ABSTRACT

Title of Dissertation: EFFICIENT STRING AND GRAPH ALGORITHMS IN MODERN SETTINGS

Jacob Gilbert
Doctor of Philosophy, 2025

Dissertation Directed by: Professor MohammadTaghi Hajiaghayi
Department of Computer Science

Fundamental computer science problems such as edit distance have been well-studied for decades, and algorithms for these problems have become established and widespread. In modern times, however, achieving even faster algorithms for more specific scenarios finds increasing value as dataset sizes become larger and computer application settings become more varied. While a polynomial-time solution in the static sequential setting may be known and straightforward to implement, it may not be quick in an online setting or take advantage of structural properties inherent to the problem setting that could lead to a more efficient solution. Parameterized algorithms and algorithms specifically designed for parallel or dynamic settings help address these gaps in algorithm utility and understanding, bringing a fresh lens to old problems.

In this work, we start by studying algorithms for string, tree, and Dyck edit distance with runtimes parameterized by the distance itself, termed the "bounded-distance" regime. We begin by giving an $\mathcal{O}(n + \mathrm{poly}(k))$-time algorithm for tree edit distance, answering a longstanding open question on the existence of such an algorithm (where k represents the distance). Expanding on

our tree edit distance approach, we then give the first $\mathcal{O}(n+\mathrm{poly}(k))$-time algorithms for weighted string, weighted tree, and weighted Dyck edit distance.

Beyond the bounded-distance regime, we give dynamic solutions for tree and Dyck edit distance. For each, we provide novel dynamic reductions to string edit distance matching state-of-the-art approximation factors from the static setting. To do so, we utilize a novel dynamic heavy-light decomposition implementation of our own complemented by a dynamic strings data structure from the literature.

Finally, we consider edit distance and related string problems in the Massively Parallel Computations (MPC) model. We give an approximation algorithm for edit distance in subpolynomial rounds with subpolynomial approximation factor, and conjecture on the hardness of the problem in MPC. We then end with constant-round MPC algorithms for longest palindrome, longest common substring, and suffix tree construction.

EFFICIENT STRING AND GRAPH ALGORITHMS IN MODERN SETTINGS

by

Jacob Gilbert

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Advisory Committee:
Professor MohammadTaghi Hajiaghayi, Chair/Advisor
Professor Uzi Vishkin, Dean's Representative
Dr. S. Cenk Sahinalp
Professor William Gasarch
Professor David Mount

© Copyright by
Jacob Gilbert
2025

Dedication

To my friends, your patience allows me to better myself every day.

Also, thank you to Ms. Blackstone for bumping up my AP Computer Science grade from an A- to an A, which felt very important at the time but in retrospect, may not have really been.

Acknowledgments

I would like to first thank my advisor MohammadTaghi Hajiaghayi for always helping me throughout these many years, especially at the beginning when I was still new to the research world. Similarly, thank you to Max Bender and Kirk Pruhs for helping me onto the academic path as an inexperienced undergraduate. Additionally, this work could not have been completed without the generous help of my co-authors Max Bender, Kirk Pruhs, Aditya Krishnan, MohammadTaghi Hajiaghayi, Alireza Farhadi, Saeed Seddighin, Hamed Saleh, Debarati Das, Tomasz Kociumaka, Barna Saha.

Special thanks goes to my graduate committee members, Uzi Vishkin, William Gasarch, David Mount, and my future postdoctorate advisor S. Cenk Sahinalp.

Finally thank you to Tom Hurst, Migo Gui, Harald Räcke, Jan Olkowski, Chih Hao Wu, Salem Malikic, Marina Knittel, Max Springer, Hamza Iseric, Auguste Gezalyan, Ananth Hari, and Yuelin Liu for your help on many things outside of this thesis along the way.

I could not have done it without you all.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents iv

Chapter 1: Introduction 1
- 1.1 Bounded-Distance Regime . . . 2
  - 1.1.1 Unweighted Tree Edit Distance in the Bounded Regime . . . 3
  - 1.1.2 Weighted String, Tree, and Dyck Edit Distance . . . 4
- 1.2 Dynamic Dyck and Tree Edit Distance . . . 6
- 1.3 Massively Parallel String Problems . . . 7
  - 1.3.1 MPC Edit Distance . . . 8
  - 1.3.2 Massively Parallel Suffix Tree and Related Problems . . . 9
- 1.4 Roadmap . . . 10

I Bounded Distance Regime 11

Chapter 2: O(n+poly(k))-time Algorithm for Bounded Tree Edit Distance 12
- 2.1 Introduction . . . 12
- 2.2 Technical Overview . . . 17
- 2.3 Preliminaries . . . 25
  - 2.3.1 Strings . . . 25
  - 2.3.2 Edit-distance Alignments . . . 26
  - 2.3.3 Trees and Forests . . . 29
  - 2.3.4 Parenthesis Representation of Forests . . . 30
- 2.4 Labeling Refinements . . . 32
  - 2.4.1 Look-Ahead Refinement . . . 32
  - 2.4.2 Compatibility Refinement . . . 40
- 2.5 Periodicity Reduction in Strings . . . 41
- 2.6 Horizontal Periodicity Reduction . . . 43
- 2.7 Vertical Periodicity Reduction . . . 56
- 2.8 Full Periodicity Reduction . . . 75
- 2.9 Main Algorithm . . . 79
  - 2.9.1 Partial Forest Matching . . . 79

2.9.2 Tree Edit Distance of Shallow Forests . . . 85
2.9.3 Level Sampling . . . 87
2.10 Conclusion . . . 90

Chapter 3: Weighted Edit Distance Computation: Strings, Trees and Dyck 91
3.1 Introduction . . . 91
3.1.1 Related Work . . . 93
3.1.2 Our Contribution . . . 96
3.1.3 Overview . . . 98
3.2 String Edit Distance . . . 110
3.2.1 Preliminaries . . . 110
3.2.2 Edit-Distance Alignments and Weighted Edit Distance . . . 111
3.2.3 Combinatorial Foundations . . . 115
3.2.4 Algorithm . . . 119
3.3 Tree Edit Distance . . . 125
3.3.1 Preliminaries . . . 125
3.3.2 Forest Alignments and Weighted Forest Edit Distance . . . 127
3.3.3 Combinatorial Foundations . . . 128
3.3.4 Algorithms . . . 139
3.4 Dyck Edit Distance . . . 151
3.4.1 Preliminaries . . . 151
3.4.2 Dyck Language Alignments and Weighted Dyck Edit Distance . . . 152
3.4.3 Periodicity Reduction . . . 159
3.4.4 Algorithm . . . 163
3.5 Deferred Proofs from Section 3.2 . . . 168

II Dynamic Tree and Dyck Distance 174

Chapter 4: Dynamic Dyck and Tree Edit Distance: Decompositions and Reductions to String Edit Distance 175
4.1 Introduction . . . 175
4.1.1 Roadmap . . . 180
4.1.2 Technical Overview . . . 181
4.1.3 Related Works . . . 205
4.2 Preliminaries . . . 210
4.2.1 Parenthesis Strings . . . 211
4.2.2 Tree Edit Distance Preliminaries . . . 212
4.3 Approximate Dynamic Dyck Edit Distance . . . 216
4.3.1 Dynamic Heavy-Light Decomposition . . . 225
4.4 Static and Dynamic Tree to String Edit Distance . . . 236
4.4.1 $\tilde{\mathcal{O}}(\sqrt{n})$-approximate Reduction to String Edit Distance . . . 237
4.4.2 String to Tree Alignment . . . 240
4.4.3 String to Tree Alignment Construction . . . 242
4.4.4 Tree Alignment Constraint Analysis . . . 248

4.4.5 Alignment Cost Analysis . . . 255
4.4.6 Approximate Dynamic Tree Edit Distance . . . 264
4.5 k versus $k^2 \log n$ Approximate Tree Edit Distance . . . 270
4.5.1 $\tilde{\mathcal{O}}(k)$-time implementation . . . 275
4.5.2 Dynamic Approximation Algorithm . . . 284
4.6 Further Results on Approximating Dynamic Dyck Edit Distance . . . 287
4.6.1 Large Dyck Edit Distance . . . 288
4.6.2 Small Dyck Edit Distance . . . 292
4.6.3 Combined Algorithm . . . 303
4.7 Exact Dynamic k-Dyck Edit Distance . . . 303
4.8 Missing Proofs of Section 4.3 . . . 310
4.8.1 Lazy Propagation . . . 310

III MPC String Problems 316

Chapter 5: Upper and Lower Bounds for Edit Distance in Space-Efficient MPC 317
5.1 Introduction . . . 317
5.1.1 Our Contribution and Techniques . . . 320
5.1.2 Other Related Works . . . 331
5.2 Preliminaries . . . 332
5.2.1 String Notations . . . 332
5.2.2 Trees . . . 333
5.3 Exact MPC Edit Distance . . . 333
5.4 Approximate MPC Edit Distance . . . 339
5.4.1 Proof of Lemma 5.4.11 . . . 353
5.5 Orthogonal Vector to Edit Distance in MPC . . . 356

Chapter 6: Massively Parallel Location-Sensitive String Problems 360
6.1 Introduction . . . 360
6.1.1 Further Related Works . . . 363
6.1.2 Problem Definition . . . 365
6.2 KMR-Based Solutions . . . 367
6.3 Our Results and Techniques . . . 370
6.3.1 LCPQ Oracle . . . 371
6.3.2 LPS, LCS, and the Periodicity Properties . . . 375
6.3.3 Massively Parallel Suffix Tree using LCPQ . . . 376
6.3.4 Suffix Tree in AMPC . . . 377
6.4 MPC Implementation Details . . . 378
6.4.1 Block-based Data Structures . . . 379
6.4.2 Modular Partitioning . . . 380
6.4.3 Weighted Load Balancing . . . 382
6.5 Longest Common Prefix Query Oracle . . . 383
6.6 Compressed LCPQ Oracle . . . 388
6.7 Longest Palindrome in MPC . . . 391

6.8 Longest Common Substring in MPC 401
6.9 Massively Parallel Suffix Tree 409
6.9.1 Suffix Array with the LCPQ Oracle 409
6.9.2 Suffix Tree in AMPC 412
6.10 Conclusion 415
6.11 KMR Suffix Array and Tree Proofs 416
6.11.1 Converting Suffix Array to Suffix Tree 418
6.11.2 LCS with Suffix Tree in MPC 427
6.11.3 LPS with Suffix Tree in MPC 429

Bibliography **433**

Chapter 1: Introduction

As computer scientists are met with increasingly challenging applications and an ever-expanding range of use cases, modern theoretical questions ask us to reinvent our algorithmic approaches to find new solutions to old problems. Dataset sizes are larger than ever, and problem domains are often highly specified to their task. Even longstanding polynomial time solutions taught in textbooks may not always be fast enough or the most well-suited solution to contemporary computer science scenarios. In response to these concerns, computer scientists are more often considering practical contexts and theoretical framings to devise more efficient algorithms tailored for modern settings. In recent years, a large part of algorithmic computer science literature has centered on parallel algorithms, streaming and dynamic algorithms, fixed-parameter tractable algorithms, and more beyond standard sequential, static algorithms.

This thesis attempts to dig into a few such classic problems and find new algorithmic approaches for faster asymptotic bounds in specific settings. The following work namely covers the string, Dyck, and tree edit distance problems as well as longest palindrome, longest common substring, and suffix tree. We consider edit distance in the bounded-distance regime, when there is an upper bound on the edit distance of the input instance, as well as the dynamic setting. We consider both edit distance and suffix tree in the Massively Parallel Computations model. Each of the following sections includes an explanation of the problem settings and their motivations as well as a

brief summary of our algorithmic results and approaches.

1.1 Bounded-Distance Regime

In the first part of this work, we study string edit distance and two of its variants, Dyck and tree edit distance in the bounded-distance regime. String edit distance is a fundamental measure of string similarity and finds uses in many real-world settings ranging from command-line applications to genomics. Given two strings, informally their edit distance is the minimum number of character insertions, deletions and substitutions needed to transform one string into the other. A well-known $\mathcal{O}(n^2)$-time dynamic programming solution solves string edit distance in a straightforward manner.

In the *bounded-distance regime*, we consider when there is a bound k on the edit distance of a pair of strings, and our goal is to either compute the distance exactly or output that the distance exceeds k. Fixed-parameter tractable algorithms dependent on k can be quite desirable, as many real-world scenarios are interested in finding distances between similar inputs, and so, k can be small, even constant. An $\mathcal{O}(nk)$-algorithm follows naturally from an observation that the dynamic programming table from the quadratic-time solution only needs to be filled on k diagonals on either side of the main diagonal, as cells outside of these diagonals must have distance larger than k. This solution was pushed further to a neat $\mathcal{O}(n + k^2)$ runtime thanks to Landau and Vishkin [Landau and Vishkin, 1988] by doing linear time pre-processing to quickly answer string queries and fill out each diagonal in $\mathcal{O}(k)$ time. As long as $k = \mathcal{O}(\sqrt{n})$, the $\mathcal{O}(n + k^2)$-time solution keeps pace with the original dynamic programming approach, and if k is logarithmic or even constant, it can greatly outspeed the original dynamic programming approach.

Alongside string edit distance, we study Dyck and tree edit distance. Dyck edit distance is a variant in which given a parenthesis string, the goal is to find the minimum number of parenthesis insertions, deletions, and substitutions needed to balance the string. This variant of edit distance finds uses in programming languages like HTML and JSON, in which tags must be balanced, and various other problems, such as in the closely related RNA folding problem in which nucleotides must be balanced according to specific criteria. The final variant, tree edit distance, asks for the minimum number of node insertions and deletions as well as label substitutions needed to transform one tree into another. Tree edit distance also finds applications in computational biology, such as when comparing tumor cell phylogenies, as well as databases, compiler optimization, and more.

Unlike string edit distance, while both Dyck and tree edit distance have been studied since the 1970's [Aho and Peterson, 1972, Selkow, 1977], progress beyond cubic time dynamic programming solutions was slow-going. However, recently, an $\mathcal{O}(n + k^{4+\epsilon})$-time solution for Dyck edit distance was found using a trapezoid decomposition of the parenthesis string complemented by fast matrix multiplication [Fried et al., 2024]. On the other hand, prior to this work, the best-known result for tree edit distance in the bounded-distance regime is $\tilde{\mathcal{O}}(nk^2)$ [Akmal and Jin, 2021]. This leaves a clear open question as to the existence of an $\mathcal{O}(n + \text{poly}(k))$-time algorithm for tree edit distance.

1.1.1 Unweighted Tree Edit Distance in the Bounded Regime

Our first significant contribution in this work is answering this question affirmatively with a tree edit distance algorithm running in time $\tilde{\mathcal{O}}(n+k^{15})$[1]. Our main algorithmic contribution behind

[1] $\tilde{\mathcal{O}}$ notation denotes hidden polylogarithmic factors.

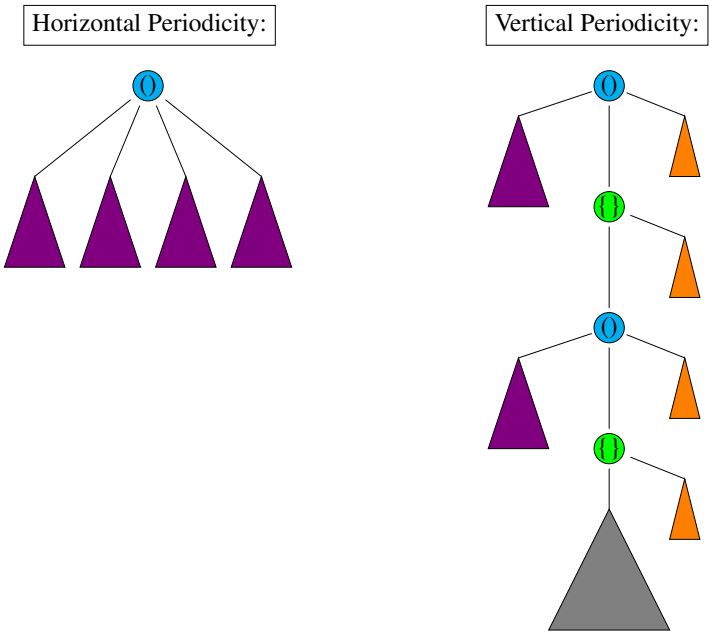


Figure 1.1: Pictured left: horizontal periodicity with string representation "([...][...][...][...])". Pictured right: vertical periodicity with string representation "([...]{([...]{[...][...]}[...])[...]}[...])".

this result is the identification of two types of periodicity in trees, which we denote *horizontal* and *vertical* (see Figure:1.1 for examples). If both input trees have matching periodic sections of the same type with large number of repetitions, most repetitions can be removed without affecting the final distance value leaving fewer unique valid alignments between the remaining tree nodes. After removing periodicity, the trees can be decomposed into smaller subproblems with bounded height, which may be solved in $\mathcal{O}(\text{poly}(k))$-time each using existing tree edit distance solutions.

1.1.2 Weighted String, Tree, and Dyck Edit Distance

Inspired by our approach to tree edit distance, we extend our methods to the weighted variants of edit distance. Weighted edit distance typically assigns non-uniform values to each edit operation dependent on the labels involved in the edit. In real-life applications, one pair of char-

acters may be considered more "similar" than a different pair of characters, e.g. a is closer to s on a standard keyboard than it is to k, which may be useful to consider in an auto-correct feature. Unfortunately, there are only few known improvements beyond a standard dynamic-programming approach for string edit distance, and even fewer for Dyck and tree edit distance.

In this work, we present:

1. an $\mathcal{O}(n + k^5)$-time algorithm for weighted string edit distance,
2. an $\mathcal{O}(n + k^7 \log k)$-time algorithm for weighted tree edit distance,
3. an $\mathcal{O}(n + k^{12})$-time algorithm for weighted Dyck edit distance.

Notably our weighted tree edit distance algorithm also improves upon the runtime of our previous algorithm for unweighted instances. We found that all three problems allow for a kernelization routine, in which an edit distance instance is reduced to an equivalent instance of size $\mathcal{O}(\mathrm{poly}(k))$ in linear time. To find the kernel, first we identify and remove any long matching periodic sections in the input pair, as done in the previous unweighted tree edit distance algorithm. In the resulting instances with limited periodicity, we observe that any remaining long matching sections must be matched by any valid alignment, up to an $\mathcal{O}(\mathrm{poly}(k))$-size boundary, and can therefore be reduced to just their boundaries. By finding a decomposition of the input instance into $\mathcal{O}(k)$ long matching sections and $\mathcal{O}(k)$ remaining unmatched characters (or nodes, for tree edit distance), we can then use our previous observation to reduce all matching sections, so the total instance becomes a kernel of size $\mathcal{O}(\mathrm{poly}(k))$. We can then use standard algorithms from the literature to calculate the edit distance on the kernel in time $\mathcal{O}(\mathrm{poly}(k))$.

1.2 Dynamic Dyck and Tree Edit Distance

In the second part of this work, we move on from the bounded-distance regime and study *dynamic* algorithms for edit distance. In the dynamic setting, we are interested in maintaining solutions to problem instances that incur dynamic updates. For example, dynamic string edit distance seeks to maintain the distance calculation of a given pair of strings using only minimal time to update the calculation when either input string undergoes a dynamic character insertion or deletion. Dynamic algorithms are desirable in applications that allow users to modify the input at any time. In real-world settings, datasets may be added to or removed from or may be fed slowly into an algorithm, and as such, being able to efficiently maintain solutions in response to dynamic updates can be critical. Dynamic string edit distance with $\tilde{\mathcal{O}}(n)$ update time has been recently solved [Charalampopoulos et al., 2020b], but no prior non-trivial algorithms for dynamic Dyck and tree edit distance are known.

Our main contributions in the dynamic setting are novel approximate reductions from Dyck and tree edit distance instances to string edit distance instances that can be maintained in poly-logarithmic time. For Dyck edit distance, we are able to give an $\mathcal{O}(\log n)$-approximate reduction, matching the static state-of-the-art Dyck-to-string reduction. For tree edit distance, we give a novel $\tilde{\mathcal{O}}(\sqrt{n})$-approximate reduction, which beats the previous best tree-to-string reduction with approximation factor $\mathcal{O}(n^{3/4})$ [Akutsu et al., 2010a]. Dynamic reductions to string edit distance are especially powerful, as any current or future dynamic algorithms for string edit distance may be used to solve instances of Dyck and tree edit distance using our reductions.

A novel dynamic heavy-light decomposition is at the center of both reductions. Given a tree, a node of the tree is heavy if the logarithm of its subtree size is equal to the logarithm of its parent's

subtree size. Like a typical heavy-light decomposition, with this definition any root-to-leaf path has at most $\mathcal{O}(\log n)$ light nodes and heavy paths. With a careful analysis, we show that a dynamic node insertion or deletion only incurs $\mathcal{O}(\log n)$ updates to the heavy-light decomposition. These changes occur specifically at the boundaries of heavy paths, and can all be found and updated in $\mathcal{O}(\log^5)$ time.

In addition to the above reductions, we also return to the bounded-distance regime and give a novel dynamic approximation algorithm for tree edit distance with $\mathcal{O}(k^2 \log n)$-approximation factor and $\tilde{\mathcal{O}}(1)$ update time. The algorithm takes a similar, though much simpler, approach to our previous bounded-distance regime algorithms. We recursively decompose one of the input trees and try to greedily match the resulting pieces with perfectly matching pieces of the other tree. When two pieces are matched, they are removed from further consideration except a boundary of size $\mathcal{O}(k)$ on either end of the pieces. We show that executing this recursive approach for at most $\mathcal{O}(k \log n)$ iterations leads to an alignment between the two trees with our desired approximation factor, and we can amortize the update time across $\mathcal{O}(k)$ dynamic edits to achieve $\tilde{\mathcal{O}}(1)$ update time.

1.3 Massively Parallel String Problems

In the final part of this work, we study a few classic string problems in the Massively Parallel Computations model. The Massively Parallel Computations (MPC) model is the theoretical backing behind popular distributed processing frameworks MapReduce, Spark, and Hadoop. The goal of these applications is to be able to efficiently process large datasets that may be stored across multiple machines. Efficient algorithms designed for MPC are desirable as the size of real datasets

trends larger as technological capabilities improve.

In MPC, there are M machines connected in a complete graph network and each has sublinear memory, i.e., memory size $S = \mathcal{O}(n^{1-\epsilon})$ memory for some $0 < \epsilon < 1$. Machines can communicate in rounds to any other machine as long as the total size of all messages passed in a single round by a machine is no larger than its local memory size. The bottleneck of MPC algorithms tends to be the number of communication rounds, and so, the goal is often to find algorithms working in constant or at worst logarithmic rounds. Furthermore, in this work we are interested in *space-efficient* algorithms, which refers to MPC algorithms where the total memory used across all machines is $M \cdot S = \tilde{\mathcal{O}}(n)$.

1.3.1 MPC Edit Distance

We continue our studies of string edit distance, now in space efficient MPC. As a starting point, the dynamic programming table from the classic $\mathcal{O}(n^2)$-time solution can be broken into tiles of size S by S and then computed in diagonal fashion. Unfortunately, this exact solution takes n^ϵ communication rounds. In order to provide any improvement without exceeding nearly linear total memory, we adapt the sublinear-time approximation algorithm of Bringmann, Cassis, Fischer, and Nakos [Bringmann et al., 2022a] to MPC. With this, we achieve a subpolynomial-round algorithm with subpolynomial approximation factor and nearly linear total memory, with a trade-off parameter between these three metrics that can be adjusted. The algorithm works by partitioning one of the input strings into individual characters and then recursively combining pieces of the partition. For each piece in the partition we maintain an edit distance approximation between the piece and several pieces in the other input string. Each time we combine a set of

smaller pieces into a larger piece, we calculate the new edit distance approximations from the approximations of the small pieces. The main challenge of the MPC adaptation that we overcome is that the number of approximations per recursive level in the algorithm is chosen randomly, with some levels even requiring $\Omega(n)$ approximation calculations, though not necessarily stored simultaneously. We therefore cannot assign machines at the start of the algorithm, and instead give a dynamic scheduling routine to efficiently assign machines to approximation calculations as needed.

In addition to the aforementioned upper bounds, we conjecture that exact edit distance has a lower bound of $n^{\epsilon-\Omega(1)}$ communication rounds in MPC. To inform our conjecture, we look at the *orthogonal vector* problem in MPC, which asks if any pair of vectors are orthogonal given two sets of n vectors each. Orthogonal vector is a well-established problem in fine-grained complexity, and there are seemingly no meaningful improvements in MPC for this simple problem beyond a trivial approach that multiplies all pairs of vectors in $n^{\epsilon-\Omega(1)}$ rounds, which we call the *Orthogonal Vector Conjecture*. We show that the sequential reduction from orthogonal vector to edit distance holds in MPC in constant rounds, as well as a set of other problems, e.g. graph diameter, longest common substring, etc, and therefore, we give a class of problems we conjecture are polynomial-round hard in MPC.

1.3.2 Massively Parallel Suffix Tree and Related Problems

Finally, we end this work by considering the problem of suffix tree along with related problems longest palindrome and longest common substring in MPC. Given a string, the *suffix tree* is a trie-like data structure storing the sorted suffixes of the input string. Suffix trees are useful data

structures in algorithms for a number of other applications, viz., longest palindrome and longest common substring. *Longest palindrome* asks to identify the titular longest palindrome in a given input string, and *longest common substring* seeks the longest substring common to a pair of input strings.

We achieve an $\mathcal{O}(1/\epsilon)$-round suffix tree construction in MPC with $\mathcal{O}(n^{1+\epsilon})$ total memory, and give constant-round MPC algorithms for longest palindrome and longest common substring using suffix tree. We then study longest palindrome and longest common substring independently. We are able to improve the round complexity to exactly $\mathcal{O}(1)$ for both without using suffix tree and improve the total memory for longest palindrome to $\tilde{\mathcal{O}}(n)$ as well. At the heart of all three of these algorithms is a novel longest common prefix query oracle that we use to answer batches of longest common prefix queries of our input strings. The algorithms rely on the oracle to find maximum-length matching substrings in constant rounds, a key step for each problem.

1.4 Roadmap

The first part of this thesis is concerned with the bounded-distance regime. In Chapter 2, we discuss an $\mathcal{O}(n + \mathrm{poly}(k))$-time algorithm for tree edit distance. In Chapter 3, we discuss $\mathcal{O}(n + \mathrm{poly}(k))$-time algorithms for weighted variants of edit distance. In the second part of the thesis, contained in Chapter 4, we discuss dynamic Dyck and dynamic tree edit distance. In the third and final part of the thesis, we discuss MPC string algorithms. In Chapter 5, we show upper and lower bounds for edit distance in MPC. Finally, in Chapter 6, we give results and related proofs for suffix tree, longest palindrome, and longest common substring in MPC.

Part I

Bounded Distance Regime

Chapter 2: O(n+poly(k))-time Algorithm for Bounded Tree Edit Distance

2.1 Introduction

Computing the *edit distance* of two strings is one of the most fundamental problems in theoretical computer science and combinatorial optimization studied since the 1960s. *Tree edit distance* is a natural generalization of edit distance in which the task is to compute a measure of dissimilarity between two (unweighted) rooted trees with node labels in which every insertion, deletion, or relabeling operation has unit cost. Tree edit distance, first introduced by Selkow [Selkow, 1977], extends the applications of edit distance to areas such as computational biology (e.g., analysis of RNA molecules, where the secondary structure of RNA is represented as a rooted tree) [Bille, 2005, Gusfield, 1997, Harel and Tarjan, 1984, Shapiro and Zhang, 1990, Landau et al., 1998], structured data analysis (e.g., XML) [Buneman et al., 2003, Chawathe, 1999, Ferragina et al., 2009], image analysis [Bunke and Shearer, 1998], and compiler optimization [Demaine et al., 2010]. Perhaps the most notable recent application of tree edit distance is in NoSQL big databases, such as MongoDB [MongoDB, 2022], where each row of the database is a JSON document represented as a labeled rooted tree, and finding dissimilarity between two rows is a basic operation.

The computational aspect of tree edit distance is also widely studied. Tai [Tai, 1979] gave the first solution for tree edit distance that runs in time $\mathcal{O}(n^6)$, where n is the total number of nodes in both trees. This running time was later improved in a series of works to $\mathcal{O}(n^4)$ [Zhang and Shasha,

1989], $\mathcal{O}(n^3 \log n)$ [Klein, 1998], and $\mathcal{O}(n^3)$ [Demaine et al., 2010]. Very recently, Mao [Mao, 2021] broke the cubic barrier by showing an $\mathcal{O}(n^{2.9546})$-time algorithm via a reduction to max-plus product of bounded-difference matrices. In a follow-up preprint, Dürr further improved the running time to $\mathcal{O}(n^{2.9149})$ [Dürr, 2023].

Boroujeni, Ghodsi, Hajiaghayi, and Seddighin [Boroujeni et al., 2019] presented a $(1+\epsilon)$-approximation algorithm for tree edit distance that runs in $\tilde{\mathcal{O}}(n^2)$ time.[1] They also obtained an $\mathcal{O}(\sqrt{n})$-factor approximation algorithm that runs in $\tilde{\mathcal{O}}(n)$ time. Very recently, Seddighin and Seddighin [Seddighin and Seddighin, 2022] gave an $\mathcal{O}(n^{1.99})$-time $(3+\epsilon)$-approximation algorithm for tree edit distance.

The problem has also been considered from the lower-bound perspective. In particular, Bringmann, Gawrychowski, Mozes, and Weimann [Bringmann et al., 2020] proved that the cubic running time barrier for weighted tree edit distance cannot be broken unless APSP admits a truly subcubic time solution and weighted k-clique admits an $\mathcal{O}(n^{k-\epsilon})$-time solution. The existence of such a lower bound was previously conjectured by Abboud [Abboud, 2014] in a collection of open problems in fine-grained complexity.

In contrast to tree edit distance, approximation algorithms for string edit distance have been subject to many studies [Andoni et al., 2010, Andoni and Onak, 2012, Bar-Yossef et al., 2004, Batu et al., 2006, Haeupler et al., 2019, Indyk, 2001, Landau et al., 1998, Boroujeni et al., 2021, Chakraborty et al., 2020, Andoni and Nosatzki, 2020]. After a series of recent developments [Boroujeni et al., 2021, Chakraborty et al., 2020, Goldenberg et al., 2020, Koucký and Saks, 2020, Brakensiek and Rubinstein, 2020], the current best bound is an algorithm by Andoni and Nosatzki [Andoni and Nosatzki, 2020], for any constant $\epsilon > 0$, provides a constant-factor approx-

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylogarithmic factors.

imation of edit distance between strings of length n in time $\mathcal{O}(n^{1+\epsilon})$.

Given a parameter k as an upper bound on the distance, an $\mathcal{O}(n+k^2)$-time algorithm for edit distance is known since the 1980s, due to Myers [Myers, 1986] and Landau and Vishkin [Landau and Vishkin, 1988], who combined suffix trees with an elegant greedy algorithm. The existence of such an $\tilde{\mathcal{O}}(n+\mathrm{poly}(k))$-time algorithm for tree edit distance (even for unlabeled trees) remained open despite persistent effort from researchers in the field. In particular, this question was posed by Mao [Mao, 2021] and Akmal and Jin [Akmal and Jin, 2021]. The current fastest algorithm for the problem runs in $\tilde{\mathcal{O}}(nk^2)$ time [Akmal and Jin, 2021], improving on the previous results of $\mathcal{O}(nk^3)$ time by Touzet [Touzet, 2005]. In this paper, we answer this open question affirmatively by providing an $\tilde{\mathcal{O}}(n+k^{15})$-time algorithm for bounded tree edit distance.

The previous algorithms for computing tree edit distance [Tai, 1979, Zhang and Shasha, 1989, Klein, 1998, Demaine et al., 2010, Mao, 2021] are dynamic-programming-based procedures, and the solutions for bounded tree edit distance [Touzet, 2005, Akmal and Jin, 2021] are obtained by appropriately pruning the set of states in earlier general-purpose algorithms. Our strategy is very different: the main effort is to greedily match all but $\mathcal{O}(\mathrm{poly}(k))$ nodes of the input trees so that any polynomial-time algorithm can be used to solve the residual instances of the problem. The greedy approach is sufficiently powerful only for trees avoiding certain synchronized periodicity, and therefore we start with a preprocessing step that eliminates appropriate periodic structures. Although such an approach is fairly simple to realize for strings, implementing it on trees requires several novel components of their own interest. This is because, so far, the underlying techniques have not been used on trees, and this setting brings many challenges absent in the context of strings. For example, periodicity in trees comes in two flavors, which we name *vertical* and *horizontal*, and we provide efficient procedures detecting both kinds. Another obstacle is that, whereas greedily

matching two characters yields two independent instances of the string edit distance problem, greedily matching two nodes does not produce two independent instances of the tree edit distance problem, and thus we need a dedicated algorithm to optimally extend a partial alignment to a complete alignment of two trees. For more details, see the technical overview in Section 2.2.

Finally, a problem related to (tree) edit distance is the *Dyck edit distance* problem, in which, given a sequence of n parentheses, the task is to find the minimum number of edits (character insertions, deletions, and substitutions) needed to make the sequence well-balanced. The Dyck edit distance has numerous applications [Harrison, 1978, Kozen, 1997] and has been subject to many theoretical studies designing exact [Bringmann et al., 2019, Chi et al., 2022a, Backurs and Onak, 2016, Fried et al., 2022, Dürr, 2023] and approximation algorithms [Saha, 2014, Das et al., 2022b]. It is also known that this problem is at least as hard as Boolean matrix multiplication [Abboud et al., 2018]. Though Dyck edit distance problem has a different flavor than tree edit distance (since the goal is to find the minimum number of edits to completion, i.e., to a target of well-balanced parentheses), the existence of an $\tilde{\mathcal{O}}(n+\mathrm{poly}(k))$-time algorithm for the bounded version was still a very important open problem (motivated by fixing hierarchical data files, such as XML and JSON). Backurs and Onak [Backurs and Onak, 2016] solved the open problem by providing an exact algorithm for Dyck edit distance that runs in $\mathcal{O}(n+k^{16})$ time, which has recently been improved by Fried, Golan, Kociumaka, Kopelowitz, Porat, and Starikovskaya [Fried et al., 2022] to run in $\mathcal{O}(n+k^5)$ time, and further to $\tilde{\mathcal{O}}(n+k^{4.5442})$ using fast matrix multiplication [Fried et al., 2022, Dürr, 2023]. Dyck edit distance falls under the umbrella of a general language edit distance problem [Bringmann et al., 2019], which, however, is at least as hard as Boolean matrix multiplication already for $k=0$.

Our Result

While in edit distance, the goal is to transform a string S into another string S', in tree edit distance the goal is to transform a tree T into another tree T' using the least number of edit operations. In the most common version of the problem, it is assumed that both trees T and T' are rooted and that there is a left-to-right order between the children of any node. Moreover, each node has a label (independent of its the degree). The elementary operations are node deletion, node insertion, and node relabeling. In node deletion, we remove a node v and replace it with all of its children, preserving their order. The reverse of node deletion is node insertion, which allows us to select a consecutive set of siblings and bring them under a new node v which appears at the previous position of the relocated nodes. A node relabeling simply modifies the label of an existing node.

In fact, we solve a slightly more general problem of computing the edit distance between two labeled forests, which are defined as sequences of labeled, rooted, and ordered trees. For two labeled forests F and G, we denote their tree edit distance with $\mathsf{ted}(F, G)$. Moreover, for a threshold k, we denote with $\mathsf{ted}_{\leq k}(F, G)$ a value equal to $\mathsf{ted}(F, G)$ (if it is at most k) or ∞ (otherwise). The main result of this paper is summarized in the following theorem:

Theorem 2.1.1. *There exists a randomized algorithm that, given forests F, G of total size n and an integer $k \in \mathbb{Z}_+$, computes $\mathsf{ted}_{\leq k}(F, G)$ in $\mathcal{O}(n \log n + k^{15} \log k \log n)$ time correctly with high probability.*

2.2 Technical Overview

Given two strings $X, Y \in \Sigma^{\leq n}$ and a threshold $k \in \mathbb{Z}_{\geq 0}$, the classic Landau–Vishkin algorithm [Landau and Vishkin, 1988] computes $\mathsf{ed}_{\leq k}(X, Y)$ in time $\mathcal{O}(n + k^2)$. The algorithm uses dynamic programming: for each $i \in [0\,..\,k]$ and $j \in [-k\,..\,k]$, it computes an index $d_{i,j} = \max\{x : \mathsf{ed}(X[0\,..\,x), Y[0\,..\,x + j)) \leq i\}$. In terms of the standard quadratic-size DP table, $d_{i,j}$ can be interpreted as the row of the farthest cell on the jth diagonal that can be reached with cost at most i. After a linear-time reprocessing of X, Y, each value $d_{i,j}$ can be computed in $\mathcal{O}(1)$ time; thus, the algorithm takes $\mathcal{O}(n + k^2)$ time in total. From the definition of $d_{i,j}$, we can observe that the alignments produced by the Landau–Vishkin algorithm satisfy the following greedy property: if a prefix $X[0\,..\,x)$ is aligned to a prefix $Y[0\,..\,y)$ and the characters $X[x] = Y[y]$ match, then these two characters are also aligned (instead of being deleted). This greedy matching strategy is crucial in achieving $\mathcal{O}(n + \mathrm{poly}(k))$ time complexity as it allows the algorithm to focus on $\mathcal{O}(\mathrm{poly}(k))$ mismatches and quickly slide through the bulk of the input strings.

A natural question is whether a similar greedy strategy can be applied in the context of the edit distance of two labeled forests F and G. While there could be several ways to formalize such a greedy property, all definitions should capture the following scenario: if the leftmost roots of F and G have the same label, we should be able to greedily match these roots. Unfortunately, this is not the case, as illustrated in Fig. 2.1.

However, it is fairly obvious that if the entire leftmost trees of F and G are the same, then these trees can be matched greedily. In general, as proved in Proposition 2.4.4, if, while constructing an alignment, we are given a chance to match a node u in F with a node v in G (or to delete one or both of these nodes), then we can greedily match the two nodes provided that the entire subtrees

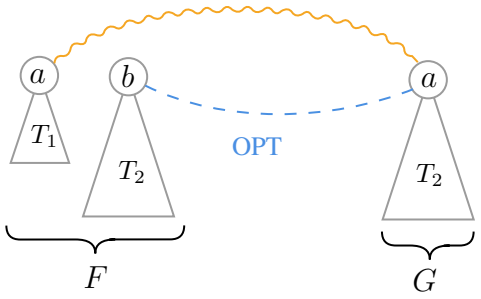


Figure 2.1: An example of simple greedy strategy where the alignment matches the leftmost roots, both having labels a. However, the unique optimal alignment deletes the entire tree T_1 and substitutes the label at the node of T_2 from b to a.

rooted at u and v, respectively, are identical. We call alignments following this principle *greedy alignments*. Unfortunately, these subtrees can be large. Thus, first we need to design a process that, for any node $u \in V_F$ (or $v \in V_G$), can efficiently encode the information (e.g., structure, labeling) of the subtree rooted at u. We do it using *Look-Ahead Labeling* discussed next.

Look-Ahead Labelling: Given unlabeled forests F, G and a labeling $\lambda : V_\mathsf{F}, V_\mathsf{G} \to \Sigma$ (here, V_F stands for the node set of F) and a parameter d, the depth-d look-ahead labeling $\lambda' = L(\lambda, d)$ satisfies the following property for any pair of nodes $u, v \in V_F \cup V_G$: $\lambda'(u) = \lambda'(v)$ if and only if $\mathsf{ted}(\mathsf{sub}_{\leq d}(u), \mathsf{sub}_{\leq d}(v)) = 0$. Here, $\mathsf{sub}_{\leq d}(u)$ denotes the λ-labeled subtree rooted at u trimmed to depth d.

To construct λ', we first define the parentheses representation of F and G with respect to labeling λ. For forest F with labeling λ, this is denoted as $\mathsf{P}_\lambda(\mathsf{F})$ and can be defined using the following recursion: If F consists of trees $\mathsf{T}_1, \ldots, \mathsf{T}_m$, then $\mathsf{P}_\lambda(\mathsf{F}) = \bigodot_{i=1}^{m} \mathsf{P}_\lambda(\mathsf{T}_i)$, where $\mathsf{P}_\lambda(\mathsf{T}_i) = (_{\lambda(v_i)} \cdot \mathsf{P}_\lambda(\mathsf{F}_i) \cdot)_{\lambda(v_i)}$ and $\cdot$ as well as $\bigodot$ denote concatenation. Here, v_i is the root of T_i and F_i is the forest obtained from T_i by removal of v_i. To create λ' in linear time, we replace the label of each node $u \in V_\mathsf{F} \cup V_\mathsf{G}$ with a Karp–Rabin fingerprint of $P_\lambda(\mathsf{sub}_{\leq d}(v))$. Note by setting d larger than the heights of F and G, we can create a labeling that for each node encodes the entire

subtree rooted at that node. The details can be found in Section 2.4.1.

As an additional challenge, it turns out that this greedy approach is beneficial only if the input trees F, G avoid certain periodic structures, which may be present in the input forests. Thus, we first design an $\tilde{\mathcal{O}}(n)$-time algorithm that, given arbitrary labelled forests F, G, produces a pair of forests F', G' such that $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$ and both F', G' avoid synchronized *horizontal* and *vertical* k-periodicity, the two types of periodicity which we describe in the following paragraphs. Moreover, with an $\tilde{\mathcal{O}}(n + k^3)$-time post-processing, we also compute an alignment $\tilde{\mathcal{A}}$ mapping $\mathsf{P}(F')$ to $\mathsf{P}(G')$ such that if we consider any optimal alignment $\mathcal{B}$ between F' and G' (and identify it with the corresponding alignment of $\mathsf{P}(F')$ and $\mathsf{P}(G')$), then $\tilde{\mathcal{A}}$ and $\mathcal{B}$ differ at $\mathcal{O}(k^4)$ positions. Next, we discuss the key ideas of the periodicity reduction process. We remark that this is the first algorithm that provides an efficient way of reducing periodicity in trees or forests, and it can be of independent interest.

Periodicity Reduction: One of the most important novel contributions of our algorithm is the definition and identification of periodicity in forests. An integer p is *period* of a string S if $S[i] = S[i + p]$ holds for all $i \in [0\,..\,n - p)$; we then call $\frac{|S|}{p}$ the *exponent* of this period of S. If $p \leq k$, then S is called k-periodic. Ideally, we would like to identify and reduce the exponent of periodic regions of forests F and G somehow. Given two strings S and T, if there exist periodic fragments $Q = S[\alpha_s\,..\,\beta_s) = T[\alpha_t\,..\,\beta_t)$ such that $|\alpha_s - \alpha_t| \leq 2k$, we say that there are k-synchronized occurrences of Q in S and T. Synchronized periodicity in strings has previously been studied for string edit distance algorithms that take advantage of such periodicity when constructing optimal alignments (e.g., in [Kociumaka et al., 2021]), but never in forests. Unfortunately, reducing periodic fragments in the parentheses representations of forests does not preserve

tree edit distance. In particular, if a periodic substring is unbalanced, then any edit to an opening parenthesis in the periodic region also needs to be applied to the corresponding closing parenthesis, which may lie outside the periodic region. We propose and identify two new types of "balanced" periodicity in forests, *horizontal* and *vertical* periodicity. By constructing forests F' and G' which avoid k-synchronized horizontal and vertical k-periodicity in F and G without affecting the total edit distance between the forests, we prove that we also avoid any periodicity in the underlying parentheses representations of F and G (with an appropriate labeling).

In Section 2.6, the first type of periodicity we identify and reduce is *horizontal periodicity*. If a sequence of subtrees of a given node repeats periodically, then we consider the repeated forest as a horizontal period. Therefore, horizontal periodicity appears in the parentheses representation of F and G as a balanced periodic fragment. Since horizontal periods are already balanced, any edits can be applied locally within the horizontally periodic fragment. By computing all maximal periodic fragments in the parentheses representations of F and G, we can find and reduce all horizontally periodic regions at nearby locations in both forests. In order to avoid interactions between overlapping horizontally periodic regions, we are careful to only reduce horizontal periodicity with large exponent and small period. Algorithm 3 gives detailed pseudocode for the $\tilde{\mathcal{O}}(n)$-time construction of forests F' and G' which avoid k-synchronized horizontal k-periodicity from input forests F and G.

In Section 2.7, the second type of periodicity we identify and reduce is *vertical periodicity*. Vertical periodicity occurs in a forest when there is some path in which the subtrees to the left and right of the path repeat periodically at subsequent levels of the path (see Fig. 2.4). To find all vertical periods in a forest F, we identify nodes whose corresponding opening and closing parenthesis are each contained in periodic substrings of the parentheses representation $\mathsf{P}_\lambda(\mathsf{F})$. Once

we find all vertical periods of F and G, we use orthogonal range queries (in two dimensions) to find nodes $u \in V_F$ and $v \in V_G$ which are both contained in a vertically periodic region and whose opening parentheses and closing parentheses are in similar locations of $\mathsf{P}_\lambda(\mathsf{F})$ and $\mathsf{P}_\lambda(\mathsf{G})$. Once we find all such pairs of nodes, we can reduce the exponent of the periodic path as in the case of horizontal periods. Algorithm 7 performs these vertical periodic reductions in $\tilde{\mathcal{O}}(n)$-time without introducing any new horizontal k-periodicity, and it outputs forests F' and G' which avoid both k-synchronized horizontal and vertical k-periodicity.

In Section 2.8, we prove that since F' and G' avoid k-synchronized horizontal and vertical k-periodicity of large exponent, then $\mathsf{P}_\lambda(\mathsf{F}')$ and $\mathsf{P}_\lambda(\mathsf{G}')$ avoids all k-synchronized $2k$-periodicity of large exponent, where λ is an appropriate refinement of the depth-$8k$ look-ahead labeling. We show that any periodic substring of a forest's parentheses representation must be either horizontal or vertical period depending on whether the period string is balanced or unbalanced. At the end of the section, using an $\mathcal{O}(n + k^3)$-time DP algorithm described in Section 2.3, we compute an alignment $\tilde{\mathcal{A}}$ of $\mathsf{P}_\lambda(F')$ and $\mathsf{P}_\lambda(G')$ that differs from any optimal tree alignment of F' and G' on at most k^4 characters.

Given the two trees F' and G' that avoid synchronized horizontal k-periodicity and synchronized vertical k-periodicity, in Section 2.9.5 we design an algorithm using the greedy strategy that computes $\mathsf{ted}_{\le k}(F', G')$ in time $\tilde{\mathcal{O}}(n + h^2k^7)$, where h is an upper bound on the height of F' and G'. We provide a brief sketch of the algorithm in the following.

Tree Edit Distance of Shallow Forests: Here, we first show that if we consider any optimal alignment $\mathcal{A}$ between F' and G', then, for all but $2hk$ pairs of nodes (u, v) where $u \in V_{F'}$ and $v \in V_{G'}$, $\mathcal{A}$ matches u with v and the subtree rooted at u is identical as the one rooted at v. If we assume

that F' and G' avoid synchronized horizontal k-periodicity, this allows constructing a large set of matching pairs $M \subseteq \{(u, v) : u \in V_F, v \in V_G\}$ (of size $|V_F| - \mathcal{O}(h^2k^4)$) that is common to all optimal greedy alignments; i.e., each pair $(u, v) \in M$ is matched by every greedy alignment and the subtrees rooted at u and v are at distance 0. The time required to construct M is $\mathcal{O}(n + hk^2)$.

Given this partial matching M common to every optimal greedy alignment, our next objective is to extend it in order to construct an optimal alignment between F' and G'. An analogous task is relatively straightforward for strings: given a partial matching, we can first partition the strings along this matching and then independently compute the optimal distance between subsequent pieces. This strategy fails for trees, but we still provide a linear-time algorithm that, given a pair of forests F' and G' and a non-crossing partial matching M between them, constructs forests F'' and G'' such that $\mathsf{ted}^M_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F'', G'')$ and $|F''| + |G''| = \mathcal{O}(k(|F'| + |G'| - 2|M|))$. Here, $\mathsf{ted}^M_{\leq k}(F', G')$ is the minimum of the cost of all alignments $\mathcal{A}$ between F', G' such that each pair of nodes in M is also matched by $\mathcal{A}$ (or ∞ if that cost exceeds k).

Thus, following this construction and using the fact that the partial matching M is common to every optimal greedy alignment between F', G' (thus $\mathsf{ted}^M_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F', G')$) and $|M| = |V_F| - \mathcal{O}(h^2k^4)$, in linear time we can construct trees F'', G'' such that $\mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F'', G'')$ and $|F''| + |G''| = \mathcal{O}(h^2k^5)$. Next using the $\tilde{\mathcal{O}}(nk^2)$ time algorithm of [Akmal and Jin, 2021], we compute $\mathsf{ted}_{\leq k}(F'', G'')$ in time $\tilde{\mathcal{O}}(h^2k^7)$. Below, we discuss how to extend a partial matching to a complete alignment.

Partial Forest Matching: As mentioned earlier, here, given a pair of forests F' and G' and a non-crossing partial matching M, our objective is to construct forests F'' and G'' such that $\mathsf{ted}^M_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F'', G'')$ and $|F''| + |G''| = \mathcal{O}(k(|F'| + |G'| - 2|M|))$. Also, if the height of F'' (or G'') is

$h > 2$, then there is a length $h-2$ top down-path in F' avoiding nodes from M. Thus, if M is large and hits all long paths, then this significantly reduces both the size and the depth of the input forests while preserving their distance.

We start by partitioning F' and G' along the matching pairs in M, creating forests $\hat{F}$ and $\hat{G}$, and a set of partial matching pairs $\hat{M}$ between $\hat{F}, \hat{G}$. Our aim is to ensure that each node appearing in a matching pair of $\hat{M}$ is a leaf node of some tree in $\hat{F}$ (or $\hat{G}$) and $\mathsf{ted}^{M}(F', G') = \mathsf{ted}^{\hat{M}}(\hat{F}, \hat{G})$. For this, we mark all the nodes that are present in M and assign each node to the nearest marked proper ancestor. This creates a partition of V_F and of V_G. Next for each subset S of nodes in the partition, we create a tree by deleting all the nodes in $V_F \setminus S$ from F, and we add the tree to $\hat{F}$ (while preserving the order of the nodes as present in F). Similarly create $\hat{G}$ from G'. Note that, by construction, each node present in M appears as a leaf node of some tree in $\hat{F}$ (or in $\hat{G}$). We also copy all the matching pairs from M to $\hat{M}$. Lastly to ensure $\mathsf{ted}^{M}(F', G') = \mathsf{ted}^{\hat{M}}(\hat{F}, \hat{G})$, between each pair of consecutive trees in $\hat{F}$ and their corresponding trees in $\hat{G}$, we insert a pair of single-node trees whose roots are also added to $\hat{M}$. Now, each node present in $\hat{M}$ appears as a leaf node of some tree in $\hat{F}$ (or in $\hat{G}$) and, if height of $\hat{F}$ (or $\hat{G}$) is $h > 1$, then there is a length $h-1$ top down path in F' (or G') avoiding nodes from M.

In the second step we further reduce $\hat{M}$ by removing redundant matching pairs. The idea here is that, for each matching pair $(\hat{u}, \hat{v}) \in \hat{M}$, if their immediate left siblings u and v (respectively) are also matched by $\hat{M}$, i.e., $(u, v) \in \hat{M}$, then we can discard $(\hat{u}, \hat{v})$ to create a new set $\bar{M}$ and create forests $\bar{F}$ and $\bar{G}$ by deleting all the nodes present in some discarded matching pair. This is because (u, v) serves as a representative of $(\hat{u}, \hat{v})$ and thus reintroducing them would not violate the non-crossing property of the matching set, and we can ensure $\mathsf{ted}^{\bar{M}}(\bar{F}, \bar{G}) = \mathsf{ted}^{\hat{M}}(\hat{F}, \hat{G}) = \mathsf{ted}^{M}(F', G')$. We also show $|\hat{M}| \leq 2/5(|\hat{F}| + |\hat{G}| + 1)$.

Lastly, using $\bar{F}, \bar{G}$, and $\bar{M}$, we construct forests F'', G'' such that $\mathsf{ted}^{\bar{M}}_{\leq k}(\bar{F}, \bar{G}) = \mathsf{ted}_{\leq k}(F'', G'')$. To construct F'' and G'', we attach a gadget containing $k+1$ uniquely-labeled children to each node present in $\bar{M}$. As the cost of an optimal alignment of F'' and G'' is bounded by k, it should match at least one node from each gadget; following this, we can show the alignment will indeed match all the nodes from the gadget and from set $\bar{M}$ thus proving $\mathsf{ted}^{\bar{M}}_{\leq k}(\bar{F}, \bar{G}) = \mathsf{ted}_{\leq k}(F'', G'')$. Moreover, from the above construction and using the bound of $|\hat{M}|$, we can show $|F''| + |G''| = \mathcal{O}(\min(|F'| + |G'| + k|M|, k(|F'| + |G'| - 2|M|)))$. Also, by the gadget construction, we ensure the height of F'' (or G'') is at most the height of $\bar{F}$ (or $\bar{G}$) plus one. Thus, if the height of F'' (or G'') is $h > 2$, then there is a length $h-2$ top down path in F' (or G', respectively) avoiding M. We provide the details in Section 2.9.1.

To summarize, given arbitrary labelled forests F, G and a threshold k, we first design an algorithm that in time $\tilde{\mathcal{O}}(n)$ produces a pair of forests F', G' such that $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$ and F', G' avoid synchronized horizontal and vertical k-periodicity. At $\tilde{\mathcal{O}}(n + k^3)$ time, this algorithm also computes an alignment $\tilde{\mathcal{A}}$ between $\mathsf{P}(F')$ and $\mathsf{P}(G')$ such that, any optimal alignment $\mathcal{B}$ between F' and G', differs from $\tilde{\mathcal{A}}$ at $\mathcal{O}(k^4)$ positions. Next, using the greedy strategy and the partial matching technique, we design an algorithm that can compute $\mathsf{ted}_{\leq k}(F', G')$ in time $\tilde{\mathcal{O}}(n + h^2k^7)$, given that the heights of F' and G' is at most h. However, the challenge is that the height h can be much larger than k. To solve this issue, instead of directly computing the distance between F', G', we design Section 2.9.3, which first creates a partial matching $\tilde{M}$ using random sampling within alignment $\tilde{\mathcal{A}}$, and then extends this partial matching to construct an optimal alignment between F' and G'. The random sampling constructs $\tilde{M}$ in such a way so that, given F', G', and $\tilde{M}$, the partial matching reduction generates forests F'', G'' where $\mathsf{ted}^{\tilde{M}}_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F'', G'')$

and the heights of F'' and G'' are bounded by $O(k^4)$. Thus, one can compute the distance in time $\tilde{\mathcal{O}}(n + k^{15})$. We now give a sketch of the sampling technique.

Level Sampling Set the height threshold $h = O(k^4)$. Select the sampling parameter r from $[0\,..\,h)$ uniformly at random. Mark all the nodes in F' and G' at depth congruent to r modulo h. Next, we create a partial matching $\tilde{M}$ as follows: for each marked node $u \in V_{F'}$ (or in $V_{G'}$), add (u, v) to $\tilde{M}$, where $v \in V_{G'}$ (or $v \in V_{F'}$) and $\tilde{\mathcal{A}}$ matches the parentheses representing u with the parentheses representing v. Apply partial matching reduction on F', G' and $\tilde{M}$ to create forests F'', G'' given (i) every marked node is present in $\tilde{M}$ (ii) $|\tilde{M}| = O(n/k^4)$. Using the bound on $|\tilde{M}|$, we claim $|F''| + |G''| = O(n)$. Moreover, neither of F'', G'' has a top-down length-h path avoiding $\tilde{M}$. Hence, the height of F'', G'' is at most $h + 1 = O(k^4)$. Thus, we can compute $\mathsf{ted}_{\le k}(F'', G'')$ using the shallow forest algorithm in time $\tilde{\mathcal{O}}(n+\mathrm{poly}(k))$. Note that the partial matching reduction ensures $\mathsf{ted}^{\tilde{M}}_{\le k}(F', G') = \mathsf{ted}_{\le k}(F'', G'')$. As $\mathsf{ted}(F', G') \le k$ and any optimal alignment between F' and G' and $\tilde{\mathcal{A}}$ differs in at most $O(k^4)$ matching pairs, following our sampling strategy we further can argue that $\mathsf{ted}_{\le k}(F', G') = \mathsf{ted}^{\tilde{M}}_{\le k}(F', G')$ holds with constant probability. To ensure concentration, we repeat this $\Theta(\log n)$ times and report the minimum distance computed.

2.3 Preliminaries

2.3.1 Strings

A *string* $Y \in \Sigma^n$ is a sequence of $|Y| := n$ characters from an *alphabet* Σ. For $i \in [0\,..\,n)$, we denote the ith character of Y with $Y[i]$. The *reverse* of a string Y is $\overline{Y} := Y[n-1]Y[n-2]\cdots Y[0]$. We say that a string X *occurs* as a *substring* of a string Y if $X = Y[i]\cdots Y[j-1]$

holds for some indices $0 \leq i \leq j \leq |Y|$. We denote the underlying *occurrence* of X as $Y[i\,..\,j)$. Formally, $Y[i\,..\,j)$ is a *fragment* of Y that can be represented using a reference to Y as well as its endpoints i, j. The fragment $Y[i\,..\,j)$ can be alternatively denoted as $Y[i\,..\,j-1]$, $Y(i-1\,..\,j-1]$, or $Y(i-1\,..\,j)$. A fragment of the form $Y[0\,..\,j)$ is a *prefix* of Y, whereas a fragment of the form $Y[i\,..\,n)$ is a *suffix* of Y.

An integer $p \in [1\,..\,n]$ is a *period* of a string $Y \in \Sigma^n$ if $Y[i] = Y[i+p]$ holds for all $i \in [0\,..\,n-p)$. In this case, the prefix $Y[0\,..\,p)$ is called a *string period* of Y. By $\mathsf{per}(Y)$ we denote the smallest period of Y. The exponent of a string Y is defined as $\exp(Y) := \frac{|Y|}{\mathsf{per}(Y)}$, and we say that a string Y is *periodic* if $\exp(Y) \geq 2$.

For a string Y and an integer $m \geq 0$, we define the mth power of Y, denoted Y^m, as the concatenation of m copies of Y. For a string $Y \in \Sigma^n$, we define a *forward rotation* $\mathsf{rot}(Y) = Y[1]\cdots Y[n-1]Y[0]$. In general, a *cyclic rotation* $\mathsf{rot}^s(Y)$ with *shift* $s \in \mathbb{Z}$ is obtained by iterating rot or the inverse operation rot^{-1}. A non-empty string $Y \in \Sigma^n$ is *primitive* if it is distinct from its non-trivial rotations, i.e., if $Y = \mathsf{rot}^s(Y)$ holds only when s is a multiple of n. A string Y is primitive if and only if it cannot be expressed as $Y = X^m$ for some string X and integer $m > 1$.

2.3.2 Edit-distance Alignments

The *edit distance* (also known as the *Levenshtein distance*) $\mathsf{ed}(X, Y)$ between two strings X and Y is defined as the smallest number of character insertions, deletions, and substitutions required to transform X to Y.

Equivalently, the edit distance $\mathsf{ed}(X, Y)$ can be defined as the cost of the cheapest *alignment* $\mathcal{A} : X \to Y$. There are many ways to formally define an alignment; all of them, however, need to

identify which characters of X are *deleted* and which characters of Y are *inserted*. The remaining characters are *aligned* (either *substituted* or *matched*) in the left-to-right order. Since parts of this work rely on the techniques of [Kociumaka et al., 2021], we chose to stick to their formalization.

Definition 2.3.1. A sequence $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ is an *alignment* of a string $X \in \Sigma^*$ onto a string $Y \in \Sigma^*$, denoted $\mathcal{A} : X \to Y$, if $(x_0, y_0) = (0, 0)$, $(x_m, y_m) = (|X|, |Y|)$, and $(x_{t+1}, y_{t+1}) \in \{(x_t + 1, y_t + 1), (x_t + 1, y_t), (x_t, y_t + 1)\}$ for $t \in [0\,..\,m)$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m} : X \to Y$, for every $t \in [0\,..\,m)$:

- If $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t)$, we say that $\mathcal{A}$ *deletes* $X[x_t]$.
- If $(x_{t+1}, y_{t+1}) = (x_t, y_t + 1)$, we say that $\mathcal{A}$ *inserts* $Y[y_t]$.
- If $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t + 1)$, we say that $\mathcal{A}$ *aligns* $X[x_t]$ and $Y[y_t]$, denoted $X[x_t] \sim_{\mathcal{A}} Y[y_t]$. If additionally $X[x_t] = Y[y_t]$, we say that $\mathcal{A}$ *matches* $X[x_t]$ and $Y[y_t]$, denoted $X[x_t] \simeq_{\mathcal{A}} Y[y_t]$. Otherwise, we say that $\mathcal{A}$ *substitutes* $X[x_t]$ for $Y[y_t]$.

The *cost* of an edit distance alignment $\mathcal{A}$ is the total number characters that $\mathcal{A}$ deletes, inserts, or substitutes. We denote the cost by $\mathsf{ed}_{\mathcal{A}}(X, Y)$. The cost of an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ is at least its *width*, defined as $\max_{t=0}^{m} |x_t - y_t|$. Observe that $\mathsf{ed}(X, Y)$ can be defined as the minimum cost of an alignment of X and Y, that is, $\mathsf{ed}(X, Y) = \min_{\mathcal{A}:X\to Y} \mathsf{ed}_{\mathcal{A}}(X, Y)$. An alignment of X and Y is *optimal* if its cost is equal to $\mathsf{ed}(X, Y)$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ of $X, Y \in \Sigma^+$, we partition the elements (x_t, y_t) of $\mathcal{A}$ into *matches* (for which $X[x_t] \simeq_{\mathcal{A}} Y[y_t]$) and *breakpoints* (the remaining elements). We denote the set of matches and breakpoints by $\mathsf{M}_{\mathcal{A}}(X, Y)$ and $\mathsf{B}_{\mathcal{A}}(X, Y)$, respectively. Observe that $|\mathsf{B}_{\mathcal{A}}(X, Y)| = 1 + \mathsf{ed}_{\mathcal{A}}(X, Y)$.

We say that a set $M \subseteq \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$ is *non-crossing* if there are no distinct pairs $(x, y), (x', y') \in$

M with $x \leq x'$ and $y \geq y'$. Observe that, for every alignment $\mathcal{A}$ of X, Y, the set $\{(x, y) \in [0\,..\,|X|) \times [0\,..\,|Y|) : X[x] \sim_{\mathcal{A}} Y[y]\}$ is non-crossing. We further say that a non-crossing set $M \subseteq [0\,..\,|X|) \times [0\,..\,|Y|)$ is a *non-crossing matching* of strings X, Y if $X[x] = Y[y]$ holds for every $(x, y) \in M$. Note that, for every alignment $\mathcal{A}$ of X, Y, the set $\mathsf{M}_{\mathcal{A}}(X, Y)$ is a non-crossing matching.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ of X and Y, for every $\ell, r \in [0\,..\,m]$ with $\ell \leq r$, we say that $\mathcal{A}$ *aligns* $X[x_\ell\,..\,x_r)$ to $Y[y_\ell\,..\,y_r)$, denoted $X[x_\ell\,..\,x_r) \sim_{\mathcal{A}} Y[y_\ell\,..\,y_r)$. If there is no breakpoint (x_t, y_t) with $t \in [\ell\,..\,r)$, we further say that $\mathcal{A}$ *matches* $X[x_\ell\,..\,x_r)$ to $Y[y_\ell\,..\,y_r)$, denoted $X[x_\ell\,..\,x_r) \simeq_{\mathcal{A}} Y[y_\ell\,..\,y_r)$.

Definition 2.3.2 (Greedy alignment [Kociumaka et al., 2021])**.** We say that an alignment $\mathcal{A}$ of two strings $X, Y \in \Sigma^*$ is *greedy* if $X[x] \neq Y[y]$ holds for every $(x, y) \in \mathsf{B}_{\mathcal{A}}(X, Y)$ such that $x \neq |X|$ and $y \neq |Y|$.

As proven in [Kociumaka et al., 2021], any two strings $X, Y \in \Sigma^*$ have an optimal alignment that is also greedy.

For strings $X, Y \in \Sigma^*$ and integers $k \geq w \geq 0$, we denote by $\mathsf{A}_{k,w}(X, Y)$ the family of alignments of $\mathcal{A} : X \to Y$ whose cost is at most k and whose width is at most w. Furthermore, we define $\mathsf{GA}_{k,w}(X, Y)$ as the set consisting of all greedy alignments in $\mathsf{A}_{k,w}(X, Y)$. The following result can be obtained using a straightforward adaptation of the Landau–Vishkin algorithm [Landau and Vishkin, 1988]. The only difference compared to the baseline implementation is that the DP table is artificially restricted to diagonals $[-w\,..\,w]$.

Lemma 2.3.3. *Given strings $X, Y \in \Sigma^{\leq n}$ and integers $k, w \in \mathbb{Z}_{\geq 0}$, one can in $\mathcal{O}(n + kw)$ time decide whether $\mathsf{A}_{k,w}(X, Y) \neq \emptyset$. If the answer is positive, the algorithm reports a witness*

alignment $\mathcal{A} \in \mathsf{GA}_{k,w}(X, Y)$.

2.3.3 Trees and Forests

Definition 2.3.4 (Forest)**.** We say that a forest F is a (possibly empty) sequence of non-empty rooted ordered trees $\mathsf{T}_1, \ldots, \mathsf{T}_m$. Formally, each such tree T_i consists of a root node v_i and a forest F_i representing the descendants of v_i. We write V_{F} to denote the node set of F.

For a node $v \in V_{\mathsf{F}}$ in a forest F, we denote the parent of v by $\mathsf{parent}_{\mathsf{F}}(v)$. If v is a root, then we set $\mathsf{parent}_{\mathsf{F}}(v) = \perp_{\mathsf{F}}$; consistently with [Mao, 2021, Akmal and Jin, 2021], we refer to $\perp_{\mathsf{F}}$ as the *virtual root* of F. We denote $\bar{V}_{\mathsf{F}} = V_{\mathsf{F}} \cup \{\perp_{\mathsf{F}}\}$ as the set of nodes in F including the virtual root. For each node $v \in \bar{V}_{\mathsf{F}}$, we denote the sequence of children of v by $\mathsf{children}_{\mathsf{F}}(v)$. Observe that the forest F is uniquely specified by the lists $\mathsf{children}_{\mathsf{F}}(v)$ for $v \in \bar{V}_{\mathsf{F}}$ and these lists, in total, contain every node $u \in V_{\mathsf{F}}$ exactly once.

We define the *height* of a forest F, denoted $\mathsf{height}(\mathsf{F})$, as the maximum number of nodes on the root-to-leaf path in F. Thus, an empty forest has height 0, whereas a forest consisting of roots only has height 1.

Definition 2.3.5 (Labelled forest)**.** A *labeled forest* F consists of a forest F and a labeling function $\lambda : V_{\mathsf{F}} \to \Sigma$ assigning labels (from an integer *alphabet* $\Sigma = [0\,..\,\sigma)$) to nodes of F.

In this work, we typically consider two labeled forests F, G. We then assume that the underlying unlabeled forests F, G have disjoint node sets and a joint labeling $\lambda : V_{\mathsf{F}} \cup V_{\mathsf{G}} \to \Sigma$.

Definition 2.3.6 (Tree edit distance)**.** The following operations are jointly called *edits* of a labeled forest F:

Substitution (relabeling) Change the label of a node in F with another symbol from Σ.

Deletion Delete a node v from F. As a result, the children of v become the children of $\mathsf{parent}_F(v)$ while preserving their order. Formally, we modify $\mathsf{children}_F(\mathsf{parent}_F(v))$ by replacing v with $\mathsf{children}_F(v)$.

Insertion Insert a labeled node v into F so that deletion of v produces F again. Formally, an insertion is specified by a $u \in \bar{V}_{\mathsf{F}}$ (which will become the parent of v), a (possibly empty) fragment of $\mathsf{children}_F(u)$ (identifying nodes which will become the children of v), and the label of the new node.

The tree edit distance $\mathsf{ted}(F, G)$ is defined as the minimum number of edits (listed above) required to transform one forest to the other.

Just like the string edit distance, the tree edit distance admits an equivalent definition in terms of *alignments* $\mathcal{A} : F \to G$. Such an alignment must specify which nodes in F are *deleted* and which nodes in G are *inserted*. The remaining nodes are *aligned* (either *relabeled* or *matched*). Unlike for string edit distance, where the alignment must only be consistent with the left-to-right ordering, here, the alignment must the consistent with both the pre-order and the post-order forest traversal.

2.3.4 Parenthesis Representation of Forests

Aiming to adapt string techniques into the forest setting, we map each forest to a string through the *parentheses representation* (closely related to the *bi-order traversal* considered in [Mao, 2021]).

Definition 2.3.7 (Parenthesis representation)**.** For every unlabeled forest F, we define the *paren-*

theses representation of F, denoted $\mathsf{P}(\mathsf{F}) \in \{\,(\,,\,)\,\}^*$, using the following recursion: If F consists of trees $\mathsf{T}_1, \ldots, \mathsf{T}_m$, where each T_i consists of a root node v_i and a forest F_i representing the descendants of v_i, then $\mathsf{P}(\mathsf{F}) = \bigodot_{i=1}^{m} \mathsf{P}(\mathsf{T}_i)$, where $\mathsf{P}(\mathsf{T}_i) = (\cdot \mathsf{P}(\mathsf{F}_i) \cdot)$ and $\cdot$ as well as $\bigodot$ denote concatenation.

For a labelling $\lambda : V_{\mathsf{F}} \to \Sigma$, we further set $\mathsf{P}_\lambda(\mathsf{F}) \in (\{\,(\,,\,)\,\} \times \Sigma)^*$ so that $\mathsf{P}_\lambda(\mathsf{F}) = \bigodot_{i=1}^{m} \mathsf{P}_\lambda(\mathsf{T}_i)$ and $\mathsf{P}_\lambda(\mathsf{T}_i) = {}_{\lambda(v_i)}(\cdot \mathsf{P}_\lambda(\mathsf{F}_i) \cdot)_{\lambda(v_i)}$. For a labeled forest $F = (\mathsf{F}, \lambda)$, we also write $\mathsf{P}(F) := \mathsf{P}_\lambda(\mathsf{F})$.

Observe that $\mathsf{P}(\mathsf{F})$ forms a well-parenthesized sequence and there is a bijection between nodes of F and the pairs of matching parentheses in $\mathsf{P}(\mathsf{F})$. We define functions $o_{\mathsf{F}}, c_{\mathsf{F}} : V_{\mathsf{F}} \to [0\,..\,2|\mathsf{F}|)$ so that the opening and the closing parenthesis representing u are located at positions $o_{\mathsf{F}}(u)$ and $c_{\mathsf{F}}(u)$, respectively.

Definition 2.3.8. We say that $\mathcal{A} : \mathsf{P}(\mathsf{F}) \to \mathsf{P}(\mathsf{G})$ is a *tree alignment* of forests F and G, denoted $\mathcal{A} : \mathsf{F} \to \mathsf{G}$ if the following *consistency* conditions are satisfied for each $u \in V_{\mathsf{F}}$:

- either $\mathcal{A}$ deletes both $\mathsf{P}(\mathsf{F})[o_{\mathsf{F}}(u)]$ and $\mathsf{P}(\mathsf{F})[c_{\mathsf{F}}(u)]$, or
- there exists $v \in V_{\mathsf{G}}$ such that $\mathsf{P}(\mathsf{F})[o_{\mathsf{F}}(u)] \sim_{\mathcal{A}} \mathsf{P}(\mathsf{G})[o_{\mathsf{G}}(v)]$ and $\mathsf{P}(\mathsf{F})[c_{\mathsf{F}}(u)] \sim_{\mathcal{A}} \mathsf{P}(\mathsf{G})[c_{\mathsf{G}}(v)]$.

We denote the set of all tree alignments $\mathcal{A} : \mathsf{F} \to \mathsf{G}$ with $\mathsf{TA}(\mathsf{F}, \mathsf{G}) \subseteq \mathsf{A}(\mathsf{P}(\mathsf{F}), \mathsf{P}(\mathsf{G}))$.

Definition 2.3.9. Consider a joint labeling λ of forests F, G and an alignment $A \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$. We denote $\mathsf{ted}_{\lambda,\mathcal{A}}(\mathsf{F}, \mathsf{G}) := \frac{1}{2}\mathsf{ed}_{\mathcal{A}}(\mathsf{P}_\lambda(\mathsf{F}), \mathsf{P}_\lambda(\mathsf{G}))$. In terms of the underlying labeled forests F, G, we express this as $\mathsf{ted}_{\mathcal{A}}(F, G) := \frac{1}{2}\mathsf{ed}_{\mathcal{A}}(\mathsf{P}(F), \mathsf{P}(G))$.

We observe that Definitions 2.3.6 and 2.3.9 are equivalent as each tree edit operation can be represented using two string edit operations. For example, a deletion of a node $v \in V_F$ corresponds to deleting the characters at positions $o_F(u)$ and $c_F(u)$ in $P(F)$.

Observation 2.3.10. *For any two forests F, G, we have* $\mathsf{ted}(F,G) = \min_{\mathcal{A}\in\mathsf{TA}(F,G)} \mathsf{ted}_{\mathcal{A}}(F,G)$.

Furthermore, for an integer $k \in \mathbb{Z}_{\geq 0}$, we denote $\mathsf{TA}_k(F,G) = \{\mathcal{A} \in \mathsf{TA}(F,G) : \mathsf{ted}_{\mathcal{A}}(F,G) \leq k\}$. We also denote

$$\mathsf{ted}_{\leq k}(F,G) = \begin{cases} \mathsf{ted}(F,G) & \text{if } \mathsf{ted}(F,G) \leq k, \\ \infty & \text{otherwise.} \end{cases}$$

2.4 Labeling Refinements

Consider two joint labelings $\lambda, \lambda' : V_{\mathsf{F}} \cup V_{\mathsf{G}} \to \Sigma$ of forests F, G. We say that λ is a *refinement* of λ' if $\lambda'(u) = \lambda'(v)$ holds for any two nodes $u, v \in V_{\mathsf{F}} \cup V_{\mathsf{G}}$ such that $\lambda(u) = \lambda(v)$. If simultaneously λ is a refinement of λ' and λ' is a refinement of λ, we say that λ is equivalent to λ'.

2.4.1 Look-Ahead Refinement

For a node v of an unlabeled forest F, let $\mathsf{sub}(v)$ denote the subtree of F rooted at v. Further, for an integer $d \in \mathbb{Z}_+$, let $\mathsf{sub}_{<d}(v)$ denote the subtree of $\mathsf{sub}(v)$ consisting of nodes at distance less than d from the root v.

Definition 2.4.1 (Look-ahead refinement)**.** Let λ be a joint labeling of forests F, G. We say that λ' is a *depth-d look-ahead refinement* of λ if, for any $u, v \in V_{\mathsf{F}} \cup V_{\mathsf{G}}$, we have $\lambda'(u) = \lambda'(v)$ if and only if $\mathsf{P}_{\lambda}(\mathsf{sub}_{<d}(u)) = \mathsf{P}_{\lambda}(\mathsf{sub}_{<d}(v))$. We use $\mathsf{L}(\lambda, d)$ to denote a depth-d look-ahead refinement of λ (chosen arbitrarily up to equivalence of labelings).

Lemma 2.4.2. *Consider forests F, G, their joint labeling λ, an alignment $\mathcal{A} \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$, and an integer $d \in \mathbb{Z}_+$. Then,*

$$\mathsf{ted}_{\mathsf{L}(\lambda,d),\mathcal{A}}(\mathsf{F}, \mathsf{G}) \leq d \cdot \mathsf{ted}_{\lambda,\mathcal{A}}(\mathsf{F}, \mathsf{G}).$$

Proof. Let the *level* of a node in $V_{\mathsf{F}} \cup V_{\mathsf{G}}$ denote its distance to the root. We *mark* some nodes of F, G and prove that $\mathsf{ted}_{\mathsf{L}(\lambda,d),\mathcal{A}}(\mathsf{F}, \mathsf{G})$ is bounded by the number of marks.

1. If $\mathcal{A}$ deletes a node $u \in V_{\mathsf{F}}$ at level ℓ or aligns such a node to a node $v \in V_{\mathsf{G}}$ with $\lambda(u) \neq \lambda(v)$, then we mark all ancestors of u at levels in $(\ell - d \,..\, \ell]$.
2. If $\mathcal{A}$ inserts a node $v \in V_{\mathsf{G}}$ at level ℓ, then we mark all ancestors of v at levels in $(\ell - d \,..\, \ell]$.

Clearly, the total number of marks does not exceed $d \cdot \mathsf{ted}_{\lambda,\mathcal{A}}(\mathsf{F}, \mathsf{G})$.

It remains to prove that each unit of $\mathsf{ted}_{\mathsf{L}(\lambda,d),\mathcal{A}}(\mathsf{F}, \mathsf{G})$ can be charged to a marked node. If $\mathcal{A}$ deletes a node in $V_{\mathsf{F}} \cup V_{\mathsf{G}}$, then this node is already marked. We shall prove that if $\mathcal{A}$ aligns a node $u \in V_{\mathsf{F}}$ with a node $v \in V_{\mathsf{G}}$ such that $\mathsf{L}(\lambda, d)(u) \neq \mathsf{L}(\lambda, d)(v)$, then either u or v is marked. For a proof by contradiction, suppose that this is not the case. In particular, this means that $\mathcal{A}$ does not delete nor insert any node in $\mathsf{sub}_{<d}(u)$ and $\mathsf{sub}_{<d}(v)$. Consequently, these trees are isomorphic, i.e., $\mathsf{P}(\mathsf{sub}_{<d}(u)) = \mathsf{P}(\mathsf{sub}_{<d}(v))$, and $\mathcal{A}$ aligns nodes according to this isomorphism. Moreover, λ assigns the same label to any two nodes matched by this isomporhism (otherwise, u would have been marked), and thus $\mathsf{P}_\lambda(\mathsf{sub}_{<d}(u)) = \mathsf{P}_\lambda(\mathsf{sub}_{<d}(v))$, contradicting the assumption that $\mathsf{L}(\lambda, d)(u) \neq \mathsf{L}(\lambda, d)(v)$. □

Lemma 2.4.3. *Given forests* F, G *of total size* n*, their joint labeling* λ*, and an integer* $d \in \mathbb{Z}_{\geq 0}$*, a labeling equivalent to* $\mathsf{L}(\lambda, d)$ *can be constructed in* $\mathcal{O}(n)$ *using a randomized algorithm correct with high probability.*

Proof. We replace the label of each node $v \in V_{\mathsf{F}} \cup V_{\mathsf{G}}$ with the Karp–Rabin fingerprint [Karp and Rabin, 1987] of $\mathsf{P}_\lambda(\mathsf{sub}_{<d}(v))$. Choosing a sufficiently large prime number $p = n^{\Theta(1)}$ and a uniformly random seed $r \in [0 \,..\, p)$, we can guarantee that, with high probability, the fingerprints will have no collisions among the n strings $\mathsf{P}_\lambda(\mathsf{sub}_{<d}(v))$.

As far as the implementation is concerned, for each node v at level ℓ, we first identify all its descendants $v_1, \ldots, v_m$ at level $\ell + d$. Such lists can be produced in linear time by a traversal of F and G that maintains the ancestors of the currently visited node in an array indexed by the level of the ancestor. When visiting a node at level $\ell \geq d$, we add it to the list of descendants of the ancestor at level $\ell - d$. Next, we note that $\mathsf{P}_\lambda(\mathsf{sub}_{<d}(v))$ can be obtained from $\mathsf{P}_\lambda(\mathsf{sub}(v))$ by cutting out the fragments representing $\mathsf{P}_\lambda(\mathsf{sub}(v_i))$. Thus, given the list $v_1, \ldots, v_m$, we can represent $\mathsf{P}_\lambda(\mathsf{sub}_{<d}(v))$ as the concatenation of $m + 1$ fragments of $\mathsf{P}_\lambda(\mathsf{sub}(v))$. The fingerprint of each fragment of $\mathsf{P}_\lambda(F)$ and $\mathsf{P}_\lambda(G)$ can be constructed in $\mathcal{O}(1)$ time and the Karp–Rabin fingerprints support concatenations in $\mathcal{O}(1)$ time. Consequently, the desired fingerprints can be constructed in $\mathcal{O}(n)$ time in total. □

Next, we show that there exists an optimum alignment that becomes greedy if we refine the labeling using look-ahead $d \geq \max\{\mathsf{height}(F), \mathsf{height}(G)\}$.

Proposition 2.4.4. *Let λ be a joint labeling of unlabeled forests F, G of height at most h and let $\hat{\lambda} = \mathsf{L}(\lambda, h)$. Then, there exists a tree alignment $\mathcal{A} \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$ of optimum cost $\mathsf{ted}_{\lambda,\mathcal{A}}(\mathsf{F}, \mathsf{G}) = \mathsf{ted}_\lambda(\mathsf{F}, \mathsf{G})$ such that $\mathcal{A} \in \mathsf{GA}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$.*

Proof. Consider a tree alignment $\mathcal{A} \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$ of optimum cost $\mathsf{ted}_{\lambda,\mathcal{A}}(\mathsf{F}, \mathsf{G}) = \mathsf{ted}_\lambda(\mathsf{F}, \mathsf{G})$. We perform the following steps repeatedly for $2|\mathsf{F}|$ rounds: at round i, we construct a tree alignment $\mathcal{A}_i \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$ such that $\mathsf{ted}_{\lambda,\mathcal{A}_i}(F, G) = \mathsf{ted}_{\lambda,\mathcal{A}}(F, G)$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] \neq \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$ for every $(x, y) \in \mathsf{B}_{\mathcal{A}_i}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G})) \cap ([0\,..\,i) \times [0\,..\,2|\mathsf{G}|))$. Trivially, $\mathcal{A}_0 := \mathcal{A}$ satisfies this condition for $i = 0$.

Round i. At round i, we construct $\mathcal{A}_i$ given $\mathcal{A}_{i-1} = (x_t, y_t)_{t \in [0\,..\,m]}$. If $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] \neq \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$ already holds for every $(x, y) \in \mathsf{B}_{\mathcal{A}_{i-1}}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G})) \cap ([0\,..\,i) \times [0\,..\,2|\mathsf{G}|))$, then we simply set $\mathcal{A}_i := \mathcal{A}_{i-1}$. Otherwise, let (x, y) be the leftmost breakpoint with $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$. By the

assumptions on $\mathcal{A}_{i-1}$, we must have $x = i - 1$.

Case 1. First, we consider the case where $(x, y) = (o_{\mathsf{F}}(u), o_{\mathsf{G}}(v))$ for some $(u, v) \in V_{\mathsf{F}} \times V_{\mathsf{G}}$. Let $(\bar{x}, \bar{y}) = (c_{\mathsf{F}}(u), c_{\mathsf{G}}(v))$ and note that, due to $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$ and $\hat{\lambda} = \mathsf{L}(\lambda, h)$, we have $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x \,..\, \bar{x}] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y \,..\, \bar{y}]$. We define $q := \min\{t : x_t > \bar{x} \text{ and } y_t > \bar{y}\}$ and create $\mathcal{A}_i$ so that it:

- aligns $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[0 \,..\, x)$ against $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[0 \,..\, y)$ in the same way as $\mathcal{A}_{i-1}$ did,
- matches $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x \,..\, \bar{x}]$ against $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y \,..\, \bar{y}]$,
- deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})(\bar{x} \,..\, x_q)$ and inserts $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})(\bar{y} \,..\, y_q)$ (at least one of these fragments is empty),
- aligns $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x_q \,..\, 2|\mathsf{F}|)$ against $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y_q \,..\, 2|\mathsf{G}|)$ in the same way as $\mathcal{A}_{i-1}$ did.

Formally, $\mathcal{A}_i$ is defined as follows, where $\odot$ denotes concatenation of sequences and $p \in [0 \,..\, m]$ is such that $(x, y) = (x_p, y_p)$:

$$\mathcal{A}_i = (x_t, y_t)_{t \in [0 \,..\, p)} \odot (x + \delta, y + \delta)_{\delta \in [0 \,..\, \bar{x} - x]} \odot (\hat{x}, y_q)_{\hat{x} \in (\bar{x} \,..\, x_q)} \odot (x_q, \hat{y})_{\hat{y} \in (\bar{y} \,..\, y_q)} \odot (x_t, y_t)_{t \in [q \,..\, m]}.$$

First, we show $\mathcal{A}_i \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$. For this, take a node $u' \in V_{\mathsf{F}}$ and consider several cases:

- $\boldsymbol{o_{\mathsf{F}}(u') \in [x \,..\, \bar{x}]}$ or $\boldsymbol{c_{\mathsf{F}}(u') \in [x \,..\, \bar{x}]}$. In this case, $u' \in \mathsf{sub}(u)$ and thus both $o_{\mathsf{F}}(u') \in [x \,..\, \bar{x}]$ and $c_{\mathsf{F}}(u') \in [x \,..\, \bar{x}]$. Moreover, due to $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x \,..\, \bar{x}] \simeq_{\mathcal{A}_i} \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y \,..\, \bar{y}]$, in this case, $\mathcal{A}_i$ matches u' with the corresponding node of the subtree $\mathsf{sub}(v)$ identical to $\mathsf{sub}(u)$.
- $\boldsymbol{o_{\mathsf{F}}(u'), c_{\mathsf{F}}(u') \in (\bar{x} \,..\, x_q)}$. In this case, $\mathcal{A}_i$ deletes both $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$.
- $\boldsymbol{o_{\mathsf{F}}(u') \in (\bar{x} \,..\, x_q)}$ and $\boldsymbol{c_{\mathsf{F}}(u') \in [x_q \,..\, 2|\mathsf{F}|)}$. In this case, $\mathcal{A}_i$ deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ and handles $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$ in the same way as $\mathcal{A}_{i-1}$ did. Thus, we must prove that $\mathcal{A}_{i-1}$ deleted u'. For a proof by contradiction, suppose that $\mathcal{A}_{i-1}$ aligned u' with some node $v' \in V_{\mathsf{G}}$. By the non-crossing property of $\mathcal{A}_{i-1}$, due to $(x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, we must have

$o_{\mathsf{G}}(v') \in [y\,..\,y_q)$ and $c_{\mathsf{G}}(v') \in [y_q\,..\,2|\mathsf{G}|)$. Moreover, since $(\bar{x}\,..\,x_q) \neq \emptyset$, we must have $(\bar{y}\,..\,y_q) = \emptyset$, which means that $o_{\mathsf{G}}(v') \in [y\,..\,\bar{y}] = [y\,..\,y_q)$. Consequently, $v' \in \mathsf{sub}(v)$ and hence $c_{\mathsf{G}}(v') \in [y\,..\,\bar{y}] = [y\,..\,y_q)$; this contradicts $c_{\mathsf{G}}(v') \in [y_q\,..\,2|\mathsf{G}|)$.

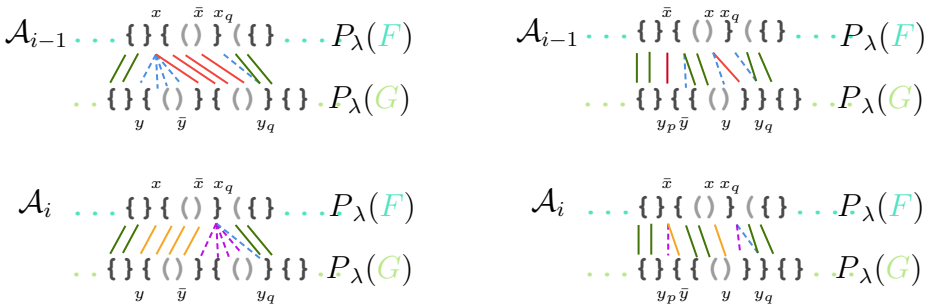
- $\boldsymbol{o_{\mathsf{F}}(u') \in [0\,..\,x)}$ and $\boldsymbol{c_{\mathsf{F}}(u') \in (\bar{x}\,..\,x_q)}$. In this case, $\mathcal{A}_i$ deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$ and handles $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ in the same way as $\mathcal{A}_{i-1}$ did. Thus, we must prove that $\mathcal{A}_{i-1}$ deleted u'. For a proof by contradiction, suppose that $\mathcal{A}_{i-1}$ aligned u' with some node $v' \in V_{\mathsf{G}}$. By the non-crossing property of $\mathcal{A}_{i-1}$, due to $(x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, we must have $o_{\mathsf{G}}(v') \in [0\,..\,y)$ and $c_{\mathsf{G}}(v') \in [y\,..\,y_q)$. Moreover, since $(\bar{x}\,..\,x_q) \neq \emptyset$, we must have $(\bar{y}\,..\,y_q) = \emptyset$, which means that $c_{\mathsf{G}}(v') \in [y\,..\,\bar{y}] = [y\,..\,y_q)$. Consequently, $v' \in \mathsf{sub}(v)$ and hence $o_{\mathsf{G}}(v') \in [y\,..\,\bar{y}] = [y\,..\,y_q)$; this contradicts $o_{\mathsf{G}}(v') \in [0\,..\,y)$.
- $\boldsymbol{o_{\mathsf{F}}(u'), c_{\mathsf{F}}(u') \in [0\,..\,x) \cup [x_q\,..\,2|\mathsf{F}|)}$. In this case, $\mathcal{A}_i$ handles u' in the same way as $\mathcal{A}_{i-1}$ did.

This case analysis above is exhaustive and, in all cases, the two parentheses corresponding to u' are handled consistently. Consequently, we indeed have $\mathcal{A}_i \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$.

Next we argue that $\mathsf{ted}_{\lambda,\mathcal{A}_i}(\mathsf{F}, \mathsf{G}) \leq \mathsf{ted}_{\lambda,\mathcal{A}_{i-1}}(\mathsf{F}, \mathsf{G})$ or, equivalently, $\mathsf{ed}_{\mathcal{A}_i}(\mathsf{P}_\lambda(\mathsf{F}), \mathsf{P}_\lambda(G)) \leq \mathsf{ed}_{\mathcal{A}_{i-1}}(\mathsf{P}_\lambda(\mathsf{F}), \mathsf{P}_\lambda(G))$. The two alignments differ only on $\mathsf{P}_\lambda(\mathsf{F})[x\,..\,x_q)$ and $\mathsf{P}_\lambda(\mathsf{G})[y\,..\,y_q)$. The contribution of these fragments to the cost of $\mathcal{A}_i : \mathsf{P}_\lambda(\mathsf{F}) \rightsquigarrow \mathsf{P}_\lambda(\mathsf{G})$ equals $|(\bar{x}\,..\,x_q)| + |(\bar{y}\,..\,y_q)| = \big||(\bar{x}\,..\,x_q)| - |(\bar{y}\,..\,y_q)|\big| = \big||[x\,..\,x_q) - |[y\,..\,y_q)|\big|$. Due to $(x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, the contribution of these fragments to the cost of $\mathcal{A}_{i-1} : \mathsf{P}_\lambda(\mathsf{F}) \rightsquigarrow \mathsf{P}_\lambda(\mathsf{G})$ must be at least that large.

Finally, we note that the breakpoints of $\mathcal{A}_i$ in $[0\,..\,i) \times [0\,..\,2|\mathsf{G}|)$ satisfy the greedy property. For breakpoints to the left of (x, y), this follows by definition of (x, y). Moreover, due to $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] \simeq \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$, the pair (x, y) is not a breakpoint in $\mathcal{A}_i$; in particular, the subsequent pair

$(x+1, y+1) \in \mathcal{A}_i$ satisfies $x+1 = i$.



(a) Case 1

(b) Case 2

Figure 2.2: Example of the construction of alignment $\mathcal{A}_i$ from alignment $\mathcal{A}_{i-1}$

Case 2. Next, we consider the case where $(x, y) = (c_{\mathsf{F}}(u), c_{\mathsf{G}}(v))$ for some $(u, v) \in V_{\mathsf{F}} \times V_{\mathsf{G}}$. Define $(\bar{x}, \bar{y}) = (o_{\mathsf{F}}(u), o_{\mathsf{G}}(v))$ and note that, due to $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$ and $\hat{\lambda} = \mathsf{L}(\lambda, h)$, we have $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[\bar{x}\,..\,x] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\bar{y}\,..\,y]$. Define $q = \min\{t : x_t > x \text{ and } y_t > y\}$ and $p = \max\{t : x_t \leq \bar{x} \text{ and } y_t \leq \bar{y}\}$. We create $\mathcal{A}_i$ so that it:

- aligns $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[0\,..\,x_p)$ against $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[0\,..\,y_p)$ in the same way as $\mathcal{A}_{i-1}$ did,
- deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x_p\,..\,\bar{x})$ and inserts $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y_p\,..\,\bar{y})$ (at least one of these fragments is empty),
- matches $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[\bar{x}\,..\,x]$ against $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\bar{y}\,..\,y]$,
- deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})(x\,..\,x_q)$ and inserts $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})(y\,..\,y_q)$ (at least one of these fragments is empty),
- aligns $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x_q\,..\,2|\mathsf{F}|)$ against $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y_q\,..\,2|\mathsf{G}|)$ in the same way as $\mathcal{A}_{i-1}$ did.

Formally, $\mathcal{A}_i$ is defined as follows:

$$\mathcal{A}_i = (x_t, y_t)_{t\in[0..p)} \odot (\hat{x}, \bar{y})_{\hat{x}\in[x_p..\bar{x})} \odot (\bar{x}, \hat{y})_{\hat{y}\in[y_p..\bar{y})} \odot (\bar{x}+\delta, \bar{y}+\delta)_{\delta\in[0..x-\bar{x}]}$$
$$\odot (\hat{x}, y_q)_{\hat{x}\in(x..x_q)} \odot (x_q, \hat{y})_{\hat{y}\in(y..y_q)} \odot (x_t, y_t)_{t\in[q..m]}.$$

First, we show $\mathcal{A}_i \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$. For this, take a node $u' \in V_{\mathsf{F}}$ and consider several cases:

- $\boldsymbol{o_{\mathsf{F}}(u') \in [\bar{x} \,..\, x]}$ or $\boldsymbol{c_{\mathsf{F}}(u') \in [\bar{x} \,..\, x]}$. In this case, $u' \in \mathsf{sub}(u)$ and thus both $o_{\mathsf{F}}(u') \in [\bar{x} \,..\, x]$ and $c_{\mathsf{F}}(u') \in [\bar{x} \,..\, x]$. Moreover, due to $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[\bar{x} \,..\, x] \simeq_{\mathcal{A}_i} \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\bar{y} \,..\, y]$, in this case, $\mathcal{A}_i$ matches u' with the corresponding node of the subtree $\mathsf{sub}(v)$ identical to $\mathsf{sub}(u)$.
- $\boldsymbol{o_{\mathsf{F}}(u'), c_{\mathsf{F}}(u') \in [x_p \,..\, \bar{x}) \cup (x \,..\, x_q)}$. In this case, $\mathcal{A}_i$ deletes both $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$.
- $\boldsymbol{o_{\mathsf{F}}(u') \in [x_p \,..\, \bar{x})}$ and $\boldsymbol{c_{\mathsf{F}}(u') \in [x_q \,..\, 2|\mathsf{F}|)}$. In this case, $\mathcal{A}_i$ deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ and handles $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$ in the same way as $\mathcal{A}_{i-1}$ did. Thus, we must prove that $\mathcal{A}_{i-1}$ deleted u'. For a proof by contradiction, suppose that $\mathcal{A}_{i-1}$ aligned u' with some node $v' \in V_{\mathsf{G}}$. By the non-crossing property of $\mathcal{A}_{i-1}$, due to $(x_p, y_p), (x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, we must have $o_{\mathsf{G}}(v') \in [y_p \,..\, y)$ and $c_{\mathsf{G}}(v') \in [y_q \,..\, 2|\mathsf{G}|)$. Moreover, since $[x_p \,..\, \bar{x}) \neq \emptyset$, we must have $[y_p \,..\, \bar{y}) = \emptyset$, which means that $o_{\mathsf{G}}(v') \in [\bar{y} \,..\, y) = [y_p \,..\, y)$. Consequently, $v' \in \mathsf{sub}(v)$ and hence $c_{\mathsf{G}}(v') \in [s \,..\, y] = [y_p \,..\, y]$; this contradicts $c_{\mathsf{G}}(v') \in [y_q \,..\, 2|\mathsf{G}|)$ due to $y_q > y$.
- $\boldsymbol{o_{\mathsf{F}}(u') \in (x \,..\, x_q)}$ and $\boldsymbol{c_{\mathsf{F}}(u') \in [x_q \,..\, 2|\mathsf{F}|)}$. In this case, $\mathcal{A}_i$ deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ and handles $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$ in the same way as $\mathcal{A}_{i-1}$ did. Thus, we must prove that $\mathcal{A}_{i-1}$ deleted u'. For a proof by contradiction, suppose that $\mathcal{A}_{i-1}$ aligned u' with some node $v' \in V_{\mathsf{G}}$. By the non-crossing property of $\mathcal{A}_{i-1}$, due to $(x_p, y_p), (x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, we must have $o_{\mathsf{G}}(v') \in [y \,..\, y_q)$ and $c_{\mathsf{G}}(v') \in [y_q \,..\, 2|\mathsf{G}|)$. Moreover, since $(x \,..\, x_q) \neq \emptyset$, we must have $(y \,..\, y_q) = \emptyset$, which means that $o_{\mathsf{G}}(v') = y$; this is a contradiction because $y = c_{\mathsf{G}}(v)$.
- $\boldsymbol{o_{\mathsf{F}}(u') \in [0 \,..\, x_p)}$ and $\boldsymbol{c_{\mathsf{F}}(u') \in [x_p \,..\, \bar{x})}$. In this case, $\mathcal{A}_i$ deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$ and handles $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ in the same way as $\mathcal{A}_{i-1}$ did. Thus, we must prove that $\mathcal{A}_{i-1}$ deleted u'. For a proof by contradiction, suppose that $\mathcal{A}_{i-1}$ aligned u' with some node $v' \in V_{\mathsf{G}}$. By the non-crossing property of $\mathcal{A}_{i-1}$, due to $(x_p, y_p), (x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, we must have $o_{\mathsf{G}}(v') \in [0 \,..\, y_p)$ and $c_{\mathsf{G}}(v') \in [y_p \,..\, y)$. Moreover, since $[x_p \,..\, \bar{x}) \neq \emptyset$, we must have

$[y_q \,..\, \bar{y}) = \emptyset$, which means that $o_{\mathsf{G}}(v') \in [\bar{y} \,..\, y) = [y_p \,..\, y)$. Consequently, $v' \in \mathsf{sub}(v)$ and hence $o_{\mathsf{G}}(v') \in [\bar{y} \,..\, y] = [y_p \,..\, y]$; this contradicts $o_{\mathsf{G}}(v') \in [0 \,..\, y_p)$.

- $\boldsymbol{o_{\mathsf{F}}(u') \in [0 \,..\, x_p)}$ and $\boldsymbol{c_{\mathsf{F}}(u') \in (x \,..\, x_q)}$. In this case, $\mathcal{A}_i$ deletes $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[c_{\mathsf{F}}(u')]$ and handles $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[o_{\mathsf{F}}(u')]$ in the same way as $\mathcal{A}_{i-1}$ did. Thus, we must prove that $\mathcal{A}_{i-1}$ deleted u'. For a proof by contradiction, suppose that $\mathcal{A}_{i-1}$ aligned u' with some node $v' \in V_{\mathsf{G}}$. By the non-crossing property of $\mathcal{A}_{i-1}$, due to $(x_p, y_p), (x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, we must have $o_{\mathsf{G}}(v') \in [0 \,..\, y_p)$ and $c_{\mathsf{G}}(v') \in [y \,..\, y_q)$. Moreover, since $(x \,..\, x_q) \neq \emptyset$, we must have $(y \,..\, y_q) = \emptyset$, which means that $c_{\mathsf{G}}(v') = y$. Consequently, $v = v'$ and hence $o_{\mathsf{G}}(v') = \bar{y}$; this contradicts $o_{\mathsf{G}}(v') \in [0 \,..\, y_p)$ due to $y_p \leq \bar{y}$.
- $\boldsymbol{o_{\mathsf{F}}(u'), c_{\mathsf{F}}(u') \in [0 \,..\, x_p) \cup [x_q \,..\, 2|\mathsf{F}|)}$. In this case, $\mathcal{A}_i$ handles u' in the same way as $\mathcal{A}_{i-1}$ did.

This case analysis above is exhaustive and, in all cases, the two parentheses corresponding to u' are handled consistently. Consequently, we indeed have $\mathcal{A}_i \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$.

Next we argue that $\mathsf{ted}_{\lambda,\mathcal{A}_i}(\mathsf{F}, \mathsf{G}) \leq \mathsf{ted}_{\lambda,\mathcal{A}_{i-1}}(\mathsf{F}, \mathsf{G})$ or, equivalently, $\mathsf{ed}_{\mathcal{A}_i}(\mathsf{P}_\lambda(\mathsf{F}), \mathsf{P}_\lambda(G)) \leq \mathsf{ed}_{\mathcal{A}_{i-1}}(\mathsf{P}_\lambda(\mathsf{F}), \mathsf{P}_\lambda(G))$. The two alignments differ only in how on $\mathsf{P}_\lambda(\mathsf{F})[x_p \,..\, x_q)$ and $\mathsf{P}_\lambda(\mathsf{G})[y_p \,..\, y_q)$. The contribution of these fragments to the cost of $\mathcal{A}_i : \mathsf{P}_\lambda(\mathsf{F}) \rightsquigarrow \mathsf{P}_\lambda(\mathsf{G})$ equals $|[x_p \,..\, \bar{x})| + |[y_p \,..\, \bar{y})| + |(x \,..\, x_q)| + |(y \,..\, y_q)| = \big||[x_p \,..\, \bar{x})| - |[y_p \,..\, \bar{y})|\big| + \big||(x \,..\, x_q)| - |(y \,..\, y_q)|\big| = \big||[x_p \,..\, x) - |[y_p \,..\, y)|\big| + \big||[x \,..\, x_q) - |[y \,..\, y_q)|\big|$. Due to $(x_p, y_p)(x, y), (x_q, y_q) \in \mathcal{A}_{i-1}$, the contribution of these fragments to the cost of $\mathcal{A}_{i-1} : \mathsf{P}_\lambda(\mathsf{F}) \rightsquigarrow \mathsf{P}_\lambda(\mathsf{G})$ must be at least that large.

Finally, we shall prove that $\mathcal{A}_i$ is greedy up to index $i - 1$. For breakpoints to the left of (x_p, y_p), this follows by definition of (x, y). Moreover, none of the pairs $(\bar{x} + \delta, \bar{y} + \delta)_{\delta \in [0 \,..\, x - \bar{x}]}$ is a breakpoint and, in particular, the subsequent pair $(x + 1, y + 1) \in \mathcal{A}_i$ satisfies $x + 1 = i$.

Hence, it suffices to consider breakpoints of the form $(\hat{x}, \bar{y})$ for some $\hat{x} \in [x_p \,..\, \bar{x})$ or $(\bar{x}, \hat{y})$ for some $\hat{y} \in [y_p \,..\, \bar{y})$. These two cases are symmetric, so let us consider $(\bar{x}, \hat{y})$ for some $\hat{y} \in [y_p \,..\, \bar{y})$. For a proof by contradiction, suppose that $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[\bar{x}] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\hat{y}]$; consequently, $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[\bar{x} \,..\, x] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\hat{y} \,..\, \hat{y} + x - \bar{x}]$ because $\hat{\lambda} = \mathsf{L}(\lambda, h)$. Moreover, due to $[y_p \,..\, \bar{y}) \neq \emptyset$, so we must have $[x_p \,..\, \bar{x}) = \emptyset$, i.e., $x_p = \bar{x}$. Define $r = \min\{t \in [p \,..\, q) : y_t - x_t \geq \hat{y} - \bar{x}\}$. Note that $y_p - x_p \leq \hat{y} - \bar{x}$ and $y - x = \bar{y} - \bar{x} > \hat{y} - \bar{x}$; hence, $x_r \in [x_p \,..\, x] = [\bar{x} \,..\, x]$ and $y_r - x_r = \hat{y} - \bar{x}$. Since all breakpoints to the left of (x, y) satisfy the greedy property and since $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[\bar{x} \,..\, x] = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\hat{y} \,..\, \hat{y} + x - \bar{x}]$, we conclude that $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x_r \,..\, x] \simeq_{\mathcal{A}_{i-1}} \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y_r \,..\, \hat{y} + x - \bar{x}]$. In particular, $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] \simeq_{\mathcal{A}_{i-1}} \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[\hat{y} + x - \bar{x}]$, which contradicts $(x, y) \in \mathcal{A}_{i-1}$ by the non-crossing property of $\mathcal{A}_{i-1}$ because $\hat{y} + x - \bar{x} < \bar{y} + x - \bar{x} = y$.

After $2|\mathsf{F}|$ rounds, we construct a tree alignment $\mathcal{A}_{2|\mathsf{F}|} \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$ such that $\mathsf{ted}_{\lambda, \mathcal{A}_{2|\mathsf{F}|}}(F, G) = \mathsf{ted}_{\lambda, \mathcal{A}}(F, G)$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x] \neq \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y]$ holds for every $(x, y) \in \mathsf{B}_{\mathcal{A}_{2|\mathsf{F}|}}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G})) \cap ([0 \,..\, 2|\mathsf{F}| - 1] \times [0 \,..\, 2|\mathsf{G}|))$. Thus, $\mathcal{A}_{2|\mathsf{F}|} \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$ is an optimum tree alignment and $\mathcal{A}_{2|\mathsf{F}|} \in \mathsf{GA}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$.

□

2.4.2 Compatibility Refinement

Definition 2.4.5 (Compatibility)**.** Let λ be a joint labeling of forests F, G, and let $w \in \mathbb{Z}_{\geq 0}$. We say that nodes $u \in V_{\mathsf{F}}$ and $v \in V_{\mathsf{G}}$ are *w-compatible* if $\lambda(u) = \lambda(v)$, $|o_{\mathsf{F}}(u) - o_{\mathsf{G}}(v)| \leq w$, and $|c_{\mathsf{F}}(u) - c_{\mathsf{G}}(v)| \leq w$.

Definition 2.4.6 (Compatibilty refinement)**.** Let λ be a joint labeling of forests F, G, and let $w \in \mathbb{Z}_{\geq 0}$. We say that λ' is a *w-compatibility refinement* of λ if, for any $u, v \in V_{\mathsf{F}} \cup V_{\mathsf{G}}$, we have $\lambda'(u) = \lambda'(v)$ if and only if (u, v) belongs to the transitive closure of the w-compatibility relation.

We use $\mathsf{C}(\lambda, w)$ to denote a *w-compatibility refinement* of λ (chosen arbitrarily up to equivalence of labelings).

Lemma 2.4.7. *Given forests F, G of total size n, their joint labeling λ, and an integer w, a w-compatibility refinement labeling $\lambda' = \mathsf{C}(\lambda, w)$ can be constructed in $\mathcal{O}(n \log n)$ time.*

Proof. We process nodes according to the original label. For each label ℓ, we use a BFS-based algorithm to compute connected components with respect to the w-compatibility relation. While doing so, we maintain two instances of a data structure for dynamic orthogonal range reporting [Mortensen, 2006], one with points $(o_{\mathsf{F}}(u), c_{\mathsf{F}}(u))$ for all unvisited nodes $u \in V_{\mathsf{F}}$ with label ℓ, and one with points $(o_{\mathsf{G}}(v), c_{\mathsf{G}}(v))$ for all unvisited nodes $v \in V_{\mathsf{G}}$ with label ℓ. While we do not have direct access to edges incident to any vertex, an orthogonal range reporting query allows listing all unvisited neighbors of any vertex. Each listed node is immediately visited (and thus the corresponding point is removed from the range reporting data structure), so, in total, we have $\mathcal{O}(n)$ queries producing output of total size $\mathcal{O}(n)$. Thus, the algorithm works in $\mathcal{O}(n \log n)$ time. □

Observation 2.4.8. *Consider a joint labeling λ of forests F, G, an integer $w \in \mathbb{Z}_{\geq 0}$, and an alignment $\mathcal{A} \in \mathsf{TA}(\mathsf{F}, \mathsf{G})$ of width at most w. Then, $\mathsf{ted}_{\mathsf{C}(\lambda,w),\mathcal{A}}(\mathsf{F}, \mathsf{G}) = \mathsf{ted}_{\lambda,\mathcal{A}}(\mathsf{F}, \mathsf{G})$.*

2.5 Periodicity Reduction in Strings

In this section, we analyze the structure of alignments $\mathcal{A} \in \mathsf{GA}_{k,w}(X, Y)$ under the assumption that X, Y avoid certain periodic structure. This is loosely inspired by the proof of [Kociumaka et al., 2021, Lemma III.10].

Definition 2.5.1. We say that a pattern P has *s-synchronized occurrences* in strings X, Y if $P = X[x \mathinner{..} x + |P|) = Y[y \mathinner{..} y + |P|)$ holds for some positions x, y satisfying $|x - y| \leq s$.

Definition 2.5.2. Consider integers $s, e, \ell \in \mathbb{Z}_+$. We say that strings $X, Y \in \Sigma^*$ *avoid s-synchronized e-powers with root at most ℓ* if there is no non-empty string $Q \in \Sigma^{\leq \ell}$ such that Q^e has s-synchronized occurrences in X, Y.

Lemma 2.5.3. *Consider strings X, Y, integers $e, k, w \in \mathbb{Z}_+$, and two alignments $\mathcal{A}, \mathcal{A}' \in \mathsf{A}_{k,w}(X, Y)$. If X, Y avoid w-synchronized e-powers with root at most $2w$, then $|\mathcal{A} \setminus \mathcal{A}'| \leq 7wke$.*

Proof. Let us partition X into individual characters representing deletions or substitutions of $\mathcal{A}$ or $\mathcal{A}'$ and maximal fragments $X[x\,..\,x+\ell)$ that $\mathcal{A}$ and $\mathcal{A}'$ match perfectly: $X[x\,..\,x+\ell) \simeq_{\mathcal{A}} Y[y\,..\,y+\ell)$ and $X[x\,..\,x+\ell) \simeq_{\mathcal{A}'} Y[y'\,..\,y'+\ell)$ for some $y, y' \in [0\,..\,|Y|-\ell] \cap [x-w\,..\,x+w]$. Observe that the number of such maximal fragments is at most $2k+1$ and their total length is at least $|X| - 2k$.

If $y = y'$, then $(x+\delta, y+\delta) \in \mathcal{A} \cap \mathcal{A}'$ holds for all $\delta \in [0\,..\,\ell]$. Otherwise, $P := X[x\,..\,x+\ell)$ has w-synchronized occurrences in X, Y (starting at positions x, y) and $\mathsf{per}(P) \leq |y - y'| \leq 2w$. Denote $Q := X[x\,..\,x+\mathsf{per}(P))$ and observe that, if $\ell \geq 2we$, then Q^e is a prefix of P and thus also has w-synchronized occurrences in X, Y, contradicting our assumption. Hence, $y \neq y'$ is only possible if $\ell < 2we$. In either case, $X[x\,..\,x+\ell)$ contributes at least $\ell + 1 - 2we$ elements to $\mathcal{A} \cap \mathcal{A}'$.

Overall, the fragments contribute at least $|X|-2k+(1-2we)(2k+1) \geq |X|-4wke-2we+1$ elements to $\mathcal{A} \cap \mathcal{A}'$. Consequently, $|\mathcal{A} \setminus \mathcal{A}'| \leq |\mathcal{A}| - |\mathcal{A} \cap \mathcal{A}'| \leq |X|+1+k-(|X|-4wke-2we+1) = 4wke + 2we + k \leq 7wke$. □

Lemma 2.5.4 (Compare [Kociumaka et al., 2021, Lemma III.10])**.** *Consider strings $X, Y \in \Sigma^{\leq n}$ and integers $e, k, w \in \mathbb{Z}_+$. If X, Y avoid w-synchronized e-powers with root at most $2w$, then there exists a set M of size $|M| \geq |X| - 15wk^2e$ such that $M \subseteq \mathsf{M}_{\mathcal{A}}(X, Y)$ holds for every alignment*

$\mathcal{A} \in \mathsf{GA}_{k,w}(X, Y)$. *Moreover, such a set M can be constructed in $\mathcal{O}(n + kw)$ time.*

Proof. Let us fix an alignment $\mathcal{A} \in \mathsf{GA}_{k,w}(X, Y)$ and partition X into individual characters representing deletions or substitutions of $\mathcal{A}$ and maximal fragments $X[x\,..\,x + \ell)$ that $\mathcal{A}$ matches perfectly: $X[x\,..\,x + \ell) \simeq_{\mathcal{A}} Y[y\,..\,y + \ell)$ for some $y \in [0\,..\,|Y| - \ell]$. Observe that the number of such maximal fragments is at most $k + 1$ and their total length it at least $|X| - k$.

Let us fix a maximal fragment $X[x\,..\,x+\ell) \simeq_{\mathcal{A}} Y[y\,..\,y+\ell)$. We claim that $(x+\delta, y+\delta)$ can be added to M for every $\delta \in [7wke\,..\,\ell)$. For this, consider another alignment $\mathcal{A}' \in \mathsf{GA}_{k,w}(X, Y)$. By Lemma 2.5.3, we have $|\mathcal{A} \setminus \mathcal{A}'| \leq 7wke$. By the pigeonhole principle, this means that $(x + \delta, y + \delta) \in \mathcal{A}'$ holds for some $\delta \in [0\,..\,7wke)$. Then, the greedy nature of $\mathcal{A}'$ guarantees that $\mathcal{A}'$ must greedily match $X[x+\delta\,..\,x+\ell)$ and $Y[y+\delta\,..\,y+\ell)$ and, in particular, $X[x+7wke\,..\,x+\ell)$ and $Y[y+7wke\,..\,y+\ell)$. Hence, $X[x\,..\,x+\ell)$ contributes at least $\ell - 7wke$ pairs to M. The total contribution of all maximal fragments $X[x\,..\,x+\ell)$ is $|X| - k - (k+1) \cdot 7wke \geq |X| - 15wk^2e$.

As for the efficient algorithm, we use Lemma 2.3.3 to build $\mathcal{A}$ (if one exists) in $\mathcal{O}(n + kw)$ time. The remaining steps of the construction above can be easily implemented in $\mathcal{O}(n)$ time. □

2.6 Horizontal Periodicity Reduction

In this section, we discuss periodic sections of the input forests F and G. Specifically, we look at "horizontal periodicity," which refers to the case when the children of a given node are a periodic sequence of subtrees. If a horizontal period is repeated a large number of times in both F and G, we know that any minimum cost alignment of F and G must align the periodic subtrees in a predictable way. We then argue that we can consider two forests F' and G' that are the exact same as F and G with a bound on the number of repetitions of horizontal periodic subtrees.

For ease of analysis, we do these periodic reductions on the parentheses representations of $\mathsf{P}(F)$ and $\mathsf{P}(G)$ with tree alignments. Furthermore, we borrow some useful definitions and results from [Bannai et al., 2017] on maximal periodic substrings in strings, called runs.

Definition 2.6.1 (Runs, [Bannai et al., 2017])**.** A *run* in a string S is a triple (i, j, p) such that $S[i\,..\,j)$ is periodic with $\mathsf{per}(S[i\,..\,j)) = p$ and maximal in length, i.e., neither $S[i-1\,..\,j)$ nor $S[i\,..\,j+1)$ is a fragment with period p. We call $\frac{j-i}{p}$ the exponent of the run. Let $\mathsf{Runs}(S)$ denote the set of all runs in S.

Theorem 2.6.2 (Theorem 9 of [Bannai et al., 2017])**.** *The number of runs in a string S of length n is less than n.*

Theorem 2.6.3 (Algorithm 1 of [Bannai et al., 2017])**.** *Given a string S of length n, the set of runs $\mathsf{Runs}(S)$ can be computed in $\mathcal{O}(n)$ time.*

Using the results of the above theorems, we can find all horizontal periodic substrings of $\mathsf{P}(F)$ and $\mathsf{P}(G)$ in $\mathcal{O}(n)$ time. Ideally, we could just iterate through a list of computed runs and reduce the exponent of each run in constant time. However, some runs may be overlapping, and so by reducing one run, we may significantly change the length of a previously reduced run, or even worse introduce a new run to the string. So, by requiring that the period of runs that we reduce is small, we also can guarantee that the overlap of runs will be manageable.

Lemma 2.6.4 (Fact 2.2.4, [Kociumaka, 2018])**.** *Any two distinct runs (i_1, j_1, p_1) and (i_2, j_2, p_2) in a string S satisfy $|[i_1\,..\,j_1) \cap [i_2\,..\,j_2)| < p_1 + p_2 - \gcd(p_1, p_2)$.*

Corollary 2.6.5. *Consider two distinct runs (i_1, j_1, p_1) and (i_2, j_2, p_2) in a string S with periods $p_1, p_2 \le 4k$ and exponents $\frac{j_1-i_1}{p_1}, \frac{j_2-i_2}{p_2} \ge 8k$. If $i_1 \le i_2$, then $j_1 < j_2$ and $j_1 - i_2 < 8k$.*

Proof. By Lemma 2.6.4, we have $|[i_1 \,..\, j_1) \cap [i_2 \,..\, j_2)| < p_1 + p_2 - \gcd(p_1, p_2) < 8k$. Due to $|[i_2 \,..\, j_2)| \geq 8kp_2 \geq 8k$, this means that $[i_2 \,..\, j_2)$ is not contained in $[i_1 \,..\, j_1)$, i.e., $j_1 < j_2$. Moreover, $j_1 - i_2 = \min(j_1, j_2) - \max(i_1, i_2) \leq |[i_1 \,..\, j_1) \cap [i_2 \,..\, j_2)| < 8k$. □

Since horizontal periodic reductions take place on subtrees, we require that any periodic substring we reduce has a balanced string period in the parentheses representation. For a string X, we define a balance function $\sigma(X)$ which is mapped to the minimum number of rotations of X needed to make it balanced. If X cannot be balanced, $\sigma(X) := -1$. For example, let $X =$ `)][()(`, then $\sigma(X) = 2$ since $\mathsf{rot}^2(X) =$ `[()()]`; let $Y =$ `(((((`, then $\sigma(Y) = -1$ since there is no rotation that balances Y. We give the following lemma for computing the σ function.

Lemma 2.6.6. *$\sigma(X)$ can be computed in time $\mathcal{O}(|X|)$.*

Proof. To compute $\sigma(X)$, we run a folklore stack matching algorithm for checking balance on X. The algorithm iterates across the characters of X and every opening parenthesis the algorithm comes across gets pushed onto the stack. When the algorithm reaches a closing parenthesis, it pops an opening parenthesis from the stack and matches the two. We modify this basic algorithm so that if the stack is empty when a closing parenthesis is reached, it ignores the failed match and continues to the next character. Additionally, the algorithm will keep track of the largest index $m < |X|$ such that $X[m]$ is a closing parenthesis which had no match due to the stack being empty when this parenthesis was reached. Clearly we must rotate X at least $m + 1$ times in order for closing parenthesis $X[m]$ to potentially have an opening parenthesis before it in X to match to. Moreover, since m is the index of the last unmatched closing parenthesis in X then for any index $i > m$ there can only be unmatched opening parenthesis. Therefore, rotating more than $m + 1$ times will

only place unmatched opening parenthesis at the end of X with no potential matches afterwards. Thus, we rotate exactly $m+1$ times, and run the stack matching algorithm on $\mathsf{rot}^{m+1}(X)$. If the rotated string is balanced and all matching parentheses share the same parenthesis type, then $\sigma(X) = m+1$. Otherwise, $\sigma(X) := -1$. □

Note that the above lemma shows that checking balance and finding the minimum number of rotations to balance a string only takes linear time. Since we only want to find the balance of periodic substrings of length at most $4k$, then computing σ will only take time $\mathcal{O}(k)$.

From the above definition and results, we can now give our algorithm to reduce horizontal periodicity seen below in Algorithms 1, 2, and 3. The goal of Algorithm 3 is to take two input forests F and G and output forests F' and G' which are the same as the input forests except any long horizontal periodic synchronized occurrences are reduced to an exponent of at most $18k$. First, Algorithm 2 uses Algorithm 1 as a subroutine to compute all runs of the parentheses representations of input forests F and G, sorted by starting index. Before the algorithm can actually reduce any pair of runs, it must make sure the runs adhere to strict requirements. First, to make sure the overlap between runs that we want to reduce is not too large compared to the run size, we only reduce runs whose period is at most $4k$ in length and whose exponent is larger than $16k$. Algorithm 1 does exactly this and returns a filtered list of runs that meet these criteria. Afterwards, Algorithm 2 iterates over pairs of runs, one from $\mathsf{P}(F)$ and one from $\mathsf{P}(G)$ in sorted order by their starting index. Note that we only want to reduce periodic substrings who have $2k$-synchronized occurrences; otherwise we cannot align the matching periodic subtrees with less than k edits. If we find a very large run in one input forest with no synchronized occurrence in the other input forest, the algorithm should not reduce these occurrences.

After checking for these properties in each run, Algorithm 2 does one final check that the string period of the run is a balanced parentheses string, i.e., it actually corresponds to a sequence of subtrees in the original forests F and G. If a run's string period is not balanced, we do not actually have horizontal periodicity and cannot reduce these runs as easily (these runs are instead handled in Sections 2.7 and 2.8). If a run is not balanced, did not have a small enough period, or did not have a large enough exponent, the run is simply copied over to the corresponding output forest F' or G' without any changes. In the case that Algorithm 2 does find a pair of runs representing $2k$-synchronized occurrences of large horizontal periodicity, a triple representing the pair of runs and their common period and exponent is added to a set S. Then, Algorithm 3 iterates across set S and reduces all found $2k$-synchronized occurrences to only $14k$ repetitions of the string period. Algorithm 3 copies all characters in $\mathsf{P}(F)$ to a new string $\mathsf{P}(F')$ except any substrings contained in $2k$-synchronized occurrences of horizontal periodicity, in which instead the algorithm skips all but the last $14k$ repetitions of the string period. Once we have the output forests F' and G', it is still necessary to show that reduced runs have not changed the tree edit distance at all nor introduced any new horizontal periodicity. The rest of the section is devoted to the analysis of these three algorithms and their outputs.

Algorithm 1: FilterRuns(S).

```
R ← SortByStartingIndex(Runs(S));
R' ← empty list;
for (i, j, p) ∈ R do
    if p ≤ 4k and (j−i)/p ≥ 16k then
        R'.append((i, j, p));
return R';
```

We begin by proving that the runtime of Algorithm 2 is linear. Since we know that computing all runs in a string only takes $\mathcal{O}(n)$ time from Theorem 2.6.3, the call to Algorithm 1 do not cause

Algorithm 2: SyncOccurrences(F, G).

1 $R_F \leftarrow \mathsf{FilterRuns}(\mathsf{P}(F)), R_G \leftarrow \mathsf{FilterRuns}(\mathsf{P}(G))$;
2 $\ell_F, \ell_G \leftarrow 0$;
3 $S \leftarrow$ empty list;
4 **while** $\ell_F < |R_F|$ **and** $\ell_G < |R_G|$ **do**
5 $\quad (i_F, j_F, p_F) \leftarrow R_F[\ell_F], (i_G, j_G, p_G) \leftarrow R_G[\ell_G]$;
6 $\quad X \leftarrow \mathsf{P}(F)[i_F \mathinner{..} i_F + p_F), Y \leftarrow \mathsf{P}(G)[i_G \mathinner{..} i_G + p_G)$;
7 $\quad e \leftarrow \lfloor \frac{|[i_F \mathinner{..} j_F) \cap [i_G \mathinner{..} j_G)|}{p_F} \rfloor$;
8 $\quad$ **if** $e \geq 16k$ **and** $\exists \alpha \leq p_F$, $rot^\alpha(X) = Y$ **and** $\sigma(X) \geq 0$ **then**
$\quad\quad$ `// Add synchronized occurrences to set` S
9 $\quad\quad S.append((\max(i_F, i_G), p_F, e - 2k))$;
$\quad$ `// Go to a new run`
10 $\quad$ **if** $j_F < j_G$ **then**
11 $\quad\quad \ell_F \leftarrow \ell_F + 1$;
12 $\quad$ **else**
13 $\quad\quad \ell_G \leftarrow \ell_G + 1$;
14 **return** S;

any issues. As for the main loop of Algorithm 2, either the algorithm finds a reason that a pair of runs do not contain $2k$-synchronized occurrences and moves on to the next run, or the algorithm creates a synchronized occurrence triple and adds it to the output list S. Therefore, we must be careful that all the checks done in each case do not take superlinear total time.

Lemma 2.6.7. *Given forests F and G of total size n, Algorithm 2 runs in time $\mathcal{O}(n)$.*

Proof. The runtime of Algorithm 2 is mostly affected by the call to Algorithm 1 for sorting and computation of runs in step 1 as well as the number of iterations and work done in each iteration of the **while** loop spanning steps 4–14. First in step 1, by Theorem 2.6.3, computing all runs of both forests can be done in time $\mathcal{O}(n)$, and there are less than $2n$ runs total by Theorem 2.6.2. Therefore, sorting all $\mathcal{O}(n)$ runs by their starting index can be done using a radix sort in time $\mathcal{O}(n)$. The rest of Algorithm 1 filters runs in linear time to make sure the period is small enough and the exponent is large enough. Due to these filtering steps and Corollary 2.6.5, we can observe

that $|R_F|, |R_G| = \mathcal{O}(n/k)$.

We look individually at the work done in the subsequent iterations of the **while** loop. An important observation is that one of the run counters ℓ_F or ℓ_G always increments by 1 (steps 11 and 13) per iteration of the loop. Line 8 involves three checks. First, the overlap of the runs is computed, which takes constant time. Then, we check if a rotation of string period X is equal to a rotation of string period Y. Since $p_F \leq 4k$, checking all rotations takes time $\mathcal{O}(k)$ using a rolling hash function [Karp and Rabin, 1987]. The last check just requires computing $\sigma(X)$, which by Lemma 2.6.6 takes time $\mathcal{O}(|X|) = \mathcal{O}(k)$. The remaining instructions within each iteration of the **while** loop take $\mathcal{O}(1)$ time, so each iteration costs $\mathcal{O}(k)$ time in total. As mentioned earlier, $|R_F|, |R_G| = \mathcal{O}(n/k)$, and hence the entire algorithm is done in linear time. □

Now, we prove that Algorithm 2 indeed finds $2k$-synchronized occurrences of horizontal periodicity with large exponent in F and G. Any triple (i, p, e) added to set S by Algorithm 2 satisfies these requirements as per the checks in step 8. Therefore, the following proof is fairly straightforward and mostly just formalizes this intuition.

Lemma 2.6.8. *Given forests F, G, let $S = \mathsf{SyncOccurrences}(F, G)$. Then for any $(i, p, e) \in S$, there must be runs (i_F, j_F, p) in $\mathsf{P}(F)$, (i_G, j_G, p) in $\mathsf{P}(G)$ each containing $2k$-synchronized occurrences of string Q^e in $\mathsf{P}(F)$ and $\mathsf{P}(G)$ where $|Q| = p \leq 4k$, $e \geq 14k$, $i = \max(i_F, i_G)$, and Q is balanced.*

Proof. First, we observe that triples (i, p, e) are only inserted to S in step 9 of Algorithm 2. Let (i_F, j_F, p_F) and (i_G, j_G, p_G) be the runs considered by the algorithm during such an insertion, and note that $p = p_F = p_G \leq 4k$ and $i = \max(i_F, i_G)$. By symmetry, we assume without loss of generality that $i_F \leq i_G$. From the checks done in step 8, the string periods $X = \mathsf{P}(F)[i_F \,..\, i_F + p)$

and $Y = \mathsf{P}(G)[i_G \,..\, i_G + p)$ are the same up to rotation, i.e., they satisfy $\mathsf{rot}^{\alpha}(X) = Y$ for some $\alpha \in [0 \,..\, p)$. Moreover, $\sigma(X) \neq 1$, that is, the string $Q := \mathsf{rot}^{\sigma(X)}(X)$ is balanced. Observe that there exists a position $i'_F \in [i \,..\, i+p)$ such that $i'_F - i_F \equiv \sigma(X) \pmod{p}$. From step 8, we further know that $e + 2k \geq 16k$ and that the set of overlapping indices between the two runs satisfies

$$|[i \,..\, \min(j_F, j_G))| = |[i_F \,..\, j_F) \cap [i_G \,..\, j_G)| \geq p_F \left\lfloor \frac{|[i_F \,..\, j_F) \cap [i_G \,..\, j_G)|}{p_F} \right\rfloor = p(e + 2k) \geq p(e + 2).$$

Hence, Q^e occurs in $\mathsf{P}(F)$ at both position $i'_F, i'_F + p \in [i \,..\, i + 2p)$. Moreover, there exists a position $i'_G \in [i'_F \,..\, i'_F + p)$ such that $i'_G - i_G \equiv \sigma(X) - \alpha \pmod{p}$, which, due to $\mathsf{rot}^{\sigma(X)-\alpha}(Y) = Q$ and $[i'_F \,..\, i'_F + p) \subseteq [i \,..\, i + 2p)$, implies that Q^e occurs in $\mathsf{P}(G)$ at position i'_G. Since $p \leq 4k$, we conclude that Q^e has $2k$-synchronized occurrence in $\mathsf{P}(F)$ and $\mathsf{P}(G)$ at positions (i'_F, i'_G) or $(i'_F + p, i'_G)$, depending on whether $i'_G \in [i'_F \,..\, i'_F + 2k]$ or $i'_G \in (i'_F + 2k \,..\, i'_F + p)$. □

Algorithm 3: SyncReductions(F, G)

```
1  S ← SyncOccurrences(F, G);
2  s_F, s_G ← ε;
3  ℓ ← 0;
4  for (i, p, e) ∈ S do
       // Copy from start of previous synchronized occurrences
          to start of next synchronized occurrences
5      s_F ← s_F · P(F)[ℓ .. i);
6      s_G ← s_G · P(G)[ℓ .. i);
       // Reduce synchronized occurrences to exponent of 14k
7      ℓ ← i + p(e − 14k);
8  s_F ← s_F · P(F)[ℓ .. |P(F)|);
9  s_G ← s_G · P(G)[ℓ .. |P(G)|);
10 return s_F, s_G;
```

Next, we prove the main statement that we need in order to show that the outputted forests F' and G' from Algorithm 3 have the same edit distance as the original forests F and G. Conceptually,

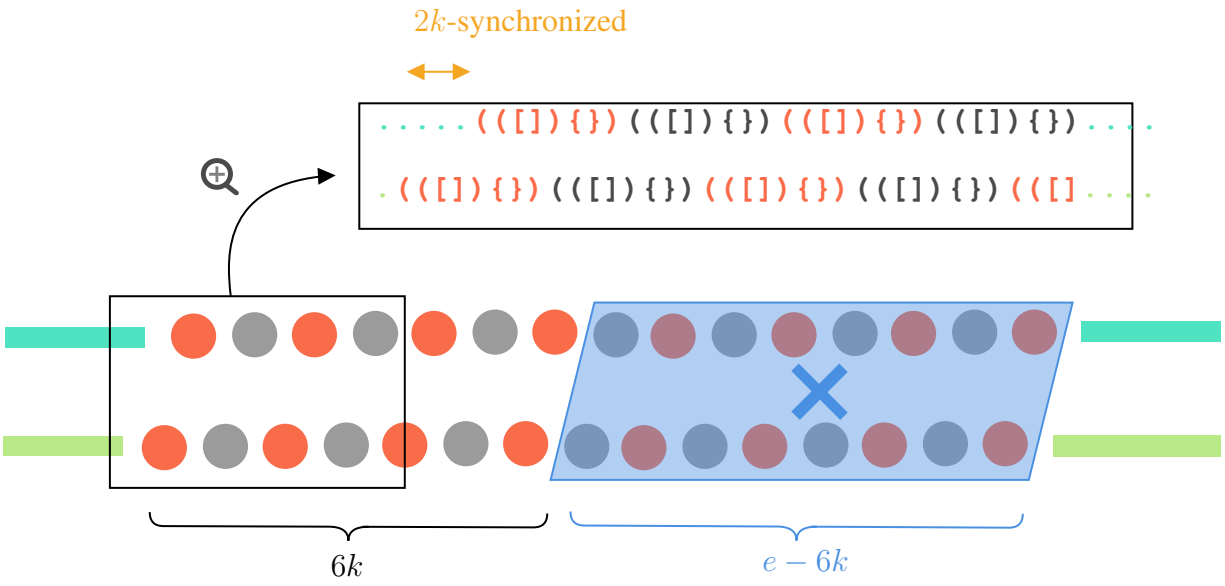


Figure 2.3: A $2k$-synchronized horizontal periodicity with $|Q| = 8$ is demonstrated. Each gray and red circle correspond to a balanced period block in $\mathsf{P}(F)$ and $\mathsf{P}(G)$ similar to the zoomed portion. According to Lemma 2.6.9 we can reduce the period exponent to $6k$ by removing the blue part of $\mathsf{P}(F)$ and $\mathsf{P}(G)$ so that the tree edit distance $\mathsf{ted}_{\leq k}(F, G)$ remains unchanged.

the proof just shows that for long synchronized occurrences of a run in both forests, any two minimal cost alignments of F and G must match long segments of these periodic sections together. In fact, we show that given any minimal cost tree alignment $\mathcal{A}$ of F and G, we can build a minimal cost tree alignment of F' and G' that follows $\mathcal{A}$ almost exactly.

Lemma 2.6.9. *Consider forests F, G such that $\mathsf{P}(F)[\alpha_F \,..\, \beta_F) = \mathsf{P}(G)[\alpha_G \,..\, \beta_G) = Q^e$ for a balanced string Q of length $0 < |Q| \leq 4k$, an integer exponent $e \geq 6k$, and indices $\alpha_F, \alpha_G, \beta_F, \beta_G$ satisfying $|\alpha_F - \alpha_G| \leq 2k$. Let F', G' be forests such that $\mathsf{P}(F') = \mathsf{P}(F)[0 \,..\, \alpha_F) \cdot Q^{e'} \cdot \mathsf{P}(F)[\beta_F \,..\, |\mathsf{P}(F)|)$ and $\mathsf{P}(G') = \mathsf{P}(G)[0 \,..\, \alpha_G) \cdot Q^{e'} \cdot \mathsf{P}(G)[\beta_G \,..\, |\mathsf{P}(G)|)$ for some integer exponent $e' \geq 6k$. Then, $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$.*

Proof. We assume without loss of generality that Q is primitive (otherwise, we replace Q by its primitive root) and denote $p := |Q| \leq 4k$. Let $\mathcal{A}$ be an optimal tree alignment such that $\mathsf{ted}(F, G) = \mathsf{ted}_{\mathcal{A}}(F, G) \leq k$.

Claim 2.6.10. *There exist* $i_F, i_G \in [0\,..\,3k]$ *such that* $(\alpha_F + i_F \cdot p, \alpha_G + i_G \cdot p) \in \mathcal{A}$ *and* $j_F, j_G \in [0\,..\,3k]$ *such that* $(\beta_F - j_F \cdot p, \beta_G - j_G \cdot p) \in \mathcal{A}$.

Proof. Let $(x_F, x_G) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $x_F \geq \alpha_F$ and $x_G \geq \alpha_G$. By symmetry between F and G, we assume without loss of generality that $x_F = \alpha_F$. Consider the $k+1$ occurrences of Q starting at positions $\alpha_F + i \cdot p$ for $i \in [0\,..\,k]$. Since Q is balanced, the alignment $\mathcal{A}$ (of cost at most k) matches at least one of them exactly; we can thus define $i_F \in [0\,..\,k]$ so that $\mathcal{A}$ matches $\mathsf{P}(F)[\alpha_F + i_F \cdot p\,..\,\alpha_F + (i_F+1) \cdot p)$ exactly to some fragment $\mathsf{P}(G)[y_G\,..\,y_G + p)$. Due to $(x_F, x_G) \in \mathcal{A}$, the non-crossing property of $\mathcal{A}$ implies that $y_G \geq \alpha_G$. Moreover, since $\mathsf{ted}_{\mathcal{A}}(F, G) \leq k$ and $\mathsf{P}(F)[\alpha_F + i_F \cdot p] \sim_{\mathcal{A}} \mathsf{P}(G)[y_G]$, we have $y_G \leq (\alpha_F + i_F \cdot p) + 2k \leq \alpha_F + kp + 2k \leq \alpha_G + kp + 4k \leq \alpha_G + 3kp$, where the last inequality follows from $p \geq 2$ (recall that Q is balanced, so its length is even). Furthermore, since Q is primitive (i.e., distinct from all its non-trivial cyclic rotations), we conclude that $y_G = \alpha_G + i_G \cdot p$ for some $i_G \in [0\,..\,3k]$. The second claim is symmetric (with respect to reversal). □

Observe that $\mathsf{P}(F)[\alpha_F + i_F p\,..\,\beta_F - j_F p) = Q^{d_F}$ for $d_F := e - j_F - i_F$ and, symmetrically, $\mathsf{P}(G)[\alpha_G + i_G p\,..\,\beta_G - j_G p) = Q^{d_G}$ for $d_G := e - j_G - i_G$. We denote $d = \min(d_F, d_G)$, observe that $d \geq e - 6k \geq 0$, and construct a tree alignment $\mathcal{A}'$ so that it

- aligns $\mathsf{P}(F)[0\,..\,\alpha_F + i_F p)$ with $\mathsf{P}(G)[0\,..\,\alpha_G + i_G p)$ in the same way as $\mathcal{A}$ does;
- matches $\mathsf{P}(F)[\alpha_F + i_F p\,..\,\alpha_F + (i_F + d)p) = Q^d$ with $\mathsf{P}(G)[\alpha_G + i_G p\,..\,\alpha_G + (i_G + d)p) = Q^d$;
- deletes $\mathsf{P}(F)[\alpha_F + (i_F + d)p\,..\,\beta_F - j_F p) = Q^{d_F - d}$ and $\mathsf{P}(G)[\alpha_G + (i_G + d)p\,..\,\beta_G - j_G p) = Q^{d_G - d}$;
- aligns $\mathsf{P}(F)[\beta_F - j_F p\,..\,|\mathsf{P}(F)|)$ with $\mathsf{P}(G)[\beta_G - j_G p\,..\,|\mathsf{P}(G)|)$ in the same way as $\mathcal{A}$ does.

Note that $\mathcal{A}'$ is a tree alignment: for any node of F, the corresponding parentheses are either both

outside $\mathsf{P}(F)[\alpha_F + i_F p \,..\, \beta_F - j_F p)$ (and then they are handled as in $\mathcal{A}$) or both contained in a single copy of Q (which is either deleted or matched perfectly to a copy of Q in $\mathsf{P}(G)$). Moreover, the cost of $\mathcal{A}'$ does not exceed the cost of $\mathcal{A}$: the two alignments only differ in how they align Q^{d_F} with Q^{d_G}, and $\mathcal{A}'$ provides an optimum alignment of these fragments.

Now, if the exponent e of $Q^e = \mathsf{P}(F)[\alpha_F \,..\, \beta_F) = \mathsf{P}(G)[\alpha_G \,..\, \beta_G)$ is modified to $e' \geq 6k$, we can interpret this as modifying exponent d of the fragments Q^d matched perfectly by $\mathcal{A}'$ to $d' = d + e' - e \geq 0$. Thus, $\mathcal{A}'$ can be trivially adapted without modifying its cost and hence $\mathsf{ted}(F', G') \leq \mathsf{ted}_{\mathcal{A}'}(F, G) = \mathsf{ted}(F, G)$. The converse inequality follows by symmetry between (F, G) and (F', G'). □

From the previous lemma, it is clear that reducing a long run does not affect edit distance. Utilizing this idea, we finally prove that the forests outputted by Algorithm 3 have the same edit distance $\mathsf{ted}_{\leq k}$ as the input forests and avoid $2k$-synchronized runs with an exponent more than $14k$.

Definition 2.6.11 (Synchronized horizontal periodicity)**.** We say that forests F, G *avoid synchronized horizontal k-periodicity* if there is no non-empty balanced string Q of length $|Q| \leq 4k$ such that Q^{18k} has $2k$-synchronized occurrences in $\mathsf{P}(F), \mathsf{P}(G)$.

Proposition 2.6.12 (Avoiding Horizontal k-Periodicity)**.** *There exists an $\mathcal{O}(n)$-time algorithm that, given labeled forests F, G of total size n and an integer $k \in \mathbb{Z}_+$, produces labeled forests F', G' that satisfy $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$ and, moreover, avoid synchronized horizontal k-periodicity.*

Proof. Let $s_F, s_G = \mathsf{SyncReductions}(F, G)$. By Lemma 2.6.8, for any triple $(i, p, e) \in S$ of Algorithm 3, there are $2k$-synchronized occurrences in F, G of some Q^e with $|Q| = p$ satisfying the constraints of Lemma 2.6.9 within runs $r_1 = (i_1, j_1, p)$ and $r_2 = (i_2, j_2, p)$ of F and G, respectively.

In step 5–7, we move the start of the synchronized occurrences forward by $(e - 14k)p$ and do not copy the skipped over indices to s_F and s_G. This is equivalent to reducing the exponent of r_1 and r_2 by $e - 14k$ since $\frac{j_1-(i_1+(e-14k)p)}{p} = \frac{j_1-i_1}{p} - (e - 14k)$. Furthermore, since the $2k$-synchronized occurrences with exponent e lie completely within r_1 and r_2, steps 5–7 are actually equivalent to reducing the synchronized occurrences of Q^e to Q^{14k} in each forest. Note that since Q is balanced, there exist forests F', G' such that $\mathsf{P}(F') = s_F$ and $\mathsf{P}(G') = s_G$. Therefore, by Lemma 2.6.9, we have that $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$.

Now, we must show that F', G' avoid synchronized horizontal k-periodicity. Suppose for contradiction that there exist $2k$-synchronized occurrences of some Q^{18k} with balanced string period Q such that $|Q| \leq 4k$ in F' and G'. If Q^{18k} was initially found in $2k$-synchronized occurrences in F and G, then since it is periodic, it must be the case that it was contained in runs $(i_F, j_F, p_F), (i_G, j_G, p_G)$, in F and G respectively, with $p_F, p_G \leq |Q| \leq 4k$ and $\frac{j_F-i_F}{p_F}, \frac{j_G-i_G}{p_G} \geq 18k$. Without loss of generality, let $j_F < j_G$. Since Q^{18k} is a $2k$-synchronized occurrence, it must be that the overlap of these runs is at least $18k - 2k = 16k$ and so, $j_F > i_G + 8k$. Let (i'_G, j'_G, p') be the run in R_G preceding (i_G, j_G, p_G). Note that by Corollary 2.6.5, $j'_G < i_G + 8k < j_F$. Furthermore, we only increment the run counter of the forest whose current run ends before the other forest's current run. Since $j'_G < j_F$ and $j_F < j_G$, we will always reach an iteration of Algorithm 2 that considers the pair of runs $(i_F, j_F, p_F), (i_G, j_G, p_G)$. Since the overlap of the two runs is at least $16k$, Algorithm 2 will therefore add triple $(\max(i_F, i_G), p_F, e - 2k)$ to S for some $e \geq 16k$. Then Algorithm 3 reduces the $2k$-synchronized occurrences containing Q^{16k} to an exponent of at most $14k$ in steps 5–7. Therefore, if any $2k$-synchronized occurrences with exponent at least $18k$ and period at most $4k$ is present in F and G, it will be reduced to an exponent of at most $14k$ in F', G'.

Now, if F', G' do not avoid synchronized horizontal k-periodicity it must be the case that

by reducing the exponent of overlapping runs, we created new $2k$-synchronized occurrences Q^{18k} where $|Q| \leq 4k$. Clearly, Q^{18k} must lie in some new run $r'_Q = (i'_Q, j'_Q, p'_Q)$ in F' since it is periodic. We will show that this is not possible for such a run to form due to the small overlap between periodic substrings with period at most $4k$. By Corollary 2.6.5, the overlap between r'_Q and any other reduced runs is at most $8k$. Since r'_Q has length at least $18k$, r'_Q may overlap at most two reduced runs. We refer to the two reduced runs as $r'_1 = (i'_1, j'_1, p'_1)$ and $r'_2 = (i'_2, j'_2, p'_2)$ and without loss of generality assume they are in $\mathsf{P}(F')$. Let $\mathsf{P}(F')[j'_1 - \beta_1 \,..\, j'_1]$ be the overlap between r'_1 and r'_Q, and similarly let $\mathsf{P}(F')[i'_2 \,..\, i'_2 + \beta_2]$ be the overlap between r'_2 and r'_Q. Since r'_1 and r'_2 are reduced, there are two corresponding runs $r_1 = (i_1, j_1, p_1), r_2 = (i_1, j_2, p_1)$ in $\mathsf{P}(F)$, such that $\mathsf{P}(F)[j_1 - \beta_1 \,..\, j_1) = \mathsf{P}(F')[j'_1 - \beta_1 \,..\, j'_1)$ and $\mathsf{P}(F)[i_2 \,..\, i_2 + \beta_2) = \mathsf{P}(F')[i'_2 \,..\, i'_2 + \beta_2)$. Note that since we do not change any characters between any runs, we also know that the middle substrings $\mathsf{P}(F')[j'_1 \,..\, i'_2] = \mathsf{P}(F)[j_1 \,..\, i_2]$ are equal as well. Combining these three substrings we have that $\mathsf{P}(F')[i'_Q \,..\, j'_Q) = \mathsf{P}(F)[j_1 - \beta_1 \,..\, i_2 + \beta_2)$, which implies that Q^{18k} is a periodic substring of $\mathsf{P}(F)$. In other words, Q^{18k} is contained in a run in $\mathsf{P}(F)$ before any reductions occur, which is a contradiction.

Finally, we discuss the runtime of Algorithm 3. First Algorithm 3 calls Algorithm 2 in step 1, and by Lemma 2.6.7, this takes time $\mathcal{O}(n)$. Now, we consider the loop in steps 4–7. We copy substrings from $\mathsf{P}(F)$ and $\mathsf{P}(G)$ of the start of one pair of synchronized occurrences to the start of the next pair of synchronized occurrences. Note that synchronized occurrences we copy have periods at most $4k$ and exponents at least $14k$, and so by Corollary 2.6.5, we know that no two synchronized occurrences will start at the same index. Therefore, we only copy each character of $\mathsf{P}(F)$ and $\mathsf{P}(G)$ at most once across the entire algorithm, and so Algorithm 3 takes time $\mathcal{O}(n + |\mathsf{P}(F)| + |\mathsf{P}(G)|) = \mathcal{O}(n)$. □

2.7 Vertical Periodicity Reduction

In order to remove periodicity from regions of $\mathsf{P}(F), \mathsf{P}(G)$ which may be unbalanced, we consider a second type of periodicity, *vertical* periodicity, in addition to horizontal periodicity. Avoiding horizontal periodicity allows us to reduce large powers of repeated balanced substrings; in this section, we essentially aim to reduce large powers of pairs of periodic substrings which are balanced together but may be unbalanced separately. We do so by finding paths of nodes in forests F and G such that the children to the left and right of the path of each node in the path are the same, which we call vertical periodicity. At the bottom of the path, we may no longer have vertical periodicity, and so in a parentheses representation of the forest we will have two separate periodic substrings, one to the left of the path and one to the right of the path. We define some useful notation for vertical periodicity as follows:

Definition 2.7.1 (Context)**.** We define a *context* as a pair $C = (C_L, C_R)$ such that $C_L \cdot C_R = \mathsf{P}(T)$ for some labeled forest T.

Definition 2.7.2 (Vertical composition)**.** For a context $C = (C_L, C_R)$ and a labeled forest F, we denote by $C\langle F\rangle$ the labeled forest H such that $\mathsf{P}(H) = C_L \cdot \mathsf{P}(F) \cdot C_R$. Similarly, for two contexts $C = (C_L, C_R)$ and $D = (D_L, D_R)$, we denote $C\langle D\rangle = (C_L D_L, D_R C_R)$.

Observe that the vertical composition of contexts is associative. For a context C and an integer $e \in \mathbb{Z}_+$, we define C^e as the context obtained by vertical composition of e copies of C. We say a forest context C *occurs* at node u of a labeled forest F if the subtree of F rooted at node u is of the form $C\langle H\rangle$ for some labeled forest H.

We say that context C has s-synchronized occurrences in labeled forests F, G if C occurs at

a node u of F and at a node v of G such that $|o_F(u) - o_G(v)| \leq s$ and $|c_F(u) - c_G(v)| \leq s$.

Definition 2.7.3 (Synchronized vertical periodicity)**.** We say that forests F, G *avoid synchronized vertical k-periodicity* if there is no context $C = (C_L, C_R)$ with $|C_L|, |C_R| \leq 4k$ such that C^{18k} has $2k$-synchronized occurrences in F, G.

To avoid synchronized vertical k-periodicity, we first compute periodic contexts which occur in forests F and G individually without concern for synchronicity. Note that if a forest has such a periodic context $C^{18k} = (C_L, C_R)^{18k}$, then that forest has two separate periodic substrings we want to identify, namely C_L^{18k} and C_R^{18k}. Additionally, from the definition of context it is clear that while C_L and C_R do not need to be balanced parentheses strings, $C_L \cdot C_R$ does have to be balanced. Therefore, C_L cannot begin with a closing parenthesis and C_R cannot end with an opening parenthesis since such parentheses would have no match in $C_L \cdot C_R$. For this reason, we will want to find periodic substrings starting at opening parentheses in $\mathsf{P}(F), \mathsf{P}(G)$ as well as periodic substrings ending at closing parentheses

For a node u in a forest F, consider $q_L \in [1\,..\,4k]$ and $e \in \mathbb{Q}$ such that q_L is a period of $\mathsf{P}(F)[o(u)\,..\,o(u) + q_L \cdot e)$ and e is maximized. More specifically, we want to find the longest periodic substring starting at $o(u)$ with period at most $4k$ and exponent at least $16k$. If $e \geq 16k$, we may have vertical periodicity that we want to avoid, and so we define an array Q_F to store these values. Let $Q_F[o(u)] := (q_L, o(u) + q_L e)$. If $e < 16k$, we do not need to worry about reducing any substring starting at $o(u)$ and so, we set a default value $Q_F[o(u)] := (1, o(u))$. Since we want to find periodic substrings ending in closing parentheses as well, we define $Q_F[c(u)] := (q_R, i)$ where $\mathsf{P}(F)(c(u) - i\,..\,c(u)]$ is the longest periodic substring ending at $c(u)$ with a period $q_R \leq 4k$ and exponent at least $16k$. Again if the exponent is less than $16k$, we define $Q_F[c(u)] := (1, c(u))$.

Lemma 2.7.4. *Given a forest F, Q_F can be computed in time $\mathcal{O}(n)$.*

Proof. Initially, we set $Q_F[\ell] := (1, \ell)$ as the default value for all $\ell \in [0\,..\,2|F|)$ and use Algorithm 1 to compute the set R_F of runs (i, j, p) of $\mathsf{P}(F)$ with period $p \leq 4k$ and exponent $\frac{j-i}{p} \geq 16k$; as discussed in Section 2.6, this costs $\mathcal{O}(n)$ time. For every run $(i, j, p) \in R_F$, take any index $\ell \in [i\,..\,j)$ such that $\mathsf{P}(F)[\ell]$ is an opening parenthesis. If the exponent of the substring $\mathsf{P}(F)[\ell\,..\,j)$, that is, $\frac{j-\ell}{p}$, is at least $16k$, we update $Q_F[\ell]$ to (p, j). By iterating over all indices in each run, we will find the longest periodic substring starting at every opening parenthesis in Q_F with exponent at least $16k$. Note that any two runs with period at most $4k$ has less than $8k$ overlapping characters by Corollary 2.6.5, and therefore, we only consider each index ℓ twice. Furthermore, we update each value $Q_F[\ell]$ at most once because the indices in the overlap of the runs cannot have an exponent of at least $16k$ in each run (otherwise the overlap would exceed $8k$).

To finish the computation of Q_F, we do the analogous steps for closing parentheses. For every run $(i, j, p) \in R_F$, we take any index $\ell \in [i\,..\,j)$. If the exponent of the substring $\mathsf{P}(F)[i\,..\,\ell] = \mathsf{P}(F)(i-1\,..\,\ell]$, that is, $\frac{\ell-(i-1)}{p}$, is at least $16k$, we update $Q_F[\ell]$ to $(p, i-1)$. Now, note that by Corollary 2.6.5, runs with period at most $4k$ can only overlap in at most $8k$ indices. Furthermore, since we only update Q_F if the exponent of a run is at least $16k = 2(8k)$, any index $\ell \leq |\mathsf{P}(F)|$ can be contained in at most two runs of period at most $4k$ and exponent at least $16k$. Therefore, iterating through all such runs of $\mathsf{P}(F)$ and computing Q_F for each index takes time $\mathcal{O}(n)$. □

We give Algorithm 4 for pseudocode on computing Q_F from forest F. Now that we have computed Q_F, we can iterate over all nodes of forest F to compute the maximal periodic context that occur at each node $u \in F$. Let $Q_F[o(u)] = (q_L, j_L)$ and $Q_F[c(u)] = (q_R, j_R)$, and denote the string periods by $P_L = \mathsf{P}(F)[o(u)\,..\,o(u) + q_L)$ and $P_R = \mathsf{P}(F)(c(u) - q_R\,..\,c(u)]$. Assume

Algorithm 4: ComputeQ(F).

$R_F \leftarrow \mathsf{FilterRuns}(\mathsf{P}(F))$;
$Q_F[\ell] \leftarrow (1, \ell) \quad \forall \ell \in [|\mathsf{P}(F)|]$;
for $(i, j, p) \in R_F$ **do**
  for $\ell \in [i \,..\, j)$ **do**
    if $\mathsf{P}(\mathsf{F})[\ell]$ *is an opening parenthesis* **and** $\frac{j-\ell}{p} \geq 16k$ **then**
      $Q_F[\ell] \leftarrow (p, j)$;
    if $\mathsf{P}(\mathsf{F})[\ell]$ *is a closing parenthesis* **and** $\frac{\ell-(i-1)}{p} \geq 16k$ **then**
      $Q_F[\ell] \leftarrow (p, i-1)$;

that there is a periodic context C^e that occurs at u such that $C = (C_L, C_R)$, e is maximized, and the subtree rooted at u is of form $C^e\langle H\rangle$. This means that for two positive integers $r_L, r_R \in \mathbb{Z}_+$, $|C_L| = r_L q_L$ and $|C_R| = r_R q_R$ since C_L and C_R could be periodic themselves. To find the correct coefficients r_L and r_R, we need to compute the number of unmatched opening parentheses in P_L and the number of unmatched closing parentheses in P_R, denoted by d_L and d_R respectively. Values d_L and d_R correspond to the depth that P_L and P_R go down in the tree. The highest node in which the runs starting at $o(u)$ and ending at $c(u)$ synchronize is located at depth $d := \mathrm{lcm}(d_L, d_R)$. By synchronizing we mean that the unmatched opening and closing parentheses are matched so that $C_L \cdot C_R$ is balanced. Hence, $C_L = P_L^{d/d_L}$ and $C_R = P_R^{d/d_R}$. After this step we need to check whether the conditions $|C_L| \leq 4k$, $|C_R| \leq 4k$, and $e \geq 16k$ still hold.

The next step is to find the maximum exponent e such that C^e occurs at node u. There is no guarantee that $\mathsf{P}(F)[j_L \,..\, j_R]$ forms a balanced substring. For example, the left run could finish earlier than the right run, or the two runs could *diverge* as illustrated in Figure 2.4. To fix this issue, we first find the vertices v_L and v_R corresponding to indices j_L and j_R in $\mathsf{P}(F)$ respectively. If v^* is the *Lowest Common Ancestor* (LCA) of v_L and v_R, then

$$e = \left\lfloor \min \left\{ \tfrac{j_L - o(u)}{|C_L|}, \tfrac{c(u) - j_R}{|C_R|}, \tfrac{c(u) - o(u) + 1}{|C_L| + |C_R|}, \tfrac{D[o(v^*)] - D[o(u)] + 1}{d} \right\} \right\rfloor$$

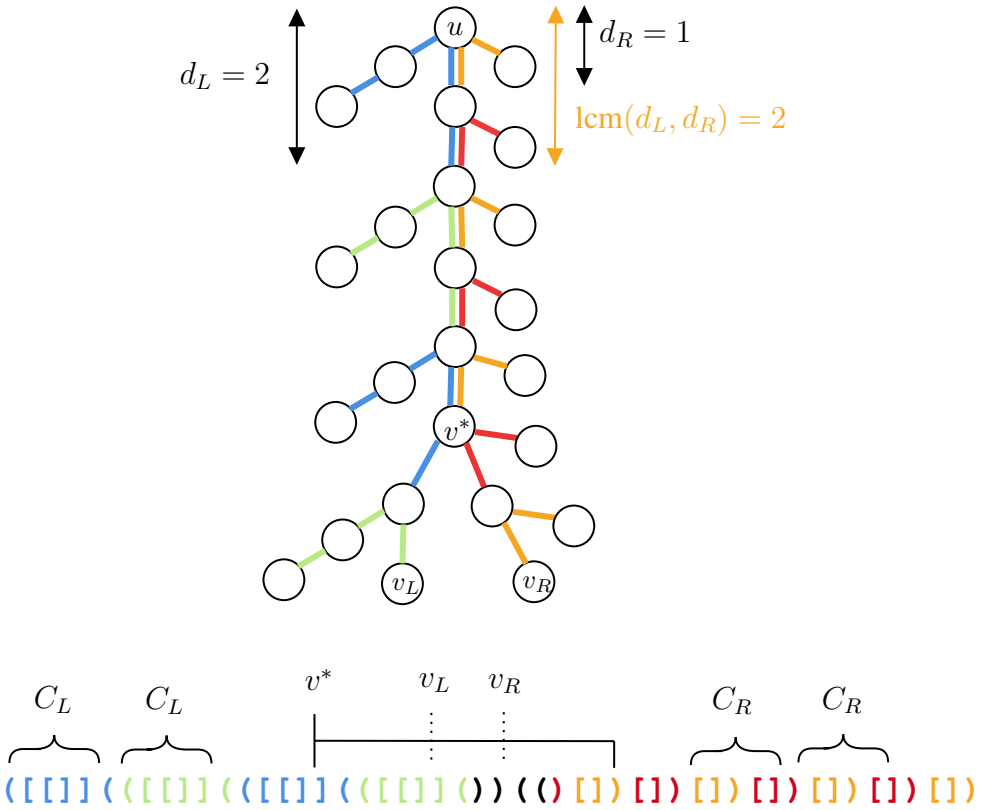


Figure 2.4: An example tree in which the left run starting at $o(u)$ and the right run ending at $c(u)$ diverge. The depth of the runs are $d_L = 2$ and $d_R = 1$. Therefore, the depth of the context is $\mathrm{lcm}(d_L, d_R) = 2$, and the left and right runs end at v_L and v_R respectively. In this case, the LCA v^* of v_L and v_R is at distance 5 from u, which indicates the exponent $e = \lfloor \frac{5+1}{2} \rfloor = 3$ of context C^e.

where the value $D[o(v)] = D[c(v)]$ is the depth of node v (i.e., the distance to the root of the corresponding tree). The arguments of the min function indicate that the left run is long enough, the right run is long enough, the subtree of u is large enough (which is needed if the left run overlaps the right run), and that the runs do not diverge too early. Again, we need to check whether $e \geq 16k$.

Definition 2.7.5. Given a forest F, let $\mathcal{C}(F)$ be a set of quadruples (u, q_L, q_R, e) such that

1. u is a node in F
2. $q_L, q_R \leq 4k$, $e \geq 16k$.
3. $C_L = \mathsf{P}(F)[o(u) \,..\, o(u) + q_L), C_R = \mathsf{P}(F)(c(u) - q_R \,..\, c(u)]$ form context $C = (C_L, C_R)$

in the subtree rooted at u.

4. C^e is a maximal context, i.e., C^{e+1} does not occur at u.

Lemma 2.7.6. *$\mathcal{C}(F)$ can be computed in $\mathcal{O}(n \log n)$ time.*

Algorithm 5: ComputeContexts(F, D).

```
1  Q_F ← ComputeQ(P(F));
2  C ← empty set;
3  for u ∈ V_F do
4      (q_L, j_L) ← Q_F[o(u)], (q_R, j_R) ← Q_F[c(u)];
       // Check for dummy Q_F values and make sure o(u) and c(u)
          are not in the same period
5      if j_L ≠ o(u) and j_R ≠ c(u) and (c(u) − o(u)) ≥ max{q_L, q_R} then
6          d_L ← D[o(u) + q_L] − D[o(u)];
7          d_R ← D[c(u) − q_R] − D[c(u)];
8          d ← lcm(d_L, d_R);
9          C_L ← P(F)[o(u) .. o(u) + q_L)^{d/d_L};
10         C_R ← P(F)(c(u) − q_R .. c(u)]^{d/d_R};
11         if |C_L| > 4k or |C_R| > 4k then
12             continue;
           // Let v_L, v_R be the nodes corresponding to j_L, j_R
13         v* ← LCA(v_L, v_R);
14         e ← ⌊min{(j_L − o(u))/|C_L|, (c(u) − j_R)/|C_R|, (c(u) − o(u) + 1)/(|C_L| + |C_R|), (D[o(v*)] − D[o(u)] + 1)/d}⌋;
15         if e ≥ 16k then
16             C.insert((u, |C_L|, |C_R|, e));
17 return C;
```

Proof. Given forest F, we showed that we can find Q_F in $\mathcal{O}(n)$ time in Lemma 2.7.4. In the rest of the algorithm, we iterate over the nodes of F. For each node $u \in V_F$, we check $Q_F[o(u)]$ and $Q_F[c(u)]$ to find a potential periodic context rooted at vertex u. If the desired context exists with period at most $4k$ and exponent at least $16k$, it satisfies the required conditions to enter $\mathcal{C}(F)$.

In order to find the period of a maximal context, we first define array $D[0\,..\,2|F|)$ so that, for each node $u \in V_F$, the value $D[o(u)] = D[c(u)]$ is the depth of node u (i.e., the distance to the root of the corresponding tree). Recall that d_L is the number of unmatched opening parentheses in P_L,

the shortest period starting from $o(u)$ which is stored in array Q_F, and similarly d_R is the number of the number of unmatched closing parenthesis in P_R. The shortest period of a maximal context is equal to $\mathrm{lcm}(d_L, d_R)$, which can be found in $\mathcal{O}(\log k)$ time.

$$d_L = D[o(u) + q_L] - D[o(u)], \tag{2.1}$$

$$d_R = D[c(u) - q_R] - D[c(u)]. \tag{2.2}$$

To find the correct exponent, we query the LCA of two nodes in the tree which indicate the end of $Q_F[o(u)]$ and $Q_F[c(u)]$; this operation takes $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing of the forest F [Harel and Tarjan, 1984, Bender et al., 2005]. Once we have the LCA v^*, we compute the exponent in $\mathcal{O}(1)$ time using the D array. □

Once we compute the set of contexts of each node in forests F and G, we next find which contexts have $2k$-synchronized occurrences that we should reduce. We can use 2-D range queries to find such synchronized occurrences since we need to check both if the opening parentheses indices and closing parenthesis indices are within $2k$ of each other.

Definition 2.7.7. Given a set of points S and a query rectangle Q in the plane, *orthogonal range successor* (ORS) is the problem of finding the point with smallest x-coordinate in $S \cap Q$. Given a quadruple $(u, q_L, q_R, e) \in \mathcal{C}(F)$ with context $C = (\mathsf{P}(F)[o(u)\,..\,o(u) + q_L), \mathsf{P}(F)(c(u) - q_R\,..\,c(u)])$ we let $\mathsf{ORS}_C(u)$ denote an ORS query which returns a node $v \in G$ such that $(v, q_L, q_R, e') \in \mathcal{C}(G), (\mathsf{P}(G)[o(v)\,..\,o(v)+q_L), \mathsf{P}(G)(c(v)-q_R\,..\,c(v)]) = C, |o(u)-o(v)| \leq 2k, |c(u)-c(v)| \leq 2k$ and no node $v' \in G$ exists satisfying these requirements with $o(v') < o(v)$. If no node satisfies the query, we return $(0, 0, 0, 0)$.

Theorem 2.7.8 (Linear time ORS queries [Gao et al., 2020])**.** *Given a set of n points in the plane, a data structure can be computed in time $\mathcal{O}(n\sqrt{\log n})$ to answer* ORS *queries in $\mathcal{O}(\lg \lg n)$ time.*

Definition 2.7.9. Given a string s, a *fingerprint hash* of s is a function $h_s : s \to [|s|^3]$ such that $h_s(s[i_1 \,..\, j_1]) = h_s(s[i_2 \,..\, j_2])$ if and only if $s[i_1 \,..\, j_1] = s[i_2 \,..\, j_2]$.

Theorem 2.7.10 (Linear time fingerprint hash [Gawrychowski, 2011])**.** *Given a string s with $n = |s|$, after $\mathcal{O}(n)$ preprocessing time, fingerprint hash values can be computed in constant time for substrings of s.*

Lemma 2.7.11. *Given forests F, G and quadruple $(u, q_L, q_R, e) \in \mathcal{C}(F)$ with context $C = (\mathsf{P}(F)[o(u) \,..\, o(u) + qL), \mathsf{P}(F)(c(u) - q_R \,..\, c(u)])$, $\mathsf{ORS}_{\mathcal{C}}(u) \in G$ can be computed in $\mathcal{O}(\lg \lg n)$ time with $\mathcal{O}(n\sqrt{\lg n})$ preprocessing.*

Proof. We will build a separate ORS data structure $\mathcal{D}_C$ for each context $C = (C_L, C_R)$ that corresponds to a quadruple of $\mathcal{C}(F)$. Any node $v \in G$ will be contained in data structure $\mathcal{D}_C$ if $(v, q_L, q_R, e) \in \mathcal{C}(G)$ corresponds has context C. To determine which point belongs to which data structure, we first preprocess $\mathsf{P}(F), \mathsf{P}(G)$ and compute a fingerprint hash $h_{F,G}$ in $\mathcal{O}(n)$ time by Theorem 2.7.10. Then, given a quadruple $(v, q_L, q_R, e) \in \mathcal{C}(G)$, determining which data structure v belongs to can be done in constant time using $h_{F,G}$. Since there is only one quadruple per node of F and G in $\mathcal{C}(F), \mathcal{C}(G)$ and n nodes total per forest, we can construct all data structures of contexts in $\mathcal{C}(F)$ in time $\mathcal{O}(n\sqrt{\lg n})$ according to Theorem 2.7.8. Input queries to these data structures will be the $2k \times 2k$ rectangle surrounding a point $(o(u), c(u))$ such that $(u, q_L, q_R, e) \in \mathcal{C}(F)$ and outputs will be quadruples $(v, q_L, q_R, e') \in \mathcal{C}(G)$ where $(o(v), c(v))$ is the orthogonal range successor of $(o(u), c(u))$ in $\mathcal{D}_C$. Note that node v will be the one with smallest index $o(v)$ out of all potential candidates satisfying the query since ORS queries return the point with smallest

x-coordinate as per Definition 2.7.7. Again, by Theorem 2.7.8, these queries can be answered in time $\mathcal{O}(\lg \lg n)$. □

Algorithm 6: VertPeriods(F, G)

```
1  i ← −1;
2  S ← ∅;
3  C(F) ← SortByStartingIndex(C(F));
4  for (u_F, q_L^F, q_R^F, e_F) ∈ C(F) do
5      if o(u_F) > i then
6          C ← (P(F)[o(u_F) .. o(u_F) + q_L^F), P(F)(c(u_F) − q_R^F .. c(u_F)]);
7          (v, q_L^G, q_R^G, e_G) ← ORS_C(u);
8          if e_G ≠ 0 then
9              e' ← min(e_F, e_G);
10             S ← S ∪ (u_F, u_G, q_L^F, q_R^F, e');
11             i ← o(u) + e'q_L^F − 12k;
12 return S;
```

Algorithm 6 and Algorithm 7 identify and output forests F', G' with reduced vertical periodicity. Algorithm 6 simply iterates through all nodes u of forest F that are the beginning of a vertical period with power at least $16k$. Then using ORS queries, we find any nodes v in G with a matching context to that of u and add the $2k$-synchronized occurrence to set S to be reduced in Algorithm 7. Algorithm 7 is fairly straightforward and simply outputs $\mathsf{P}(F')$ and $\mathsf{P}(G')$ by copying all characters of $\mathsf{P}(F), \mathsf{P}(G)$ and skipping all but $16k$ repetitions of any $2k$-synchronized vertical periods of set S. We now show that these algorithms do correctly identify vertical periods and that reducing vertical period powers does not change tree edit distance. Most of the following proofs mimic the same structures as Section 2.6 with more details since each context $C^e = (C_L, C_R)^e$ has a left C_L^e and right C_R^e part we must consider rather than a single substring Q^e.

Lemma 2.7.12. *Given forests F, G and $S = \mathsf{VertPeriods}(F, G)$, then for any $(u, v, q_L, q_R, e) \in S$ there is $2k$-synchronized occurrences of C^e at nodes u in F, v in G where $C = (\mathsf{P}(F)[o(u) \,..\, o(u)+$*

$q_L), \mathsf{P}(F)(c(u) - q_R \,..\, c(u)])$ *and* $e \geq 16k$.

Proof. The proof of this lemma is fairly straightforward from the steps of Algorithm 6. First, note that for any quintuple (u, v, q_L, q_R, e') added to S in step 10, there must be some quadruple $(u, q_L, q_R, e_F) \in \mathcal{C}(F)$ where by definition of $\mathcal{C}(F)$, $e_F \geq 16k$ and C^{e_F} occurs at node u where context $C = (\mathsf{P}(F)[o(u) \,..\, o(u) + q_L), \mathsf{P}(F)(c(u) - q_R \,..\, c(u)])$. Furthermore, by definition of $\mathsf{ORS}_C(u)$ any quadruple (v, q_L^G, q_R^G, e_G) returned in step 7 must also have an occurrence of C^{e_G} at node v where $|o(u) - o(v)| \leq 2k, |c(u) - c(v)| \leq 2k$. Moreover, (v, q_L, q_R, e_G) must be in $\mathcal{C}(G)$, and so, $e_G \geq 16k$. So, since in step 9 we set $e' = \min(e_F, e_G)$, we have that $e' \geq 16k$ and there is a $2k$-synchronized occurrence of C^e at nodes u in F and v in G. □

Algorithm 7: VertSyncReductions(F, G)

```
1  S ← VertPeriods(F, G);
2  S' ← ∅;
3  for (u_F, u_G, q_L, q_R, e) ∈ S do
4  |  S' ← S' ∪ {(o(u_F), o(u_G), q_L, e), (c(u_F) − q_R · e + 1, c(u_G) − q_R · e + 1, q_R, e)} ;
5  S' ← SortByStartingIndex(S');
6  s_F, s_G ← ε;
7  i_F, i_G ← 0;
8  for (ℓ_F, ℓ_G, q, e) ∈ S' do
9  |  s_F ← s_F · P(F)[i_F .. ℓ_F);
10 |  s_G ← s_G · P(G)[i_G .. ℓ_G);
11 |  i_F ← ℓ_F + q(e − 16k);
12 |  i_G ← ℓ_G + q(e − 16k);
13 s_F ← s_F · P(F)[i_F .. |P(F)|) ;
14 s_G ← s_G · P(G)[i_G .. |P(G)|) ;
15 return s_F, s_G;
```

Lemma 2.7.13. *Consider forests* F, G *and a context* $C = (C_L, C_R)$ *such that* $0 < |C_L|, |C_R| \leq 4k$ *and, for some integer exponent* $e \geq 10k$*, the context* C^e *has* $2k$*-synchronized occurrences in* F, G

at nodes u and v, respectively. This means that

$$\mathsf{P}(F) = \mathsf{P}(F)[0\,..\,o_F(u)) \cdot C_L^e \cdot \mathsf{P}(F)[o_F(u) + e|C_L|\,..\,c_F(u) - e|C_R|] \cdot C_R^e \cdot \mathsf{P}(F)(c_F(u)\,..\,|\mathsf{P}(F)|),$$

$$\mathsf{P}(G) = \mathsf{P}(G)[0\,..\,o_G(v)) \cdot C_L^e \cdot \mathsf{P}(G)[o_G(v) + e|C_L|\,..\,c_G(v) - e|C_R|] \cdot C_R^e \cdot \mathsf{P}(G)(c_G(v)\,..\,|\mathsf{P}(G)|).$$

We define F', G' so that the following holds for some exponent $e' \geq 10k$:

$$\mathsf{P}(F') = \mathsf{P}(F)[0\,..\,o_F(u)) \cdot C_L^{e'} \cdot \mathsf{P}(F)[o_F(u) + e|C_L|\,..\,c_F(u) - e|C_R|] \cdot C_R^{e'} \cdot \mathsf{P}(F)(c_F(u)\,..\,|\mathsf{P}(F)|),$$

$$\mathsf{P}(G') = \mathsf{P}(G)[0\,..\,o_G(v)) \cdot C_L^{e'} \cdot \mathsf{P}(G)[o_G(v) + e|C_L|\,..\,c_G(v) - e|C_R|] \cdot C_R^{e'} \cdot \mathsf{P}(G)(c_G(v)\,..\,|\mathsf{P}(G)|).$$

Then, $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$.

Proof. We assume without loss of generality that C is primitive (if C can be expressed as an integer power of a smaller context, we should consider that context instead) and denote $q_L := |C_L| \leq 4k$ and $q_R := |C_R| \leq 4k$. For $i \in [0\,..\,e)$, let u_i be the node of F with $o_F(u_i) = o_F(u) + iq_L$ (and $c_F(u_i) = c_F(u) - iq_R$) and let v_i be the node of G with $o_G(v_i) = o_G(v) + iq_L$ (and $c_G(v_i) = c_G(v) - iq_R$). Moreover, let $\mathcal{A}$ be an optimal tree alignment such that $\mathsf{ted}(F, G) = \mathsf{ted}_{\mathcal{A}}(F, G) \leq k$.

Claim 2.7.14. *There exist $i_F, i_G \in [0\,..\,5k]$ such that*

$$(o_F(u) + i_F \cdot q_L, o_G(v) + i_G \cdot q_L), (1 + c_F(u) - i_F \cdot q_R, 1 + c_G(v) - i_G \cdot q_R) \in \mathcal{A}.$$

Moreover, there exist $j_F, j_G \in [0\,..\,5k]$ such that

$$(o_F(u) + (e - j_F)q_L, o_G(v) + (e - j_G)q_L), (1 + c_F(u) - (e - j_F)q_R, 1 + c_G(v) - (e - j_G)q_R) \in \mathcal{A}.$$

Proof. Let $(x_F, x_G) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $x_F \geq o_F(u)$ and $x_G \geq o_G(v)$. By symmetry between F and G, we may assume without loss of generality that $x_F = o_F(u)$. The context C occurs at each of the nodes $u_0, \ldots, u_k$ and, since the occurrences are disjoint, the alignment $\mathcal{A}$ (of cost at most k) must match one of these occurrences perfectly. We pick the index $i_F \in [0\,..\,k]$ of one such perfectly matched occurrence and denote the node matched to u_{i_F} by w. In particular, $\mathsf{P}(F)[o_F(u_{i_F})\,..\,o_F(u_{i_F}) + q_L) \simeq_{\mathcal{A}} \mathsf{P}(G)[o_G(w)\,..\,o_G(w) + q_L)$ and $\mathsf{P}(F)(c_F(u_{i_F}) - q_R\,..\,c_F(u_{i_F})] \simeq_{\mathcal{A}} \mathsf{P}(G)(c_G(w) - q_R\,..\,c_G(w)]$. Since $(x_F, x_G) \in \mathcal{A}$, we must have $o_G(w) \geq o_G(v)$ by the non-crossing property of $\mathcal{A}$. At the same time, $o_G(w) \leq o_F(u_{i_F}) + 2k \leq o_F(u) + kq_L + 2k \leq o_G(v) + kq_L + 4k \leq o_G(v) + 5kq_L$. Similarly, $c_G(w) \geq c_G(v) - 5kq_R$, which also implies $c_G(w) \leq c_G(v)$.

Our next goal is to show that $w = v_{i_G}$ for some $i_G \in [0\,..\,5k]$. For a proof by contradiction, suppose that $o_G(v_i) < o_G(w) < o_G(v_{i+1})$ for some $i \in [0\,..\,5k)$. Due to $c_G(w) > c_G(v) - 5kq_R$, this also implies that $c_G(v_i) > c_G(w) > c_G(v_{i+1})$, i.e., that w is a node on the path between v_i and v_{i+1}. Suppose that the length of this path is ℓ and the node w is at distance ℓ' from v_i. Hence, $\mathsf{P}(G)[o_G(v_i)\,..\,o_G(w))$ has ℓ' unmatched opening parentheses out of the ℓ unmatched opening parentheses in C_L. Moreover, $\mathsf{P}(G)[o_G(v_i)\,..\,o_G(w)) \cdot \mathsf{P}(G)[o_G(w)\,..\,o_G(v_{i+1})) = C_L = \mathsf{P}(G)[o_G(w)\,..\,o_G(v_{i+1})) \cdot \mathsf{P}(G)[o_G(v_i)\,..\,o_G(w))$, and thus there is a primitive string Q_L such that $\mathsf{P}(G)[o_G(w)\,..\,o_G(v_{i+1}))$ and $\mathsf{P}(G)[o_G(v_i)\,..\,o_G(w))$ are both powers of Q_L. The number of unmatched opening parentheses is Q_L must be a common divisor of ℓ and ℓ', i.e., C_L can be expressed as a string power with exponent $\ell/\gcd(\ell, \ell')$. A symmetric argument shows that C_R can be expressed as a string power with exponent $\ell/\gcd(\ell, \ell')$. Overall, we conclude that C can be expressed as a context power with exponent $\ell/\gcd(\ell, \ell')$, contradicting the primitivity of C. Hence, $w = v_{i_G}$ for some $i_G \in [0\,..\,5k]$ holds as claimed and, in particular, $o_G(w) = o_G(v) + i_G q_L$ and

$c_G(w) = c_G(v) - i_G q_R$.

The proof of the second part of the claim is analogous. □

Observe that $\mathsf{P}(F)[o_F(u) + i_F q_L \,..\, o_F(u) + (e - j_F)q_L) = C_L^{d_F}$ and $\mathsf{P}(F)(c_F(u) - (e - j_F)q_R \,..\, c_F(u) - i_F q_L] = C_R^{d_F}$ for $d_F := e - j_F - i_F$. Symmetrically, $\mathsf{P}(G)[o_G(v) + i_G q_L \,..\, o_G(v) + (e - j_G)q_L) = C_L^{d_G}$ and $\mathsf{P}(G)(c_G(v) - (e - j_G)q_R \,..\, c_G(v) - i_G q_L] = C_R^{d_G}$ for $d_G := e - j_G - i_G$. We denote $d = \min(d_F, d_G)$, observe that $d \geq e - 10k \geq 0$, and construct a tree alignment $\mathcal{A}'$ so that it:

- aligns $\mathsf{P}(F)[0 \,..\, o_F(u) + i_F q_L)$ with $\mathsf{P}(G)[0 \,..\, o_G(v) + i_G q_L)$ in the same way as $\mathcal{A}$ does;
- matches $\mathsf{P}(F)[o_F(u) + i_F q_L \,..\, o_F(u) + (i_F + d)q_L) = C_L^d$ with $\mathsf{P}(G)[o_G(v) + i_G q_L \,..\, o_G(v) + (i_G + d)q_L) = C_L^d$;
- deletes $\mathsf{P}(F)[o_F(u) + (i_F + d)q_L \,..\, o_F(u) + (e - j_F)q_L) = C_L^{d_F - d}$ and $\mathsf{P}(G)[o_G(v) + (i_G + d)q_L \,..\, o_G(v) + (e - j_G)q_L) = C_L^{d_G - d}$;
- aligns $\mathsf{P}(F)[o_F(u) + (e - j_F)q_L \,..\, c_F(u) - (e - j_F)q_R]$ with $\mathsf{P}(G)[o_G(v) + (e - j_G)q_L \,..\, c_G(v) - (e - j_G)q_R]$ in the same way as $\mathcal{A}$ does;
- deletes $\mathsf{P}(F)(c_F(u) - (e - j_F)q_R \,..\, c_F(u) - (i_F + d)q_R] = C_R^{d_F - d}$ and $\mathsf{P}(G)(c_G(v) - (e - j_G)q_R \,..\, c_G(v) - (i_G + d)q_R] = C_R^{d_G - d}$;
- matches $\mathsf{P}(F)(c_F(u) - (i_F + d)q_R \,..\, c_F(u) - i_F q_R] = C_R^d$ with $\mathsf{P}(G)(c_G(v) - (i_G + d)q_R \,..\, c_G(v) - i_G q_R] = C_R^d$;
- aligns $\mathsf{P}(F)(c_F(u) - i_F q_R \,..\, |\mathsf{P}(F)|)$ with $\mathsf{P}(G)(c_G(v) - i_G q_R \,..\, |\mathsf{P}(G)|)$ in the same way as $\mathcal{A}$ does.

This definition makes it clear that $\mathcal{A}'$ is an edit-distance (string) alignment. In terms of the forests F and G, the alignment $\mathcal{A}'$ can be interpreted so that it:

- perfectly matches the occurrences of C at nodes $u_{i_F}, \ldots, u_{i_F+d-1}$ to the occurrences of C at nodes $v_{i_G}, \ldots v_{i_G+d-1}$, respectively;
- deletes the occurrences of C at nodes $u_{i_F+d}, \ldots u_{i_F+d_F-1}$ and $v_{i_G+d}, \ldots v_{i_G+d_G-1}$;
- handles the remaining parts of F and G in the same way as $\mathcal{A}$ does.

This interpretation makes it clear that $\mathcal{A}'$ is a tree alignment. Moreover, $\mathcal{A}$ and $\mathcal{A}'$ only differ in how they align C^{d_F} to C^{d_G}, and $\mathcal{A}'$ provides an optimum alignments of these contexts.

Now, if the exponent e of the context C^e is modified to $e' \geq 10k$, we can interpret this as modifying the exponent d of the contexts C^d matched perfectly by $\mathcal{A}'$ to $d' = d + e' - e \geq 0$. Thus, $\mathcal{A}'$ can be trivially adapted without modifying its cost, and hence $\mathsf{ted}(F', G') \leq \mathsf{ted}_{\mathcal{A}'}(F, G) = \mathsf{ted}(F, G)$. The converse inequality follows by symmetry between (F, G) and (F', G'). □

Now we've shown that we correctly identify $2k$-synchronized occurrences of vertical periodicity and that reducing such occurrences to an exponent of at least $10k$ does not change the tree edit distance. Before giving the main proposition of the section, we show that any two contexts $C = (C_L, C_R)$ we add to our set S in Algorithm 6 overlap in at most $16k$ characters in each C_L and C_R. Thus, when reducing the exponents of repeated contexts to an exponent of $16k$ in Algorithm 7, we do not have to worry about attempting to remove the same periodic regions twice.

Claim 2.7.15. *Given two distinct quintuples* $\gamma = (u_F, u_G, q_L, q_R, e), \gamma' = (u'_F, u'_G, q'_L, q'_R, e') \in S$, *it is true that*

- $|[o(u_F) \,..\, o(u_F) + q_L e) \cap [o(u'_F) \,..\, o(u'_F) + q'_L e')| \leq 12k$
- $|[o(u_F) \,..\, o(u_F) + q_L e) \cap (c(u'_F) - q'_R e' \,..\, c(u'_F)]| \leq 12k$
- $|(c(u_F) - q_R e \,..\, c(u_F)] \cap (c(u'_F) - q'_R e' \,..\, c(u'_F)]| \leq 12k$
- $|[o(u_G) \,..\, o(u_G) + q_L e) \cap [o(u'_G) \,..\, o(u'_G) + q'_L e')| \leq 16k$

- $|[o(u_G)\,..\,o(u_G)+q_Le) \cap (c(u'_G)-q'_Re'\,..\,c(u'_G)]| \leq 16k$
- $|(c(u_G)-q_Re\,..\,c(u_G)] \cap (c(u'_G)-q'_Re'\,..\,c(u'_G)]| \leq 16k$

Proof. If the intersection of any pairs of ranges is empty in any of the six properties of the claim, we are done since $0 \leq 12k$. Therefore, we assume that all intersections are non-empty. In this proof, when discussing contexts $C = (C_L, C_R)$ we call C_L the *left part* of the context and C_R the *right part*.

We start by proving the first property of the claim. Conceptually, this property is showing that no pair γ, γ' of $2k$-synchronized occurrences of vertical periodicity added to our set S overlap very much in forest F in their left parts of their contexts. For this properties, without loss of generality we assume $o(u_F) < o(u'_F)$ and so $|o(u_F)\,..\,o(u_F)+q_Le) \cap [o(u'_F)\,..\,o(u'_F)+q'_Le')| = (o(u_F)+q_Le) - o(u'_F)$. Note that Algorithm 6 considers nodes in left-to-right order according to their opening parenthesis index due to the sort in step 3. Therefore, γ is added to S before γ', and then variable i is set to $o(u_F)+eq_L-12k$. Therefore, by step 5 of Algorithm 6, $o(u'_F) > o(u_F)+eq_L-12k$ and $(o(u_F)+q_Le)-o(u'_F) < (o(u_F)+q_Le)-(o(u_F)+q_Le-12k) = 12k$.

We now move on to the second property, which conceptually is showing no pair γ, γ' of $2k$-synchronized occurrences of vertical periodicity in set S overlap between the left part of γ and the right part of γ'. Note that by Definition 2.7.5, $\mathsf{P}(F)[o(u_F)\,..\,o(u_F)+q_Le)$ and $\mathsf{P}(F)[c(u'_F)-q'_Re'\,..\,c(u_F)]$ are both periodic and part of some runs R and R', respectively. If $R \neq R'$, then already we have our desired property by Corollary 2.6.5. If $R = R'$, then we reach a contradiction since the string period of R cannot contain both unmatched opening parentheses and unmatched closing parentheses as required to be part of both a left context ($\mathsf{P}(F)[o(u_F)\,..\,o(u_F)+q_Le)$) and a right context ($\mathsf{P}(F)[c(u'_F)-q'_Re'\,..\,c(u_F)]$).

For the third property, now we must show that γ and γ' do not overlap in the right parts of their contexts. Unfortunately, this is not as straightforward as the first case since step 5 and step 11 of Algorithm 6 only directly handle the left parts of contexts. Without loss of generality, we assume $c(u_F) > c(u'_F)$.

Case i) The runs containing $\mathsf{P}(F)(c(u_F) - q_R e \,..\, c(u_F)]$ and $\mathsf{P}(F)(c(u'_F) - q'_R e' \,..\, c(u_F)]$ are different. In this case we are done by Corollary 2.6.5 as in the proof of the second property.

Case ii) The runs containing $\mathsf{P}(F)(c(u_F) - q_R e \,..\, c(u_F)]$ and $\mathsf{P}(F)(c(u'_F) - q'_R e' \,..\, c(u_F)]$ are the same, denoted R^c_F, but the runs containing $\mathsf{P}(F)[o(u_F) \,..\, o(u_F) + q_L e)$ and $\mathsf{P}(F)[o(u'_F) \,..\, o(u'_F) + q'_L e')$ are different, denoted R^o_F and $R^{o'}_F$, respectively. For contradiction let us assume the intersection $|(c(u_F) - q_R e \,..\, c(u_F)] \cap (c(u'_F) - q'_R e' \,..\, c(u_F)]|$ is greater than $8k$. Since $q_R, q'_R \leq 4k$ by Definition 2.7.5, then at least two repetitions of the period string of R^C_F is contained in the $8k$ intersection. Therefore, there must be two repetitions of each period string in the overlap of $R^o_F, R^{o'}_F$ as well. However, by Lemma 2.6.4, this is a contradiction for whichever run has the larger period.

Case iii) The runs containing $\mathsf{P}(F)(c(u_F) - q_R e \,..\, c(u_F)]$ and $\mathsf{P}(F)(c(u'_F) - q'_R e' \,..\, c(u_F)]$ are the same, denoted R^c_F, and the runs containing $\mathsf{P}(F)[o(u_F) \,..\, o(u_F) + q_L e)$ and $\mathsf{P}(F)[o(u'_F) \,..\, o(u'_F) + q'_L e')$ are the same, denoted R^o_F. Furthermore, we consider the runs of corresponding contexts occurring at nodes $u_G, u'_G \in G$. Let $R^c_G, R^{c'}_G$ be the runs containing $\mathsf{P}(G)(c(u_G) - q_R e \,..\, c(u_G)]$ and $\mathsf{P}(G)(c(u'_G) - q'_R e' \,..\, c(u_G)]$, respectively, and let $R^o_G, R^{o'}_G$ be the runs containing $\mathsf{P}(G)[o(u_G) \,..\, o(u_G) + q_L e)$ and $\mathsf{P}(G)[o(u'_G) \,..\, o(u'_G) + q'_L e')$, respectively. We assume without loss of generality that $c(u'_F) < c(u_F)$. We split this case into 3 sub-cases.

Case iiia) The runs $R^c_G, R^{c'}_G$ are different runs. In this case, both $R^c_G, R^{c'}_G$ have at most $8k$ characters in their intersection by Corollary 2.6.5, and so, $c(u'_G) \leq c(u_G) - eq_R + 8k$. Hence $c(u'_F) \leq c(u'_G) + 2k \leq (c(u_G) - eq_R + 8k) + 2k \leq c(u_F) + 2k - eq_R + 10k \leq c(u_F) - eq_R + 12k$,

which implies that $c(u'_F) - (c(u_F) - eq_R) \leq 12k$.

Case iiib) The runs $R^c_G, R^{c'}_G$ are the same runs, but $R^o_G, R^{o'}_G$ are different runs. This sub-case is akin to case ii) by applying the observation that $c(u'_F) \leq o(u'_G) + 2k$ and $c(u_F) \leq o(u_F) + 2k$ since (u_F, u_G) and (u'_F, u'_G) denote $2k$-synchronized occurrences. For contradiction, let us assume the intersection of $|(c(u_F) - q_R e \,..\, c(u_F)] \cap (c(u'_F) - q'_R e' \,..\, c(u_F)]|$ is at least $12k$. Then, $|(c(u_G) - q_R e \,..\, c(u_G)] \cap (c(u'_G) - q'_R e' \,..\, c(u_G)]| \geq |(c(u_F) - q_R e \,..\, c(u_F)] \cap (c(u'_F) - q'_R e' \,..\, c(u_F)]| - 4k \geq 8k$. Since $q_R, q'_R \leq 4k$ by Definition 2.7.5, there must be two repetitions of each period string in the overlap of $R^o_G, R^{o'}_G$ as well. However, by Lemma 2.6.4, this is a contradiction for whichever run has the larger period.

Case iiic) The runs $R^c_G, R^{c'}_G$ are the same runs, and $R^o_G, R^{o'}_G$ are the same runs. In this case, Algorithm 6 would have found maximal $2k$-synchronized occurrences represented by $\gamma'' = (u_F, u_G, q_L, q_R, e'')$ which cannot be extended (i.e. some run ends which contains part of these synchronized occurrences). If $o(u'_F) > o(u_F) + e''q_L - 12k$, then this would lead to a contradiction since e' is at least $16k \geq 12k$ and therefore, γ'' would not be maximal. Hence, this entire sub-case cannot occur.

Finally, we discuss the fourth through sixth properties. Observe that since γ and γ' represent $2k$-synchronized occurrences, the intersections of any two ranges considered in these properties cannot be more than $4k$ more than their corresponding intersections in F. Therefore, by properties 1-3, we have properties 4-6. □

Proposition 2.7.16 (Avoiding Vertical k-Periodicity)**.** *There exists an $\mathcal{O}(n \log n)$-time algorithm that, given labeled forests F, G of total size n and an integer $k \in \mathbb{Z}_+$, produces forests labeled forests F', G' that satisfy $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$ and, moreover, avoid both synchronized*

horizontal k-periodicity and synchronized vertical k-periodicity.

Proof. We consider Algorithm 7 to prove this proposition. First we find $S = \mathsf{VertPeriods}(\mathsf{F}, \mathsf{G})$ in step 1; by Lemma 2.7.12, any $(u, v, q_L, q_R, e) \in S$ represents a $2k$-synchronized occurrence of C^e where $C_L = \mathsf{P}(F)[o(u)\,..\,o(u) + q_L), C_R = \mathsf{P}(G)(c(u) - q_R\,..\,c(u)], C = (C_L, C_R)$ at nodes u and v. In step 4, we identify the start of the specific periodic substrings we want to reduce the exponent of, and construct a set S' with quadruples containing these starting indices, the length of the periods, and the exponent to be reduced. In step 5 we sort these quadruples by their indices and iterate through them in order from left to right. For each quadruple (ℓ_F, ℓ_G, q, e), in step 11 and step 12, we move the start of these synchronized occurrences ahead by $q(e - 16k)$. Clearly, this is equivalent to shortening the power e of the synchronized occurrences of C to a power of $16k$ exactly since

$$\frac{(\ell_F + qe) - (\ell_F + q(e - 16k))}{q} = 16k.$$

We also note that by Claim 2.7.15, no two quadruples of S' will represent overlapping synchronized occurrences with more than $16k$ overlapping characters. Therefore, since $C_L \cdot C_R$ is a balanced string, we can construct forests F', G' such that $\mathsf{P}(F') = s_F, \mathsf{P}(G') = s_G$ and by Lemma 2.7.13, $\mathsf{ted}_{\le k}(F, G) = \mathsf{ted}_{\le k}(F', G')$.

Now, we want to show that F', G' avoid vertical k-periodicity. Assume for contradiction that there is some $2k$-synchronized occurrences of C^e in F', G' with $e \ge 18k$. First we consider the case when the $2k$-synchronized occurrences already were present in F', G' at some nodes $u \in F, v \in G$. If node u was directly considered in an iteration of Algorithm 6 and node v with quadruple (v, q_L, q_R, e_v) was returned by ORS query on u, then we would have added these occurrences to S and reduced exponent e to at most $14k$. It may be the case however that an

earlier node v' with quadruple $(v', q_L, q_R, e_{v'})$ where $o(v') < o(v)$ was returned by the ORS query as per Definition 2.7.7. However, since the ORS checks for queries within a range of $4k$ in the x-coordinate, then $o(v) - o(v') \leq 4k$. Since $o(v) - o(v') \leq 4k$ and $e_v, e_{v'} \geq 16k$, by Corollary 2.6.5 we know that $o(v)$ and $o(v')$ must be part of the same maximal run. Since the contexts starting at $o(v)$ and $o(v')$ overlap in at least $16 - 4k > 8k$, then at least two full periods are in the overlap since $q_L \leq 4k$. Also, $c(v)$ and $c(v')$ must have two full periods in the overlap of the contexts which occur at v and v'. As a result, $c(v)$ and $c(v')$ must also be in the same run, otherwise Lemma 2.6.4 is violated. Thus, $e_{v'} \geq e_v$, and so a quintuple (u, v', q_L, q_R, e') will have been added to S with $e' \geq e$. In other words, the occurrence of C^e at u would have been avoided if u was considered in an iteration of Algorithm 6.

Now if u is not considered in an iteration of Algorithm 6, then there must be some quintuple (x, z, q_L, q_R, e') added to S with $o(x) < o(u)$ and $o(x) + e'q_L - 12k \geq o(u)$ according to step 5 and step 11. By Corollary 2.6.5, $o(u)$ and $o(x)$ must be in the same run. Furthermore, since $(o(x) + e'q_L) - o(u) \geq 12k \geq 8k$ and we only consider periods of length at most $4k$, two full repetitions of the period overlap in F between the $2k$-synchronized occurrences considered at x, z and u, v. Therefore, $c(u)$ and $c(x)$ must be part of the same run, otherwise Lemma 2.6.4 is violated. Similarly in forest G, since we know that u, v and x, z are $2k$-synchronized then $(o(z)+e'q_L)-o(v) \geq 12k-4k = 8k$ and so two full repetitions of the period overlap in G as well . Thus, $o(v)$ and $o(x)$ are part of the same run and $c(v)$ and $c(x)$ are part of the same run as well. So, the $2k$-synchronized occurrences at x, z will either fully overlap that of the $2k$-synchronized occurrences at u, v, or stop at $o(u) + eq_L$ or $o(v) + eq_L$ if either of these indices are the end of the vertical period. In either case the exponent of the $2k$-synchronized occurrences at u, v will be reduced to less than $16k$.

We have shown that if a $2k$-synchronized C^{18k} occurs in F, G, we will reduce it. However, it is possible that such an occurrence comes as a result of other reductions. In this case, by Corollary 2.6.5, such a synchronized occurrence in F', G' can overlap at most two reduced vertical periods C_1^{18k}, C_2^{18k} since both contexts had powers of at least $18k$ in F, G. Since the overlap with each context is at most $16k$ and we leave $16k$ repetitions of C_1 and C_2 in F', G', any context C^{18k} in F', G' must have also occurred in F, G. As shown previously, C^{18k} must have been reduced to C^{16k}, which is a contradiction. Therefore, all $2k$-synchronized occurrences of vertical periods have been reduced in F', G' to a power of $16k$. Note that by the same argument, no new horizontal periodicity can occur in F', G' as well.

Algorithm 6 takes $\mathcal{O}(n \log n)$ time to compute S by Lemmas 2.7.6 and 2.7.11. We add one quintuple to S per node in F, so $|S| \leq n$. Therefore, Algorithm 7 takes time $\mathcal{O}(n \log n)$ to compute forests F', G'. □

2.8 Full Periodicity Reduction

Lemma 2.8.1. *Let λ be a joint labeling of forests F, G resulting in labeled forests F, G, and let $\hat{\lambda} = \mathsf{C}(\mathsf{L}(\lambda, 8k), 2k)$ for some $k \in \mathbb{Z}_+$. If F, G avoid both synchronized horizontal k-periodicity and synchronized vertical k-periodicity, then $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})$ avoid $2k$-synchronized $(22k+2)$-powers with root at most $4k$.*

Proof. For a proof by contradiction, suppose that there is a string $\hat{Q}$ of length $\hat{q} \in (0\,..\,4k]$ such that $\hat{Q}^{22k+2} = \mathsf{P}_{\hat{\lambda}}(\mathsf{F})[x\,..\,x + (22k+2)\hat{q}) = \mathsf{P}_{\hat{\lambda}}(\mathsf{G})[y\,..\,y + (22k+2)\hat{q})$ for some positions x, y with $|x - y| \leq 2k$. Note that $\hat{\lambda}$ is a refinement of $\mathsf{L}(\lambda, 8k)$, which, in turn, is a refinement of $\lambda' := \mathsf{L}(\lambda, 4k)$. In particular, $\mathsf{P}_{\lambda'}(\mathsf{F})[x\,..\,x + (22k+2)\hat{q}) = \mathsf{P}_{\lambda'}(\mathsf{G})[y\,..\,y + (22k+2)\hat{q})$ is a string

with period $\hat{q}$. Let q be the shortest period of this string and let Q be the underlying string period.

If Q contains the same number of opening and closing parentheses, then, since Q^2 occurs in a balanced string $\mathsf{P}_{\lambda'}(\mathsf{F})$, we conclude that some cyclic rotation of Q is balanced, and therefore F, G do not avoid synchronized horizontal k-periodicity (because λ' is a refinement of λ). Thus, by symmetry, we may assume without loss of generality that Q contains more opening than closing parentheses. Suppose that $Q[\delta]$ is the leftmost unmatched opening parenthesis within Q and define nodes $u_0, \ldots, u_{22k+1} \in V_{\mathsf{F}}$ such that $o_{\mathsf{F}}(u_i) = x + \delta + iq$ and nodes $v_0, \ldots, v_{22k+1} \in V_{\mathsf{G}}$ such that $o_{\mathsf{G}}(v_i) = y + \delta + iq$. Observe that $c_{\mathsf{F}}(u_0) > \cdots > c_{\mathsf{F}}(u_{22k+1}) > o_{\mathsf{F}}(u_{22k+1})$ and $c_{\mathsf{G}}(v_0) > \cdots > c_{\mathsf{G}}(v_{22k+1}) > o_{\mathsf{G}}(v_{22k+1})$.

Next, we prove two claims regarding $2k$-compatibility with respect to $\mathsf{L}(\lambda, 8k)$, which we simply refer to as compatibility in the remainder of the proof.

Claim 2.8.2. *If there exists $i \in [0\,..\,22k]$ and a node $u' \in V_{\mathsf{F}}$ with $o_{\mathsf{F}}(u') \in [o_{\mathsf{F}}(u_0)\,..\,o_{\mathsf{F}}(u_{22k})]$ such that $\hat{\lambda}(u_i) = \hat{\lambda}(u')$, then $u' = u_{i'}$ for some $i' \in [0\,..\,22k]$.*

Proof. Note that $\hat{\lambda}(u_i) = \hat{\lambda}(u')$ implies $\mathsf{P}_{\lambda}(\mathsf{sub}_{<8k}(u')) = \mathsf{P}_{\lambda}(\mathsf{sub}_{<8k}(u_i))$ and $\mathsf{P}_{\lambda'}(\mathsf{sub}_{<4k}(u')) = \mathsf{P}_{\lambda'}(\mathsf{sub}_{<4k}(u_i))$. We thus have $\mathsf{P}_{\lambda'}(\mathsf{F})[o_{\mathsf{F}}(u')\,..\,o_{\mathsf{F}}(u') + q) = \mathsf{P}_{\lambda'}(\mathsf{F})[o_{\mathsf{F}}(u_i)\,..\,o_{\mathsf{F}}(u_i) + q)$ because the common size of $\mathsf{sub}_{<4k}(u')$ and $\mathsf{sub}_{<4k}(u_i)$ is at least $4k \geq q$. Since q is the shortest period of $\mathsf{P}_{\lambda'}(\mathsf{F})[o_{\mathsf{F}}(u_0)\,..\,o_{\mathsf{F}}(u_{22k}) + q)$, we conclude that $o_{\mathsf{F}}(u') - o_{\mathsf{F}}(u_0)$ is a multiple of q, and hence $u' = u_{i'}$ for some $i' \in [0\,..\,22k]$. □

Claim 2.8.3. *For all $i \in [0\,..\,22k)$, we have $c_{\mathsf{F}}(u_i) \leq c_{\mathsf{F}}(u_{i+1}) + 4k$ and $c_{\mathsf{G}}(v_i) \leq c_{\mathsf{G}}(v_{i+1}) + 4k$*

Proof. We focus on the claim regarding F (the claim regarding G is symmetric). For a proof by contradiction, suppose that $c_{\mathsf{F}}(u_i) > c_{\mathsf{F}}(u_{i+1}) + 4k$ holds for some $i \in [0\,..\,22k)$. By construction, the node u_i shares the $\hat{\lambda}$-label with its descendant whose opening parenthesis is located at position

$o_{\mathsf{F}}(u_i) + \hat{q}$. Consider the underlying path in the compatibility graph, let $u' \in V_{\mathsf{F}}$ be the last node on this path that is an ancestor of u_i (possibly $u' = u_i$), and let $u'' \in V_{\mathsf{F}}$ be the subsequent node of F on this path. By definition of compatibility, we have $|o_{\mathsf{F}}(u') - o_{\mathsf{F}}(u'')| \leq 4k$ and $|c_{\mathsf{F}}(u') - c_{\mathsf{F}}(u'')| \leq 4k$. Moreover, since u' is an ancestor of u_i, we have $o_{\mathsf{F}}(u') \leq o_{\mathsf{F}}(u_i)$ and $c_{\mathsf{F}}(u') \geq c_{\mathsf{F}}(u_i)$. We conclude the proof by deriving a contradiction for every possible location of $o_{\mathsf{F}}(u'')$.

- If $o_{\mathsf{F}}(u'') \leq o_{\mathsf{F}}(u_i)$, then either $c_{\mathsf{F}}(u'') \leq o_{\mathsf{F}}(u_i)$, which implies $c_{\mathsf{F}}(u'') \leq o_{\mathsf{F}}(u_i) < o_{\mathsf{F}}(u_{i+1}) < c_{\mathsf{F}}(u_{i+1}) < c_{\mathsf{F}}(u_i) - 4k \leq c_{\mathsf{F}}(u') - 4k \leq c_{\mathsf{F}}(u'')$ (a contradiction) or $c_{\mathsf{F}}(u'') \geq c_{\mathsf{F}}(u_i)$, which means that u'' is an ancestor of u_i and contradicts the choice of u'.
- If $o_{\mathsf{F}}(u'') \in (o_{\mathsf{F}}(u_i) \mathinner{..} o_{\mathsf{F}}(u_{i+1}))$, then a contradiction follows from Claim 2.8.2
- If $o_{\mathsf{F}}(u'') \in [o_{\mathsf{F}}(u_{i+1}) \mathinner{..} c_{\mathsf{F}}(u_{i+1})]$, then u'' is an ancestor of u_{i+1}, which means that $c_{\mathsf{F}}(u'') \leq c_{\mathsf{F}}(u_{i+1}) < c_{\mathsf{F}}(u_i) - 4k \leq c_{\mathsf{F}}(u') - 4k \leq c_{\mathsf{F}}(u'')$ (a contradiction).
- If $o_{\mathsf{F}}(u'') > c_{\mathsf{F}}(u_{i+1})$, then $o_{\mathsf{F}}(u'') > c_{\mathsf{F}}(u_{i+1}) > o_{\mathsf{F}}(u_{i+1}) + 4k > o_{\mathsf{F}}(u_i) + 4k \geq o_{\mathsf{F}}(u') + 4k \geq o_{\mathsf{F}}(u'')$, which is also a contradiction. □

The nodes $(u_i)_{i \in [0 \mathinner{..} 22k]}$ and $(v_i)_{i \in [0 \mathinner{..} 22k]}$ share the same λ'-label, so the subtrees $\mathsf{P}_\lambda(\mathsf{sub}_{<4k}(u_i))$ and $\mathsf{P}_\lambda(\mathsf{sub}_{<4k}(v_i))$ are all isomorphic. Consequently, by Claim 2.8.3, the context $C := (C_L, C_R) := (\mathsf{P}_\lambda(\mathsf{F})[o_{\mathsf{F}}(u_0) \mathinner{..} o_{\mathsf{F}}(u_1)), \mathsf{P}_\lambda(\mathsf{F})(c_{\mathsf{F}}(u_1) \mathinner{..} c_{\mathsf{F}}(u_0)])$ satisfies $0 < |C_L|, |C_R| < 4k$ and occurs at all nodes $(u_i)_{i \in [0 \mathinner{..} 22k)}$ and $(v_i)_{i \in [0 \mathinner{..} 22k)}$. Its power C^{22k} occurs in F at node u_0 and in G at node v_0.

Observe that, for every $i \in [0 \mathinner{..} 22k]$, the node v_i must be compatible with some $u' \in V_{\mathsf{F}}$. If $i \in [4k \mathinner{..} 18k]$, then $o_{\mathsf{F}}(u') \geq o_{\mathsf{G}}(v_i) - 2k \geq o_{\mathsf{F}}(u_i) - 4k \geq o_{\mathsf{F}}(u_{i-4k})$ and $o_{\mathsf{F}}(u') \leq o_{\mathsf{G}}(v_i) + 2k \leq o_{\mathsf{F}}(u_i) + 4k \leq o_{\mathsf{F}}(u_{i+4k})$, so $o_{\mathsf{F}}(u') \in [o_{\mathsf{F}}(u_{i-4k}) \mathinner{..} o_{\mathsf{F}}(u_{i+4k})]$. By Claim 2.8.2, this means that $u' = u_j$ for some $j \in [i-4k \mathinner{..} i+4k]$. In particular, $|o_{\mathsf{F}}(u_i) - o_{\mathsf{G}}(v_j)| \leq 2k$ and $|c_{\mathsf{F}}(u_i) - c_{\mathsf{G}}(v_j)| \leq 2k$. Since $(o_{\mathsf{F}}(u_t))_{t \in [0 \mathinner{..} 22k)}$ and $(o_{\mathsf{G}}(v_t))_{t \in [0 \mathinner{..} 22k)}$ form arithmetic progressions with difference $|C_L|$

and $(c_{\mathsf{F}}(u_t))_{i\in[0..22k)}$ and $(c_{\mathsf{G}}(v_i))_{t\in[0..22k)}$ form arithmetic progressions with difference $-|C_R|$, we conclude that there exists $\delta := j - i \in [-4k..4k]$ such that $|o_{\mathsf{F}}(u_t) - o_{\mathsf{G}}(v_{t+\delta})| \le 2k$ and $|c_{\mathsf{F}}(u_t) - c_{\mathsf{G}}(v_{t+\delta})| \le 2k$ hold whenever $t, t+\delta \in [0..22k]$. This means that C^{18k} has $2k$-synchronized occurrences in F and G (at nodes u_0, v_δ if $\delta \ge 0$, and at nodes $u_{-\delta}, v_0$ otherwise), contradicting the assumption that F, G avoid synchronized vertical k-periodicity. □

Proposition 2.8.4. *There exists a randomized algorithm that, given forests F, G and a threshold $k \in \mathbb{Z}_+$, produces forests F', G' and an alignment $\mathcal{A} : \mathsf{P}(F') \rightsquigarrow \mathsf{P}(G')$ such that:*

- $\mathsf{ted}_{\le k}(F, G) = \mathsf{ted}_{\le k}(F', G')$, *and*
- $|\mathcal{A} \triangle \mathcal{B}| \le 5376k^4$ *for every alignment* $\mathcal{B} \in \mathsf{TA}_k(F', G')$.

The running time is $\mathcal{O}(n \log n + k^3)$ and the algorithm is correct w.h.p.

Proof. The forests F', G' are produced by horizontal (Proposition 2.6.12) and vertical (Proposition 2.7.16) periodicity reduction; this guarantees $\mathsf{ted}_{\le k}(F, G) = \mathsf{ted}_{\le k}(F', G')$. Let F', G' be the underlying unlabeled forests and let λ' be their joint labeling. We use Lemmas 2.4.3 and 2.4.7 to construct $\hat{\lambda} = \mathsf{C}(\mathsf{L}(\lambda', 8k), 2k)$ and strings $\mathsf{P}_{\hat{\lambda}}(\mathsf{F}')$, $\mathsf{P}_{\hat{\lambda}}(\mathsf{G}')$. Note that the width of any $\mathcal{B} \in \mathsf{TA}_k(F', G')$ does not exceed $2k$ and, by Lemma 2.4.2 and Observation 2.4.8, we have $\mathsf{ed}_{\mathcal{B}}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}'), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}')) \le 2k \cdot 8k \le 16k^2$. Thus, $\mathcal{B} \in \mathsf{A}_{16k^2,2k}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}'), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}'))$. We construct $\mathcal{A}$ in $\mathcal{O}(n + k^3)$ time using Lemma 2.3.3 as an arbitrary alignment in $\mathsf{GA}_{16k^2,2k}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}'), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}'))$; if there is no such alignment, then $\mathsf{A}_{16k^2,2k}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}'), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}')) = \emptyset$ and hence $\mathsf{TA}_k(F', G') = \emptyset$. By Lemmas 2.5.3 and 2.8.1, we have $|\mathcal{A} \triangle \mathcal{B}| \le 7 \cdot 2k \cdot 16k^2 \cdot (22k + 2) \le 5376k^4$. □

2.9 Main Algorithm

2.9.1 Partial Forest Matching

Let F and G be labeled forests. We say that a set $M \subseteq V_F \times V_G$ is *non-crossing* if the set $\bigcup_{(u,v)\in M}\{(o_F(u), o_F(v)), (c_F(u), c_F(v))\} \subseteq \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$ is non-crossing. Furthermore, we say that $M \subseteq V_F \times V_G$ is a *non-crossing matching* of forests F, G if $\bigcup_{(u,v)\in M}\{(o_F(u), o_F(v)), (c_F(u), c_F(v))\}$ is a non-crossing matching of $\mathsf{P}(F), \mathsf{P}(G)$.

We say that a tree alignment $\mathcal{A} \in \mathsf{TA}(F, G)$ *aligns* $u \in V_F$ with $v \in V_G$, denoted $u \sim_{\mathcal{A}} v$, if $\mathsf{P}(F)[o_F(u)] \sim_{\mathcal{A}} \mathsf{P}(G)[o_G(v)]$ and $\mathsf{P}(F)[c_F(u)] \sim_{\mathcal{A}} \mathsf{P}(G)[c_G(v)]$. If, additionally, u and v have the same labels, we say that $\mathcal{A}$ *matches* u and v, denoted $u \simeq_{\mathcal{A}} v$. For a set $M \subseteq V_F \times V_G$, we define $\mathsf{TA}^M(F, G) = \{\mathcal{A} \in \mathsf{TA}(F, G) : u \simeq_{\mathcal{A}} v \text{ for each } (u, v) \in M\}$; note that $\mathsf{TA}^M(F, G) = \emptyset$ unless M is a non-crossing matching. Moreover, we denote $\mathsf{ted}^M(F, G) = \min_{\mathcal{A}\in\mathsf{TA}^M(F,G)} \mathsf{ted}_{\mathcal{A}}(F, G)$.

In this section, we consider the problem of computing $\mathsf{ted}^M(F, G)$ given labeled forests F, G and a non-crossing matching $M \subseteq V_F \times V_G$. The following lemma provides a reduction that restricts the heights of F, G and lets us assume that $M \subseteq L_F \times L_G$, where $L_F \subseteq V_F$ is the set of leaves of F and $L_G \subseteq V_G$ is the set of leaves of G.

Lemma 2.9.1. *There exists a linear-time algorithm that, given labeled forests F, G and a non-crossing matching $M \subseteq V_F \times V_G$, produces labeled forests F', G' and a non-crossing matching $M' \subseteq L_{F'} \times L_{G'}$ such that:*

- $\mathsf{ted}^{M'}(F', G') = \mathsf{ted}^M(F, G)$*;*
- $|F'| = |M| + |F|$*,* $|G'| = |M| + |G|$*, and* $|M'| = 2|M|$*;*
- *if the height of F' (G') is $h > 1$, then there is an $(h-1)$-node top-down path in F (respec-*

tively, G) avoiding nodes participating in M.

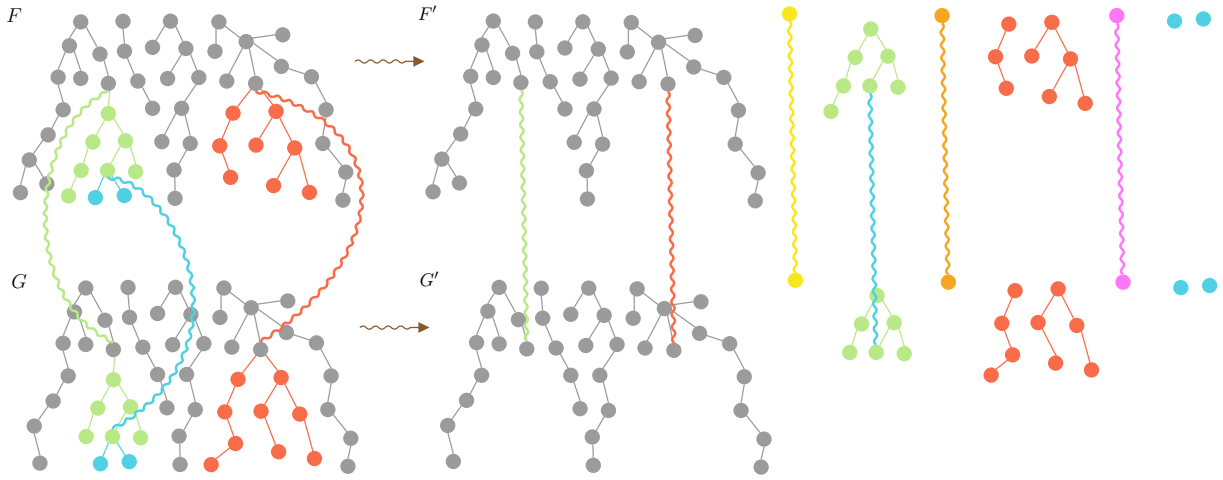


Figure 2.5: An illustration of the transformation implemented in Lemma 2.9.1. The non-crossing matching is represented using colorful wavy lines. In the input forests F and G, each node inherits its color from the nearest marked ancestor.

Proof. Let us denote $M = \{(u_i, v_i) : i \in [1\,..\,m]\}$ and *mark* all nodes participating in M. We decompose $V_F = \bigcup_{i=0}^{m} V_{F_i}$ so that $u \in V_{F_0}$ if u does not have any proper marked ancestor and, for every $i \in [1\,..\,m]$, we have $u \in V_{F_i}$ if u_i is the nearest proper marked ancestor of u. We further define forests $F_0, \ldots, F_m$ so that F_i is obtained from F by deleting all vertices in $V_F \setminus V_{F_i}$.

As far as the implementation is concerned, a top-down traversal of F allows classifying each node $u \in V_F$ into one of the classes V_{F_i}. This is because every root node belongs to V_{F_0}, every node with a marked parent u_i belongs to the class V_{F_i}, and every other node belongs to the same class as its unmarked parent. Consequently, the forest F can be decomposed into the forests $F_0, \ldots, F_m$ in linear time. Symmetrically, we decompose the forest G into analogously defined forests $G_0, \ldots, G_m$.

Due to the fact that M is non-crossing, this yields a decomposition $M = \bigcup_{i=0}^{m} M_i$, where $M_i = M \cap (V_{F_i} \times V_{G_i})$. Furthermore, if $u \sim_{\mathcal{A}} v$ holds for an alignment $\mathcal{A} \in \mathsf{TA}^M(F, G)$, then the

set $M \cup \{(u,v)\} \subseteq V_F \times V_G$ is non-crossing, and thus there exists $i \in [0\,..\,m]$ such that $u \in V_{F_i}$ and $v \in V_{G_i}$. Consequently, we have $\mathsf{ted}^M(F,G) = \sum_{i=0}^{m} \mathsf{ted}^{M_i}(F_i,G_i)$.

In the second step of the algorithm, we create nodes $\hat{u}_1,\ldots,\hat{u}_m$ and $\hat{v}_1,\ldots,\hat{v}_m$, all sharing the same label. The forest F' is obtained as a (horizontal) concatenation $F' := F_0{\cdot}\hat{u}_1{\cdot}F_1 \cdots \hat{u}_m{\cdot}F_m$, where each node $\hat{u}_i$ is interpreted as a single-node forest. Symmetrically, $G' := G_0 \cdot \hat{v}_1 \cdot G_1 \cdots \hat{v}_m \cdot G_m$. Finally, we set $M' := M \cup \{(\hat{u}_i,\hat{v}_i) : i \in [1\,..\,m]\}$. The forests F', G' and the set M' can be easily constructed in linear time.

The construction trivially yields $|F'| = |F|+|M|$, $|G'| = |G|+|M|$, and $|M'| = 2|M|$. Since the separator nodes are included in M', we further have $\mathsf{ted}^{M'}(F',G') = \sum_{i=0}^{m} \mathsf{ted}^{M_i}(F_i,G_i) = \mathsf{ted}^M(F,G)$. The separator nodes are leaves and $M_i \subseteq L_{F_i} \times L_{G_i}$ holds for each $i \in [0\,..\,m]$ (by definition of the decompositions of F and G), so we have $M' \subseteq L_{F'} \times L_{G'}$. Finally, observe that if the height of F' is $h > 1$, then one of the forests F_i has an h-node root-to-leaf path. Trimming the leaf, we obtain an $(h-1)$-node top-down path avoiding marked nodes. By construction of F_i, this path is also present in F. A symmetric reasoning lets us characterize the height of G'. □

The following reduction prunes unnecessary leaves. This is useful if $M \subseteq L_F \times L_G$ is very large.

Lemma 2.9.2. *There exists a linear-time algorithm that, given labeled forests F, G and a non-crossing matching $M \subseteq L_F \times L_G$ produces labeled forests F', G' and a non-crossing matching $M' \subseteq L_{F'} \times L_{G'}$ such that:*

- *F' and G' are obtained by deleting some leaves in F and G, respectively;*
- *$M' = M \cap (V_{F'} \times V_{G'}) = M \cap (V_F \times V_{G'}) = M \cap (V_{F'} \times V_G)$ satisfies $|M'| \leq \frac{2}{5}(|F'|+|G'|+1)$;*
- *$\mathsf{ted}^{M'}(F',G') = \mathsf{ted}^M(F,G)$.*

Proof. The algorithm identifies a subset $\hat{M}$ of *redundant leaf pairs* and constructs (F', G', M') so that $M' = M \setminus \hat{M}$ whereas the forests F' and G' are obtained be deleting all nodes participating in $\hat{M}$. Specifically, $\hat{M}$ contains all pairs $(\hat{u}, \hat{v}) \in M$ such that $\hat{u}$ and $\hat{v}$ have immediate left siblings u and v, respectively, such that $(u, v) \in M$. The construction trivially yields $\mathsf{ted}^{M'}(F', G') \leq \mathsf{ted}^{M}(F, G)$. As for the inverse inequality, we shall prove that every alignment $\mathcal{A}' \in \mathsf{ted}^{M'}(F', G')$, can be converted to an alignment $\mathcal{A} \in \mathsf{ted}^{M}(F, G)$ of the same cost. For this, we simply extend $\mathcal{A}'$ so that $\hat{u} \simeq_{\mathcal{A}} \hat{v}$ holds for all $(\hat{u}, \hat{v}) \in \hat{M}$. A simple inductive argument (reintroducing $(\hat{u}, \hat{v}) \in \hat{M}$ in the left-to-right order) shows that this does not introduce any crossings among the aligned pairs of vertices. This is because any existing pair crossing $(\hat{u}, \hat{v})$ would also cross the pair (u, v) of immediate left siblings.

Next, we note that if $(u', v') \in M'$, then one of the following cases holds:

- u' or v' is the leftmost root of F' (resp. G');
- u' or v' is the leftmost child of its parent (and the parent is not participating in M');
- u' or v' has an immediate left sibling that is not participating in M'.

The number of nodes not participating in M' is $|F'|+|G'|-2|M'|$, and each of them can be charged twice (by its right sibling and by its leftmost child). Hence, $|M'| \leq 2(1 + |F'| + |G'| - 2|M'|)$, and this simplifies to $|M'| \leq \frac{2}{5}(|F'| + |G'| + 1)$.

Finally, we observe that F', G', and M' can be easily constructed in linear time. □

The last result of this section reduces computing $\mathsf{ted}^{M}_{\leq k}(F, G)$ to computing $\mathsf{ted}_{\leq k}(F', G')$ for some F' and G'.

Lemma 2.9.3. *There exists an algorithm that, given labeled forests F, G, a non-crossing matching $M \subseteq L_F \times L_G$, and an integer $k \in \mathbb{Z}_{\geq 0}$, produces labeled forests F', G' satisfying the following*

conditions:

- $\mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}^M_{\leq k}(F, G)$*;*
- $|F'| = |F| + (k+1)|M|$ *and* $|G'| = |G| + (k+1)|M|$*;*
- *The height of* F' *(*G'*) does not exceed the height of* F *(resp.,* G*) plus one.*

The running time of the algorithm is $\mathcal{O}(|F'| + |G'|)$.

Proof. Let $M = \{(u_i, v_i) : i \in [1\,..\,m]\}$. For each $i \in [1\,..\,m]$, we attach $k+1$ new nodes, $u_{i,0}, \ldots, u_{i,k}$, as children of u_i and $k+1$ new nodes, $v_{i,0}, \ldots, v_{i,k}$, as children of v_i. The nodes $u_{i,j}$ and $v_{i,j}$ share a unique label $\$_{i,j}$ that is not present anywhere else in the constructed forests F', G'. This completes the description of the constructed forests F', G'. It is easy to see that F', G' can be constructed in time proportional to their sizes, which are $|F| + (k+1)|M|$ and $|G| + (k+1)|M|$, respectively. Moreover, the height of F' (resp. G') may increase compared to the height of F (G) by at most one (since we attach new leaves only to existing nodes).

Thus, it remains to prove that $\mathsf{ted}^M_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$. For this, let us first observe that $\mathsf{ted}(F', G') \leq \mathsf{ted}^M(F, G)$. This is because, any $\mathcal{A} \in \mathsf{TA}^M(F, G)$ can be extended to an alignment $\mathcal{A}' \in \mathsf{TA}(F', G')$ such that $u_{i,j} \simeq_{\mathcal{A}'} v_{i,j}$ holds for all $i \in [1\,..\,m]$ and $j \in [0\,..\,k]$. If any of the newly matched pairs $(u_{i,j}, v_{i,j})$ crossed some other pair of vertices aligned by $\mathcal{A}'$, already (u_i, v_i), with $u_i \simeq_{\mathcal{A}} v_i$, would cross that pair. As for the converse inequality, consider an alignment $\mathcal{A}' \in \mathsf{TA}(F', G')$ of cost at most k. By Proposition 2.4.4, we can choose $\mathcal{A}'$ so that it can be interpreted as a greedy alignment with respect to (unbounded) look-ahead labels. We shall prove that, for each $i \in [1\,..\,m]$, we have $u_i \simeq_{\mathcal{A}'} v_i$ and $u_{i,j} \simeq_{\mathcal{A}'} v_{i,j}$ for $j \in [0\,..\,k]$. As a result, a matching $\mathcal{A} \in \mathsf{TA}^M(F, G)$ of the same cost can be obtained by restricting $\mathcal{A}'$ to the nodes of F and G.

Let us fix $i \in [1\,..\,m]$. As the cost of $\mathcal{A}'$ is at most k, this alignment must match the node $u_{i,\hat{\jmath}}$ for some $\hat{\jmath} \in [0\,..\,k]$. The only vertex in G' sharing the label with $u_{i,\hat{\jmath}}$ is $v_{i,\hat{\jmath}}$, so we must have $u_{i,\hat{\jmath}} \simeq_{\mathcal{A}'} v_{i,\hat{\jmath}}$ and, in particular, $\mathsf{P}(F')[c_{F'}(u_{i,\hat{\jmath}})] \simeq_{\mathcal{A}'} \mathsf{P}(G')[c_{G'}(v_{i,\hat{\jmath}})]$. Furthermore, observe that the nodes (u_i, v_i) and the nodes $(u_{i,j}, v_{i,j})$ for $j \in [0\,..\,k]$ not only share their regular labels, but also their (unbounded) look-ahead labels. Consequently, the greedy nature of $\mathcal{A}'$ yields $\mathsf{P}(F')[c_{F'}(u_{i,\hat{\jmath}})\,..\,c_{F'}(u_i)] \simeq_{\mathcal{A}'} \mathsf{P}(G')[c_{G'}(v_{i,\hat{\jmath}})\,..\,c_{G'}(v_i)]$. Since $\mathcal{A}'$ is a tree alignment, $\mathsf{P}(F')[c_{F'}(u_i)] \simeq_{\mathcal{A}'} \mathsf{P}(G')[c_{G'}(v_i)]$ implies $\mathsf{P}(F')[o_{F'}(u_i)] \simeq_{\mathcal{A}'} \mathsf{P}(G')[o_{G'}(v_i)]$. Using the greedy nature of $\mathcal{A}'$ once again, we finally conclude that $\mathsf{P}(F')[o_{F'}(u_i)\,..\,c_{F'}(u_i)] \simeq_{\mathcal{A}'} \mathsf{P}(G')[o_{G'}(v_i)\,..\,c_{G'}(v_i)]$. Thus, $u_i \simeq_{\mathcal{A}'} v_i$ and $u_{i,j} \simeq_{\mathcal{A}'} v_{i,j}$ for $j \in [0\,..\,k]$ hold as claimed. □

We conclude with a corollary that applies Lemmas 2.9.1 to 2.9.3 one after another.

Corollary 2.9.4. *There exists an algorithm that, given labeled forests F, G, a non-crossing matching $M \subseteq V_F \times V_G$, and an integer $k \in \mathbb{Z}_{\geq 0}$, produces labeled forests F', G' such that:*

- $\mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}^M_{\leq k}(F, G)$;
- $|F'| + |G'| = \mathcal{O}(\min(|F| + |G| + k|M|, 1 + (k+1)(|F| + |G| - 2|M|)))$;
- *if the height of F' (or G') is $h > 2$, then there is an $(h-2)$-node top-down path in F (respectively, G) avoiding nodes participating in M.*

The running time of the algorithm is $\mathcal{O}(|F| + |G| + |F'| + |G'|)$.

Proof. As hinted above, the algorithm behind Corollary 2.9.4 simply chains the procedures underlying Lemmas 2.9.1 to 2.9.3. Let us denote the intermediate results by $(\hat{F}, \hat{G}, \hat{M})$ and $(\bar{F}, \bar{G}, \bar{M})$, respectively. Due to $\mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}^{\bar{M}}_{\leq k}(\bar{F}, \bar{G})$ and $\mathsf{ted}^{\bar{M}}(\bar{F}, \bar{G}) = \mathsf{ted}^{\hat{M}}(\hat{F}, \hat{G}) = \mathsf{ted}^{M}(F, G)$, we conclude that $\mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}^{M}_{\leq k}(F, G)$ holds as claimed. As for the heights, observe that if the height of F' is $h > 2$, then the heights of $\bar{F}$ are $\hat{F}$ are at least $h - 1 > 1$. By Lemma 2.9.1,

this means that F contains a top-down path of at least $h-2$ nodes none of which participate in M. A symmetric argument can be used to bound the height of G'.

It remains to bound $|F'|+|G'|$. Applying Lemma 2.9.3, Lemma 2.9.2, and Lemma 2.9.1 in subsequent steps, we obtain

$$|F'|+|G'| \leq |\bar{F}|+|\bar{G}|+2(k+1)|\bar{M}| \leq |\hat{F}|+|\hat{G}|+2(k+1)|\hat{M}| \leq |F|+|M|+|G|+|M|+4(k+1)|M| \\ = \mathcal{O}(|F|+|G|+k|M|).$$

However, we further have

$$2|\bar{M}| \leq |\bar{F}|+|\bar{G}| \leq |\bar{F}|+|\bar{G}|-10|\bar{M}|+10|\bar{M}| \leq |\bar{F}|+|\bar{G}|-10|\bar{M}|+4(|\bar{F}|+|\bar{G}|+1) \\ = 5(|\bar{F}|+|\bar{G}|-2|\bar{M}|)+4 = 5(|\hat{F}|+|\hat{G}|-2|\hat{M}|)+4 = 5(|F|+|G|-2|M|)+4,$$

and thus

$$|F'|+|G'| \leq |\bar{F}|+|\bar{G}|+2(k+1)|\bar{M}| \leq (k+2)(|\bar{F}|+|\bar{G}|) = \mathcal{O}(1+(k+1)(|F|+|G|-2|M|)).$$

Consequently, we indeed have $|F'|+|G'| = \mathcal{O}(\min(|F|+|G|+k|M|, 1+(k+1)(|F|+|G|-2|M|)))$. □

2.9.2 Tree Edit Distance of Shallow Forests

Theorem 2.9.5. *There exists a randomized algorithm that, given forests F, G of height at most $h \in \mathbb{Z}_+$ and a threshold $k \in \mathbb{Z}_+$, computes $\mathsf{ted}_{\leq k}(F,G)$ in $\mathcal{O}(n \log n + h^2 k^7 \log(hk))$ time correctly*

with high probability.

Proof. Let F and G be the underlying unlabeled forests and let λ be their joint labeling. Using Proposition 2.6.12, at the cost of $\mathcal{O}(n)$ time, we can assume without loss of generality that F, G avoid synchronized horizontal k-periodicity, i.e., there is no balanced string Q of length $|Q| \leq 4k$ such Q^{18k} has $2k$-synchronized occurrences in $\mathsf{P}_\lambda(\mathsf{F})$ and $\mathsf{P}_\lambda(\mathsf{G})$. In the next step, we construct a labeling $\hat{\lambda}$ equivalent to $\mathsf{L}(\lambda, h)$ (using Lemma 2.4.3) as well as the strings $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})$. Since $\hat{\lambda}$ is a refinement of λ, we conclude that there is no balanced string Q of length $|Q| \leq 4k$ such Q^{18k} has $2k$-synchronized occurrences in $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})$ and $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})$. Moreover, $\hat{\lambda}$ assigns distinct labels to nodes in any root-to-leaf path; thus, for any unbalanced string Q, even Q^2 cannot occur in $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})$ or $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})$. Consequently, $\mathsf{P}_{\hat{\lambda}}(\mathsf{F})$ or $\mathsf{P}_{\hat{\lambda}}(\mathsf{G})$ avoid $2k$-synchronized $18k$-powers of length at most $4k$. This lets us apply Lemma 2.5.4 to build a set M_P of size $|M_\mathsf{P}| \geq 2|F| - 15 \cdot (18k) \cdot (2hk)^2 \cdot 2k \geq 2|F| - \mathcal{O}(h^2k^4)$ such that $M_\mathsf{P} \subseteq \mathsf{M}_\mathcal{A}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$ holds for every $\mathcal{A} \in \mathsf{GA}_{2hk,2k}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$; the construction of M_P costs $\mathcal{O}(n + hk^2)$ time.

Claim 2.9.6. *Let $M = \{(u, v) \in V_F \times V_G : (o_F(u), o_G(v)) \in M_\mathsf{P}$ and $(c_F(u), c_G(v)) \in M_\mathsf{P}\}$. If $\mathsf{ted}(F, G) \leq k$, then M is a non-crossing matching of size $|M| \geq |F| - \mathcal{O}(h^2k^4)$ and, moreover, $\mathsf{ted}^M(F, G) = \mathsf{ted}(F, G)$.*

Proof. By Proposition 2.4.4, there is an optimum alignment $\mathcal{A} \in \mathsf{TA}(F, G)$ of cost $\mathsf{ted}_\mathcal{A}(F, G) = \mathsf{ted}(F, G)$ such that $\mathcal{A} \in \mathsf{GA}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$. If $\mathsf{ted}(F, G) \leq k$, then the width of $\mathcal{A}$ is at most $2k$ and, by Lemma 2.4.2, its cost $\mathsf{ed}_\mathcal{A}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$ does not exceed $2hk$. Hence, $\mathcal{A} \in \mathsf{GA}_{2hk,2k}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$, which means that $M_\mathsf{P} \subseteq \mathsf{M}_\mathcal{A}(\mathsf{P}_{\hat{\lambda}}(\mathsf{F}), \mathsf{P}_{\hat{\lambda}}(\mathsf{G}))$ and, in particular, $\mathcal{A} \in \mathsf{TA}^M(F, G)$. Consequently, $\mathsf{ted}^M(F, G) = \mathsf{ted}(F, G)$ and the number of vertices $u \in V_\mathsf{F}$ not participating in M does not exceed $2|F| - |M_\mathsf{P}| = \mathcal{O}(h^2k^4)$. □

In the light of Claim 2.9.6, we construct M and check whether it is a non-crossing matching of size $|M| \geq |F| - \mathcal{O}(h^2k^4)$; if it is not, we report that $\mathsf{ted}(F,G) > k$. The remaining task is to compute $\mathsf{ted}^M_{\leq k}(F,G)$. For this, we use Corollary 2.9.4, which reduces this problem to computing $\mathsf{ted}_{\leq k}(F',G')$, where $|F'|+|G'| = \mathcal{O}(k \cdot h^2k^4)$ because $|F|+|G|-2|M| = \mathcal{O}(h^2k^4)$; this reduction costs $\mathcal{O}(n + h^2k^5)$ time. As for the final step of determining $\mathsf{ted}_{\leq k}(F',G')$, we employ the algorithm of [Akmal and Jin, 2021], whose running time is $\mathcal{O}(h^2k^5 \cdot k^2 \cdot \log(h^2k^5)) = \mathcal{O}(h^2k^7 \log(hk))$. Overall, our procedure takes $\mathcal{O}(n + h^2k^7 \log(hk))$ time. □

2.9.3 Level Sampling

Theorem 2.1.1. *There exists a randomized algorithm that, given forests F, G of total size n and an integer $k \in \mathbb{Z}_+$, computes $\mathsf{ted}_{\leq k}(F,G)$ in $\mathcal{O}(n \log n + k^{15} \log k \log n)$ time correctly with high probability.*

Algorithm 8: `TreeEditDistance`(F, G, k)

Input: labeled forests F, G, integer threshold $k \in \mathbb{Z}_+$
Output: $\mathsf{ted}_{\leq k}(F,G)$

$(F', G', \mathcal{A}) :=$ `FullPeriodicityReduction`(F, G, k); `// Proposition 2.8.4`
$d := \infty$;
$h := 21508k^4$;
for $i := 0$ **to** $\Theta(\log(|F| + |G|))$ **do**
  Pick $r_i \in [0\,..\,h)$ uniformly at random;
  Mark nodes in F', G' at depths $\equiv r_i \pmod h$;
  $M_i := \{(u,v) \in V_{F'} \times V_{G'} : \mathsf{P}(F')[o_{F'}(u)] \simeq_{\mathcal{A}} \mathsf{P}(G')[o_{G'}(v)]$, and $\mathsf{P}(F')[c_{F'}(u)] \simeq_{\mathcal{A}} \mathsf{P}(G')[c_{G'}(v)]$, and u or v is marked$\}$;
  if $|M_i| \leq \frac{4}{h}(|F'| + |G'|)$ **and** *all marked nodes participate in* M_i **then**
    $(F_i, G_i) :=$ `PartialMatchingReduction`(F', G', M_i, k); `// Corollary 2.9.4`
    $d_i :=$ `ShallowTreeEditDistance`(F_i, G_i, k); `// Theorem 2.9.5`
    $d := \min(d, d_i)$;
return d;

Proof. As a first step, we apply Proposition 2.8.4, which produces forests F', G' and an alignment $\mathcal{A} : \mathsf{P}(F') \rightsquigarrow \mathsf{P}(G')$. Next, we pick $h := 21508k^4$ and draw $s = \Theta(\log n)$ uniformly random values $r_1, \ldots, r_s \in [0\,..\,h)$. As described below, each of these values r_i is either discarded or results in an upper bound on $\mathsf{ted}_{\leq k}(F', G')$. We mark all nodes in F' and G' whose depths are congruent to r_i modulo h. We then attempt using $\mathcal{A}$ to construct a non-crossing matching M_i in which all marked nodes participate. For this, we add to M_i every pair of nodes $(u, v) \in V_{F'} \times V_{G'}$ such that $\mathsf{P}(F')[o_{F'}(u)] \simeq_{\mathcal{A}} \mathsf{P}(G')[o_{G'}(v)]$, $\mathsf{P}(F')[c_{F'}(u)] \simeq_{\mathcal{A}} \mathsf{P}(G')[c_{G'}(v)]$, and u or v is marked. By construction, this guarantees that M_i is a non-crossing matching. However, we discard M_i if $|M_i| > \frac{4}{h}(|F'| + |G'|)$ or there exists a marked node that does not participate in M_i. If M_i is not discarded, we use Corollary 2.9.4 to build forests F_i, G_i such that $\mathsf{ted}_{\leq k}(F_i, G_i) = \mathsf{ted}^{M_i}_{\leq k}(F', G')$, and then we compute $d_i := \mathsf{ted}_{\leq k}(F_i, G_i)$ using Theorem 2.9.5. The final answer d is defined as the minimum among the values d_i computed for non-discarded matchings M_i.

Let us analyze the correctness of this approach. Proposition 2.8.4 provides the following guarantees with high probability:

- $\mathsf{ted}_{\leq k}(F, G) = \mathsf{ted}_{\leq k}(F', G')$,
- for every alignment $\mathcal{B} \in \mathsf{TA}_k(F', G')$, we have $|\mathcal{A} \triangle \mathcal{B}| \leq 5376k^4$.

Due to $\mathsf{ted}^{M_i}(F', G') \geq \mathsf{ted}(F', G')$, the reported value d must clearly satisfy $d \geq \mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F, G)$. The main challenge is to prove the converse inequality. This task is trivial if $\mathsf{ted}(F, G) > k$. Otherwise, it boils down to showing that, with high probability, there exists $i \in [1\,..\,s]$ such that $d_i \leq \mathsf{ted}_{\leq k}(F, G)$. Let us fix an optimum alignment $\mathcal{B} \in \mathsf{TA}(F', G')$. We shall first prove that $d_i \leq \mathsf{ted}_{\leq k}(F, G)$ holds conditioned on the following events:

- $|M_i| \leq \frac{4}{h}(|F'| + |G'|)$;

- the alignment $\mathcal{B}$ that does not make any edits on the marked nodes and the parentheses corresponding to the marked nodes do not contribute to $\mathcal{A} \triangle \mathcal{B}$.

Since $\mathcal{B}$ does not make any edits on the marked nodes, all the marked nodes participate in the following non-crossing matching: $\{(u, v) \in V_{F'} \times V_{G'} : u \simeq_{\mathcal{B}} v \text{ and } u \text{ or } v \text{ is marked}\}$. Furthermore, this matching is equal to M_i because the parentheses corresponding to marked nodes do not contribute to $\mathcal{A} \triangle \mathcal{B}$. Consequently, $\mathsf{ted}^{M_i}(F', G') \leq \mathsf{ted}_{\mathcal{B}}(F', G') \leq k$, and thus $d_i = \mathsf{ted}_{\leq k}(F_i, G_i) = \mathsf{ted}^{M_i}_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F', G') = \mathsf{ted}_{\leq k}(F, G)$. It remains to prove that the favorable events hold with high probability for at least one $i \in [1 \,..\, s]$. For this, we analyze the complementary bad events. For a random remainder modulo h, each individual node in $V_{F'} \cup V_{G'}$ is marked with probability $\frac{1}{h}$. Hence, in expectation, the number of marked nodes is $\frac{1}{h}(|F'| + |G'|)$. Given that each pair in M_i contains a marked node, this means that $\mathbb{E}[|M_i|] \leq \frac{1}{h}(|F'| + |G'|)$. By Markov's inequality, we conclude that $|M_i| > \frac{4}{h}(|F'| + |G'|)$ holds with probability at most $\frac{1}{4}$. Moreover, there are at most $2k$ nodes affected by the edits of $\mathcal{B}$ and at most $10752k^4$ further nodes may contribute to $\mathcal{B} \triangle \mathcal{A}$. By the union bound, at least one of these nodes is marked with probability at most $\frac{2k+10752k^4}{h} \leq \frac{1}{2}$. Overall, for each $i \in [1 \,..\, s]$, the probability of the bad events does not exceed $\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$. Due to $s = \Theta(\log n)$, the probability that the bad events hold for all $i \in [1 \,..\, s]$ does not exceed $(\frac{3}{4})^s = n^{-\Theta(1)}$. Thus, the algorithm is correct with high probability.

Let us complete the proof with the running time analysis. Applying Proposition 2.8.4 costs $\mathcal{O}(n \log n + k^3)$ time. For each $i \in [1 \,..\, s]$, the set M_i can be constructed and verified in $\mathcal{O}(n)$ time. If r_i is not discarded, then $|M_i| = \mathcal{O}(\frac{1}{h}(|F'| + |G'|)) = \mathcal{O}(\frac{n}{k^4})$ and neither F' nor G' contains an h-node top-down path avoiding nodes participating in M_i. Consequently, the application of Corollary 2.9.4 takes $\mathcal{O}(n)$ time and produces forests of size $\mathcal{O}(n)$ and height at most $h + 1 =$

$\mathcal{O}(k^4)$. The final application of Theorem 2.9.5 thus costs $\mathcal{O}(n + k^{15} \log k)$ time. Since all steps, except for the preprocessing of Proposition 2.8.4, are repeated $\mathcal{O}(\log n)$ times, the overall time complexity of our algorithm is $\mathcal{O}(n \log n + k^{15} \log k \log n)$. □

2.10 Conclusion

This paper gives an $\tilde{\mathcal{O}}(n + \mathrm{poly}(k))$-time algorithm for bounded tree edit distance, solving an open problem posed in [Akmal and Jin, 2021, Mao, 2021]. Multiple natural improvements and extensions of this result might be feasible. An immediate direction for future work is to significantly reduce the polynomial dependency on k, which currently stays at k^{15}, akin to the recent progress for Dyck edit distance [Fried et al., 2022]. Another open question is whether the weighted version of tree edit distance admits an $\tilde{\mathcal{O}}(n + \mathrm{poly}(k))$-time algorithm (assuming that the cost of each edit is at least one). To the best of our knowledge, no such algorithm is known even for weighted string edit distance.

Acknowledgment

Barna Saha and Tomasz Kociumaka were partly supported by NSF 1652303, 1909046, and HDR TRIPODS Phase II grant 2217058. MohammadTaghi Hajiaghayi and Jacob Gilbert were partly supported by NSF CCF grants 2114269 and 2218678.

Chapter 3: Weighted Edit Distance Computation: Strings, Trees and Dyck

3.1 Introduction

String edit distance and its several variants have been studied for decades since the 1960s [Levenshtein, 1965, Needleman and Wunsch, 1970, Wagner and Fischer, 1974]. Historically, most work on these problems assumed that the edit operations have unit weights in order to simplify the problem and streamline theoretical results. Till date, the fastest exact algorithm for unweighted edit distance is due to Myers [Myers, 1986] and Landau and Vishkin [Landau and Vishkin, 1988], who obtained an $\mathcal{O}(n + k^2)$-time solution for two strings of length n with an upper bound k on their edit distance. This bound is now known to be optimal (up to subpolynomial factors) under the Strong Exponential Hypothesis [Backurs and Indyk, 2018]. Over the years, the Holy-Grail result of [Myers, 1986, Landau and Vishkin, 1988] has inspired many developments on fast approximation algorithms for (unweighted) string edit distance [Chakraborty et al., 2020, Goldenberg et al., 2019, Goldenberg et al., 2020, Kociumaka and Saha, 2020] and similar $\tilde{\mathcal{O}}(n + \text{poly}(k))$-time[1] algorithms for generalizations such as the (unweighted) Dyck and tree edit distances [Backurs and Onak, 2016, Fried et al., 2022, Das et al., 2022a]. However, almost all real-world applications require weighted edit distance, where different weights are assigned to different edit operations (insertions, deletions, and substitutions), and the weights may vary with the characters being

[1]The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses factors polylogarithmic in the input size n.

edited [Wagner and Fischer, 1974, Zhang and Shasha, 1989, Kurtz, 1996, Gusfield, 1997, Peris and Marzal, 2002, Jurafsky and Martin, 2009, Fontan et al., 2016, Skiena, 2020, Koide et al., 2020, Gerlach et al., 2021, Chumachenko et al., 2022]. As a result, there is a major gap between the theoretical results of prior research and real-world utility of these results. In this paper, we bridge this fundamental gap between the understanding of unweighted and weighted edit distance: We provide the first non-trivial algorithm computing the weighted edit distance and its generalizations to weighted tree and Dyck tree edit distance.

More specifically, in this paper we propose the first $\mathcal{O}(n+\mathrm{poly}(k))$-time algorithm for exact weighted edit distance computation in which, given a weight function $w : \Sigma\cup\{\varepsilon\}\times\Sigma\cup\{\varepsilon\} \to \mathbb{R}_{\geq 0}$ (normalized so that $w(a,b) \geq 1$ for $a \neq b$), the goal is to find an alignment that minimizes the total weight of edit operations (insertions, deletions, and substitutions) assuming that it does not exceed a provided threshold k. Strikingly, except for the vanilla $\mathcal{O}(n^2)$-time dynamic-programming algorithm and its almost trivial $\mathcal{O}(nk)$-time implementation, none of the aforementioned developments on unweighted edit distance apply to this weighted variant. We then generalize our result to weighted tree and Dyck edit distances, bringing in several new techniques that lead to improvements even for the unweighted tree edit distance problem: As a byproduct of our results, we present a deterministic $\mathcal{O}(n + k^7 \log k)$-time solution, which is much faster than the randomized $\mathcal{O}(n \log n + k^{15} \log^2 k)$-time algorithm of Das, Gilbert, Hajiaghayi, Kociumaka, Saha, and Saleh [Das et al., 2022a].

Can this apparent lack of progress in weighted edit distance computation be explained? As we observe later, even basic properties like monotonicity, which was fundamental for efficient computation of unweighted edit distance [Myers, 1986, Landau and Vishkin, 1988], break down when considering weighted operations. This precludes any local matching approach, which seemed nec-

essary for a linear-time algorithm for bounded (unweighted) edit distance [Myers, 1986, Landau and Vishkin, 1988, Backurs and Onak, 2016, Fried et al., 2022, Das et al., 2022a]; instead, a global view of the sequences is needed to find matching substrings and yet maintain the linear runtime. Faced with such barriers, our biggest contribution is a kernelization method for weighted edit distance, not just for strings, but also for tree and Dyck edit distance instances. Interestingly, our kernels are weight-agnostic, that is, the kernelization algorithms do not need to know the weight function w. Given how fundamental weighted edit distance is, we believe our $\mathcal{O}(n + \text{poly}(k))$ algorithm for weighted edit distance will be instrumental for further significant developments in the area.

3.1.1 Related Work

String Edit Distance: *Edit distance* is one of the most fundamental problems in computer science studied since the 1960s [Levenshtein, 1965, Needleman and Wunsch, 1970, Wagner and Fischer, 1974]. In the unweighted edit distance problem, given two strings of length at most n, the goal is to find the minimum number of edit operations (insertions, deletions, and substitutions) required to transform one string into the other. Given a parameter k as an upper bound on the edit distance, an algorithm proposed in the 1980s by Myers [Myers, 1986] and Landau and Vishkin [Landau and Vishkin, 1988] achieves this task in $\mathcal{O}(n + k^2)$ time by combining suffix trees with an elegant greedy approach. As long as $k = \mathcal{O}(\sqrt{n})$, the running time of the above algorithm in linear in n. For larger values of k, approximation algorithms for edit distance have been studied extensively [Landau et al., 1998, Indyk, 2001, Bar-Yossef et al., 2004, Batu et al., 2006, Andoni and Onak, 2012, Andoni et al., 2010], especially recently [Haeupler et al., 2019, Chakraborty et al.,

2020, Goldenberg et al., 2020, Koucký and Saks, 2020, Brakensiek and Rubinstein, 2020, Boroujeni et al., 2021]. This culminated with the currently best bound by Andoni and Nosatzki [Andoni and Nosatzki, 2020], who obtained a constant-factor approximation algorithm with running time $\mathcal{O}(n^{1+\epsilon})$ time for any constant $\epsilon > 0$. All of these works require monotonicity and assume that an optimal solution can be extended easily if matching suffixes are added to both strings, none of which may hold in weighted edit distance instances. As a result, the state-of-the-art approximation algorithm for weighted edit distance, by Kuszmaul [Kuszmaul, 2019], offers much worse trade-off, with an $\mathcal{O}(n^{\tau})$-factor approximation in $\tilde{\mathcal{O}}(n^{2-\tau})$ time for any $0 \leq \tau \leq 1$.

Tree Edit Distance: The *tree edit distance* problem, first introduced by Selkow [Selkow, 1977], is a generalization of edit distance in which the task is to compute a measure of dissimilarity between two rooted ordered trees with node labels. In the unweighted version of tree edit distance, every node insertion, deletion, or relabeling operation has unit cost. The problem has numerous applications in compiler optimization [Demaine et al., 2010], structured data analysis [Chawathe, 1999, Buneman et al., 2003, Ferragina et al., 2009], image analysis [Bunke and Shearer, 1998], and computational biology [Harel and Tarjan, 1984, Shapiro and Zhang, 1990, Gusfield, 1997, Landau et al., 1998, Bille, 2005]. The current best bound on running time of an algorithm for finding exact tree edit distance is due to Dürr [Dürr, 2023] who obtained an $\mathcal{O}(n^{2.9149})$-time algorithm for the problem, after a long series of improvements from $\mathcal{O}(n^6)$ [Tai, 1979] to $\mathcal{O}(n^4)$ [Zhang and Shasha, 1989], to $\mathcal{O}(n^3 \log n)$ [Klein, 1998], to $\mathcal{O}(n^3)$ [Demaine et al., 2010], and to $\mathcal{O}(n^{2.9546})$ [Mao, 2021]. Moreover, there is a $(1+\epsilon)$-approximation algorithm for tree edit distance with running time $\tilde{\mathcal{O}}(n^2)$ time due to Boroujeni, Ghodsi, Hajiaghayi, and Seddighin [Boroujeni et al., 2019]. Recently, Seddighin and Seddighin [Seddighin and Seddighin, 2022] gave an $\mathcal{O}(n^{1.99})$-time $(3+\epsilon)$-

approximation algorithm for tree edit distance (building on a previous $\tilde{\mathcal{O}}(n)$-time $\mathcal{O}(\sqrt{n})$-factor approximation algorithm of [Boroujeni et al., 2019]). Furthermore, Das, Gilbert, Hajiaghayi, Kociumaka, Saha, and Saleh [Das et al., 2022a] obtained an $\tilde{\mathcal{O}}(n + k^{15})$-time algorithm for exact tree edit distance with an upper bound k on the distance (see also an $\tilde{\mathcal{O}}(nk^2)$-time algorithm of Akmal and Jin [Akmal and Jin, 2021], which improves upon a previous algorithm with running time $\mathcal{O}(nk^3)$ for the bounded tree edit distance problem [Touzet, 2005]).

As far as the weighted tree edit distance is concerned, the fastest algorithm, by Demaine, Mozes, Rossman, and Weimann [Demaine et al., 2010], takes $\mathcal{O}(n^3)$ time, which matches the conditional lower-bound of Bringmann, Gawrychowski, Mozes, and Weimann [Bringmann et al., 2020] (earlier conjectured by Abboud [Abboud, 2014]). Specifically, there is no truly subcubic-time algorithm for weighted tree edit distance unless APSP has a truly subcubic-time solution. The lower bound still holds for trees over a constant-size alphabet unless the weighted k-clique problem admits an $\mathcal{O}(n^{k-\epsilon})$-time algorithm.

Dyck Edit Distance: The *Dyck edit distance* problem is another variation of edit distance which falls under the umbrella of general language edit distance [Aho and Peterson, 1972, Myers, 1995, Saha, 2017, Bringmann et al., 2019] and has numerous practical applications, e.g., for fixing hierarchical data files, in particular XML and JSON files [Harrison, 1978, Kozen, 1997]. In the unweighted version of this problem, given a string of n parentheses, the goal is to find the minimum number of edits (character insertions, deletions, and substitutions) to make the string well-balanced. Several algorithms for both exact [Bringmann et al., 2019, Chi et al., 2022a, Dürr, 2023] and approximation [Saha, 2014, Das et al., 2022b] versions of the problem have been obtained. Finding exact Dyck edit distance is at least as hard as Boolean matrix multiplication [Abboud

et al., 2018]. The bounded Dyck edit problem was subject to several recent studies as well: Backurs and Onak [Backurs and Onak, 2016] obtained the first algorithm with running time $\mathcal{O}(n+k^{16})$, which was further improved to $\mathcal{O}(n+k^5)$ [Fried et al., 2022], and finally to $\mathcal{O}(n+k^{4.5442})$ using fast matrix multiplication [Fried et al., 2022, Dürr, 2023]. Except for the $\mathcal{O}(n^3)$-time exact algorithm for language edit distance [Myers, 1995], these results are not applicable to the weighted setting.

3.1.2 Our Contribution

The main contributions of our paper are new algorithms for weighted string, tree, and Dyck edit distance. We define a *weight function* as a function $w : \Sigma \cup \{\varepsilon\} \times \Sigma \cup \{\varepsilon\} \to \mathbb{R}_{\geq 0}$ such that $w(a,a) = 0$ and $w(a,b) \geq 1$ for $a \neq b$. If $a, b \in \Sigma$, then $w(a,\varepsilon)$ is the cost of deleting a, $w(\varepsilon,b)$ is the cost of inserting b, whereas $w(a,b)$ is the cost of substituting a for b. The assumption $w(a,a) = 0$ indicates that matching symbols can be aligned at no cost, whereas the assumption $w(a,b) \geq 1$ for $a \neq b$ indicates that the weights are normalized so that every edit costs at least one. A weight function is a *quasimetric* if it also satisfies the triangle inequality (which we assume for tree and Dyck edit distance). When it comes to computations on weights, we consider any uniform model in which real numbers are subject to only comparison and addition [Pettie and Ramachandran, 2005], e.g., the RAM model.

We define $\mathsf{ed}^w(X,Y)$ to be the minimum cost of an alignment of strings X and Y for weight function w. Furthermore, we define $\mathsf{ed}^w_{\leq k}(X,Y)$ as $\mathsf{ed}^w(X,Y)$ (if it is at most k) or ∞ (otherwise). We give the first weighted bounded edit distance algorithm with runtime $\mathcal{O}(n+\mathrm{poly}(k))$.

Theorem 3.1.1. *Given strings X, Y of length at most n, an integer $k \in \mathbb{Z}_+$, and a weight func-*

tion w, the value $\mathsf{ed}^w_{\le k}(X, Y)$ can be computed in $\mathcal{O}(n + k^5)$ time.

Similarly to string edit distance, we define $\mathsf{ted}^w(F, G)$ as the minimum cost of a tree alignment of forests F and G for weight function w. We define $\mathsf{ted}^w_{\le k}(F, G)$ analogously and give the first weighted tree edit distance algorithm with runtime $\mathcal{O}(n + \mathrm{poly}(k))$. In the unweighted case, our deterministic algorithm is significantly faster than the state-of-the-art randomized algorithm from [Das et al., 2022a].

Theorem 3.1.2. *Given forests F, G of length at most n, an integer $k \in \mathbb{Z}_+$, and a quasimetric w, the value $\mathsf{ted}^w_{\le k}(F, G)$ can be computed in $\mathcal{O}(n + k^{15})$ time. Moreover, $\mathsf{ted}_{\le k}(F, G)$ can be computed in $\mathcal{O}(n + k^7 \log k)$ time.*

Finally, we define $\mathsf{dyck}^w_{\le k}(X)$ to be the minimum distance $\mathsf{ed}^w_{\le k}(X, Y)$ between X and a string Y in the Dyck language. We give the first algorithm for weighted Dyck edit distance with runtime $\mathcal{O}(n + \mathrm{poly}(k))$. In this setting, the alphabet consists of opening and closing parentheses, and we need to assume that the weight function, apart from satisfying the triangle inequality, treats opening and closing parentheses of the same type similarly. This is captured in the notion of a *skewmetric* formally defined in Section 3.4.2.

Theorem 3.1.3. *Given a string X of length n, an integer $k \in \mathbb{Z}_+$, and a skewmetric w, the value $\mathsf{dyck}^w_{\le k}(X)$ can be computed in $\mathcal{O}(n + k^{12})$ time.*

We note that, although our algorithms assume k is given, one can also obtain running times analogous to those of Theorems 3.1.1 to 3.1.3 but with the sought distance instead of the threshold k. For this, it suffices to start from the largest value k that results in the running time of $\mathcal{O}(n)$, e.g., $k = \Theta(n^{1/5})$ for strings, and keep doubling the threshold k as long as the algorithm outputs

∞. The first finite outcome is guaranteed to be the sought distance and, since the running times of the subsequent iterations form a geometric progression, the overall runtime is dominated by the last iteration, where k is at most twice the sought distance.

3.1.3 Overview

The folklore algorithms to compute edit distance for unweighted and weighted instances use dynamic programming and runs in $\mathcal{O}(n^2)$ time. Given two strings X and Y, the entry $D[i,j]$ of the dynamic programming table D holds the weighted (unweighted) edit distance of prefixes of X and Y up to indices i and j respectively. That is $D[i,j] := \mathsf{ed}(X[0\,..\,i), Y[0\,..\,j))$. Then

$$D[i{+}1, j{+}1] = \min\{D[i, j{+}1]{+}1, D[i{+}1, j]{+}1, D[i, j]{+}\delta(X[i], Y[j])\} \quad \textit{:unweighted edit distance}$$

$$D[i+1, j+1] = \min\{D[i, j+1] + w(X[i], \varepsilon), D[i+1, j] + w(\varepsilon, Y[j]), D[i, j] + w(X[i], Y[j])\}.$$

:weighted edit distance

The first entry in the recursive definition corresponds to deleting $X[i]$, the second entry corresponds to inserting $Y[j]$, and the third entry corresponds to either matching or substitution ($\delta(X[i], Y[j]) = 0$ if $X[i] = Y[j]$, otherwise $\delta(X[i], Y[j]) = 1$). Clearly, $D[|X|, |Y|]$ equals the total weighted (unweighted) edit distance between X and Y, and can be computed in $\mathcal{O}(n^2)$ time.

It is possible to improve the running time to $\mathcal{O}(nk)$ if the weighted (unweighted) edit distance is bounded by $k < n$. In this case the entries corresponding to only $2k + 1$ diagonals surrounding the main diagonal of D need to be computed. However, the similarities between the developments

on unweighted and weighted edit distance computations end here.

The first major breakthrough in the unweighted edit distance computation came in the late eighties [Myers, 1986, Landau and Vishkin, 1988]. An $\mathcal{O}(n + k^2)$-time algorithm for unweighted edit distance was developed whenever edit distance is bounded by k, thereby giving a linear time algorithm for $k \leq \sqrt{n}$. The algorithm utilizes two simple but powerful properties of unweighted edit distance, namely (i) *monotonicity*: $D[i + 1, j + 1] \geq D[i, j]$, and (ii) *greedy extension*: if $X[i] = Y[j]$ then $D[i+1, j+1] = D[i, j]$. These two properties together imply that if we can find maximal equal substrings in X and Y through a preprocessing step, only $\mathcal{O}(k^2)$ entries of D need to be computed. More precisely, for each of the $2k+1$ diagonals, these are the at most $k+1$ entries with $k+1 \geq D[i+1, j+1] > D[i, j]$. The preprocessing step utilizes a linear-time construction of a suffix tree to answer any maximal equal substring queries in constant time, leading to an overall running time of $\mathcal{O}(n + k^2)$. All subsequent developments on fast approximation algorithms for unweighted string edit distance rely on the above two properties without exception.

Unfortunately, none of the above two properties hold for weighted edit distance computation. The following simple examples will make this observation clear.

1. *No monotonicity:* Let $X = \texttt{ab}$, $Y = \texttt{c}$, and $w(\texttt{a}, \texttt{c}) + w(\texttt{b}, \varepsilon) < w(\texttt{a}, \varepsilon)$. Then $D[1, 0] = w(\texttt{a}, \varepsilon)$ and $D[2, 1] = w(\texttt{a}, \texttt{c}) + w(\texttt{b}, \varepsilon) < w(\texttt{a}, \varepsilon)$.
2. *No greedy extension:* Let $X = \texttt{ab}$, $Y = \texttt{b}$, and $w(\texttt{a}, \texttt{b}) + w(\texttt{b}, \varepsilon) < w(\texttt{a}, \varepsilon)$. Then substituting `a` to `b` and deleting `b` from X is cheaper than deleting `a` and matching the subsequent `b`.

In some sense, this explains the lack of progress on weighted instances in this field. We need a very different approach and new ideas.

When k, the minimum weighted edit distance, is small for two input strings, clearly most characters of the input strings are perfectly matched and contribute no cost to the edit distance computation. The main idea of our algorithm is to find small representative instances for the input strings and then run the $\mathcal{O}(nk)$-time weighted edit distance solution on these representatives to find the original weighted edit distance. In fact, we prove that any instance of the bounded-weighted edit distance can be solved using strings of size $\mathcal{O}(k^4)$. Our algorithm constructs such an $\mathcal{O}(k^4)$-size kernel from strings of size $\mathcal{O}(n)$ in time $\mathcal{O}(n)$, and then the resulting small instances can be solved in time $\mathcal{O}(k^5)$ using the $\mathcal{O}(nk)$-time weighted extension of the dynamic programming.

We are also able to extend the idea of kernelization to weighted instances of tree and Dyck edit distances by giving the first $\mathcal{O}(n + \mathrm{poly}(k))$ algorithms for them. Notably, our algorithms are deterministic and give significant improvements over the recent randomized algorithms on unweighted tree edit distance [Das et al., 2022a]. We show it is possible to compute small $\mathcal{O}(\mathrm{poly}(k))$-size kernels from the original instances of each problem in linear time, and then run dynamic programming based algorithms to compute the final edit distance values.

To find such kernels, we utilize substrings that have *synchronized occurrences* in both input strings X and Y, that is, they occur in X and Y at positions x and y, respectively, satisfying $|x - y| = \mathcal{O}(k)$. Our kernelization algorithm first tries to cover the input strings (almost entirely) with $\mathcal{O}(k)$ pairs of synchronized occurrences. If this is impossible, then we conclude that the edit distance must be large, that is, $\mathsf{ed}^w_{\leq k}(X, Y) = \infty$. Otherwise, we apply a novel notion of *edit-distance equivalence* so that synchronized occurrences of a substring P can be substituted with synchronized occurrences of an equivalent substring P' without affecting the edit distance $\mathsf{ed}^w_{\leq k}(X, Y)$. To this end, we provide a linear-time algorithm that, given any string P, computes an edit-distance equivalent string P' of size $\mathrm{poly}(k)$.

A similar notion of equivalent pieces is also central to our algorithms for weighted tree and Dyck edit distance. Our three algorithms all utilize the following high-level steps:

1. Partition the input objects into $\mathcal{O}(k)$ pieces most of which can be paired up to form synchronized occurrences.
2. If the algorithm failed to find sufficiently long synchronized occurrences, report that the edit distance exceeds k.
3. Otherwise, for every pair of synchronized occurrences, substitute the original piece with a small equivalent replacement.
4. Solve the resulting small instance with a known dynamic-programming algorithm.

3.1.3.1 Weighted String Edit Distance

We now describe how to obtain Theorem 3.1.1 by implementing the aforementioned high-level scheme.

Edit-Distance Equivalent Strings The biggest technical contribution behind our weighted edit distance algorithm is a linear-time procedure (of Corollary 3.2.14) that, given a string P, computes an equivalent string of length $\mathcal{O}(k^3)$. In the first phase, it eliminates *k-periodicity*: as long as the processed string contains a fragment of the form Q^{4k+1} with $|Q| \in [1\,..\,2k]$, this fragment is replaced by Q^{4k}. As shown in Lemma 3.2.9, the strings Q^{4k+1} and Q^{4k} are equivalent, so this step preserves equivalence with the input string P. Eventually, the first phase results in a string that *avoids k-periodicity* and is equivalent with P (see Fig. 3.1 for example). It is implemented in Lemma 3.2.13, where the underlying algorithm processes the input string P from left to right and removes the first copy of Q for every encountered fragment of the form Q^{4k+1} with $|Q| \in [1\,..\,2k]$.

In Lemma 3.2.11, we prove that if P avoids k-periodicity and satisfies $|P| \geq 42k^3$, then it is equivalent to $P[0\,..\,21k^3) \cdot P[|P| - 21k^3\,..\,|P|)$, that is, the concatenation of its prefix of length $21k^3$ and its suffix of length $21k^3$ (with the characters in the middle removed). For this, we consider synchronized occurrences of P in strings X and Y and an optimal alignment $\mathcal{A}$ of cost $\mathsf{ed}^w(X, Y) \leq k$ that maps X onto Y. We observe that $\mathcal{A}$ must perfectly match a length-$10k^2$ fragment within the length-$21k^3$ prefix of the occurrence of P in X. Moreover, since P avoids k-periodicity, $\mathcal{A}$ can only match this fragment to the corresponding fragment of the occurrence of P in Y. Symmetrically, $\mathcal{A}$ must match the two copies of a length-$10k^2$ fragment within the length-$21k^3$ suffix of P. We conclude that $\mathcal{A}$ aligns the two copies of $P[d\,..\,|P| - e)$ for some $d, e \in [0\,..\,21k^3]$. Since $\mathcal{A}$ is optimal, it must perfectly match the two copies of $P[d\,..\,|P| - e)$. Thus, $P[21k^3\,..\,|P| - 21k^3)$ can be removed from the synchronized occurrences of P in X and Y without affecting the cost $\mathsf{ed}^w_{\leq k}(X, Y)$. Consequently, if the first phase returns a string of length at least $42k^3$, then the algorithm of Corollary 3.2.14 removes all but the leading $21k^3$ and the trailing $21k^3$ characters of that string (see Fig. 3.2 for example).

Linear-Time Kernel In order to apply the notion of edit-distance equivalence, we need to identify synchronized occurrences within X and Y. To this end, we check whether $\mathsf{ed}(X, Y) \leq k$. If this is not the case, then $\mathsf{ed}^w(X, Y) \geq \mathsf{ed}(X, Y) > k$ holds for every normalized weight function w, and thus we already know that $\mathsf{ed}^w_{\leq k}(X, Y) = \infty$. If $\mathsf{ed}(X, Y) \leq k$, on the other hand, then we construct an optimal unweighted $\mathcal{A}$ alignment mapping X onto Y. As formally proved in Fact 3.2.7, the unedited characters of X form at most $k + 1$ fragments that $\mathcal{A}$ matches perfectly. Each of these fragments of X forms a synchronized occurrence together with its image under $\mathcal{A}$ in Y. Thus, we can replace the synchronized occurrences with occurrences of an equivalent string of length

$\mathcal{O}(k^3)$. Since we have partitioned X and Y into $\mathcal{O}(k)$ edited characters plus $\mathcal{O}(k)$ synchronized occurrences, this yields strings X' and Y' of length $\mathcal{O}(k^4)$ satisfying $\mathsf{ed}^w_{\leq k}(X', Y') = \mathsf{ed}^w_{\leq k}(X, Y)$. In order to construct $\mathcal{A}$ efficiently, we use the $\mathcal{O}(n + k^2)$ unweighted edit distance algorithm of [Myers, 1986, Landau and Vishkin, 1988]. However, if $n \leq k^4$, then we do not need to worry about reducing the size of X and Y in the first place and therefore do not construct an optimal unweighted alignment; otherwise, $\mathcal{O}(n + k^2) = \mathcal{O}(n)$ and constructing the $\mathcal{O}(k^4)$-size kernel takes linear time; see Theorem 3.2.15 for details on our kernel for weighted string edit distance.

As mentioned earlier, once we have a kernel (X', Y') of size $\mathcal{O}(k^4)$, we can run the $\mathcal{O}(nk)$-time weighted edit-distance algorithm to compute $\mathsf{ed}^w_{\leq k}(X', Y') = \mathsf{ed}^w_{\leq k}(X, Y)$ in $\mathcal{O}(k^5)$ time.

3.1.3.2 Weighted Tree Edit Distance

Our algorithm for weighted tree edit distance follows the same high-level approach. However, compared to the string edit distance, two major challenges arise. First, the structure of periodicity is much richer and requires two notions: *horizontal periodicity* of *forests* and *vertical periodicity* of *contexts*. As a result, we need separate definitions of tree-edit-distance equivalence for forests and contexts. Nevertheless, assuming that the weight function w satisfies the triangle inequality, we can still construct equivalent forests and contexts of size $\mathcal{O}(k^3)$ and $\mathcal{O}(k^4)$, respectively. The second challenge is that the state-of-the-art algorithm for computing the unweighted tree edit distance is randomized and takes $\mathcal{O}(n \log n + \mathrm{poly}(k))$ time rather than $\mathcal{O}(n + \mathrm{poly}(k))$ time. Thus, in order to achieve a deterministic linear-time kernel, we need another method for identifying large synchronizing occurrences. Our workaround is to shrink the input in multiple iterations (essentially halving the size each time) rather than in a single shot. This way, we can still

obtain a kernel of size $\mathcal{O}(k^5)$, which is asymptotically as small as we would get from an optimum unweighted alignment.

Periodicity in Trees Intuitively, the two types of periodicity in trees correspond to the two ways to interpret strings as trees. For a string X, the *horizontal embedding* constructs a tree with $|X|$ leafs attached to the root and labeled by subsequent characters of X, whereas the *vertical embedding* constructs a path with $|X|$ nodes labeled by subsequent characters of X. Similarly, forest algebras (see [Bojańczyk and Walukiewicz, 2008] for a survey) in formal language theory involve two natural monoids: a *horizontal monoid* of forests (with concatenation, denoted $\cdot$) and a *vertical monoid* of contexts (with composition, denoted $\star$). A context can be defined as a tree with a single *hole* in some leaf, and contexts can be composed by placing one of them in the hole of the other. Moreover, placing a forest in the hole of a context yields a forest. In order to formalize these notions and easily port combinatorial and algorithmic tools designed for strings, we interpret forests as balanced strings of parentheses; see Section 3.3.1.

Following [Das et al., 2022a], a horizontal power is the concatenation of multiple copies of the same forest, whereas a vertical power is the composition of multiple copies of the same context; see Fig. 1.1 for an example. More specifically, we say that a forest contains horizontal k-periodicity if it has a subforest of the form Q^{4k+1} for some forest Q of size $|Q| \leq 4k$, whereas a context contains vertical k-periodicity if it can be expressed as a composition of several contexts, including Q^{6k+1} for some context Q of size $|Q| \leq 8k$.

Tree-Edit-Distance Equivalent Forests The first ingredient of our algorithm for weighted tree edit distance is a linear-time procedure that, given a forest P, constructs an equivalent forest of

size $\mathcal{O}(k^3)$. The first phase of this subroutine eliminates horizontal k-periodicity: as long as the processed forest contains a subforest of the form Q^{4k+1} with $|Q| \in [1\,..\,4k]$, this subforest is replaced by Q^{4k}. As shown in Lemma 3.3.5, the forests Q^{4k+1} and Q^{4k} are equivalent, so this step preserves equivalence with the input forest P. An efficient implementation of this phase relies on the fact that, if P is interpreted as a string, then horizontal k-periodicity can be interpreted as a substring of the form Q^{4k+1} for a sufficiently short *balanced* string Q. Thus, we can reuse Lemma 3.2.13 to obtain a forest equivalent with P that avoids horizontal k-periodicity.

In Lemma 3.3.7, we show the equivalence of any two forests of size at least $74k^3$ that avoid horizontal k-periodicity.[2] Based on this result, if horizontal periodicity reduction yields a forest of size at least $74k^3$, we return a canonical forest of size exactly $74k^3$; see Lemma 3.3.17 for details.

Tree-Edit-Distance Equivalent Contexts Our next ingredient is a linear-time algorithm that, given a context P, constructs an equivalent context of size $\mathcal{O}(k^4)$. First, we use the previous procedure for every maximal forest in P (that does not contain the hole). Then, we eliminate vertical k-periodicity: as long as P contains a context of the form Q^{6k+1} with $|Q| \in [1\,..\,8k]$, this context is replaced by Q^{6k}. As shown in Lemma 3.3.10, the contexts Q^{6k+1} and Q^{6k} are equivalent, so this step preserves equivalence with the input context P. For an efficient implementation, the *spine*, i.e., the path from the root of P to the hole, is interpreted as a string, with each character encoding the label of the underlying node and the subtrees attached there to the left and to the right of the spine. This way, vertical-k periodicity can be interpreted as periodicity in the constructed string, and hence Lemma 3.2.13 can be used again.

In Lemma 3.3.12, we show the equivalence of any two contexts of size at least $578k^4$ that

[2]This statement is stronger that its counterpart for strings, Lemma 3.2.11, because we now assume that the weight function w satisfies the triangle inequality.

avoid vertical k-periodicity and subforests of size more than $74k^3$. Thus, if vertical periodicity reduction yields a context of size at least $578k^4$, we replace it with a canonical context of size exactly $578k^4$; see Lemma 3.3.18 for details.

Linear-Time Kernel As for strings, in order to apply the notion of tree-edit-distance equivalence, we need to identify synchronized occurrences of forests and contexts within the input forests F and G. As mention above, in order to obtain a deterministic linear-time kernel, we cannot use the algorithm of [Das et al., 2022a] to obtain a tree alignment mapping F to G with at most k edits. Instead, we develop an iterative workaround. At each step, we decompose F into $\mathcal{O}(k)$ contexts and forests (jointly called pieces) of size at most $\frac{n}{2k}$ each; see Lemma 3.3.15 for details. Next, we maximize the number of pieces (from the decomposition) that admit disjoint synchronized occurrences in G; Lemma 3.3.16 implements this step in $\mathcal{O}(n + k^4)$ time using dynamic programming. If $\mathsf{ted}(F, G) \leq k$, then no more than k of the pieces are left unmatched (an optimal alignment may edit at most k pieces). We replace the matched pieces with equivalent pieces of size $\mathcal{O}(k^5)$, obtaining forests of size at most $\frac{n}{2} + \mathcal{O}(k^5)$, where the first term corresponds to the unmatched pieces; see Theorem 3.3.19. As long as $n = \omega(k^5)$, this procedure essentially halves the input size. Hence, as shown in Corollary 3.3.20, this still yields a linear-time algorithm producing forests F' and G' of size $\mathcal{O}(k^5)$ such that $\mathsf{ted}^w_{\leq k}(F', G') = \mathsf{ted}^w_{\leq k}(F, G)$.

Once we have such a kernel (F', G') of size $\mathcal{O}(k^5)$, we can run the cubic-time weighted edit-distance algorithm [Demaine et al., 2010] to compute $\mathsf{ted}^w_{\leq k}(X', Y') = \mathsf{ted}^w_{\leq k}(X, Y)$ in $\mathcal{O}(k^{15})$ time, for a total runtime of $\mathcal{O}(n+k^{15})$. Additionally, we significantly improve the state-of-the-art of the unweighted tree edit distance problem by using the $\mathcal{O}(nk^2 \log n)$-time algorithm from [Akmal and Jin, 2021], which gives us a total runtime of $\mathcal{O}(n + k^7 \log k)$ for unweighted tree edit distance.

3.1.3.3 Weighted Dyck Edit Distance

In the final section of our paper, the weighted Dyck edit distance algorithm follows a similar approach to that of the string and tree edit distance algorithm. However, many of the proofs and details are specific to Dyck edit distance problem and come with their own set of intricacies and difficulties that we outline in the following.

Given a string X over an alphabet $\Sigma = T \cup \overline{T}$ (where T and $\overline{T}$ are the sets of opening and closing parentheses, respectively), an integer $k \in \mathbb{Z}_+$, and a skewmetric weight function w representing the cost of each edit operation (parenthesis insertion, deletion, and substitution), our objective is to compute the minimum weight of a sequence of edits that convert X to a well-parenthesized expression over Σ provided the total weight of all edits is bounded by k. In this work we design a deterministic algorithm that achieves this goal in $\mathcal{O}(n + k^{12})$ time. For the unweighted counterpart of this problem, the recent solution of [Fried et al., 2022, Dürr, 2023] computes the Dyck edit distance in time $\mathcal{O}(n + k^{4.5442})$. That algorithm, consistently with its predecessors [Backurs and Onak, 2016, Fried et al., 2022], starts with a greedy preprocessing step that exhaustively removes any two adjacent characters $X[i]X[i+1]$ such that $X[i]$ is an opening parenthesis and $X[i+1]$ is a closing parenthesis of the same type. Following a simple argument, it can be shown that the Dyck edit distance of the preprocessed string stays exactly the same as the input string X.

Preprocessing We tried to follow a similar approach for the weighted version, but it turns out that such a simple analysis is not enough to construct a reduced string. For example, let the input string be `({()`. For a general weight function w, it is not evident that the optimal matching should

always match the last two parentheses. In fact, if we consider a weight function where the cost of substituting { with) is 10 whereas the cost of substituting (with } is 5, then any optimal matching should match the first and the last parentheses instead of the last two. Thus, in this work, we consider our weight function w to be a skewmetric. Formally, we assume that w satisfies the triangle inequality and skew-symmetry, that is, $w(p_1, p_2) = w(\overline{p_2}, \overline{p_1})$ holds for all $p_1, p_2 \in \Sigma \cup \{\varepsilon\}$, where $\overline{p}$ is the parenthesis complementary to p (and $\overline{\varepsilon} = \varepsilon$). Following this property of w, we show that one can apply a similar greedy preprocessing (as described for the unweighted version) to reduce X to a string X' while preserving the weighted Dyck edit distance. Our argument is substantially more elaborate, though, and follows a case-by-case analysis depending on the structure of the other alternate alignments (Claim 3.4.7). Nevertheless, it is trivial to observe the greedy preprocessing can be done in linear time.

Dyck-Edit-Distance Equivalent Strings Next, following a similar strategy as described for string edit distance, we further reduce X' to generate a string X'' of length $\mathcal{O}(k^4)$ while preserving the weighted Dyck edit distance. For this first we introduce the concept of k-synchronicity. A substring P containing only opening parentheses and a substring $\overline{P}$ containing only closing parentheses are k-synchronized if $\overline{P}$ appears after P, they are of same length and their height difference is at most $2k$. Following this and the non-crossing property of Dyck matching, first we argue that if the lengths of $P, \overline{P}$ are large and the distance is bounded by k, then there exist a substring $\ell \in P$ that is matched with a substring $\ell' \in \overline{P}$ in the optimal alignment (we fix one for the analysis purpose). Now if we replace P with $P \setminus \ell$ and P' with $P' \setminus \ell'$ then in the resulting string the distance stays the same (Fact 3.4.12). Following this, for any two k-synchronized substrings $P, \overline{P}$, we can reduce their periodicity as follows: if $P = Q^e$ and $\overline{P} = \overline{Q^e}$, (where Q is a primitive string with large

exponent e) then at least one occurrence of Q is matched with its reverse complement counterpart $\overline{Q}$ in $\overline{P}$. Thus, we can remove the matched part while not changing the distance, and it reduces the exponent by one. Repeat this until e become small (Lemma 3.4.13).

Next assuming that $P, \overline{P}$ avoid periodicity, it can be shown that there exists a pair of indices $i, j \in [0\,..\,78k^3]$ such that $P[i]$ is matched with, $\overline{P}[|P| - 1 - i]$ and $P[|P| - 1 - j]$ is matched with, $\overline{P}[j]$ in the optimal alignment. Thus, following the fact that $|P| = |\overline{P}|$ and the non-crossing property of the Dyck optimal alignment, all the indices between i and $|P| - 1 - j$ are also matched, and thus removing these matched characters from both $P, \overline{P}$ does not affect the Dyck edit distance. Consequently, we replace each k-synchronized pairs with substrings of length just $156k^3$ (replace $P, \overline{P}$ with their first and last $78k^3$ characters) to generate a string X'' such that the weighted Dyck edit distance of X and X'' is the same (Lemma 3.4.15, Corollary 3.4.17).

Linear-Time Kernel Lastly, we show if the distance is bounded by k, then X can be partitioned in time $\mathcal{O}(n + k^5)$ into $\mathcal{O}(k)$ disjoint k-synchronized pairs of substrings (plus $\mathcal{O}(k)$ individual characters) and thus the total length of X'' is bounded by $\mathcal{O}(k^4)$. Start by preprocessing input string X to generate X'. Next, we check if $\mathsf{ded}(X') \leq k$ and, if so, we compute an unweighted optimal Dyck alignment $\mathcal{M}$ of X' in time $\mathcal{O}(n + k^5)$ [Fried et al., 2022]. Then, we argue any pair of substrings of X' that are matched by $\mathcal{M}$ are k-synchronized. Thus, using $\mathcal{M}$, we identify the set of maximal substrings from T^* and $\overline{T}^*$ that are matched by $\mathcal{M}$. A substring is maximal in a sense that either the substring itself or its matched counterpart can not be extended to the right or left without paying an edit. As unweighted Dyck edit distance is no more than the weighted version and hence, assuming cost of $\mathcal{M}$ is bounded by k, we can show string X' can be partitioned into $\mathcal{O}(k)$ different k-synchronized pairs. Also, these maximal fragments can be found in linear

time with a left-to-right scan of X'. Subsequently, we create a string X'' from X' as follows: (i) for each k-synchronized pairs we reduce them following the algorithm as discussed above and add two corresponding strings each of length $O(k^3)$ (ii) add all the characters that are edited by $\mathcal{M}$ just the same to X'' (Theorem 3.4.19).

Finally, we compute the weighted Dyck edit distance of X'' using the dynamic program algorithm of [Myers, 1995] in time $\mathcal{O}(k^{12})$.

3.2 String Edit Distance

3.2.1 Preliminaries

A *string* $Y \in \Sigma^n$ is a sequence of $|Y| := n$ characters from an *alphabet* Σ. For $i \in [0\,..\,n)$, we denote the ith character of Y with $Y[i]$. We say that a string X *occurs* as a *substring* of a string Y if $X = Y[i] \cdots Y[j-1]$ holds for some integers $0 \leq i \leq j \leq |Y|$. We denote the underlying *occurrence* of X as $Y[i\,..\,j)$. Formally, $Y[i\,..\,j)$ is a *fragment* of Y that can be represented using a reference to Y as well as its endpoints i, j. The fragment $Y[i\,..\,j)$ can be alternatively denoted as $Y[i\,..\,j-1]$, $Y(i-1\,..\,j-1]$, or $Y(i-1\,..\,j)$. A fragment of the form $Y[0\,..\,j)$ is a *prefix* of Y, whereas a fragment of the form $Y[i\,..\,n)$ is a *suffix* of Y.

Theorem 3.2.1 (LCE queries [Landau and Vishkin, 1988, Farach-Colton et al., 2000])**.** *Strings X, Y can be preprocessed in linear time so that the following* longest common extension *(LCE) queries can be answered in $\mathcal{O}(1)$ time: given positions $x \in [0\,..\,|X|]$ and $y \in [0\,..\,|Y|]$, compute the largest ℓ such that $X[x\,..\,x+\ell) = Y[y\,..\,y+\ell)$.*

As mentioned in Section 3.1.3, high-power periodicity plays a key role in our algorithms,

which we may now formally define for strings here. An integer $p \in [1\,..\,n]$ is a *period* of a string $Y \in \Sigma^n$ if $Y[i] = Y[i+p]$ holds for all $i \in [0\,..\,n-p)$. In this case, the prefix $Y[0\,..\,p)$ is called a *string period* of Y. By $\mathsf{per}(Y)$ we denote the smallest period of Y. The exponent of a string Y is defined as $\exp(Y) := \frac{|Y|}{\mathsf{per}(Y)}$, and we say that a string Y is *periodic* if $\exp(Y) \geq 2$.

Theorem 3.2.2 (2-Period queries [Kociumaka et al., 2015, Bannai et al., 2017])**.** *A string X can be preprocessed in linear time so that one can decide in constant time whether any given fragment $X[i\,..\,j)$ is periodic and, if so, compute its shortest period $\mathsf{per}(X[i\,..\,j))$.*

For a string Y and an integer $m \geq 0$, we define the mth power of Y, denoted Y^m, as the concatenation of m copies of Y. A non-empty string $Y \in \Sigma^n$ is *primitive* if it cannot be expressed as $Y = X^m$ for some string X and integer $m > 1$. For a string $Y \in \Sigma^n$, we define a *forward rotation* $\mathsf{rot}(Y) = Y[1]\cdots Y[n-1]Y[0]$. In general, a *cyclic rotation* $\mathsf{rot}^s(Y)$ with *shift* $s \in \mathbb{Z}$ is obtained by iterating rot or the inverse operation rot^{-1}. A string Y is primitive if and only if it is distinct from its non-trivial rotations, i.e., if $Y = \mathsf{rot}^s(Y)$ holds only when s is a multiple of n.

3.2.2 Edit-Distance Alignments and Weighted Edit Distance

In this subsection, we discuss alignments and their weighted cost, which provide a formal way to describe a sequence of edits needed to transform a string X into Y.

Definition 3.2.3. A sequence $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ is an *alignment* of a fragment $X[x\,..\,x')$ onto a fragment $Y[y\,..\,y')$ if $(x_0, y_0) = (x, y)$, $(x_m, y_m) = (x', y')$, and $(x_{t+1}, y_{t+1}) \in \{(x_t+1, y_t+1), (x_t+1, y_t), (x_t, y_t+1)\}$ for $t \in [0\,..\,m)$. The set of all alignments of $X[x\,..\,x')$ onto $Y[y\,..\,y')$ is denoted with $\mathsf{A}(X[x\,..\,x'), Y[y\,..\,y'))$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m} \in \mathsf{A}(X[x\,..\,x'), Y[y\,..\,y'))$, for every $t \in [0\,..\,m)$, we say that

- $\mathcal{A}$ *deletes* $X[x_t]$ if $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t)$.
- $\mathcal{A}$ *inserts* $Y[y_t]$ if $(x_{t+1}, y_{t+1}) = (x_t, y_t + 1)$.
- $\mathcal{A}$ *aligns* $X[x_t]$ to $Y[y_t]$, denoted by $X[x_t] \sim_{\mathcal{A}} Y[y_t]$, if $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t + 1)$.
- $\mathcal{A}$ *matches* $X[x_t]$ with $Y[y_t]$, denoted by $X[x_t] \simeq_{\mathcal{A}} Y[y_t]$, if $X[x_t] \sim_{\mathcal{A}} Y[y_t]$ and $X[x_t] = Y[y_t]$.
- $\mathcal{A}$ *substitutes* $X[x_t]$ for $Y[y_t]$ if $X[x_t] \sim_{\mathcal{A}} Y[y_t]$ but $X[x_t] \neq Y[y_t]$.

Insertions, deletions, and substitutions are jointly called *(character) edits*.

Example 3.2.4. For an example of an alignment, consider strings $X = \texttt{abc}$ and $Y = \texttt{bd}$. One optimal alignment $\mathcal{A}$ might be $\{(0,0),(1,0),(2,1),(3,2)\}$. The pairs $(0,0),(1,0)$ represent a deletion of $X[0] = \texttt{a}$ by $\mathcal{A}$. The pairs $(1,0),(2,1),(3,2)$ signify that $\mathcal{A}$ aligns $X[1\,..\,2] \sim_{\mathcal{A}} Y[0\,..\,1]$, i.e. $\texttt{bc} \sim_{\mathcal{A}} \texttt{bd}$. Moreover, $X[1]$ is matched to $Y[0]$ since $X[1] = Y[0] = \texttt{b}$ while $X[2]$ is substituted for $Y[1]$ since $X[2] = \texttt{c} \neq \texttt{d} = Y[1]$.

For an alphabet Σ, we define $\bar{\Sigma} := \Sigma \cup \{\varepsilon\}$, where ε is the empty string over Σ. We say that a function $w : \bar{\Sigma} \times \bar{\Sigma} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a *weight function* if $w(a,a) = 0$ holds for all $a \in \bar{\Sigma}$. The *cost* of an alignment $\mathcal{A} \in \mathsf{A}(X[x\,..\,x'), Y[y\,..\,y'))$ with respect to a weight function w, denoted $\mathsf{ed}^{w}_{\mathcal{A}}(X[x\,..\,x'), Y[y\,..\,y'))$, is defined as the total cost of edits that $\mathcal{A}$ performs, where:

- the cost of deleting $X[x]$ is $w(X[x], \varepsilon)$,
- the cost of inserting $Y[y]$ is $w(\varepsilon, Y[y])$,
- the cost of substituting $X[x]$ for $Y[y]$ is $w(X[x], Y[y])$.

The *width* of an alignment $(x_t, y_t)_{t=0}^{m} \in \mathsf{A}(X[x\,..\,x'), Y[y\,..\,y'))$ is defined as $\max_{t=0}^{m} |x_t - y_t|$.

We usually consider alignments of the entire string $X[0\,..\,|X|)$ onto the entire string $Y[0\,..\,|Y|)$, and we denote the set of all such alignment with $\mathsf{A}(X,Y) = \mathsf{A}(X[0\,..\,|X|), Y[0\,..\,|Y|))$. The weighted edit distance of strings $X, Y \in \Sigma^*$ with respect to a weight function w is defined as $\mathsf{ed}^w(X,Y) = \min_{\mathcal{A}\in\mathsf{A}(X,Y)} \mathsf{ed}^w_{\mathcal{A}}(X,Y)$. For $k \in \mathbb{R}_{\geq 0}$, we also denote

$$\mathsf{ed}^w_{\leq k}(X,Y) = \begin{cases} \mathsf{ed}^w(X,Y) & \text{if } \mathsf{ed}^w(X,Y) \leq k, \\ \infty & \text{otherwise.} \end{cases}$$

In the literature, the (weighted) edit distance of X and Y is sometimes defined as the minimum cost of a sequence of edits that transform X into Y. As shown in the following fact (whose technical proof is deferred to Section 3.5), this sequence-based view is equivalent to our alignment-based view provided that w is a *quasimetric*, that is, it satisfies the triangle inequality $w(a,b) + w(b,c) \geq w(a,c)$ for every $a, b, c \in \bar{\Sigma}$. The assumption of w being quasimetric can be made without loss of generality in the sequence-based view (a single character can be edited multiple times, so one can replace w by its distance closure without affecting the edit distances). Our alignment-based view, on the other hand, is more general and captures weighted edit distances violating the triangle inequality.

Fact 3.2.5. *If w is a quasimetric on $\bar{\Sigma}$, then ed^w is a quasimetric on Σ^*. In this case, $\mathsf{ed}^w(X,Y)$ can be equivalently defined as the minimum cost of a sequence of edits transforming X into Y.*

Although our algorithm for strings works for any weight function, its tree and Dyck counterparts assume that w is a quasimetric. Specifically, they rely on the following fact proved in Section 3.5.

Fact 3.2.6. *Consider a string X and its fragment $X[i\,..\,j)$. Then, for every quasimetric w, we have* $\mathsf{ed}^w(X, X[i\,..\,j)) = \mathsf{ed}^w(X[0\,..\,i) \cdot X[j\,..\,|X|), \varepsilon)$.

While our main results are on the weighted version of edit distance, our algorithm relies on unweighted edit distance procedures as well. If w is the discrete metric on $\bar{\Sigma}$ (that is, for every $a, b \in \bar{\Sigma}$, we have $w(a, b) = 0$ if $a = b$ and $w(a, b) = 1$ otherwise), then we drop the superscript w in ed^w and $\mathsf{ed}^w_{\mathcal{A}}$. This yields the unit-cost edit distance (also known as the unweighted edit distance or the Levenshtein distance). We consider weight function w to be *normalized* that is $w(a, b) \geq 1$ holds for all $a, b \in \bar{\Sigma}$ with $a \neq b$. In this case, $\mathsf{ed}^w_{\mathcal{A}}(X, Y) \geq \mathsf{ed}_{\mathcal{A}}(X, Y)$ holds for all strings $X, Y \in \Sigma^*$ and alignments $\mathcal{A} \in \mathsf{A}(X, Y)$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^m \in \mathsf{A}(X, Y)$, for every $\ell, r \in [0\,..\,m]$ with $\ell \leq r$, we say that $\mathcal{A}$ *aligns* $X[x_\ell\,..\,x_r)$ to $Y[y_\ell\,..\,y_r)$, denoted $X[x_\ell\,..\,x_r) \sim_{\mathcal{A}} Y[y_\ell\,..\,y_r)$. In this case, for any weight function w, we write $\mathsf{ed}^w_{\mathcal{A}}(X[x_\ell\,..\,x_r), Y[y_\ell\,..\,y_r))$ to denote the cost of the induced alignment of $X[x_\ell\,..\,x_r)$ onto $Y[y_\ell\,..\,y_r)$. If $\mathsf{ed}^w_{\mathcal{A}}(X[x_\ell\,..\,x_r), Y[y_\ell\,..\,y_r)) = 0$, we say that $\mathcal{A}$ *matches* $X[x_\ell\,..\,x_r)$ with $Y[y_\ell\,..\,y_r)$, denoted $X[x_\ell\,..\,x_r) \simeq_{\mathcal{A}} Y[y_\ell\,..\,y_r)$.

Fact 3.2.7. *Consider $k \in \mathbb{Z}_{\geq 0}$, strings X, Y, and an alignment $\mathcal{A} \in \mathsf{A}(X, Y)$ of cost $\mathsf{ed}_{\mathcal{A}}(X, Y) \leq k$. Then, the string X can be partitioned into at most k individual characters (that $\mathcal{A}$ deletes or substitutes) and at most $k + 1$ fragments that $\mathcal{A}$ matches perfectly to fragments of Y.*

Proof. Let $\mathcal{A} = (x_t, y_t)_{t=0}^m$ and let $t_1 < \cdots < t_e$ be the indices in $[0\,..\,m)$ corresponding to edits in $\mathcal{A}$. Then, the maximal fragments that $\mathcal{A}$ matches perfectly are $X[0\,..\,x_{t_1})$, $X[x_{t_i+1}\,..\,x_{t_{i+1}})$ for $i \in [1\,..\,e)$, and $X[x_{t_e+1}\,..\,|X|)$. Moreover, $\mathcal{A}$ deletes or substitutes $X[x_{t_i}]$ for every $i \in [1\,..\,e]$ such that $x_{t_i+1} > x_{t_i}$. Each edit contributes one unit to the cost of $\mathcal{A}$, so the decomposition contains at most $e \leq k$ edited characters and $e + 1 \leq k + 1$ fragments matched perfectly. □

3.2.3 Combinatorial Foundations

Before giving our algorithms for weighted string edit distance, we discuss edit distance equivalent substrings, one of our main technical contributions.

Definition 3.2.8. For $k \in \mathbb{Z}_{\geq 0}$ and a weight function w, strings P, P' are called $\mathsf{ed}^w_{\leq k}$-*equivalent* if

$$\mathsf{ed}^w_{\leq k}(X, Y) = \mathsf{ed}^w_{\leq k}(X[0\,..\,p_X) \cdot P' \cdot X[p_X + |P|\,..\,|X|), Y[0\,..\,p_Y) \cdot P' \cdot Y[p_Y + |P|\,..\,|Y|))$$

holds for all strings X and Y in which P occurs at positions p_X and p_Y, respectively, satisfying $|p_X - p_Y| \leq k$. We say that such occurrences of P in X and Y are k-*synchronized occurrences*.

First, we prove that changing the power of a periodic substring does not change the edit distance cost of any synchronized occurrences of that substring. Second, we prove that we only need to consider the small prefixes and suffixes of substrings when calculating the edit distance. In both cases, we are able to show that any optimal alignment must align such large substrings (first periodic and then non-periodic) that have synchronized occurrences in edit distance instances, and so, we do not have to worry about most of these large substrings when calculating edit distance. See Fig. 3.1 and Fig. 3.2 for examples of these two edit distance equivalent steps.

Lemma 3.2.9. *Let $k \in \mathbb{Z}_+$, let Q be a string, and let $e, e' \in \mathbb{Z}_{\geq 4k}$. Then, Q^e and $Q^{e'}$ are $\mathsf{ed}^w_{\leq k}$-equivalent for every weight function w.*

Proof. We assume without loss of generality that Q is primitive. (If $Q = R^m$ for $m \in \mathbb{Z}_{\geq 2}$, then $Q^e = R^{me}$ and $Q^{e'} = R^{me'}$ can be interpreted as powers of R rather than powers of Q.) Suppose that Q^e occurs in strings X and Y at positions p_X and p_Y, respectively, satisfying $|p_X - p_Y| \leq k$.

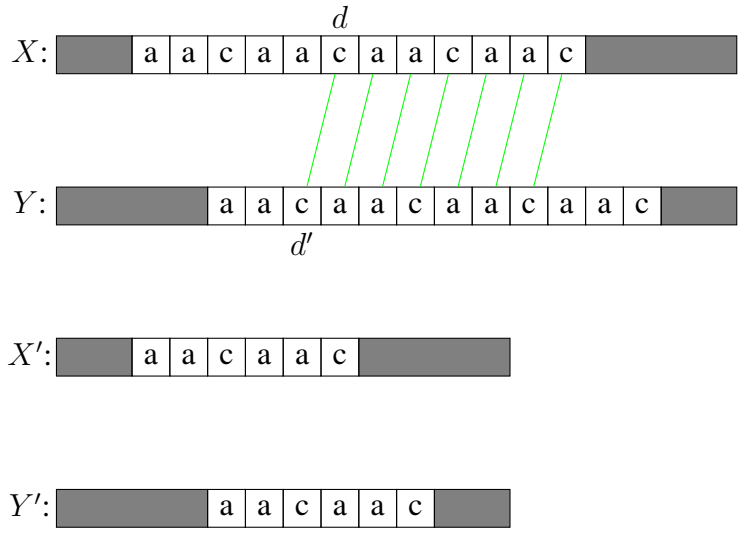


Figure 3.1: Periodicity reduction in X and Y with an optimal alignment depicted by lines connecting characters of the two strings. At indices d, d' in X and Y respectively, the periodic substring is fully aligned (depicted by green lines), and so, we may reduce the power of these periodic substrings to construct X' and Y' with $\mathsf{ed}(X, Y) = \mathsf{ed}(X', Y')$.

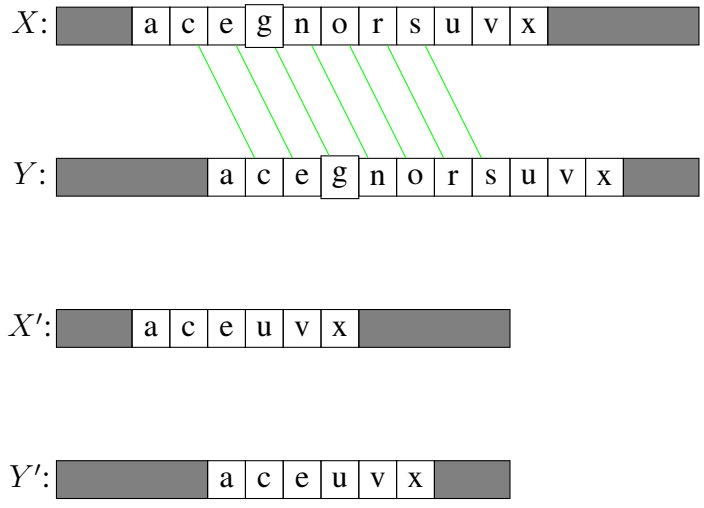


Figure 3.2: For any synchronized occurrences of a substring P that avoids k-periodicity, any optimal alignment (depicted by lines connecting characters of the two strings) must match most of the inner characters of P (see green lines). We can construct strings X', Y' removing these matched characters such that $\mathsf{ed}(X, Y) = \mathsf{ed}(X', Y')$.

Denote $X' = X[0\,..\,p_X) \cdot Q^{e'} \cdot X[p_X + |Q^e|\,..\,|X|)$ and $Y' = Y[0\,..\,p_Y) \cdot Q^{e'} \cdot Y[p_Y + |Q^e|\,..\,|Y|)$. Moreover, let $q = |Q|$ and let $\mathcal{A} \in \mathsf{A}(X, Y)$ be an alignment such that $\mathsf{ed}^w(X, Y) = \mathsf{ed}^w_{\mathcal{A}}(X, Y) \leq k$.

Claim 3.2.10. *There exist $i_X, i_Y \in [0\,..\,3k]$ such that*

$$X[p_X + i_X \cdot q\,..\,p_X + (i_X + 1) \cdot q) \simeq_{\mathcal{A}} Y[p_Y + i_Y \cdot q\,..\,p_Y + (i_Y + 1) \cdot q).$$

Proof. Let $(t_X, t_Y) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $t_X \geq p_X$ and $t_Y \geq p_Y$. By symmetry between X and Y, we assume without loss of generality that $t_X = p_X$. Consider the $k + 1$ occurrences of Q in X starting at positions $p_X + i \cdot q$ for $i \in [0\,..\,k]$. The alignment $\mathcal{A}$ matches at least one of them exactly; we can thus define $i_X \in [0\,..\,k]$ so that $\mathcal{A}$ matches $X[p_X + i_X \cdot q\,..\,p_X + (i_X + 1) \cdot q)$ exactly to some fragment $Y[s_Y\,..\,s_Y + q)$. Due to $(t_X, t_Y) \in \mathcal{A}$, the non-crossing property of $\mathcal{A}$ implies that $s_Y \geq t_Y \geq p_Y$. Moreover, since $\mathsf{ed}_{\mathcal{A}}(X, Y) \leq k$ and $X[p_X + i_X \cdot q] \simeq_{\mathcal{A}} Y[s_Y]$, we have $s_Y \leq (p_X + i_X \cdot q) + k \leq p_X + kq + k \leq p_Y + kq + 2k \leq p_Y + 3kq$. Furthermore, since Q is primitive (i.e., distinct from all its non-trivial cyclic rotations), we conclude that $s_Y = p_Y + i_Y \cdot q$ for some $i_Y \in [0\,..\,3k]$. □

Now, if $Q^e = X[p_X\,..\,p_X + e \cdot q) = Y[p_Y\,..\,p_Y + e \cdot q)$ is replaced with $Q^{e'}$ for $e' \geq e - 1$, we can interpret this as replacing $Q = X[p_X + i_X \cdot q\,..\,p_X + (i_X + 1) \cdot q) = Y[p_Y + i_Y \cdot q\,..\,p_Y + (i_Y + 1) \cdot q)$ with $Q^{1+e'-e}$. By Claim 3.2.10, $\mathcal{A}$ can be trivially adapted without modifying its cost, and hence $\mathsf{ed}^w(X', Y') \leq \mathsf{ed}^w_{\mathcal{A}}(X, Y) = \mathsf{ed}^w(X, Y)$. If $e' < e - 1$, we repeat the above argument to decrement the exponent e one step at a time, still concluding that $\mathsf{ed}^w(X', Y') \leq \mathsf{ed}^w(X, Y)$. In either case, the converse inequality follows by symmetry between (X, Y, e) and (X', Y', e'). □

We say that a string avoids k-periodicity if it does not contain any substring of the form Q^{4k+1} with $|Q| \in [1\,..\,2k]$.

Lemma 3.2.11. *Let $k \in \mathbb{Z}_+$ and let P, P' be strings of lengths at least $42k^3$ such that $P[0\,..\,21k^3) = P'[0\,..\,21k^3)$ and $P[|P| - 21k^3\,..\,|P|) = P'[|P'| - 21k^3\,..\,|P'|)$ avoid k-periodicity. Then, P and P' are $\mathsf{ed}^w_{\leq k}$-equivalent for every weight function w.*

Proof. Suppose that P occurs in strings X and Y at positions p_X and p_Y, respectively, satisfying $|p_X - p_Y| \leq k$. Denote $X' = X[0\,..\,p_X) \cdot P' \cdot X[p_X + |P|\,..\,|X|)$ and $Y' = Y[0\,..\,p_Y) \cdot P' \cdot Y[p_Y + |P|\,..\,|Y|)$. Moreover, let $\mathcal{A} \in \mathsf{A}(X, Y)$ be an alignment such that $\mathsf{ed}^w(X, Y) = \mathsf{ed}^w_{\mathcal{A}}(X, Y) \leq k$.

Claim 3.2.12. *There exist $d, e \in [0\,..\,21k^3]$ such that*

$$X[p_X + d\,..\,p_X + |P| - e) \sim_{\mathcal{A}} Y[p_Y + d\,..\,p_Y + |P| - e).$$

Proof. Let us partition $X[p_X\,..\,p_X + 21k^3)$ into individual characters representing deletions or substitutions of $\mathcal{A}$ and maximal fragments that $\mathcal{A}$ matches perfectly (to fragments of Y). By Fact 3.2.7, the number of such maximal fragments is at most $k + 1$ and their total length is at least $21k^3 - k \geq 20k^3$. Hence, one of these fragments is of length at least $\frac{20k^3}{k+1} \geq 10k^2$. Thus, let $R := X[r_X\,..\,r_X + |R|)$ be a fragment of length at least $10k^2$ contained in $X[p_X\,..\,p_X + 21k^3)$ that $\mathcal{A}$ matches perfectly to $Y[r_Y\,..\,r_Y + |R|)$. Moreover, let $r'_Y := r_X + p_Y - p_X$. If $r_Y = r'_Y$, then we set $d := r_X - p_X = r_Y - p_Y$ so that $(p_X + d, p_Y + d) \in \mathcal{A}$. Otherwise, both $Y[r_Y\,..\,r_Y + |R|)$ and $Y[r'_Y\,..\,r'_Y + |R|)$ are occurrences of R in Y. Moreover, $0 < |r_Y - r'_Y| \leq |r_Y - r_X| + |r'_Y - r_X| \leq \mathsf{ed}^w_{\mathcal{A}}(X, Y) + |p_Y - p_X| \leq 2k$. Hence, $\mathsf{per}(R) \leq |r_Y - r'_Y| \leq 2k$ and $\exp(R) \geq \frac{|R|}{2k} \geq 4k + 1$. Since $Y[r'_Y\,..\,r'_Y + |R|)$ is contained in $Y[p_Y\,..\,p_Y + 21k^3) = P[0\,..\,21k^3)$, this contradicts the

assumption about $P[0\,..\,21k^3)$ avoiding k-periodicity.

A symmetric argument shows that $(p_X+|P|-e, p_Y+|P|-e)$ holds for some $e \in [0\,..\,21k^3]$, which lets us conclude that $X[p_X + d\,..\,p_X + |P| - e) \sim_{\mathcal{A}} Y[p_Y + d\,..\,p_Y + |P| - e)$. □

By Claim 3.2.12, we have $X[p_X + d\,..\,p_X + |P| - e) \sim_{\mathcal{A}} Y[p_Y + d\,..\,p_Y + |P| - e)$. Both fragments match $P[d\,..\,|P| - e)$, so the optimality of $\mathcal{A}$ guarantees $X[p_X + d\,..\,p_X + |P| - e) \simeq_{\mathcal{A}} Y[p_Y + d\,..\,p_Y + |P| - e)$. Hence, if $P = X[p_X\,..\,p_X + |P|) = Y[p_Y\,..\,p_Y + |P|)$ is replaced with P', we can interpret this as $P[d\,..\,|P|-e) = X[p_X+d\,..\,p_X+|P|-e) = Y[p_Y+d\,..\,p_Y+|P|-e)$ with $P'[d\,..\,|P'|-e)$. Since $X[p_X+d\,..\,p_X+|P|-e) \simeq_{\mathcal{A}} Y[p_Y+d\,..\,p_Y+|P|-e)$, the alignment $\mathcal{A}$ can be trivially adapted without modifying its cost, and therefore $\mathsf{ed}^w(X', Y') \le \mathsf{ed}^w_{\mathcal{A}}(X, Y) = \mathsf{ed}^w(X, Y)$. The converse inequality follows by symmetry between (X, Y, P) and (X', Y', P'). □

3.2.4 Algorithm

The following lemma lets us transform any string P to a string P' that avoids k-periodicity and is $\mathsf{ed}^w_{\le k}$-equivalent to P for every weight function w. It is stated in a general form so that it can be reused in subsequent sections.

Lemma 3.2.13. *Let $e \in \mathbb{Z}_+$ and let $\mathcal{Q}$ be a family of primitive strings of length at most e. There is an algorithm that repeatedly transforms an input string P by replacing an occurrence of Q^{e+1} (for some $Q \in \mathcal{Q}$) with an occurrence of Q^e, arriving at a string P' that does not contain any occurrence of Q^{e+1} (for any $Q \in \mathcal{Q}$). Moreover, this algorithm can be implemented in linear time using a constant-time oracle that tests whether a given primitive fragment of P belongs to $\mathcal{Q}$.*

Proof. At preprocessing, we construct data structures for LCE and 2-Period queries in P; see Theorems 3.2.1 and 3.2.2. In the main phase, our algorithm scans the string P from left to right

Algorithm 9: Caps the exponent of every power of $Q \in \mathcal{Q}$ occurring in P to at most e.

```
PeriodicityReduction(P, e, 𝒬):
    R ← ε;
    r ← 0;
    while r < |P| do
        if r + 2e ≤ |P| and P[r..r + 2e) is periodic then q ← per(P[r..r + 2e));
        else q ← 1;
        m ← max{ℓ : P[r..r + ℓ) = P[r + q..r + q + ℓ)};
        if m ≥ eq and P[r..r + q) ∈ 𝒬 then
            r ← r + q;
        else
            R ← R · P[r];
            r ← r + 1;
    return R;
```

maintaining a string R and an index $r \in [0\,..\,|P|]$ such that $R \cdot P[r\,..\,|P|)$:

- is obtained from P by repeatedly replacing an occurrence of Q^{e+1} (for some $Q \in \mathcal{Q}$) with an occurrence of Q^e,
- does not contain any occurrence of Q^{e+1} (for $Q \in \mathcal{Q}$) starting at position smaller than $|R|$.

We initialize the process with $R := \varepsilon$ and $r := 0$. At each step, we test if $P[r\,..\,r+2q)$ is periodic; if so, we retrieve its shortest period q; otherwise, we set $q := 1$. Then, we further check whether $P[r\,..\,r+q) \in \mathcal{Q}$ and $P[r\,..\,r+eq) = P[r+q\,..\,r+q+eq)$. If both tests are successful, we move the index r to position $r + q$. Otherwise, we append $P[r]$ to R and increment r.

Let us analyze the correctness of this algorithm. First, suppose that $P[r\,..\,|P|)$ does not have a prefix of the form Q^{e+1} for any $Q \in \mathcal{Q}$. In particular, $P[r\,..\,r+q) \notin \mathcal{Q}$ or $m < eq$. Thus, our algorithm appends $P[r]$ to R and increments r. The invariant remains satisfied because $R \cdot P[r\,..\,|P|)$ did not change and $P[r\,..\,|P|)$ had no prefix of the form Q^{e+1} for any $Q \in \mathcal{Q}$.

Next, suppose that $P[r\,..\,|P|)$ has a prefix of the form Q^{e+1} for some $Q \in \mathcal{Q}$. If $|Q| \neq 1$, then $|Q|$ is the shortest period of $P[r\,..\,r+2e)$ because Q is primitive and $|Q| \leq e$. If $|Q| = 1$, on

the other hand, then $P[r\,..\,r+2e)$ either has period 1 or at least $e+2$. In all cases, the algorithm correctly identifies $q = |Q|$. Moreover, the subsequent tests whether $P[r\,..\,r+q)$ belongs to $\mathcal{Q}$ and $m \geq eq$ are successful. Hence, the algorithm transforms $R \cdot P[r\,..\,|P|)$ into $R \cdot P[r+q\,..\,|P|)$, which is a valid operation because the prefix Q^{e+1} of $P[r\,..\,|P|)$ is replaced with the prefix Q^e of $P[r+q\,..\,|P|)$. Thus, it remains to prove that $R \cdot P[r+q\,..\,|P|)$ does not contain any occurrence of $\hat{Q}^{e+1}$ (for any $\hat{Q} \in \mathcal{Q}$) starting at position smaller than $|R|$. Since $R \cdot P[r\,..\,|P|)$ did not contain such an occurrence, the occurrence of $\hat{Q}^{e+1}$ would need to end at position $|R|+m$ or larger. The fragment $P[r+q\,..\,r+q+m)$ thus has periods q and $\hat{q} := |\hat{Q}|$. Moreover, by primitivity of Q and $\hat{Q}$, the Periodicity Lemma [Fine and Wilf, 1965] implies $q = \hat{q}$ due to $m \geq eq \geq e+q-1 \geq \hat{q}+q-1$. However, this means that q is a period of $P[r\,..\,r+q+m]$, contradicting the definition of m.

The overall running time is linear, including the preprocessing and the query time of the data structures of Theorems 3.2.1 and 3.2.2, because each iteration of the **while** loop costs constant time. □

Corollary 3.2.14. *There exists a linear-time algorithm that, given a string P and an integer $k \in \mathbb{Z}_+$, constructs a string of length at most $42k^3$ that is $\mathsf{ed}^w_{\leq k}$-equivalent to P for every weight function w.*

Algorithm 10: Construct a string of length at most $42k^3$ that is $\mathsf{ed}^w_{\leq k}$-equivalent to P.

```
1 StringReduction(P, k):
2     P' ← PeriodicityReduction(P, 4k, {Q ∈ Σ+ : |Q| ≤
        2k and Q is primitive});
3     if |P'| ≥ 42k^3 then return P'[0..21k^3) · P'[|P'| − 21k^3..|P'|);
4     else return P';
```

Proof. We set $P' := \texttt{PeriodicityReduction}(P, 4k, \mathcal{Q})$ with $\mathcal{Q}$ consisting of all primitive strings of length in $[1\,..\,2k]$. We return $P'' := P'[0\,..\,21k^3) \cdot P'[|P'|-21k^3\,..\,|P'|)$ or P' de-

pending on whether $|P'| \geq 42k^3$ or not. By Lemmas 3.2.9 and 3.2.13, the string P' is $\mathsf{ed}^w_{\leq k}$-equivalent to P and avoids k-periodicity. Thus, if $|P'| \leq 42k^3$, then the algorithm is correct. Otherwise, Lemma 3.2.11 implies that P'' is $\mathsf{ed}^w_{\leq k}$-equivalent to P' (and, by transitivity, to P) because $P'[0\,..\,21k^3)$ and $P'[|P'| - 21k^3\,..\,|P'|)$ avoid k-periodicity. Due to Lemma 3.2.13, the running time is linear (a primitive fragment belongs to $\mathcal{Q}$ if and only if its length does not exceed $2k$, which takes $\mathcal{O}(1)$ time to test). □

Theorem 3.2.15. *There exists a linear-time algorithm that, given strings X, Y and an integer $k \in \mathbb{Z}_+$, constructs strings X', Y' of lengths at most $85k^4$ such that $\mathsf{ed}^w_{\leq k}(X, Y) = \mathsf{ed}^w_{\leq k}(X', Y')$ holds for every weight function w.*

Algorithm 11: Construct strings X', Y' of length at most $85k^4$ such that $\mathsf{ed}^w_{\leq k}(X, Y) = \mathsf{ed}^w_{\leq k}(X', Y')$

```
1  StringKernel(X, Y, k):
2      if |X| ≤ 85k^4 and |Y| ≤ 85k^4 then return (X, Y);
3      if ed(X, Y) > k then return (a^{k+1}, ε) for some a ∈ Σ;
4      Let (x_t, y_t)_{t=0}^m ∈ A(X, Y) be an alignment satisfying ed_A(X, Y) ≤ k;
5      X', Y', P ← ε;
6      for t ← 0 to m do
7          if t < m and x_{t+1} > x_t and y_{t+1} > y_t and X[x_t] = Y[y_t] then
8              P ← P · X[x_t]
9          else
10             P ← StringReduction(P, k);
11             X' ← X' · P;
12             Y' ← Y' · P;
13             P ← ε;
14             if t < m and x_{t+1} > x_t then X' ← X' · X[x_t];
15             if t < m and y_{t+1} > y_t then Y' ← Y' · Y[y_t];
16     return (X', Y')
```

Proof. Our procedure is implemented as Algorithm 11. First, if X and Y are already of length at most $85k^4$, then we return X and Y unchanged. If $\mathsf{ed}(X, Y) > k$, we return strings a^{k+1} and

ε, where $a \in \Sigma$ is an arbitrary character. If $\mathsf{ed}(X, Y) \leq k$, we construct an alignment $\mathcal{A} := (x_t, y_t)_{t=0}^{m} \in \mathsf{A}(X, Y)$ of (unweighted) cost at most k. We then build the output strings X' and Y' during a left-to-right scan of the alignment $\mathcal{A}$: We append to X' and Y' every character of X and Y (respectively) that $\mathcal{A}$ edits. Moreover, for every pair of maximal fragments in X and Y that $\mathcal{A}$ matches perfectly, we apply the reduction of Corollary 3.2.14 and append the resulting string to both X' and Y'.

Let us now prove that the resulting instance (X', Y') satisfies $\mathsf{ed}^{w}_{\leq k}(X, Y) = \mathsf{ed}^{w}_{\leq k}(X', Y')$. This is trivial when the algorithm returns (X, Y) in Algorithm 11. If $\mathsf{ed}(X, Y) > k$, then $\mathsf{ed}_{\leq k}(X, Y) = \infty = \mathsf{ed}_{\leq k}(a^{k+1}, \varepsilon)$ and thus also $\mathsf{ed}^{w}_{\leq k}(X, Y) = \infty = \mathsf{ed}^{w}_{\leq k}(a^{k+1}, \varepsilon)$ because the weighted edit distance with a normalized weight function is at least as large as the unweighted edit distance. In the remaining case of $\mathsf{ed}(X, Y) \leq k$, we maintain an invariant that $|X'| - |Y'| = x_t - y_t$ and $\mathsf{ed}^{w}_{\leq k}(X, Y) = \mathsf{ed}^{w}_{\leq k}(X' \cdot P \cdot X[x_t \mathinner{..} x_m), Y' \cdot P \cdot Y[y_t \mathinner{..} y_m))$ hold at the beginning of every iteration of the **for** loop as well as after every execution of Line 10 and Line 13. It is easy to see that the strings $X' \cdot P \cdot X[x_t \mathinner{..} x_m)$ and $Y' \cdot P \cdot Y[y_t \mathinner{..} y_m)$ change only at Line 10, when P is replaced with $\texttt{StringReduction}(P, k)$. The correctness of this step follows directly from the definition of $\mathsf{ed}^{w}_{\leq k}$-equivalence (Definition 3.2.8) since $\texttt{StringReduction}(P, k)$ is $\mathsf{ed}^{w}_{\leq k}$-equivalent to P.

Next, we show that the returned strings are of length at most $85k^4$. This is clear when the algorithm terminates at Line 2 or 3. Otherwise, we apply Fact 3.2.7 to observe that X is decomposed into at most k characters that $\mathcal{A}$ deletes or substitutes (which are copied to X') and at most $k + 1$ maximal fragments that $\mathcal{A}$ matches perfectly to fragments of Y (which are copied to X' after applying $\texttt{StringReduction}$). By the guarantee of Corollary 3.2.14, we conclude that $|X'| \leq k + (k + 1) \cdot 42k^3 \leq 85k^4$. Symmetrically, we have $|Y'| \leq 85k^4$.

It remains to analyze the time complexity of our procedure. We use the Landau–Vishkin

algorithm [Landau and Vishkin, 1988] to check whether $\mathsf{ed}(X, Y) \leq k$ and, if so, construct the alignment $\mathcal{A}$. This costs $\mathcal{O}(n + k^2)$ time, which is $\mathcal{O}(n)$ because we perform this step only if $n \geq k^4 \geq k^2$. The scan of the alignment $\mathcal{A}$ takes $\mathcal{O}(m) = \mathcal{O}(n)$ time, including the applications of Corollary 3.2.14, which operate on strings of total length at most n. □

Having reduced the string lengths to $\mathcal{O}(k^4)$, we can use the classic dynamic programming [Wagner and Fischer, 1974] to compute $\mathsf{ed}^w_{\leq k}(X, Y)$ in $\mathcal{O}(k^8)$ time. However, since w is a normalized, the running of [Wagner and Fischer, 1974] can be reduced to $\mathcal{O}(nk)$. For completeness, we describe this improvement below.

Proposition 3.2.16. *Given strings X, Y of length at most n, an integer $k \in \mathbb{Z}_+$, and a weight function w, the value $\mathsf{ed}^w_{\leq k}(X, Y)$ can be computed in $\mathcal{O}(nk)$ time.*

Proof. Recall that the algorithm of [Wagner and Fischer, 1974] maintains a table $D[0\,..\,|X|, 0\,..\,|Y|]$ such that $D[i, j] = \mathsf{ed}^w(X[0\,..\,i), Y[0\,..\,j))$ holds for each $i \in [0\,..\,|X|]$ and $j \in [0\,..\,|Y|]$. We have $D[0, 0] = 0$, whereas the remaining entries are constructed in $\mathcal{O}(1)$ time each using the following formula:

$$D[i, j] = \min \begin{cases} D[i-1, j] + w(X[i-1], \varepsilon) & \text{if } i > 0, \\ D[i, j-1] + w(\varepsilon, Y[j-1]) & \text{if } j > 0, \\ D[i-1, j-1] + w(X[i-1], Y[j-1]) & \text{if } i, j > 0. \end{cases} \tag{3.1}$$

In order to compute $\mathsf{ed}^w_{\leq k}(X, Y)$, we use a modified table $D'[0\,..\,|X|, 0\,..\,|Y|]$ such that $D'[0, 0] = 0$, $D'[i, j] = \infty$ if $|i - j| > k$, whereas the remaining entries are computed using (3.1) (with D replaced by D'). A straightforward inductive argument shows that $D'[i, j] \geq D[i, j]$ holds for all $i \in [0\,..\,|X|]$ and $j \in [0\,..\,|Y|]$ and, moreover, $D[i, j] \leq k$ implies $D'[i, j] = D[i, j]$. For $|i - j| > k$, this is true because w is normalized and thus $D[i, j] = \mathsf{ed}^w(X[0\,..\,i), Y[0\,..\,j)) \geq$

$\mathsf{ed}(X[0\,..\,i), Y[0\,..\,j)) \geq |i-j| > k$. For $|i-j| \leq k$, on the other hand, the argument is based on the inductive hypothesis and the fact that the weight function w has non-negative values. The entries $D'[i,j] = \infty$ for $|i-j| > k$ can be set implicitly, which reduces the running time to $\mathcal{O}(nk)$. □

Theorem 3.1.1. *Given strings X, Y of length at most n, an integer $k \in \mathbb{Z}_+$, and a weight function w, the value $\mathsf{ed}^w_{\leq k}(X,Y)$ can be computed in $\mathcal{O}(n+k^5)$ time.*

Proof. We first apply Theorem 3.2.15 to build strings X', Y' of length $\mathcal{O}(k^4)$ such that $\mathsf{ed}^w_{\leq k}(X',Y') = \mathsf{ed}^w_{\leq k}(X,Y)$. Then, we compute $\mathsf{ed}^w_{\leq k}(X',Y')$ using Proposition 3.2.16. The running times of these two steps are $\mathcal{O}(n)$ and $\mathcal{O}(k^4 \cdot k) = \mathcal{O}(k^5)$, respectively, for a total of $\mathcal{O}(n+k^5)$. □

3.3 Tree Edit Distance

3.3.1 Preliminaries

For an alphabet Σ, we define a set $\mathsf{P}_\Sigma := \bigcup_{a\in\Sigma}\{(_a,)_a\}$ of parentheses with labels over Σ. A *forest* with node labels over Σ is a *balanced* string of parentheses over Σ. Formally, the set of forests with labels over Σ is defined as the smallest subset $\mathcal{F}_\Sigma \subseteq \mathsf{P}^*_\Sigma$ satisfying the following conditions:

- $\varepsilon \in \mathcal{F}_\Sigma$,
- $F \cdot G \in \mathcal{F}_\Sigma$ for every $F, G \in \mathcal{F}_\Sigma$,
- $(_a \cdot F \cdot)_a \in \mathcal{F}_\Sigma$ for every $F \in \mathcal{F}_\Sigma$ and $a \in \Sigma$.

For a forest F, we define the set of *nodes* V_F as the set of pairs $(i,j) \in [0\,..\,|F|)$ such that $F[i]$ is an opening parenthesis, $F[j]$ is a closing parenthesis, and $F[i\,..\,j]$ is balanced. For a node

$u = (i, j) \in V_F$, we denote the positions of the opening and the closing parenthesis by $o(u) := i$ and $c(u) := j$. A forest F is a tree if $(0, |F| - 1) \in V_F$.

Fact 3.3.1. *A forest F can be preprocessed in linear time so that one can test in constant time whether any given fragment $F[i\,..\,j)$ is balanced.*

Proof. Let us define the height function $H : [0\,..\,|F|] \to \mathbb{Z}$ so that $H(i)$ equals the number of opening parentheses in $F[0\,..\,i)$ minus the number of closing parentheses in $F[0\,..\,i)$. Since F is balanced, the fragment $F[i\,..\,j)$ is balanced if and only if $H(i) = H(j) = \min_{m \in [i\,..\,j]} H(m)$. This condition can be tested in $\mathcal{O}(1)$ time after linear-time preprocessing using range minimum queries (RMQ) [Harel and Tarjan, 1984]. □

A context with node labels over Σ is a pair $C = \langle C_L; C_R \rangle \in \mathsf{P}_\Sigma \times \mathsf{P}_\Sigma$ such that $C_L \cdot C_R$ is a tree. The node set V_C of a context C is identified with the node set of the underlying tree $C_L \cdot C_R$. The *depth* of a context C is the number of nodes $u \in V_C$ whose opening parenthesis belongs to C_L and closing parenthesis belongs to C_R, that is, $o(u) < |C_L| < c(u)$.

The (vertical) composition of contexts C, D results in a context $C \star D := \langle C_L \cdot D_L; D_R \cdot C_R \rangle$. Moreover, vertical composition of a context C and a forest F results in a tree $C \star F := C_L \cdot F \cdot C_R$. A context C is primitive if it cannot be expressed as vertical composition of $e \geq 2$ copies of the same context.

A context C *occurs* in a forest F at node $u \in V_F$ if $C_L = F[o(u)\,..\,o(u) + |C_L|)$ and $C_R = F(c(u) - |C_R|\,..\,c(u)]$, or equivalently, $F[o(u)\,..\,c(u)] = C \star G$ for some forest G

3.3.2 Forest Alignments and Weighted Forest Edit Distance

We begin our discussion of weighted tree edit distance by formally defining forest alignments, which are similar to alignments on strings with just a few additional restrictions to make sure the alignments make valid edits on forests.

Definition 3.3.2. We say that an alignment $\mathcal{A} \in \mathsf{A}(F, G)$ is a *forest alignment* of forests F and G if the following *consistency* conditions are satisfied for each $u \in V_F$:

- either $\mathcal{A}$ deletes both $F[o(u)]$ and $F[c(u)]$, or
- there exists $v \in V_G$ such that $F[o(u)] \sim_{\mathcal{A}} G[o(v)]$ and $F[c(u)] \sim_{\mathcal{A}} G[c(v)]$.

The set of all forests alignments of F onto G is denoted with $\mathsf{TA}(F, G) \subseteq \mathsf{A}(F, G)$.

Define $\overline{\mathsf{P}_\Sigma} = \mathsf{P}_\Sigma \cup \varepsilon$ and a mapping $\lambda : \overline{\mathsf{P}_\Sigma} \to \bar{\Sigma}$ such that $\lambda(\,(_a) = \lambda()_a) = a$ for each $a \in \Sigma$, and $\lambda(\varepsilon) = \varepsilon$. For a weight function $w : \bar{\Sigma} \times \bar{\Sigma} \to \mathbb{R}_{\geq 0}$, we define a corresponding weight function $\tilde{w} : \overline{\mathsf{P}_\Sigma} \times \overline{\mathsf{P}_\Sigma} \to \mathbb{R}_{\geq 0}$ so that $\tilde{w}(p, q) = w(\lambda(p), \lambda(q))$ for all $p, q \in \overline{\mathsf{P}_\Sigma}$. The *cost* of a forest alignment $\mathcal{A} \in \mathsf{TA}(F, G)$ with respect to a weight function w is defined as $\mathsf{ted}^w_{\mathcal{A}}(F, G) := \frac{1}{2}\mathsf{ed}^{\tilde{w}}_{\mathcal{A}}(F, G)$. Moreover, for any two forests F, G, we define the *weighted tree edit distance* $\mathsf{ted}^w(F, G) = \min_{\mathcal{A} \in \mathsf{TA}(F,G)} \mathsf{ted}^w_{\mathcal{A}}(F, G)$, and for a threshold $k \in \mathbb{R}_{\geq} 0$, we set

$$\mathsf{ted}^w_{\leq k}(F, G) = \begin{cases} \mathsf{ted}^w(F, G) & \text{if } \mathsf{ted}^w(F, G) \leq k, \\ \infty & \text{otherwise.} \end{cases}$$

The superscript is omitted if w is the discrete metric over $\bar{\Sigma}$.

Fact 3.3.3. *If w is a quasimetric on $\bar{\Sigma}$, then ted^w is a quasimetric on $\mathcal{F}_\Sigma$. In that case, $\mathsf{ted}^w(F, G)$*

can be equivalently defined as the minimum cost of a sequence of edits transforming F into G, where inserting a node with label b costs $w(\varepsilon, b)$, deleting a node with label a costs $w(a, \varepsilon)$, and changing a node label from a to b costs $w(a, b)$.

Proof. Consider arbitrary forests $F, G, H \in \mathcal{F}_\Sigma$ as well as alignments $A = (x_t, y_t)_{t=0}^{m} \in \mathsf{TA}(F, G)$ and $B = (\hat{y}_t, \hat{z}_t)_{t=0}^{\hat{m}} \in \mathsf{TA}(G, H)$. We can construct the product alignment $\mathcal{A} \otimes \mathcal{B}$ as in the proof of Fact 3.2.5, which has $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(F, H) \leq \mathsf{ed}^w_{\mathcal{A}}(F, G) + \mathsf{ed}^w_{\mathcal{B}}(G, H)$. Therefore, it remains to prove that $\mathcal{A} \otimes \mathcal{B}$ is a tree alignment.

Consider an arbitrary node $u_F \in V_F$. If $\mathcal{A}$ deletes u_F (that is, it deletes both characters $F[o(u_F)]$ and $F[c(u_F)]$), then $\mathcal{A} \otimes \mathcal{B}$ also deletes u_F; see Case 2 in the recursive definition of $\mathcal{A} \otimes \mathcal{B}$. The other possibility is that $\mathcal{A}$ aligns u_F with some node $u_G \in V_G$ (that is, it aligns $F[o(u_F)]$ with $G[o(u_G)]$ and $F[c(u_F)]$ with $G[c(u_G)]$). If $\mathcal{B}$ deletes u_G, then $\mathcal{A} \otimes \mathcal{B}$ deletes u_F; see Case 6 in the recursive definition of $\mathcal{A} \otimes \mathcal{B}$. Finally, if $\mathcal{B}$ aligns u_G with some node $u_H \in V_H$, then $\mathcal{A} \otimes \mathcal{B}$ aligns u_F with u_H; see Case 7 in the recursive definition of $\mathcal{A} \otimes \mathcal{B}$. □

3.3.3 Combinatorial Foundations

3.3.3.1 Forests

Similar to our discussion of weighted string edit distance, before giving our tree edit distance algorithms we prove the existence of small edit distance equivalent forests for synchronized occurrences of large subforests in the input instance forests.

Definition 3.3.4. For $k \in \mathbb{Z}_{\geq 0}$ and a weight function w, forests P, P' are called $\mathsf{ted}^w_{\leq k}$*-equivalent*

if

$$\mathsf{ted}^w_{\le k}(F,G) = \mathsf{ted}^w_{\le k}(F[0\,..\,p_F)\cdot P'\cdot F[p_F+|P|\,..\,|F|), G[0\,..\,p_G)\cdot P'\cdot G[p_G+|P|\,..\,|G|))$$

holds for all forests F and G in which P occurs at positions p_F and p_G, respectively, satisfying $|p_F - p_G| \le 2k$.

Lemma 3.3.5. *Let $k \in \mathbb{Z}_+$, let Q be a forest, and let $e, e' \in \mathbb{Z}_{\ge 4k}$. Then, Q^e and $Q^{e'}$ are $\mathsf{ted}^w_{\le k}$-equivalent for every normalized weight function w.*

Proof. We assume without loss of generality that Q is primitive. (If $Q = R^m$ for $m \in \mathbb{Z}_{\ge 2}$, then $Q^e = R^{me}$ and $Q^{e'} = R^{me'}$ can be interpreted as powers of R rather than powers of Q.) Suppose that Q^e occurs in forests F and G at positions p_F and p_G, respectively, satisfying $|p_F - p_G| \le 2k$. Denote $F' = F[0\,..\,p_F)\cdot Q^{e'}\cdot F[p_F+|Q^e|\,..\,|F|)$ and $G' = G[0\,..\,p_G)\cdot Q^{e'}\cdot G[p_G+|Q^e|\,..\,|G|)$. Moreover, let $q = |Q|$ and let $\mathcal{A}$ be a forest alignment such that $\mathsf{ted}^w(F,G) = \mathsf{ted}^w_{\mathcal{A}}(F,G) \le k$.

Claim 3.3.6. *There exist $i_F, i_G \in [0\,..\,3k]$ such that*

$$F[p_F + i_F\cdot q\,..\,p_F + (i_F+1)\cdot q) \simeq_{\mathcal{A}} G[p_G + i_G\cdot q\,..\,p_G + (i_G+1)\cdot q).$$

Proof. Let $(f_b, g_b) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_b \ge p_F$ and $g_b \ge p_G$. By symmetry between F and G, we assume without loss of generality that $f_b = p_F$. Consider the $k+1$ occurrences of Q in F starting at positions $p_F + i\cdot q$ for $i \in [0\,..\,k]$. Since Q is balanced, the alignment $\mathcal{A}$ (of unweighted cost at most k) matches at least one of them exactly; we can thus define $i_F \in [0\,..\,k]$ so that $\mathcal{A}$ matches $F[p_F + i_F\cdot q\,..\,p_F + (i_F+1)\cdot q)$ exactly to some fragment $G[g_a\,..\,g_a + q)$. By definition of b, we have $a \ge b$ and thus $g_a \ge g_b \ge p_G$. Moreover, since

$\mathsf{ted}_{\mathcal{A}}(F,G) \le k$ and $F[p_F + i_F \cdot q] \sim_{\mathcal{A}} G[g_a]$, we have $g_a \le (p_F + i_F \cdot q) + 2k \le p_F + kq + 2k \le p_G + kq + 4k \le p_G + 3kq$, where the last inequality follows from $q \ge 2$ (recall that Q is balanced, so its length is even). Furthermore, since Q is primitive (i.e., distinct from all its non-trivial cyclic rotations), we conclude that $g_a = p_G + i_G \cdot q$ for some $i_G \in [0\,..\,3k]$. □

Now, if $Q^e = F[p_F\,..\,p_F + e \cdot q) = G[p_G\,..\,p_G + e \cdot q)$ is replaced with $Q^{e'}$ for $e' \ge e - 1$, we can interpret this as replacing $Q = F[p_F + i_F \cdot q\,..\,p_F + (i_F + 1) \cdot q) = G[p_G + i_G \cdot q\,..\,p_G + (i_G + 1) \cdot q)$ with $Q^{1+e'-e}$. By Claim 3.3.6, $\mathcal{A}$ can be trivially adapted without modifying its cost, and hence $\mathsf{ted}^w(F',G') \le \mathsf{ted}^w_{\mathcal{A}}(F,G) = \mathsf{ted}^w(F,G)$. If $e' < e - 1$, we repeat the above argument to decrement the exponent one step at a time, still concluding that $\mathsf{ted}^w(F',G') \le \mathsf{ted}^w(F,G)$. In either case, the converse inequality follows by symmetry between (F,G,e) and (F',G',e'). □

We say that a forest F avoids horizontal k-periodicity if there is no forest Q of length $|Q| \in [1\,..\,4k]$ such that Q^{4k+1} occurs in F.

Lemma 3.3.7. *Let $k \in \mathbb{Z}_+$ and let P, P' be forests of length $|P|, |P'| \ge 74k^3$ avoiding horizontal k-periodicity. Then, P and P' are $\mathsf{ted}^w_{\le k}$-equivalent for every normalized quasimetric w.*

Proof. Suppose that P occurs in forests F and G at positions p_F and p_G, respectively, satisfying $|p_F - p_G| \le 2k$. Denote $F' = F[0\,..\,p_F) \cdot P' \cdot F[p_F + |P|\,..\,|F|)$ and $G' = G[0\,..\,p_G) \cdot P' \cdot G[p_G + |P|\,..\,|G|)$.

Let $\mathcal{A} = (f_t, g_t)_{t=0}^m$ be an alignment such that $\mathsf{ted}^w(F,G) = \mathsf{ted}^w_{\mathcal{A}}(F,G) \le k$. Moreover, let $(f_a, g_a) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_a \ge p_F$ or $g_a \ge p_G$, and let $(f_b, g_b) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_b \ge p_F + |P|$ and $g_b \ge p_G + |P|$. We construct an alignment $\mathcal{A}'$ so that it:

- aligns $F[0\,..\,f_a)$ with $G[0\,..\,g_a)$ in the same way as $\mathcal{A}$ does;
- deletes $F[f_a\,..\,p_F)$ and inserts $G[g_a\,..\,p_G)$ (at least one of these fragments is empty);
- matches $F[p_F\,..\,p_F+|P|) = P$ with $G[p_G\,..\,p_G+|P|) = P$;
- deletes $F[p_F+|P|\,..\,f_b)$ and inserts $G[p_G+|P|\,..\,g_b)$ (at least one of these fragments is empty);
- aligns $F[f_b\,..\,|F|)$ with $G[g_b\,..\,|G|)$ in the same way as $\mathcal{A}$ does.

To prove that $\mathcal{A}'$ is a forest alignment, let us consider several possibilities for a node u in F.

- If u is inside $P = F[p_F\,..\,p_F+|P|)$, then $\mathcal{A}'$ matches u to the corresponding node inside $P = G[p_G\,..\,p_G+|P|)$.
- If u is outside $P = F[p_F\,..\,p_F+|P|)$ and $\mathcal{A}$ aligns u to a node v of G inside $P = G[p_G\,..\,p_G+|P|)$, then $o(u), c(u) \in [f_a\,..\,p_F) \cup [p_F+|P|\,..\,f_b)$ because of the non-crossing property of $\mathcal{A} \ni (f_a, g_a), (f_b, g_b)$. Hence, $\mathcal{A}'$ deletes u.
- If u is outside $P = F[p_F\,..\,p_F+|P|)$ and $\mathcal{A}$ aligns u to a node v of G outside $P = G[p_G\,..\,p_G+|P|)$, then $o(u), c(u) \in [0\,..\,f_a) \cup [f_b\,..\,|F|)$ because of the non-crossing property of $\mathcal{A} \ni (f_a, g_a), (f_b, g_b)$. Hence, $\mathcal{A}'$ also aligns u to v.
- If u is outside $P = F[p_F\,..\,p_F+|P|)$ and $\mathcal{A}$ deletes u, then $\mathcal{A}'$ also deletes u.

Our next goal is to prove that $\mathsf{ted}^w_{\mathcal{A}'}(F,G) \le \mathsf{ted}^w_{\mathcal{A}}(F,G)$. This relies on the following claim.

Claim 3.3.8. *There exists* $t \in [a\,..\,b]$ *such that* $f_t - g_t = p_F - p_G$.

Proof. Let us partition $P = F[p_F\,..\,p_F+|P|)$ into individual characters representing deletions or substitutions of $\mathcal{A}$ and maximal fragments that $\mathcal{A}$ matches perfectly (to fragments of G). By Fact 3.2.7, the number of such fragments is at most $2k+1$ and their total length is at least $|P|-2k$.

Hence, one of these fragments, denoted $R = F[r_F \,..\, r_F + |R|)$, is of length at least $\frac{|P|-2k}{2k+1} \geq 24k^2$. Suppose that the fragment of G matched perfectly to R is $G[r_G \,..\, r_G + |R|)$. If $r_F - p_F = r_G - p_G$, the claim holds for t such that $(f_t, g_t) = (r_F, r_G)$. Otherwise, we note that R has period $q := |(r_F - p_F) - (r_G - p_G)| \in [1 \,..\, 4k]$. Let $q = R[0 \,..\, q)$ and observe that $\frac{|R|}{q} \geq \frac{24k^2}{4k} \geq 4k + 2$. Hence, Q^{4k+2} is a substring of P; since P avoids horizontal k-periodicity, we conclude that no cyclic rotation of Q is balanced.

As Q is a substring of a balanced string P, this means that the number of opening parentheses in Q does not match the number of closing parentheses in Q. By symmetry (up to reversal), we assume without loss of generality that Q has more opening than closing parentheses. Thus, there exists a node u in F such that $o(u) \in [r_F \,..\, r_F + q)$ yet $c(u) \geq r_F + |R|$. In particular, $c(u) - o(u) \geq |R| - q \geq 24k^2 - 4k > 8k$. Let v, v' be the nodes in G matched with u by $\mathcal{A}$ and $\mathcal{A}'$, respectively. Note that $|o(v) - o(v')| \leq 4k$ and $|c(v) - c(v')| \leq 4k$. Due to $c(v') - o(v') = c(u) - o(u) > 8k$, we conclude that v is ancestor of v' or vice versa. In either case, we have $0 \geq (o(v') - o(v)) \cdot (c(v') - c(v)) = ((o(u) - o(v)) - (p_F - p_G)) \cdot ((c(u) - c(v)) - (p_F - p_G))$. The value $(f_t - g_t) - (p_F - p_G)$ can change by at most one for subsequent indices t. The sign of this value is different when $(f_t, g_t) = (o(u), o(v))$ and $(f_t, g_t) = (c(u), c(v))$, so it must be equal to 0 at some intermediate index t. □

The alignments $\mathcal{A}$ and $\mathcal{A}'$ only differ in how they align $F[f_a \,..\, f_t)$ with $G[g_a \,..\, g_t)$ and $F[f_t \,..\, f_b)$ with $G[g_t \,..\, g_b)$, and, by Fact 3.2.6, $\mathcal{A}'$ provides an optimum alignment of these fragments. Now, if $P = F[p_F \,..\, p_F + |P|) = G[p_G \,..\, p_G + |P|)$ is modified to P', then $\mathcal{A}'$ can be trivially adapted without modifying its cost and hence $\mathsf{ted}^w(F', G') \leq \mathsf{ted}^w_{\mathcal{A}'}(F, G) = \mathsf{ted}^w(F, G)$. The converse inequality follows by symmetry between (F, G, P) and (F', G', P'). □

3.3.3.2 Contexts

Definition 3.3.9. For $k \in \mathbb{Z}_{\geq 0}$ and a weight function w, contexts $P = \langle P_L; P_R \rangle$ and $P' = \langle P'_L; P'_R \rangle$ are called $\mathsf{ted}^w_{\leq k}$-*equivalent* if

$$\mathsf{ted}^w_{\leq k}(F, G) = \mathsf{ted}^w_{\leq k}(F[0\,..\,o(u)) \cdot P'_L \cdot F[o(u) + |P_L|\,..\,c(u) - |P_R|] \cdot P'_R \cdot F(c(u)\,..\,|F|),$$
$$G[0\,..\,o(v)) \cdot P'_L \cdot G[o(v) + |P_L|\,..\,c(u) - |P_R|] \cdot P'_R \cdot G(c(v)\,..\,|G|))$$

holds for all forests F and G in which P occurs at nodes u and v, respectively, satisfying $|o(u) - o(v)| \leq 2k$ and $|c(u) - c(v)| \leq 2k$.

Lemma 3.3.10. *Let $k \in \mathbb{Z}_+$, let Q be a context, and let $e, e' \in \mathbb{Z}_{\geq 6k}$. Then, Q^e and $Q^{e'}$ are $\mathsf{ted}^w_{\leq k}$-equivalent for every normalized weight function w.*

Proof. We assume without loss of generality that Q is primitive. (If $Q = R^m$ for $m \in \mathbb{Z}_{\geq 2}$, then $Q^e = R^{me}$ and $Q^{e'} = R^{me'}$ can be interpreted as powers of R rather than powers of Q.) Let $Q = \langle Q_L; Q_R \rangle$ with $q_L = |Q_L|$ and $q_R = |Q_R|$. Suppose that Q^e occurs in forests F and G at nodes u and v, respectively, satisfying $|o(u) - o(v)| \leq 2k$ and $|c(u) - c(v)| \leq 2k$. Denote

$$F' = F[0\,..\,o(u)) \cdot Q_L^{e'} \cdot F[o(u) + |Q_L^e|\,..\,c(u) - |Q_R^e|] \cdot Q_R^{e'} \cdot F(c(u)\,..\,|F|),$$
$$G' = G[0\,..\,o(v)) \cdot Q_L^{e'} \cdot G[o(v) + |Q_L^e|\,..\,c(u) - |Q_R^e|] \cdot Q_R^{e'} \cdot G(c(v)\,..\,|G|).$$

For $i \in [0\,..\,e)$, let u_i be the node of F with $o(u_i) = o(u) + i \cdot q_L$ (and $c(u_i) = c(u) - i \cdot q_R$) and let v_i be the node of G with $o(v_i) = o(v) + i \cdot q_L$ (and $c(v_i) = c(v) - i \cdot q_R$). Moreover, let $\mathcal{A}$ be an optimal forest alignment such that $\mathsf{ted}(F, G) = \mathsf{ted}_{\mathcal{A}}(F, G) \leq k$.

Claim 3.3.11. *There exist $i_F, i_G \in [0\,..\,5k]$ such that*

$$F[o(u) + i_F \cdot q_L\,..\,o(u) + (i_F + 1) \cdot q_L) \simeq_{\mathcal{A}} G[o(v) + i_G \cdot q_L\,..\,o(v) + (i_G + 1) \cdot q_L),$$

$$F(c(u) - (i_F + 1) \cdot q_R\,..\,c(u) - i_F \cdot q_R] \simeq_{\mathcal{A}} G(c(v) - (i_F + 1) \cdot q_R\,..\,c(v) - i_G \cdot q_R].$$

Proof. Let $(f_b, g_b) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_b \geq o(u)$ and $g_b \geq o(v)$. By symmetry between F and G, we may assume without loss of generality that $f_b = o(u)$. Consider the $k + 1$ disjoint occurrences of C in F at positions $(o(u) + i \cdot q_L, c(u) + i \cdot q_R)$ for $i \in [0\,..\,k]$. The alignment $\mathcal{A}$ (of unweighted cost at most k) must match one of these occurrences perfectly to a context within G. We pick the index $i_F \in [0\,..\,k]$ of one such perfectly matched occurrence and suppose that it occurs at a node v' of G.

In particular,

$$F[o(u_{i_F})\,..\,o(u_{i_F}) + q_L) \simeq_{\mathcal{A}} G[o(v')\,..\,o(v') + q_L),$$

$$F(c(u_{i_F}) - q_R\,..\,c(u_{i_F})] \simeq_{\mathcal{A}} G(c(v') - q_R\,..\,c(v')].$$

Since $(f_b, g_b) \in \mathcal{A}$, we must have $o(v') \geq g_b \geq o(v)$ by the non-crossing property of $\mathcal{A}$. At the same time, since the unweighted cost of $\mathcal{A}$ does not exceed k, we have $o(v') \leq o(u_{i_F}) + 2k \leq o(u) + kq_L + 2k \leq o(v) + kq_L + 4k \leq o(v) + 5kq_L$. Similarly, $c(v') \geq c(v) - 5kq_R$, which also implies $c(v') \leq c(v)$.

Our next goal is to show that $v' = v_{i_G}$ for some $i_G \in [0\,..\,5k]$. For a proof by contradiction, suppose that $o(v_i) < o(v') < o(v_{i+1})$ for some $i \in [0\,..\,5k)$. Due to $c(v') > c(v) - 5kq_R$, this also implies that $c(v_i) > c(v') > c(v_{i+1})$, i.e., that v' is a node on the path between v_i and v_{i+1}. Suppose

that the length of this path is ℓ and the node v' is at distance ℓ' from v_i. Hence, $G[o(v_i)\,..\,o(v'))$ has ℓ' unmatched opening parentheses out of the ℓ unmatched opening parentheses in Q_L. Moreover, $G[o(v_i)\,..\,o(v')) \cdot G[o(v')\,..\,o(v_{i+1})) = Q_L = G[o(v')\,..\,o(v_{i+1})) \cdot G[o(v_i)\,..\,o(v'))$, and thus there is a primitive string Q_L such that $G[o(v')\,..\,o(v_{i+1}))$ and $G[o(v_i)\,..\,o(v'))$ are both powers of Q_L. The number of unmatched opening parentheses is Q_L must be a common divisor of ℓ and ℓ', i.e., Q_L can be expressed as a string power with exponent $\ell/\gcd(\ell,\ell')$. A symmetric argument shows that Q_R can be expressed as a string power with exponent $\ell/\gcd(\ell,\ell')$. Overall, we conclude that C can be expressed as a context power with exponent $\ell/\gcd(\ell,\ell')$, contradicting the primitivity of C. Hence, $v' = v_{i_G}$ for some $i_G \in [0\,..\,5k]$ holds as claimed and, in particular, $o(v') = o(v) + i_G q_L$ and $c(v') = c(v) - i_G q_R$. □

Now, if the occurrences of Q^e at nodes u, v are replaced with $Q^{e'}$ for $e' \geq e-1$, we can interpret this as replacing the occurrences of Q at nodes u_{i_F}, v_{i_G} with $Q^{1+e'-e}$. By Claim 3.3.11, $\mathcal{A}$ can be trivially adapted without modifying its cost, and hence $\mathsf{ted}^w(F',G') \leq \mathsf{ted}^w_{\mathcal{A}}(F,G) = \mathsf{ted}^w(F,G)$. If $e' < e-1$, we repeat the above argument to decrement the exponent one step at a time, still concluding that $\mathsf{ted}^w(F',G') \leq \mathsf{ted}^w(F,G)$. In either case, the converse inequality follows by symmetry between (F,G,e) and (F',G',e'). □

We say that a context $P = \langle P_L; P_R\rangle$ avoids vertical k-periodicity if it cannot be expressed as $P = C \star Q^{6k+1} \star D$ for some contexts C, Q, D satisfying $|Q| \in [1\,..\,8k]$.

Lemma 3.3.12. *Let $k \in \mathbb{Z}_+$, let $P = \langle P_L; P_R\rangle, P' = \langle P'_L; P'_R\rangle$ be contexts of length $|P_L| + |P_R|, |P'_L| + |P'_R| \geq 578k^4$ that avoid vertical k-periodicity and whose halves do not contain any balanced substring of length more than $74k^3$. Then, P and P' are $\mathsf{ted}^w_{\leq k}$-equivalent for every normalized weight function w.*

Proof. Suppose that P occurs in forests F and G at nodes u and v, respectively, satisfying $|o(u) - o(v)| \leq 2k$ and $|c(u) - c(v)| \leq 2k$, Denote

$$F' = F[0\,..\,o(u)) \cdot P'_L \cdot F[o(u) + |P_L|\,..\,c(u) - |P_R|] \cdot P'_R \cdot F(c(u)\,..\,|F|),$$

$$G' = G[0\,..\,o(v)) \cdot P'_R \cdot G[o(v) + |P_L|\,..\,c(v) - |P_R|] \cdot P'_R \cdot G(c(v)\,..\,|G|).$$

Let $\mathcal{A} = (f_t, g_t)_{t=0}^{m}$ be an optimal forest alignment such that $\mathsf{ted}(F, G) = \mathsf{ted}_{\mathcal{A}}(F, G) \leq k$. Moreover, let $(f_a, g_a) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_a \geq o(u)$ or $g_a \geq o(v)$, $(f_b, g_b) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_b \geq o(u) + |P_L|$ and $g_b \geq o(v) + |P_L|$, $(f_c, g_c) \in \mathcal{A}$ be the leftmost element of $\mathcal{A}$ such that $f_c > c(u) - |P_R|$ or $g_c > c(v) - |P_R|$, and let (f_d, g_d) be the leftmost element of $\mathcal{A}$ such that $f_d > c(u)$ and $g_d > c(v)$. We construct an alignment $\mathcal{A}'$ so that it:

- aligns $F[0\,..\,f_a)$ with $G[0\,..\,g_a)$ in the same way as $\mathcal{A}$ does;
- deletes $F[f_a\,..\,o(u))$ and inserts $G[g_a\,..\,o(v))$ (at least one of these fragments is empty);
- matches $F[o(u)\,..\,o(u) + |P_L|) = P_L$ with $G[o(v)\,..\,o(v) + |P_L|) = P_L$;
- if $b > c$, deletes $F[o(u) + |P_L|\,..\,c(u) - |P_R|]$ and inserts $G[o(v) + |P_L|\,..\,c(v) - |P_R|]$;
- if $b \leq c$, deletes $F[o(u) + |P_L|\,..\,f_b)$ and inserts $G[o(v) + |P_L|\,..\,g_b)$ (at least one of these fragments is empty);
- if $b \leq c$, aligns $F[f_b\,..\,f_c)$ with $G[g_b\,..\,g_c)$ in the same way as $\mathcal{A}$ does;
- if $b \leq c$, deletes $F[f_c\,..\,c(u) - |P_R|]$ and inserts $G[g_c\,..\,c(v) - |P_R|]$ (at least one of these fragments is empty);
- matches $F(c(u) - |P_R|\,..\,c(u)] = P_R$ with $G(c(v) - |P_R|\,..\,c(v)] = P_R$;
- deletes $F(c(u)\,..\,f_d)$ and inserts $G(c(v)\,..\,g_d)$ (at least one of these fragments is empty);
- aligns $F[f_d\,..\,|F|)$ with $G[g_d\,..\,|G|)$ in the same way as $\mathcal{A}$ does.

To prove that $\mathcal{A}'$ is a forest alignment, let us consider several possibilities for a node u' in F.

- If u' belongs to $P = \langle F[o(u)\,..\,o(u)+|P_L|); F(c(u)-|P_R|\,..\,c(u)]\rangle$, then $\mathcal{A}'$ matches u' to the corresponding node that belongs to $P = \langle G[o(v)\,..\,o(v)+|P_L|); G(c(v)-|P_R|\,..\,c(v)]\rangle$.
- If u' is outside $F[o(u)\,..\,c(u)]$ and $\mathcal{A}$ aligns u' to a node v' of G inside $G[o(v)\,..\,c(v)]$, then $o(u'), c(u') \in [f_a\,..\,o(u)) \cup (c(u)\,..\,f_d)$ because of the non-crossing property of $\mathcal{A} \ni (f_a, g_a), (f_d, g_d)$. Hence, $\mathcal{A}'$ deletes u'.
- If u' is outside $F[o(u)\,..\,c(u)]$ and $\mathcal{A}$ aligns u' to a node v' of G outside $G[o(v)\,..\,c(v)]$, then $o(u'), c(u') \in [0\,..\,f_a) \cup [f_d\,..\,|F|)$ because of the non-crossing property of $\mathcal{A} \ni (f_a, g_a)$, (f_d, g_d). Hence, $\mathcal{A}'$ also aligns u' to v'.
- If u' is outside $F[o(u)\,..\,c(u)]$ and $\mathcal{A}$ deletes u', then $\mathcal{A}'$ also deletes u'.
- If $b > c$ and u' is inside $F[o(u)+|P_L|\,..\,c(u)-|P_R|]$, then $\mathcal{A}'$ deletes u'.
- If $b \leq c$, u' is inside $F[o(u)+|P_L|\,..\,c(u)-|P_R|]$, and $\mathcal{A}$ aligns u' to a node v' of G outside $G[o(v)+|P_L|\,..\,c(v)-|P_R|]$, then $o(u'), c(u') \in [o(u)+|P_L|\,..\,f_b) \cup [f_c\,..\,c(u)-|P_R|]$ because of the non-crossing property of $\mathcal{A} \ni (f_b, g_b), (f_c, g_c)$. Hence, $\mathcal{A}'$ deletes u'.
- If $b \leq c$, u' is inside $F[o(u)+|P_L|\,..\,c(u)-|P_R|]$, and $\mathcal{A}$ aligns u' to a node v' of G inside $G[o(v)+|P_L|\,..\,c(v)-|P_R|]$, then $o(u'), c(u') \in [f_b\,..\,f_c)$ because of the non-crossing property of $\mathcal{A} \ni (f_b, g_b), (f_c, g_c)$. Hence, $\mathcal{A}'$ also aligns u' to v'.
- If $b \leq c$, u' is inside $F[o(u)+|P_L|\,..\,c(u)-|P_R|]$, and $\mathcal{A}$ deletes u', then $\mathcal{A}'$ also deletes u'.

Let us now prove that $\mathsf{ted}^w_{\mathcal{A}'}(F,G) \leq \mathsf{ted}^w_{\mathcal{A}}(F,G)$. This relies on the following claim.

Claim 3.3.13. *There exist $t_L \in [a\,..\,b]$ such that $f_{t_L} - g_{t_L} = o(u) - o(u)$ and $t_R \in [c\,..\,d]$ such that $f_{t_L} - g_{t_L} = c(u) - c(u)$.*

Proof. By symmetry (up to reversal), we can focus without loss of generality on the first claim.

Moreover, by symmetry between F and G, we can assume without loss of generality that $f_a = o(u)$; in particular, this implies $f_a - g_a \geq o(u) - o(v)$. If there exists $t \in [a\,..\,b]$ such that $f_t - g_t \leq o(u) - o(v)$, then, since $f_t - g_t$ may change by at most one for subsequent positions, there is also $t_L \in [a\,..\,b]$ such that $f_{t_L} - g_{t_L} = o(u) - o(v)$, Consequently, it remains to consider the case when $f_t - g_t > o(u) - o(v)$ holds for all $t \in [a\,..\,b]$.

Let us express P as a vertical composition of e contexts $P = P_0 \star \cdots \star P_{e-1}$, where e is the depth of P. Observe that the occurrences of P at node u in F and v in G, for each $i \in [0\,..\,e)$, induce occurrences of P_i at some nodes u_i in F and v_i in G. Since $F(o(u_i)\,..\,o(u_i) + |P_{i,L}|)$ and $F(c(u_i) - |P_{i,R}|\,..\,c(u_i))$ are balanced, we conclude that $|P_i| \leq 2 \cdot (74k^3 + 1) \leq 150k^3$. We can decompose $[0\,..\,e)$ into at most k individual indices i such that $\mathcal{A}$ does not match perfectly the occurrence of P_i at v_i and at most $k+1$ intervals $[i\,..\,i')$ such that $\mathcal{A}$ matches the occurrence $P_i \star \cdots \star P_{i'-1}$ at v_i perfectly to a context in F. Let us choose such an interval $[i\,..\,i')$ maximizing $|P_i \star \cdots \star P_{i'-1}|$; this length is at least $\frac{578k^4 - k \cdot 150k^3}{k+1} \geq 214k^3$. Let $i'' \in [i\,..\,i')$ be the maximum index such that $|P_{i''} \star \cdots \star P_{i'-1}| > 8k$; note that $|P_i \star \cdots \star P_{i''-1}| \geq 214k^3 - (150k^3 + 8k) \geq 56k^2$.

For each $j \in [i\,..\,i'']$, denote by u'_j be the node of matched with v_j by $\mathcal{A}$. Note that $|o(u'_j) - o(u_j)| \leq 4k$ and $|c(u'_j) - c(u_j)| \leq 4k$. Moreover, $o(u'_j) - o(v_j) > o(u) - o(v) = o(u_j) - o(v_j)$ implies $o(u'_j) > o(u_j)$. Since $|P_j \star \cdots \star P_{i'-1}| > 8k$, we conclude that $u'_j = u_{j'}$ for some $j' \in [j\,..\,i')$. Moreover, if $j > i$, then u'_j must be a child of u'_{j-1}. Hence, there exists $\delta > 0$ such that $u'_j = u_{j+\delta}$ holds for all $j \in [i\,..\,i'']$. For $j \in [i\,..\,i'')$, this implies $P_j = P_{j+\delta}$ and that both halves of $P_j \star \cdots \star P_{j+\delta-1}$ are of length at most $4k$. In particular, if we define $Q = P_i \star \cdots \star P_{i+\delta-1}$, then, due to $\frac{|P_i \star \cdots \star P_{i''-1}|}{8k} \geq \frac{56k^2}{8k} \geq 6k+1$, we conclude that Q^{6k+1} occurs in F and G at positions u_i and v_i, respectively. This contradicts the assumption that P avoids vertical periodicity. $\square$

The alignments $\mathcal{A}$ and $\mathcal{A}'$ only differ in how they align the following fragments:

- $F[f_a \,..\, f_{t_L})$ with $G[g_a \,..\, g_{t_L})$: here, $\mathcal{A}'$ matches one fragment perfectly with a suffix of the other; by Fact 3.2.6, this is optimal.
- $F[f_{t_L} \,..\, f_{t_R})$ with $G[g_{t_L} \,..\, g_{t_R})$ if $b > c$: here, the cost of $\mathcal{A}'$ is equal to the cost of deleting $F[o(u)+|P_L| \,..\, c(u)-|P_R|]$ and inserting $G[o(v)+|P_L| \,..\, c(v)-|P_R|]$. By Fact 3.2.6, these two costs do not exceed the cost of $\mathcal{A}$ aligning $F[f_{t_L} \,..\, f_c)$ with $G[g_{t_L} \,..\, g_c)$ and aligning $F[f_b \,..\, f_{t_R})$ with $G[g_b \,..\, g_{t_R})$.
- $F[f_{t_L} \,..\, f_b)$ with $G[g_{t_L} \,..\, g_b)$ if $b \leq c$: here, $\mathcal{A}'$ matches one fragment perfectly with a prefix of the other; by Fact 3.2.6, this is optimal.
- $F[f_c \,..\, f_{t_R})$ with $G[g_c \,..\, f_{t_R})$ if $b \leq c$: here, $\mathcal{A}'$ matches one fragment perfectly with a suffix of the other; by Fact 3.2.6, this is optimal.
- $F[f_{t_R} \,..\, f_d)$ with $G[g_{t_R} \,..\, g_d)$: here, $\mathcal{A}'$ matches one fragment perfectly with a prefix of the other; by Fact 3.2.6, this is optimal.

If the occurrences of P at nodes u in F and v in G are modified to occurrences of P', then $\mathcal{A}'$ can be trivially adapted without modifying its cost and hence $\mathsf{ted}^w(F', G') \leq \mathsf{ted}^w_{\mathcal{A}'}(F, G) = \mathsf{ted}^w(F, G)$. The converse inequality follows by symmetry between (F, G, P) and (F', G', P'). □

3.3.4 Algorithms

We say a *piece* of a forest F is a balanced fragment $F[i \,..\, j)$ or a pair of fragments $\langle F[i \,..\, i'); F[j' \,..\, j)\rangle$ that form a context, that is, $F[i \,..\, j)$ is a tree and $F[i' \,..\, j')$ is balanced. For a fragment $F[i \,..\, j)$, we denote the set of pieces contained in $F[i \,..\, j)$ by $\mathcal{P}(F[i \,..\, j))$. Moreover, let $\mathcal{P}(F) = \mathcal{P}(F[0 \,..\, |F|))$.

Definition 3.3.14. A set $\mathsf{D} \subseteq \mathcal{P}(F[i \,..\, j))$ is a *piece decomposition* of a balanced fragment $F[i \,..\, j)$

of a forest F if it satisfies one of the following conditions:

- $\mathsf{D} = \emptyset$ and $i = j$;
- $\mathsf{D} = \{F[i\,..\,j)\}$ and $i < j$;
- $\mathsf{D} = \mathsf{D}_L \cup \mathsf{D}_R$ for some piece decompositions D_L of $F[i\,..\,m)$ and D_R of $F[m\,..\,j)$, where $m \in (i\,..\,j)$.
- $\mathsf{D} = \{\langle F[i\,..\,i'); F[j'\,..\,j)\rangle\} \cup \mathsf{D}'$ for a context $\langle F[i\,..\,i'); F[j'\,..\,j)\rangle \in \mathcal{P}(F)$ and a piece decomposition D' of $F[i'\,..\,j')$.

Lemma 3.3.15. *There exists a linear-time algorithm that, given a forest F and an integer $t \geq 2$, constructs a piece decomposition D of F consisting of at most $\max(1, \frac{6|F|}{t} - 1)$ pieces of length at most t each.*

Algorithm 12: $\mathcal{D}(i, j)$: Construct a decomposition of a balanced fragment $F[i\,..\,j)$.

```
1  if j = i then return ∅;
2  if j ≤ i + t then return {F[i..j)};
3  i' ← i; j' ← j;
4  while true do                    // F[i'..j') is balanced and (i' − i) + (j − j') ≤ t
5      Let m ∈ [i'..j'] be such that F[i'..m) is a tree;
6      if (m − i) + (j − j') ≤ t then i' ← m;
7      else if m < j' and (i' − i) + (j − m) ≤ t then j' ← m;
8      else if F[i..j) is not a tree then return D(i, i') ∪ D(i', m) ∪ D(m, j') ∪ D(j', j);
9      else if m = j' and (i' + 1 − i) ≤ (j − j' + 1) ≤ t then i' ← i' + 1; j' ← j' − 1;
10     else return {⟨F[i..i'); F[j'..j)⟩} ∪ D(i', m) ∪ D(m, j');
```

Proof. Algorithm 12 provides a recursive procedure that, for every balanced fragment $F[i\,..\,j)$ of F, constructs a piece decomposition $\mathcal{D}(i, j)$ of $F[i\,..\,j)$ consisting of pieces of size at most t. In the corner cases of $j = i$ and $j \in (i\,..\,i + t]$, we return $\mathcal{D}(i, j) = \emptyset$ and $\mathcal{D}(i, j) = \{F[i\,..\,j)\}$, respectively. Otherwise, we iteratively grow fragments $F[i\,..\,i')$ and $F[j'\,..\,j)$ (initially empty) maintaining the following invariants:

(a) $F[i' \,..\, j')$ is balanced;

(b) $|F[i \,..\, i')| + |F[j' \,..\, j)| \leq t$;

(c) if $F[i \,..\, j)$ is a tree, then $F[i \,..\, i') = F[j' \,..\, j) = \varepsilon$ or $\langle F[i \,..\, i'); F[j' \,..\, j)\rangle$ is a context;

(d) if $F[i \,..\, j)$ is not a tree, then $F[i \,..\, i')$ and $F[j' \,..\, j)$ are balanced.

At each iteration, we identify a position $m \in (i' \,..\, j']$ such that $F[i' \,..\, m)$ is a tree (such a position always exists due to $j - i > t$ and by invariants (a), (b)).

(1) We set $i' \leftarrow m$ as long as it would not violate invariant (b).

(2) If $m \neq j'$, we set $j \leftarrow m$ as long as it would not violate invariant (b).

(3) If $m = j'$ and $F[i \,..\, j)$ is a tree, we set $(i', j') \leftarrow (i' + 1, j' - 1)$ as long as it would not violate invariant (b).

(4) Otherwise, we return $\mathcal{D}(i, j) := \{\langle F[i \,..\, i'); F[j' \,..\, j)\rangle\} \cup \mathcal{D}(i', m) \cup \mathcal{D}(m, j')$ if $F[i \,..\, j)$ is a tree and $\mathcal{D}(i, j) := \mathcal{D}(i, i') \cup \mathcal{D}(i', m) \cup \mathcal{D}(m, j') \cup \mathcal{D}(j', j)$ if $F[i \,..\, j)$ is not a tree.

It is easy to see that cases (1)–(3) preserve the invariants and hence case (4) results in a valid piece decomposition with pieces of size at most t.

Next, we prove that the number of pieces is at most $\max(1, \frac{6(j-i)}{t} - 3)$ if $F[i \,..\, j)$ is a tree and at most $\max(1, \frac{6(j-i)}{t} - 1)$ otherwise. This holds trivially if $j \leq i + t$, where Algorithm 12 terminates at Line 1 or 2. If $F[i \,..\, j)$ is a tree of size $j - i > t$, then Algorithm 12 terminates at Line 10. We consider several sub-cases:

1. If $m - i' \leq \frac{2t}{3}$ and $j' - m \leq \frac{t}{3}$, then $|\mathcal{D}(i, j)| \leq 3 < \frac{6(j-i)}{t} - 3$ because $j - i > t$.
2. If $m - i' \leq \frac{2t}{3}$ and $j' - m > \frac{t}{3}$, then $(m - i) + (j - j') > t$ because the test in Line 6 failed. Hence, $j' - m < j - i - t$ and $|\mathcal{D}(i, j)| \leq 2 + |\mathcal{D}(m, j')| \leq 2 + \frac{6(j'-m)}{t} - 1 < \frac{6(j-i-t)}{t} + 1 < \frac{6(j-i)}{t} - 3$.

3. If $m - i' > \frac{2t}{3}$ and $j' - m = 0$, then $(i' + 1 - i) + (j - j' + 1) > t$ because the test in Line 9 failed. Hence, $j' - i' \leq j - i - t + 1$ and $|\mathcal{D}(i,j)| \leq 1 + |\mathcal{D}(i',j')| \leq 1 + \frac{6(j'-i')}{t} - 3 \leq \frac{6(j-i-t+1)}{t} - 2 < \frac{6(j-i)}{t} - 3$.
4. If $m - i' > \frac{2t}{3}$ and $0 < j' - m \leq \frac{t}{3}$, then $(i' - i) + (j - m) > t$ because the test in Line 7 failed. Hence, $m - i' < j - i - t$ and $|\mathcal{D}(i,j)| \leq 2 + |\mathcal{D}(i',m)| \leq 2 + \frac{6(m-i')}{t} - 3 < \frac{6(j-i-t)}{t} - 1 < \frac{6(j-i)}{t} - 3$.
5. If $m - i' > \frac{2t}{3}$ and $j' - m > \frac{t}{3}$, then $|\mathcal{D}(i,j)| \leq 1 + |\mathcal{D}(i',m)| + |\mathcal{D}(m,j')| \leq 1 + \frac{6(m-i')}{t} - 3 + \frac{6(j'-m)}{t} - 1 \leq \frac{6(j-i)}{t} - 3$.

If $F[i\,..\,j)$ is not a tree, then Algorithm 12 terminates at Line 8. We consider several sub-cases:

1. If $m - i' \leq \frac{2t}{3}$ and $j' - m \leq \frac{t}{3}$, then $|\mathcal{D}(i,j)| \leq 4 < \frac{6(j-i)}{t} - 1$ because $j - i > t$.
2. If $m - i' \leq \frac{2t}{3}$ and $j' - m > \frac{t}{3}$, then $(m - i) + (j - j') > t$ because the test in Line 6 failed. Hence, $j' - m < j - i - t$ and $|\mathcal{D}(i,j)| \leq 3 + |\mathcal{D}(m,j')| \leq 3 + \frac{6(j'-m)}{t} - 1 < \frac{6(j-i-t)}{t} + 2 < \frac{6(j-i)}{t} - 1$.
3. If $m - i' > \frac{2t}{3}$ and $j' - m = 0$, then $|\mathcal{D}(i,j)| \leq 2 + |\mathcal{D}(i',j')| \leq 2 + \frac{6(j'-i')}{t} - 3 \leq \frac{6(j-i)}{t} - 1$.
4. If $m - i' > \frac{2t}{3}$ and $0 < j' - m \leq \frac{t}{3}$, then $(i' - i) + (j - m) > t$ because the test in Line 7 failed. Hence, $m - i' < j - i - t$ and $|\mathcal{D}(i,j)| \leq 3 + |\mathcal{D}(i',m)| \leq 3 + \frac{6(m-i')}{t} - 3 < \frac{6(j-i-t)}{t} < \frac{6(j-i)}{t} - 1$.
5. If $m - i' > \frac{2t}{3}$ and $j' - m > \frac{t}{3}$, then $|\mathcal{D}(i,j)| \leq 2 + |\mathcal{D}(i',m)| + |\mathcal{D}(m,j')| \leq 2 + \frac{6(m-i')}{t} - 3 + \frac{6(j'-m)}{t} - 1 \leq \frac{6(j-i)}{t} - 2 < \frac{6(j-i)}{t} - 1$.

It remains to provide a linear-time implementation of our algorithm. We assume that there are bidirectional pointers between the opening and the closing parentheses representing the same node. Such pointers can be constructed using a linear-time stack-based preprocessing of the input

forest F. Each iteration of the **while** loop increases $j - j' + i' - i$ (except for the final one), so a single call to the $\mathcal{D}(i, j)$ function costs $\mathcal{O}(t)$ due to invariant (b). The total number of calls is $\mathcal{O}(|\mathcal{D}(0, |F|)|) = \mathcal{O}(\frac{1}{t} \cdot |F|)$, so the overall running time, including preprocessing, is $\mathcal{O}(|F|)$. □

Lemma 3.3.16. *Given forests F and G of total size n, a piece decomposition D of F, and an integer $s \in \mathbb{Z}_+$, one can find in $\mathcal{O}(n + |\mathsf{D}|s^3)$ time a maximum-size set $S \subseteq \mathsf{D} \times \mathcal{P}(G)$ that, for some alignment $\mathcal{A} \in \mathsf{TA}(F, G)$ of width at most s, contains only pairs of pieces that $\mathcal{A}$ matches perfectly.*

Algorithm 13: Compute a maximum-size element of $\mathcal{S}(D_{i,j}, G[i' \,..\, j'))$.

```
1  Pairs(D_{i,j}, G[i' .. j')):
2      S ← ∅;
3      if i' < min(j', i + s) then S ⇐^max Pairs(D_{i,j}, G[i' + 1 .. j'));
4      if j' > max(i', j − s) then S ⇐^max Pairs(D_{i,j}, G[i' .. j' − 1));
5      if D_{i,j} = {F[i .. j)} and F[i .. j) = G[i' .. j') then
6          S ⇐^max {(F[i .. j), G[i' .. j'))};
7      if D_{i,j} = D_{i,m} ∪ D_{m,j} for some m ∈ (i .. j) then
8          foreach m' ∈ [m − s .. m + s] ∩ [i' .. j'] do
9              S ⇐^max Pairs(D_{i,m}, G[i' .. m')) ∪ Pairs(D_{m,j}, G[m' .. j'));
10     if D_{i,j} = {⟨F[i .. i + ℓ); F[j − r .. j)⟩} ∪ D_{i+ℓ,j−r} then
11         if F[i .. i + ℓ) = G[i' .. i' + ℓ) and F[j − r .. j) = G[j' − r .. j') and
            G[i' + ℓ .. j' − r) is balanced then
12             S ⇐^max {(⟨F[i .. i + ℓ); F[j − r .. j)⟩, ⟨G[i' .. i' + ℓ); G[j' − r .. j')⟩)} ∪
                Pairs(D_{i+ℓ,j−r}, G[i' + ℓ .. j' − r))
13         S ⇐^max Pairs(D_{i+ℓ,j−r}, G[max(i + ℓ − s, i') .. min(j − r + s, j')));
14     return S;
```

Proof. For a piece decomposition $\mathsf{D}_{i,j}$ of a balanced fragment $F[i \,..\, j)$ and a fragment $G[i' \,..\, j')$, let $\mathcal{S}(D_{i,j}, G[i' \,..\, j'))$ be the family of all subsets of $\mathsf{D}_{i,j} \times \mathcal{P}(G[i' \,..\, j'))$ that, for some alignment $\mathcal{A} \in \mathsf{A}(F[i \,..\, j), G[i' \,..\, j'))$ of width at most s, contain only pairs of pieces that $\mathcal{A}$ matches perfectly. Algorithm 13 implements a recursive procedure $\texttt{Pairs}(\mathsf{D}_{i,j}, G[i' \,..\, j'))$ that computes a maximum-size element of $\mathcal{S}(D_{i,j}, G[i' \,..\, j'))$ assuming that $i' \in [i-s \,..\, i+s]$ and $j' \in [j-s \,..\, j+s]$.

It uses an $S \overset{\max}{\leftarrow} S'$ operator that assigns $S \leftarrow S'$ if $|S'| > |S|$. The algorithm returns the largest of the following candidates:

1. $\emptyset$. This is trivially valid because every alignment $\mathcal{A} \in \mathsf{A}(F[i\,..\,j), G[i'\,..\,j'))$ of width at most s witnesses $\emptyset \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.
2. $\texttt{Pairs}(D_{i,j}, G[i'+1\,..\,j'))$ if $i' < \min(j', i+s)$. Let $S = \texttt{Pairs}(D_{i,j}, G[i'+1\,..\,j'))$ with a witness alignment $\mathcal{A}' \in \mathsf{A}(F[i\,..\,j), G[i'+1\,..\,j'))$. An alignment obtained from $\mathcal{A}'$ by prepending (i, i'), which corresponds to inserting $Y[i']$, witnesses $S \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.
3. $\texttt{Pairs}(D_{i,j}, G[i'\,..\,j'-1))$ if $j' > \max(i', j-s)$. Let $S = \texttt{Pairs}(D_{i,j}, G[i'\,..\,j'-1))$ with a witness alignment $\mathcal{A}' \in \mathsf{A}(F[i\,..\,j), G[i'\,..\,j'-1))$. An alignment obtained from $\mathcal{A}'$ by appending (j, j'), which corresponds to inserting $Y[j'-1]$, witnesses $S \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.
4. $\{(F[i\,..\,j), G[i'\,..\,j'))\}$ if $\mathsf{D}_{i,j} = \{F[i\,..\,j)\}$ and $F[i\,..\,j) = G[i'\,..\,j')$. The alignment $(i+t, j+t)_{t=0}^{i'-i} \in \mathsf{A}(F[i\,..\,j), G[i'\,..\,j'))$ witnesses $\{(F[i\,..\,j), G[i'\,..\,j'))\} \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.
5. $\texttt{Pairs}(D_{i,m}, G[i'\,..\,m')) \cup \texttt{Pairs}(D_{m,j}, G[m'\,..\,j'))$ if $\mathsf{D}_{i,j} = \mathsf{D}_{i,m} \cup \mathsf{D}_{m,j}$ for some $m \in (i\,..\,j)$ and $m' \in [m-s\,..\,m+s] \cap [i'\,..\,j']$. Denote $S_L = \texttt{Pairs}(D_{i,m}, G[i'\,..\,m'))$ and $S_R = \texttt{Pairs}(D_{m,j}, G[m'\,..\,j'))$ with witness alignments $\mathcal{A}_L \in \mathsf{A}(F[i\,..\,m), G[i'\,..\,m'))$ and $\mathcal{A}_R \in \mathsf{A}(F[m\,..\,j), G[m'\,..\,j'))$, respectively. Stitching $\mathcal{A}_L$ and $\mathcal{A}_R$ at the common endpoint (m, m') yields an alignment witnessing $S_L \cup S_R \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.
6. $\{(\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle, \langle G[i'\,..\,i'+\ell); G[j'-r\,..\,j')\rangle)\} \cup \texttt{Pairs}(D_{i+\ell,j-r}, G[i'+\ell\,..\,j'-r))$ if $\mathsf{D}_{i,j} = \{\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle\} \cup \mathsf{D}_{i+\ell,j-r}$ and $\langle G[i'\,..\,i'+\ell); G[j'-r\,..\,j')\rangle$ is a context in G matching $\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle$. Consider a set $S' = \texttt{Pairs}(D_{i+\ell,j-r}, G[i'+\ell\,..\,j'-r))$ and a witness alignment $\mathcal{A}' \in \mathsf{A}(F[i+\ell\,..\,j-r), G[i'+\ell\,..\,j'-r))$. Stitching $(i+t, i'+t)_{t=0}^{\ell}$, $\mathcal{A}'$, and $(j+t, j'+t)_{t=-r}^{0}$ at the common endpoints yields an align-

ment witnessing $S' \cup \{(\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle, \langle G[i'\,..\,i'+\ell); G[j'-r\,..\,j')\rangle)\} \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.

7. $\texttt{Pairs}(D_{i+\ell,j-r}, G[\max(i+\ell-s, i')\,..\,\min(j-r+s, j'))))$ if $\mathsf{D}_{i,j} = \{\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle\} \cup \mathsf{D}_{i+\ell,j-r}$. Let $S = \texttt{Pairs}(D_{i+\ell,j-r}, G[i'+\ell'\,..\,j'-r'))$, where $\ell' = \max(0, i+\ell-s-i')$ and $r' = \max(0, j'-j+r-s)$, with a witness alignment $\mathcal{A}' \in \mathsf{A}(F[i+\ell, j-r), G[i'+\ell', j'-r'))$. Stitching $(i+t, i'+t)_{t=0}^{\ell'}$, $(i+t, i'+\ell')_{t=\ell'}^{\ell}$, $\mathcal{A}'$, $(j+t, j'-r')_{t=-r}^{-r'}$, and $(j+t, j'+t)_{t=-r'}^{0}$ yields an alignment witnessing $S \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$.

Next, consider a maximum-size element $S \in \mathcal{S}(D_{i,j}, G[i'\,..\,j'))$ and a witness alignment $\mathcal{A} \in \mathsf{A}(F[i\,..\,j), G[i'\,..\,j'))$ of width at most s.

(a) If $\mathsf{D}_{i,j} = \emptyset$, we must have $S = \emptyset$, which is covered by candidate 1.

(b) Suppose that $\mathsf{D}_{i,j} = \{F[i\,..\,j)\}$. The case of $S = \emptyset$ is covered by candidate 1. Otherwise, $S = \{(F[i\,..\,j), G[i''\,..\,j''))\}$ for some $i'' \in [i'\,..\,i+s]$ and $j'' \in [j-s\,..\,j']$. This is covered by $i''-i'$ applications of candidate 2, $j'-j''$ applications of candidate 3, and finally an application of candidate 4.

(c) Suppose that $\mathsf{D}_{i,j} = \mathsf{D}_{i,m} \cup \mathsf{D}_{m,j}$ for some $m \in (i\,..\,j)$. Since the width of $\mathcal{A}$ does not exceed s, we must have $(m, m') \in \mathcal{A}$ for some $m' \in [m-s\,..\,m+s] \cap [i'\,..\,j']$. Consequently, S can be expressed as a union of an element of $\mathcal{S}(\mathsf{D}_{i,m}, G[i'\,..\,m'))$ and an element of $\mathcal{S}(D_{m,j}, G[m'\,..\,j'))$. This case is thus covered by candidate 5.

(d) Suppose that $\mathsf{D}_{i,j} = \{\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle\} \cup \mathsf{D}_{i+\ell,j-r}$. If S does not contain any pair of the form $(\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle, \langle G[i''\,..\,i''+\ell); G[j''-r\,..\,j'')\rangle)$, then $S \in \mathcal{S}(D_{i+\ell,j-r}, G[\max(i+\ell-s, i')\,..\,\min(j-r+s, j')))$, and this case is covered by candidate 7. Otherwise, we must have $i'' \in [\max(i', i-s)\,..\,i+s]$ and $j'' \in [j-s\,..\,\min(j', j+s)]$. This

is covered by $i'' - i'$ applications of candidate 2, $j' - j''$ applications of candidate 3, and finally an application of candidate 4 because $S \setminus \{(\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle, \langle G[i'\,..\,i'+\ell); G[j'-r\,..\,j')\rangle)\} \in \mathcal{S}(D_{i+\ell,j-r}, G[i'+\ell\,..\,j'-r))$.

This completes the proof that Algorithm 13 is correct. The sought set S is obtained via a call $\texttt{Pairs}(\mathsf{D}, G[0\,..\,|G|))$ which is valid as long as $s \geq \big||F|-|G|\big|$. Otherwise, there is no alignment $\mathcal{A} \in \mathsf{TA}(F,G)$ of width at most s, and thus we return $S = \emptyset$.

As for the efficient implementation, we use memoization to make sure that each call to `Pairs` is executed at most once. The number of calls is $\mathcal{O}(|\mathsf{D}| \cdot s^2)$ and each one performs $\mathcal{O}(s)$ instructions. In order to implement every instruction in $\mathcal{O}(1)$ time, we implement sets as persistent linked lists augmented with their size (this is valid because the arguments of every union operation are guaranteed to be disjoint). Moreover, we use Theorem 3.2.1 (for checking whether fragments of F match fragments of G) and Fact 3.3.1 (for checking whether fragments of G are balanced). Including the necessary preprocessing, the overall runtime is $\mathcal{O}(n + |\mathsf{D}|s^3)$. □

Lemma 3.3.17. *There exists a linear-time algorithm that, given a forest P and an integer $k \in \mathbb{Z}_+$, constructs a forest of length at most $74k^3$ that is $\mathsf{ted}^w_{\leq k}$-equivalent to P for every normalized quasimetric w.*

Algorithm 14: Construct a forest of length at most $74k^3$ that is $\mathsf{ted}^w_{\leq k}$-equivalent to P.

```
1 HorizontalReduction(P, k):
2     P' ← PeriodicityReduction(P, 4k, {Q ∈ Σ⁺ : |Q| ≤ 4k and Q is a primitive forest});
3     if |P'| ≥ 74k³ then return (a^{37k³})_a^{37k³} for some a ∈ Σ;
4     else return P';
```

Proof. We apply Lemma 3.2.13 with $e = 4k$ and $\mathcal{Q}$ consisting of all primitive forests of length at

most $4k$. We return $P'' := (_a^{37k^3})_a^{37k^3}$ (for an arbitrary label $a \in \Sigma$) or P' depending on whether $|P'| \leq 74k^3$ or not.

Observe that $\mathcal{Q}$ is chosen so that P' avoids horizontal k-periodicity and, by Lemma 3.3.5, P' is $\mathsf{ted}^w_{\leq k}$-equivalent to P for every normalized quasimetric w. Thus, the algorithm is correct if $|P'| < 74k^3$. Otherwise, Lemma 3.3.7 implies that P' and P'' are $\mathsf{ted}^w_{\leq k}$-equivalent for every normalized quasimetric w (both avoid horizontal k-periodicity and are of length at least $74k^3$).

The oracle testing in constant time whether a given fragment of P belongs to $\mathcal{Q}$ can be implemented using Fact 3.3.1. Thus, by Lemma 3.2.13, the overall running time is linear. □

Lemma 3.3.18. *There exists a linear-time algorithm that, given a context P and an integer $k \in \mathbb{Z}_+$, constructs a context of length at most $578k^4$ that is $\mathsf{ted}^w_{\leq k}$-equivalent to P for every normalized quasimetric w.*

Algorithm 15: Construct a context of length at most $578k^4$ that is $\mathsf{ted}^w_{\leq k}$-equivalent to P.

```
VerticalReduction(P, k):
    Let P = P_0 ⋆ ··· ⋆ P_{e−1}, where each P_i is a context of depth 1;
    for i ← 0 to e do
        Let P_i = ⟨ (_{a_i} F_i ; G_i )_{a_i} ⟩;
        P ← P · ⟨ (_{a_i} ·
          HorizontalReduction(F_i, k); HorizontalReduction(G_i, k) · )_{a_i} ⟩;
    Q ← {Q : ★_{i=0}^{|Q|−1} Q[i] is a primitive context of length at most 8k};
    P' ← PeriodicityReduction(P, 6k, Q);
    P' ← ★_{i=0}^{|P'|−1} P'[i];
    if |P'| ≥ 578k^4 then return ★_{i=0}^{17k^2−1} ⟨ (_a( (_a )_a )^i ; ( (_a )_a )^{17k^2−1−i} )_a ⟩ for some a ∈ Σ;
    else return P';
```

Proof. Let $P = P_0 \star \cdots \star P_{e-1}$, where each P_i is a context of depth 1, that is, $P_i = \langle (_{a_i} F_i ; G_i)_{a_i} \rangle$ for some label $a_i \in \Sigma$ and forests F_i, G_i. As the first step, our algorithm constructs a string $\mathbf{P}[0\,..\,e)$ whose characters are depth-1 contexts defined so that $\mathbf{P}[i] = \langle (_{a_i} F'_i ; G'_i)_{a_i} \rangle$, where forests $F'_i =$

$\texttt{HorizontalReduction}(F_i, k)$ and $G'_i = \texttt{HorizontalReduction}(G_i, k)$ are constructed using Lemma 3.3.17. Next, we transform $\mathbf{P}$ using Lemma 3.2.13 with $e = 6k$ and a family $\mathcal{Q}$ defined so that $\mathbf{P}[i\,..\,j) \in \mathcal{Q}$ if and only if $\mathbf{P}[i] \star \cdots \star \mathbf{P}[j-1]$ is a primitive context of length at most $8k$ (this implies $j - i \leq 4k$). In order to apply Lemma 3.2.13 to $\mathbf{P}$, we use linear-time string sorting [Paige and Tarjan, 1987, Andersson and Nilsson, 1994] to map characters of $\mathbf{P}$ (depth-1 contexts) to integer identifiers. By composing the contexts corresponding to the resulting string $\mathbf{P}'$, we obtain a context P'. We return $P'' := \bigstar_{i=0}^{17k^2-1} \langle\, (_a (\, (_a)\, a)^i ; (\, (_a)\, a)^{17k^2-1-i})\, a \rangle$ (for an arbitrary label $a \in \Sigma$) or P' depending on whether $|P'| \geq 578k^4$ or not.

Note that $|P''| = \sum_{i=0}^{17k^2-1}(1+2\cdot i+2\cdot(17k^2-1-i)+1) = 17k^2\cdot 2\cdot 17k^2 = 578k^4$. Thus, the resulting context (either P' or P'') is guaranteed to be of length at most $578k^4$. Let us now argue that it is $\mathsf{ted}^w_{\leq k}$-equivalent to P for every normalized quasimetric w. By Lemma 3.3.17, the forests F'_i and G'_i are $\mathsf{ted}^w_{\leq k}$-equivalent to F_i and G_i, respectively, and thus $\bigstar_{i=0}^{e-1}\mathbf{P}[i]$ is $\mathsf{ted}^w_{\leq k}$-equivalent to P. By Lemma 3.2.13, the context P' is obtained from $\bigstar_{i=0}^{e-1}\mathbf{P}[i]$ by repeatedly replacing Q^{6k+1} with Q^{6k} for primitive contexts Q of length at most $8k$. By Lemma 3.3.10, Q^{6k+1} is then $\mathsf{ted}^w_{\leq k}$-equivalent to Q^{6k}, so this operation preserves $\mathsf{ted}^w_{\leq k}$-equivalence, i.e., P' is also $\mathsf{ted}^w_{\leq k}$-equivalent to P. Moreover, each depth-1 context in $\mathbf{P}'$ originates from $\mathbf{P}$, so each forest occurring in (either half of) P' is of length at most $74k^3$. Furthermore, Lemma 3.2.13 guarantees that P' is not of the form $C \star Q^{6k+1} \star D$ for any context Q of length at most $8k$, and thus P' avoids vertical k-periodicity. By construction, P'' avoids vertical k-periodicity and its halves contain only forests of lengths at most $74k^3$ (in fact, at most $34k^2$). Consequently, Lemma 3.3.12 implies that P'' is $\mathsf{ted}^w_{\leq k}$-equivalent to P' (and, by transitivity, to P) provided that $|P''| \geq 578k^4$.

As for the running time analysis, we note that all applications of Lemma 3.3.17 concern disjoint fragments of P, so the total cost of the calls to $\texttt{HorizontalReduction}$ is linear.

Assigning integer identifiers to contexts $\mathbf{P}[i]$ and applying Lemma 3.2.13 also takes linear time. Finally, P'' is constructed only if $|P'| \geq 578k^4$, so the cost of this step is also be bounded by $\mathcal{O}(k^4) = \mathcal{O}(|P|)$. □

Theorem 3.3.19. *There exists an $\mathcal{O}(n)$-time algorithm that, given forests F, G of size at most $n \geq 12716k^5$ and an integer $k \in \mathbb{Z}_+$, constructs forests F', G' of lengths at most $\frac{n}{2} + 6358k^5$ such that $\mathsf{ted}^w_{\leq k}(F, G) = \mathsf{ted}^w_{\leq k}(F', G')$ holds for every normalized quasimetric w.*

Proof. By symmetry, we assume without loss of generality that $|F| \geq |G|$. We start by applying Lemma 3.3.15 to construct a piece decomposition D of F consisting of at most $12k - 1$ pieces of length at most $\lceil \frac{n}{2k} \rceil$ each. Next, we use Lemma 3.3.16 to identify a maximum-size set $S \subseteq \mathsf{D} \times \mathcal{P}(G)$ that, for some alignment $\mathcal{A} \in \mathsf{A}(F, G)$ of width at most $2k$, contains only pairs of pieces that $\mathcal{A}$ matches perfectly. If $|S| < |\mathsf{D}| - k$, we return $F' = (\,(_a)\,_a)^{k+1}$ and $G' = \varepsilon$ for some $a \in \Sigma$. Otherwise, for each pair of matching forests $F[i\,..\,j) = P = G[i'\,..\,j')$ in S, we use Lemma 3.3.17 to construct a forest P' of length at most $74k^3$ that is $\mathsf{ted}^w_{\leq k}$-equivalent to P for every normalized quasimetric w. We replace the occurrences of P at $F[i\,..\,j)$ and $G[i'\,..\,j')$ by occurrences of P'. Similarly, for every pair of matching contexts $\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle = P = \langle G[i'\,..\,i'+\ell); G[j'-r\,..\,j')\rangle$ in S, we use Lemma 3.3.18 to construct a context P' of length at most $578k^4$ that is $\mathsf{ted}^w_{\leq k}$-equivalent to P for every normalized quasimetric w. We replace the occurrences of P at $\langle F[i\,..\,i+\ell); F[j-r\,..\,j)\rangle$ and $\langle G[i'\,..\,i'+\ell); G[j'-r\,..\,j')\rangle$ by occurrences of P'.

If $\mathsf{ted}^w(F, G) \leq k$ holds for any normalized weight function w, then the unweighted cost of the underlying optimal alignment does not exceed k. Thus, its width is at most $2k$, and it matches perfectly all but at most k pieces of D. Consequently, in that case, $|S| \geq |\mathsf{D}| - k$. In particular, if

we return $F' = (\,(_a\,)_a)^{k+1}$ and $G' = \varepsilon$ for some $a \in \Sigma$, then $\mathsf{ted}^w_{\le k}(F,G) = \infty = \mathsf{ted}^w_{\le k}(F',G')$ holds as claimed. In that case, $|F'|, |G'| \le 2k+2 \le 4k < 6358k^5$. Otherwise, by definition of $\mathsf{ted}^w_{\le k}$-equivalence, the resulting forests F' and G' satisfy $\mathsf{ted}^w_{\le k}(F,G) = \mathsf{ted}^w_{\le k}(F',G')$. Moreover, $|F'| \le k \cdot \lceil \frac{n}{2k} \rceil + (11k-1)\cdot 578k^4 \le \frac{n}{2} + k + (11k-1)\cdot 578k^4 \le \frac{n}{2} + 11 \cdot 578 \cdot k^5 = \frac{n}{2} + 6358k^5$ and $|G'| = |F'| + |G| - |F| \le |F'| \le \frac{n}{2} + 6358k^5$.

The applications of Lemmas 3.3.15 and 3.3.16 cost $\mathcal{O}(n)$ and $\mathcal{O}(n + k^4) = \mathcal{O}(n)$ time, respectively. The calls to `HorizontalReduction` and `VerticalReduction` concern disjoint pieces of F, so their total cost is $\mathcal{O}(n)$ by Lemmas 3.3.17 and 3.3.18, respectively. □

Corollary 3.3.20. *There exists a linear-time algorithm that, given forests F, G and an integer $k \in \mathbb{Z}_+$, constructs forests F', G' of lengths at most $12717k^5$ such that $\mathsf{ted}^w_{\le k}(F,G) = \mathsf{ted}^w_{\le k}(F',G')$ holds for every normalized quasimetric w.*

Proof. We iteratively apply Theorem 3.3.19 as long as $\max(|F|,|G|) > 12717k^5$ and return the resulting pair of forests. Formally, we construct a sequence $(F_i, G_i)_{i=0}^t$ such that $(F_0, G_0) = (F, G)$ and $\mathsf{ted}^w_{\le k}(F_{i+1}, G)_{i+1} = \mathsf{ted}^w_{\le k}(F_i, G_i)$ holds for every $i \in [0\,..\,t)$. Consider forests (F_i, G_i) at iteration i. If $n_i := \max(|F_i|,|G_i|) \le 12717k^5$, we set $t := i$ and return $(F', G') := (F_i, G_i)$. Otherwise, we apply Theorem 3.3.19 to derive forests F_{i+1} and G_{i+1} of lengths at most $n_{i+1} := \max(|F_{i+1}|,|G_{i+1}|) \le \frac{1}{2}n_i + 6358k^5$ such that $\mathsf{ted}^w_{\le k}(F_{i+1}, G)_{i+1} = \mathsf{ted}^w_{\le k}(F_i, G_i)$. Since $n_{i+1} - 12716k^5 \le \frac{1}{2}(n_i - 12716k^5)$, the value n_i strictly decreases at each iteration and thus the process terminates. Moreover, the running time of each iteration is $\mathcal{O}(n_i) = \mathcal{O}(n_i - 12716k^5)$. The latter values form a geometric series dominated by the leading term at $i = 0$. Hence, the total running time is linear in the input size. □

Theorem 3.1.2. *Given forests F, G of length at most n, an integer $k \in \mathbb{Z}_+$, and a quasimetric*

w, the value $\mathsf{ted}^w_{\le k}(F, G)$ can be computed in $\mathcal{O}(n + k^{15})$ time. Moreover, $\mathsf{ted}_{\le k}(F, G)$ can be computed in $\mathcal{O}(n + k^7 \log k)$ time.

Proof. We first apply Corollary 3.3.20 to build forests F', G' of length $\mathcal{O}(k^5)$ such that $\mathsf{ted}^w_{\le k}(F, G) = \mathsf{ted}^w_{\le k}(F', G')$. Then, we compute $\mathsf{ted}^w_{\le k}(F', G')$ using the algorithm of Demaine, Mozes, Rossman, and Weimann [Demaine et al., 2010]. The running times of these two steps are $\mathcal{O}(n)$ and $\mathcal{O}((k^5)^3)$, respectively, for a total of $\mathcal{O}(n + k^{15})$. If w is the discrete metric (the unweighted case), then we compute $\mathsf{ted}_{\le k}(F', G')$ using the algorithm of Akmal and Jin [Akmal and Jin, 2021], which costs $\mathcal{O}(k^5 \cdot k^2 \cdot \log(k^5)) = \mathcal{O}(k^7 \log k)$ time. □

3.4 Dyck Edit Distance

In this section we give a deterministic algorithm that computes weighted Dyck edit distance of a given input string. Formally we show the following.

Theorem 3.1.3. *Given a string X of length n, an integer $k \in \mathbb{Z}_+$, and a skewmetric w, the value $\mathsf{dyck}^w_{\le k}(X)$ can be computed in $\mathcal{O}(n + k^{12})$ time.*

3.4.1 Preliminaries

In Dyck Language, the alphabet Σ consists of two disjoint sets T and $\overline{T}$ of *opening* and *closing* parentheses, respectively, with a bijection $f : T \to \overline{T}$ mapping each opening parenthesis to the corresponding closing parenthesis. We extend this mapping to an involution $f : T \cup \overline{T} \to T \cup \overline{T}$ and then to an involution $f : \Sigma^* \to \Sigma^*$ mapping each string $X[0]X[1] \cdots X[|X| - 1]$ to its reverse complement $\overline{X[|X| - 1]} \cdots \overline{X[1]}\,\overline{X[0]}$. Given two strings X, Y, we denote their concatenation by XY or $X \cdot Y$.

The *Dyck* language $\mathsf{Dyck}(\Sigma) \subseteq \Sigma^*$ consists of all well-parenthesized expression over Σ; formally, it can be defined using a context-free grammar whose only non-terminal S admits productions $S \to SS$, $S \to \varnothing$ (empty string), and $S \to aS\overline{a}$ for all $a \in T$.

Definition 3.4.1 (Heights)**.** Given an alphabet set Σ, define the function $h : \Sigma \to \{-1, 1\}$ where $h(a) = 1$ if $a \in \Sigma$ is an opening parenthesis and $h(a) = -1$ otherwise. Given a string $X \in \Sigma^n$, define the height of a position i where $0 \leq i \leq n$, as $H(i) = \sum_{j=0}^{i-1} h(X[j])$.

Here $H(i)$ is the difference between the number of opening parentheses and the number of closing parentheses in $X[0..i)$.

Definition 3.4.2 (Peaks and valleys)**.** Given a string $X \in \Sigma$, an index $i \in [1 \dots n)$ is called a peak if $H(i-1) < H(i) > H(i+1)$ and a valley if $H(i-1) > H(i) < H(i+1)$.

3.4.2 Dyck Language Alignments and Weighted Dyck Edit Distance

We say that $\mathcal{M} \subseteq \{(i,j) \subseteq \mathbb{Z}^2 : i < j\}$ is a *non-crossing matching* if any two distinct pairs $(i,j), (i',j') \in \mathcal{M}$ satisfy $i < j < i' < j'$ or $i < i' < j' < j$. Such a matching can also be interpreted as a function $\mathcal{M} : \mathbb{Z} \to \mathbb{Z} \cup \{\perp\}$ with $\mathcal{M}(i) = j$ if $(i,j) \in \mathcal{M}$ or $(j,i) \in \mathcal{M}$ for some $j \in \mathbb{Z}$, and $\mathcal{M}(i) = \perp$ otherwise. For a string $X \in \Sigma^*$ we define its *Dyck language alignment* to be a matching function $\mathcal{M}$ as defined above.

For two fragments $X[p\,..\,q)$ and $X[p'\,..\,q')$ of X, we write $X[p\,..\,q) \simeq_{\mathcal{M}} \overline{X[p'\,..\,q')}$ if $X[p\,..\,q) = \overline{X[p'\,..\,q')} \in T^*$ and $(r, q'-r-1)$ holds for every $r \in [p\,..\,q)$.

Similar to Section 3.2.2 we define a weight function w on $\bar{\Sigma} := \Sigma \cup \{\varepsilon\}$. We call this weight function a *skewmetric* if it satisfies the triangle inequality, that is, $w(a,b) + w(b,c) \geq w(a,c)$ holds for every $a, b, c \in \bar{\Sigma}$ and skew-symmetry, that is, $w(a,b) = w(\overline{b}, \overline{a})$ holds for every $a, b \in \bar{\Sigma}$. In the

rest of this section we assume the weight function w to be skewmetric unless stated otherwise.

Definition 3.4.3. The weighted Dyck edit distance of a string $X \in \Sigma^*$ with respect to a weight function w is the minimum edit distance $\mathsf{ed}^w(X, Y)$ between X and a string $Y \in \mathsf{Dyck}(\Sigma)$. Formally,

$$\mathsf{dyck}^w(X) = \min_{Y \in \mathsf{Dyck}(\Sigma)} \mathsf{ed}^w(X, Y).$$

For $k \in \mathbb{R}_{\geq 0}$, we also denote

$$\mathsf{dyck}^w_{\leq k}(X) = \begin{cases} \mathsf{dyck}^w(X) & \text{if } \mathsf{dyck}^w(X) \leq k, \\ \infty & \text{otherwise.} \end{cases}$$

The *cost* of an alignment $\mathcal{M} \in \mathsf{M}(X)$ with respect to a *weight function* w, denoted $\mathsf{dyck}^w_{\mathcal{M}}(X)$, is defined as

$$\mathsf{dyck}^w_{\mathcal{M}}(X) = \sum_{(i,j) \in \mathcal{M}} \mathsf{dyck}^w(X[i]X[j]) + \sum_{i \in [0..|X|): \mathcal{M}(i) = \perp} \mathsf{dyck}^w(X[i]).$$

Fact 3.4.4. *For every string X and weight function w, we have* $\mathsf{dyck}^w(X) = \min_{\mathcal{M} \in \mathsf{M}(X)} \mathsf{dyck}^w_{\mathcal{M}}(X)$.

Proof. We first show by induction on $|X|$ that $\mathsf{dyck}^w(X) \leq \mathsf{dyck}^w_{\mathcal{M}}(X)$ holds for every $\mathcal{M} \in \mathsf{M}(X)$. The claim is trivial if $|X| = 0$. If $\mathcal{M}(0) = \perp$, then we construct $\mathcal{M}' := \{(i-1, j-1) : (i, j) \in \mathcal{M}\}$ and $X' := X[1..|X|)$. By the inductive assumption, $\mathsf{dyck}^w(X) \leq \mathsf{dyck}^w(X') + \mathsf{dyck}^w(X[0]) \leq \mathsf{dyck}^w_{\mathcal{M}'}(X') + \mathsf{dyck}^w(X[0]) = \mathsf{dyck}^w_{\mathcal{M}}(X)$. If $\mathcal{M}(0) = |X| - 1$, then we construct $\mathcal{M}' := \{(i-1, j-1) : (i, j) \in \mathcal{M} \setminus \{(0, |X| - 1)\}$ and $X' = [1..|X| - 1)$. By the inductive assumption, $\mathsf{dyck}^w(X) \leq \mathsf{dyck}^w(X') + \mathsf{dyck}^w(X[0]X[|X| - 1]) \leq \mathsf{dyck}^w_{\mathcal{M}'}(X') + \mathsf{dyck}^w(X[0]X[|X|-1]) = \mathsf{dyck}^w_{\mathcal{M}}(X)$. Otherwise, we have $(0, p) \in M$ for some $p \in [1..|X|-1)$.

In this case, we construct $\mathcal{M}' := \{(i,j) \in \mathcal{M} : j \leq p\}$ and $X' := X[0\,..\,p]$, as well as $\mathcal{M}'' := \{(i-p-1, j-p-1) : (i,j) \in \mathcal{M} \text{ and } i > p\}$ and $X'' := X[p+1\,..\,|X|)$. By the inductive assumption, $\mathsf{dyck}^w(X) \leq \mathsf{dyck}^w(X') + \mathsf{dyck}^w(X'') \leq \mathsf{dyck}^w_{\mathcal{M}'}(X') + \mathsf{dyck}^w_{\mathcal{M}''}(X'') = \mathsf{dyck}^w_{\mathcal{M}}(X)$; here, the last equality follows from the fact that $|\mathcal{M}| = |\mathcal{M}'| + |\mathcal{M}''|$: any $(i,j) \in M$ with $i \leq p$ and $j > p$ would violate the non-crossing property of $\mathcal{M}$.

Next, we show by induction on $|X|$ that there exists $\mathcal{M} \in \mathsf{M}(X)$ such that $\mathsf{dyck}^w_{\mathcal{M}}(X) \leq \mathsf{dyck}^w(X)$; again, the claim is trivial for $|X| = 0$. Let us fix $Y \in \mathsf{Dyck}(\Sigma)$ and $\mathcal{A} \in \mathsf{A}(X,Y)$ such that $\mathsf{dyck}^w(X) = \mathsf{ed}^w_{\mathcal{A}}(X,Y)$. If $\mathcal{A}$ deletes $X[0]$, we consider $X' := X[1\,..\,|X|)$. The inductive assumption yields a matching $\mathcal{M}'$ such that $\mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{dyck}^w(X')$. In this case, we set $\mathcal{M} := \{(i+1, j+1) : (i,j) \in \mathcal{M}'\}$ so that $\mathsf{dyck}^w_{\mathcal{M}}(X) = \mathsf{dyck}^w(X[0]) + \mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{ed}^w(X[0], \varepsilon) + \mathsf{dyck}^w(X') \leq w(X[0], \varepsilon) + \mathsf{ed}^w(X', Y') \leq \mathsf{ed}^w_{\mathcal{A}}(X,Y) = \mathsf{dyck}^w(X)$. The case when $\mathcal{A}$ deletes $X[|X|-1]$ is symmetric, so we may assume that $\mathcal{A}$ deletes neither $X[0]$ nor $X[|X|-1]$; in particular, $Y \neq \varepsilon$.

Suppose that $Y = Y' \cdot Y''$ for some non-empty strings $Y', Y'' \in \mathsf{Dyck}(\Sigma)$. This yields a decomposition $X = X' \cdot X''$ such that $\mathsf{ed}^w(X,Y) = \mathsf{ed}^w(X',Y') + \mathsf{ed}^w(X'',Y'')$. Moreover, the optimality of Y guarantees that X' and X'' are both non-empty. The inductive assumption yields matchings $\mathcal{M}', \mathcal{M}''$ such that $\mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{dyck}^w(X')$ and $\mathsf{dyck}^w_{\mathcal{M}'}(X'') \leq \mathsf{dyck}^w(X'')$. In this case, we set $\mathcal{M} := \mathcal{M}' \cup \{(i+|X'|, j+|X'|) : (i,j) \in \mathcal{M}''\}$ so that $\mathsf{dyck}^w_{\mathcal{M}}(X) = \mathsf{dyck}^w_{\mathcal{M}'}(X') + \mathsf{dyck}^w_{\mathcal{M}'}(X'') \leq \mathsf{dyck}^w(X') + \mathsf{dyck}^w(X'') \leq \mathsf{ed}^w(X',Y') + \mathsf{ed}^w(X'',Y'') \leq \mathsf{ed}^w_{\mathcal{A}}(X,Y) = \mathsf{dyck}^w(X)$.

In the remaining case, we must have $Y = aY'\overline{a}$ for $a \in T$ and $Y' \in \mathsf{Dyck}(\Sigma)$. Let us first suppose that $\mathcal{A}$ aligns $Y[0] = a$ with $X[0]$ and $Y[|Y|-1] = \overline{a}$ with $X[|X|-1]$. In this case, $\mathcal{A}$ aligns $Y[1\,..\,|Y|-1) = Y'$ with $X' := X[1\,..\,|X|-1)$. The inductive assumption yields a matching $\mathcal{M}'$ such that $\mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{dyck}^w(X')$. In this case, we set $\mathcal{M} := \{(0, |X|-1)\} \cup \{(i+1, j+1) :$

$(i,j) \in \mathcal{M}'\}$ so that $\mathsf{dyck}^w_{\mathcal{M}}(X) = \mathsf{dyck}^w(X[0]X[|X|-1]) + \mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{ed}^w(X[0]X[|X|-1], a\overline{a}) + \mathsf{dyck}^w(X') \leq w(X[0],a) + w(X[|X|-1],\overline{a}) + \mathsf{ed}^w(X',Y') \leq \mathsf{ed}^w_{\mathcal{A}}(X,Y) = \mathsf{dyck}^w(X)$. Next, suppose that $\mathcal{A}$ aligns $Y[0] = a$ with $X[0]$ but inserts $Y[|Y|-1] = \overline{a}$. In this case, $\mathcal{A}$ aligns $Y[1\,..\,|Y|-1) = Y'$ with $X' := X[1\,..\,|X|)$. The inductive assumption yields a matching $\mathcal{M}'$ such that $\mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{dyck}^w(X')$. In this case, we set $\mathcal{M} := \{(i+1,j+1) : (i,j) \in \mathcal{M}'\}$ so that $\mathsf{dyck}^w_{\mathcal{M}}(X) = \mathsf{dyck}^w(X[0]) + \mathsf{dyck}^w_{\mathcal{M}'}(X') \leq \mathsf{ed}^w(X[0], a\overline{a}) + \mathsf{dyck}^w(X') \leq w(X[0],a) + w(\varepsilon,\overline{a}) + \mathsf{ed}^w(X',Y') \leq \mathsf{ed}^w_{\mathcal{A}}(X,Y) = \mathsf{dyck}^w(X)$. The case when $\mathcal{A}$ inserts $Y[0] = a$ and aligns $Y[|Y|-1] = \overline{a}$ with $X[|X|-1]$ is symmetric. The case when $\mathcal{A}$ inserts both $Y[0] = a$ and $Y[|Y|-1] = \overline{a}$ is impossible by optimality of Y. Finally, we note that, since $\mathcal{A}$ deletes neither $X[0]$ nor $X[|X|-1]$, the alignment $\mathcal{A}$ cannot align $Y[0]$ to any character other than $X[0]$ and $Y[|Y|-1]$ to any character other than $Y[|Y|-1]$. Thus, the case analysis above is complete. □

Claim 3.4.5. *For every $x \in \Sigma$ and skewmetric weight function w, $\mathsf{dyck}^w(x) = w(x,\epsilon) = w(\epsilon,\overline{x})$.*

Proof. We consider the following three different cases.

Case 1: x is deleted. In this case $\mathsf{dyck}^w(x) = w(x,\epsilon)$.

Case 2: $\overline{x}$ is inserted after x if $x \in T$ and before x if $x \in \overline{T}$. In this case $\mathsf{dyck}^w(x) = w(\epsilon,\overline{x}) = w(x,\epsilon)$. The last equality follows as w is skew-symmetric. Thus an insertion can be replaced with a deletion. From now on wards we assume that only allowed edits are deletion and substitutions.

Case 3: x is substituted by some $y \in \Sigma$. Here we also need to insert $\overline{y}$. Thus $\mathsf{dyck}^w(x) = w(x,y) + w(\epsilon,\overline{y}) = w(x,y) + w(y,\epsilon) \geq w(x,\epsilon)$. The second equality follows as w is skew-symmetric and the last inequality follows as w obeys triangle inequality. Trivially $\mathsf{dyck}^w(x) \leq w(x,\epsilon)$. Thus the claim follows. □

Claim 3.4.6. *For every $x, y \in \bar{\Sigma}$ and skewmetric weight function w,* $\mathsf{dyck}^w(xy) = \min_{z \in T \cup \{\epsilon\}} w(x, z) + w(y, \overline{z})$.

Proof. Let $z \in T \cup \{\epsilon\}$ minimizes $w(x, z) + w(y, \overline{z})$. It is straight forward to argue that $\mathsf{dyck}^w(xy) \leq w(x, z) + w(y, \overline{z})$ as x, y can be substituted by $z, \overline{z}$ respectively. Next we argue the converse. Following Claim 3.4.5 we assume the only allowed edits are deletions and substitutions.

Case 1: Both x and y are deleted. Here $\mathsf{dyck}^w(xy) \geq w(x, \epsilon) + w(y, \epsilon)$. The claim follows as $\epsilon \in T \cup \{\epsilon\}$ and $\overline{\epsilon} = \epsilon$.

Case 2: x is substituted by $\overline{y}$. Here $\mathsf{dyck}^w(xy) \geq w(x, \overline{y}) = w(x, \overline{y}) + w(y, y)$. Thus the claim follows as $\overline{y} \in T$.

Case 3: y is substituted by $\overline{x}$. Here $\mathsf{dyck}^w(xy) \geq w(y, \overline{x}) = w(x, x) + w(y, \overline{x})$. Thus the claim follows as $x \in T$.

Case 3: x is substituted by z and y is substituted by $\overline{z}$. Here $\mathsf{dyck}^w(xy) \geq w(x, z) + w(y, \overline{z})$. Thus the claim follows as $z \in T$. □

From now on wards we assume w to be skew-symmetric.

3.4.2.1 Preprocessing.

Given the input string $X \in \Sigma^n$, preprocess X as follows. As long as there are two neighboring indices $i, i+1$ such that $X[i+1] = \overline{X[i]}$ and $X[i] \in T$ remove them. Let the resulting string be X'. We make the following claim.

Claim 3.4.7. $\mathsf{dyck}^w(X) = \mathsf{dyck}^w(X')$.

Proof. Let $\mathcal{M}$ be an optimal alignment of X. For contradiction assume for two consecutive indices

$i, i+1$, $X[i+1] = \overline{X[i]}$, $X[i] \in T$ but $(i, i+1) \notin \mathcal{M}$. Next depending on the matching indices of $i, i+1$, we consider the following three cases.

Case 1: Let $(j, i), (i+1, k) \in \mathcal{M}$ where $j \in [0 \dots i) \cup \{\perp\}$ and $k \in (i+1 \dots |X|) \cup \{\perp\}$. In this case we create another alignment $\mathcal{M}' = \mathcal{M} \setminus \{(j, i), (i+1, k)\} \cup \{(i, i+1), (j, k)\}$. We argue $\mathsf{dyck}^w(X[j]X[k]) \leq \mathsf{dyck}^w(X[j]X[i]) + \mathsf{dyck}^w(X[i+1]X[k])$, thus proving $\mathsf{dyck}^w_{\mathcal{M}'}(X) \leq \mathsf{dyck}^w_{\mathcal{M}}(X)$. Following Claim 3.4.6, let $a, b \in T \cup \{\epsilon\}$ be such that $\mathsf{dyck}^w(X[j]X[i]) = w(X[j], a) + w(X[i], \overline{a})$ and $\mathsf{dyck}^w(X[i+1]X[k]) = w(X[i+1], b) + w(X[k], \overline{b})$. Thus,

$$\begin{aligned}
\mathsf{dyck}^w(X[j]X[i]) + \mathsf{dyck}^w(X[i+1]X[k]) &= w(X[j], a) + w(X[i], \overline{a}) + w(X[i+1], b) + w(X[k], \overline{b}) \\
&= w(X[j], a) + w(a, \overline{X[i]}) + w(X[i+1], b) + w(X[k], \overline{b}) \\
&\geq w(X[j], \overline{X[i]}) + w(X[i+1], b) + w(X[k], \overline{b}) \\
&\geq w(X[j], b) + w(X[k], \overline{b}) \\
&\geq \mathsf{dyck}^w(X[j], X[k])
\end{aligned}$$

The second equality follows as w is skew-symmetric; thus $w(X[i], \overline{a}) = w(a, \overline{X[i]})$. The third and fourth inequality follows as w follows triangle inequality and $\overline{X[i]} = X[i+1]$. The last inequality follows from Claim 3.4.6.

Case 2: Let $(k, i), (j, i+1) \in \mathcal{M}$ where $k, j \in [0 \dots i) \cup \{\perp\}$ and $j < k$. In this case we create another alignment $\mathcal{M}' = \mathcal{M} \setminus \{(k, i), (j, i+1)\} \cup \{(i, i+1), (j, k)\}$. We argue $\mathsf{dyck}^w(X[j]X[k]) \leq \mathsf{dyck}^w(X[k]X[i]) + \mathsf{dyck}^w(X[j]X[i+1])$, thus proving $\mathsf{dyck}^w_{\mathcal{M}'}(X) \leq \mathsf{dyck}^w_{\mathcal{M}}(X)$. Following Claim 3.4.6, let $a, b \in T \cup \{\epsilon\}$ be such that $\mathsf{dyck}^w(X[k]X[i]) = w(X[k], a) + w(X[i], \overline{a})$ and

$\mathsf{dyck}^w(X[j]X[i+1]) = w(X[j], b) + w(X[i+1], \overline{b})$. Thus,

$$\begin{aligned}
\mathsf{dyck}^w(X[k]X[i]) + \mathsf{dyck}^w(X[j]X[i+1]) &= w(X[k], a) + w(X[i], \overline{a}) + w(X[j], b) + w(X[i+1], \overline{b}) \\
&= w(X[k], a) + w(a, \overline{X[i]}) + w(X[j], b) + w(X[i+1], \overline{b}) \\
&\geq w(X[k], \overline{X[i]}) + w(X[i+1], \overline{b}) + w(X[j], b) \\
&\geq w(X[k], \overline{b}) + w(X[j], b) \\
&\geq \mathsf{dyck}^w(X[j], X[k])
\end{aligned}$$

Case 3: Let $(i, k), (i+1, j) \in \mathcal{M}$ where $k, j \in (i+1 \dots |X|) \cup \{\perp\}$ and $j < k$. In this case we create another alignment $\mathcal{M}' = \mathcal{M} \setminus \{(i, k), (i+1, j)\} \cup \{(i, i+1), (j, k)\}$. We argue $\mathsf{dyck}^w(X[j]X[k]) \leq \mathsf{dyck}^w(X[i]X[k]) + \mathsf{dyck}^w(X[i+1]X[j])$, thus proving $\mathsf{dyck}^w_{\mathcal{M}'}(X) \leq \mathsf{dyck}^w_{\mathcal{M}}(X)$. Following Claim 3.4.6, let $a, b \in T \cup \{\epsilon\}$ be such that $\mathsf{dyck}^w(X[i]X[k]) = w(X[i], a) + w(X[k], \overline{a})$ and $\mathsf{dyck}^w(X[i+1]X[j]) = w(X[i+1], b) + w(X[j], \overline{b})$. Thus,

$$\begin{aligned}
\mathsf{dyck}^w(X[i]X[k]) + \mathsf{dyck}^w(X[i+1]X[j]) &= w(X[i], a) + w(X[k], \overline{a}) + w(X[i+1], b) + w(X[j], \overline{b}) \\
&= w(X[i], a) + w(X[k], \overline{a}) + w(X[j], \overline{b}) + w(\overline{b}, \overline{X[i+1]}) \\
&\geq w(X[i], a) + w(X[k], \overline{a}) + w(X[j], \overline{X[i+1]}) \\
&\geq w(X[j], a) + w(X[k], \overline{a}) \\
&\geq \mathsf{dyck}^w(X[j], X[k])
\end{aligned}$$

□

The preprocessing can be done in time $O(n)$. Also, we can assume that in the preprocessed

string no two neighbouring symbols can be aligned. Following this and Claim 35 from [Backurs and Onak, 2016], we can make the following claim.

Claim 3.4.8. *Let $X \in \Sigma^n$. There exists an algorithm that preprocesses X in $O(n)$ time, and either declares $\mathsf{dyck}^w(X) > k$, or outputs a string X' of length at most n such that $\mathsf{dyck}^w(X) = \mathsf{dyck}^w(X')$ and X' has at most $2k$ valleys.*

Thus from now on wards we assume X to be preprocessed and has at most $2k$ valleys.

3.4.3 Periodicity Reduction

Definition 3.4.9. For $k \in \mathbb{Z}_{\geq 0}$ a fragments $X[a\,..\,b)$ and $X[c\,..\,d)$ of a string X are *k-synchronized* if $X[a\,..\,b) \in T^*$, $X[c\,..\,d) \in \overline{T}^*$, $b-a = d-c$, $b \leq c$, and $H(b)+H(c)-2\min_{m\in[b\,..\,c]} H(m) \leq 2k$.

Note that $X[a\,..\,b)$ and $X[c\,..\,d)$ are 0-synchronized if and only if (a, b, c, d) is a trapezoid.

Definition 3.4.10. For $k \in \mathbb{Z}_{\geq 0}$ and a skewmetric weight function w, strings $P, P' \in T^*$ are called *$\mathsf{dyck}^w_{\leq k}$-equivalent* if

$$\mathsf{dyck}^w_{\leq k}(X) = \mathsf{dyck}^w_{\leq k}(X[0\,..\,a) \cdot P' \cdot X[b\,..\,c) \cdot \overline{P'} \cdot X[d\,..\,|X|))$$

holds for every string X with k-synchronized fragments $X[a\,..\,b) = P$ and $X[c\,..\,d) = \overline{P}$.

Fact 3.4.11 (Fact 36, [Backurs and Onak, 2016])**.** *Let $\mathcal{M}$ be an alignment such that $\mathsf{dyck}^w_{\mathcal{M}}(X) \leq k$. If $X[a\,..\,b) \simeq_{\mathcal{M}} \overline{X[c\,..\,d)}$, then the fragments $X[a\,..\,b)$ and $X(c\,..\,d]$ are k-synchronized.*

Fact 3.4.12. *Consider a string X and an alignment $\mathcal{M} \in \mathsf{M}(X)$ such that $\mathsf{dyck}_{\mathcal{M}}(X) \leq k$ for some $k \in \mathbb{Z}_{\geq 0}$. Moreover, let $X[a\,..\,b)$ and $X[c\,..\,d)$ be k-synchronized fragments of length $\ell > 6k$.*

Then, there exist k-synchronized fragments $X[a'\,..\,b')$ and $X[c'\,..\,d')$ of length $\ell' \geq \frac{\ell-6k}{k+1}$, such that $X[a'\,..\,b') \simeq_{\mathcal{M}} \overline{X[c'\,..\,d')}$ and $a \leq a' \leq b' \leq b \leq c \leq c' \leq d' \leq d$. Furthermore, we then have $|(a+d)-(a'+d')| \leq 4k$.

Proof. Since $\mathcal{M}$ is non-crossing, it is disjoint with $[a\,..\,b) \times [d\,..\,|X|)$ or $[0\,..\,a) \times [c\,..\,d)$. By symmetry (up to the reverse complement), let us assume that $\mathcal{M}$ is disjoint with $[a\,..\,b)\times[d\,..\,|X|)$. Consider $x \in [a\,..\,b-4k)$ such that $X[x] \simeq_{\mathcal{M}} \overline{X[y]}$. The assumption implies that $y < d$. Moreover, $b - x = H(b) - H(x) > 4k$, so $H(x) < H(b) - 4k$. At the same time, $|H(y+1) - H(x)| \leq 2k$, so $H(y+1) < H(b) - 2k$. Since $X[a\,..\,b)$ and $X[c\,..\,d)$ are k-synchronized, this means that $y+1 \notin [b\,..\,c]$, i.e., $y \in [c\,..\,d)$. Consider the fragment $X[a\,..\,b-4k)$ and the minimal subfragment of $X[c\,..\,d)$ containing positions that $\mathcal{M}$ matches perfectly to positions $X[x]$ with $x \in [a\,..\,b-4k)$. These two fragments contain at most $2k$ positions that are deleted or matched imperfectly. The remaining positions constitute a common subsequence of $X[a\,..\,b-4k)$ and $X[c\,..\,d)$; this subsequence can be interrupted at most k times, so there is a contiguous subsequence $X[a'\,..\,b') \simeq_{\mathcal{M}} X[c'\,..\,d')$ of length at least $\frac{\ell-6k}{k+1}$. Due to $|H(a) - H(d)| \leq 2k$ and $|H(a') - H(d')| \leq 2k$, we have $4k \geq |H(a) - H(d) - H(a') + H(d')| = |a - a' + d - d'|$. □

Lemma 3.4.13. *Let $k \in \mathbb{Z}_+$, let $Q \in T^*$ be a string, and let $e, e' \in \mathbb{Z}_{\geq 8k}$. Then Q^e and $Q^{e'}$ are $\mathsf{dyck}^w_{\leq k}$-equivalent for every skewmetric weight function w.*

Proof. We assume without loss of generality that Q is primitive. (If $Q = R^m$ for $m \in \mathbb{Z}_{\geq 2}$, then $Q^e = R^{me}$ and $Q^{e'} = R^{me'}$ can be interpreted as powers of R rather than powers of Q.) Let $q = |Q|$. Consider a string X and positions p_T, $p_{\overline{T}}$ such that $Q^e = X[p_T\,..\,p_T + e\cdot q)$ and $\overline{Q^e} = X(p_{\overline{T}} - e\cdot q\,..\,p_{\overline{T}}]$ are k-synchronized fragments. Denote $X[0\,..\,p_T)\cdot Q^{e'}\cdot X[p_T + e\cdot q\,..\,p_{\overline{T}} - e\cdot q]\cdot\overline{Q^{e'}}\cdot X(p_{\overline{T}}\,..\,|X|)$. Moreover, let $\mathcal{M} \in \mathsf{M}(X)$ be an alignment such that

$\mathsf{dyck}^w(X,Y) = \mathsf{dyck}^w_{\mathcal{M}}(X,Y) \leq k$.

Claim 3.4.14. *There exist* $i_T, i_{\overline{T}} \in [0\,..\,7k]$ *such that*

$$X[p_T + i_T \cdot q\,..\,p_T + (i_T+1)\cdot q) \simeq_{\mathcal{M}} \overline{X(p_{\overline{T}} - (i_{\overline{T}}+1)\cdot q\,..\,p_{\overline{T}} - i_{\overline{T}}\cdot q]}.$$

Proof. Consider the $8k$ occurrences of Q starting at positions $p_T + i \cdot q$ for $i \in [0\,..\,7k]$ (let this fragment be P) and $8k$ occurrences of $\overline{Q}$ ending at positions $p_{\overline{T}} - i \cdot q$ for $i \in [0\,..\,7k]$ (let this fragment be $\overline{P}$). Note $P, \overline{P}$ are also are k-synchronized fragments. Thus following Fact 3.4.12, there exists at least one occurrence of Q in P, starting at index ℓ such that $\mathcal{M}$ matches it exactly with a fragment in $\overline{P}$. We can thus define $i_T \in [0\,..\,7k]$ so that $\mathcal{M}$ matches $X[p_T + i_T \cdot q\,..\,p_T + (i_T+1)\cdot q)$ exactly to some fragment $X(s_{\overline{T}} - q\,..\,s_{\overline{T}}] \in \overline{P}$. By definition of $\overline{P}$, we have $s_{\overline{T}} \geq p_{\overline{T}} - 7kq$. Furthermore, since Q is primitive (i.e., distinct from all its non-trivial cyclic rotations), we conclude that $s_{\overline{T}} = p_{\overline{T}} - i_{\overline{T}} \cdot q$ for some $i_{\overline{T}} \in [0\,..\,7k]$.

□

Now, if $Q^e = X[p_T\,..\,p_T + e\cdot q)$ is replaced with $Q^{e'}$ and $\overline{Q^e} = X(p_{\overline{T}} - e\cdot q\,..\,p_{\overline{T}}]$ is replaced with $\overline{Q^{e'}}$ for $e' \geq e-1$, we can interpret this as replacing $Q = X[p_T + i_T \cdot q\,..\,p_T + (i_T+1)\cdot q)$ with $Q^{1+e'-e}$ and $\overline{Q} = X(p_{\overline{T}} - (i_{\overline{T}}+1)\cdot q\,..\,p_{\overline{T}} - i_{\overline{T}}\cdot q]$ with $\overline{Q^{1+e'-e}}$. By Claim 3.4.14, $\mathcal{M}$ can be trivially adapted without modifying its cost, and hence $\mathsf{dyck}^w(X') \leq \mathsf{dyck}^w_{\mathcal{M}}(X) = \mathsf{dyck}^w(X)$. If $e' < e-1$, we repeat the above argument to decrement the exponent e one step at a time, still concluding that $\mathsf{dyck}^w(X') \leq \mathsf{dyck}^w(X)$. In either case, the converse inequality follows by symmetry between (X,e) and (X',e'). □

We say that a string $P \in T^*$ avoids k-periodicity if it does not contain any substring Q^{8k+1}

with $|Q| \in [1\,..\,4k]$.

Lemma 3.4.15. *Let $k \in \mathbb{Z}_+$ and let $P, P' \in T^*$ be strings of lengths at least $156k^3$ such that $P[0\,..\,78k^3) = P'[0\,..\,78k^3)$ and $P[|P|-78k^3\,..\,|P|) = P'[|P'|-78k^3\,..\,|P'|)$ avoid k-periodicity. Then, P and P' are $\mathsf{dyck}^w_{\le k}$-equivalent for every skewmetric weight function w.*

Proof. Consider a string X and positions p_T, $p_{\overline{T}}$ such that $P = X[p_T\,..\,p_T+|P|)$, $\overline{P} = X(p_{\overline{T}}-|P|\,..\,p_{\overline{T}}]$ are k-synchronized fragments. Denote $X' = X[0\,..\,p_T)\cdot P'\cdot X[p_T+|P|\,..\,p_{\overline{T}}-|P|]\cdot\overline{P'}\cdot X(p_{\overline{T}}\,..\,|X|]$. Moreover, let $\mathcal{M} \in \mathsf{M}(X)$ be an alignment such that $\mathsf{dyck}^w(X) = \mathsf{dyck}^w_{\mathcal{M}}(X) \le k$.

Claim 3.4.16. *There exist $d, e \in [0\,..\,78k^3]$ such that $(p_T+d, p_{\overline{T}}-d) \in \mathcal{M}$ and $(p_T+|P|-e, p_{\overline{T}}-|P|+e) \in \mathcal{M}$.*

Proof. By Fact 3.4.12, $X[p_T\,..\,p_T+78k^3)$ contains a fragment of length at least $\frac{78k^3-6k}{k+1} \ge 36k^2$ that $\mathcal{M}$ matches perfectly to a fragment of $X(p_{\overline{T}}-78k^3\,..\,p_{\overline{T}}]$ Thus, let $R := X[r_T\,..\,r_T+|R|)$ be a fragment of length at least $36k^2$ contained in $X[p_T\,..\,p_T+|P|)$ that $\mathcal{M}$ matches perfectly to $X[r_{\overline{T}}-|R|\,..\,r_{\overline{T}}) = \overline{R}$. Moreover, let $r'_{\overline{T}} := p_T + p_{\overline{T}} - r_T$. If $r_{\overline{T}} = r'_{\overline{T}}$, then the claim is satisfied for $d = r_T - p_T = p_{\overline{T}} - r_{\overline{T}}$. Otherwise, both $X[r_{\overline{T}}-|R|\,..\,r_{\overline{T}})$ and $X[r'_{\overline{T}}-|R|\,..\,r'_{\overline{T}})$ are occurrences of $\overline{R}$ in X. Moreover, $0 < |r_{\overline{T}} - r'_{\overline{T}}| \le |(p_{\overline{T}} - r'_{\overline{T}}) - (p_{\overline{T}} - r_{\overline{T}})| \le |(r_T - p_T) - (p_{\overline{T}} - r_{\overline{T}})| + |(r_T - p_T) - (p_{\overline{T}} - r'_{\overline{T}})| \le 2\mathsf{dyck}^w_{\mathcal{M}}(X) + 2k \le 4k$. Hence, $\mathsf{per}(\overline{R}) \le |r_{\overline{T}} - r'_{\overline{T}}| \le 4k$ and $\exp(\overline{R}) \ge \frac{|\overline{R}|}{4k} \ge 9k$. Since $X[r'_{\overline{T}}-|\overline{R}|\,..\,r'_{\overline{T}})$ is contained in $X(p_{\overline{T}}-|P|\,..\,p_{\overline{T}}] = \overline{P[0\,..\,78k^3)}$, this contradicts the assumption that $\overline{P[0\,..\,78k^3)}$ and thus $P[0\,..\,78k^3)$ avoids k-periodicity.

The second part of the claim is proved analogously. □

As $X[p_T+d\,..\,p_T+|P|-e) \in T$ and $X[p_T+d\,..\,p_T+|P|-e) = \overline{X(p_{\overline{T}}-|P|+e\,..\,p_{\overline{T}}-d]}$, the optimality of $\mathcal{M}$ guarantees that $X[p_T+d\,..\,p_T+|P|-e) \simeq_{\mathcal{M}} \overline{X(p_{\overline{T}}-|P|+e\,..\,p_{\overline{T}}-d]}$.

Hence, if $P = X[p_T \,..\, p_T + |P|) = \overline{X(p_{\overline{T}} - |P| \,..\, p_{\overline{T}}]}$ is replaced with P', we can interpret this as $P[d \,..\, |P| - e) = X[p_T + d \,..\, p_T + |P| - e) = \overline{X(p_{\overline{T}} - |P| + e \,..\, p_{\overline{T}} - d]}$ with $P'[d \,..\, |P'| - e)$. Since $X[p_T + d \,..\, p_T + |P| - e) \simeq_{\mathcal{M}} \overline{X(p_{\overline{T}} - |P| + e \,..\, p_{\overline{T}} - d]}$, the alignment $\mathcal{M}$ can be trivially adapted without modifying its cost, and therefore $\mathsf{dyck}^w(X') \leq \mathsf{dyck}^w_{\mathcal{M}}(X) = \mathsf{dyck}^w(X)$. The converse inequality follows by symmetry between (X, P) and (X', P'). □

Corollary 3.4.17. *Let $k \in \mathbb{Z}_+$. For every string $P \in T^*$, there exists a string of length at most $156k^3$ that is $\mathsf{dyck}^w_{\leq k}$-equivalent to P for every skewmetric weight function w.*

Proof. We proceed by induction on $|P|$ with the trivial base case of $|P| \leq 156k^3$. If $|P| \geq 156k^3$ and P avoids k-periodicity, then Lemma 3.4.15 implies that P is equivalent to a string $P' := P[0 \,..\, 78k^3) \cdot P[|P| - 78k^3 \,..\, |P|)$ of length $156k^3$. Thus, suppose that P contains a fragment $P[i \,..\, j) = Q^{8k+1}$ and $|Q| \in [1 \,..\, 4k]$. By Lemma 3.4.13, Q^{8k+1} is equivalent to Q^{8k}, and thus P is equivalent to a string $P' := P[0 \,..\, i) \cdot P[i + |Q| \,..\, |P|)$. By the inductive assumption, P' is equivalent to some string P'' of length at most $156k^3$, and, by transitivity of the considered equivalence, P is also equivalent to P''. □

3.4.4 Algorithm

Lemma 3.4.18. *There exists a linear-time algorithm that, given a string P and an integer $k \in \mathbb{Z}_+$, constructs a string P' of length at most $156k^3$ that is $\mathsf{dyck}^w_{\leq k}$-equivalent to P for every skewmetric weight function w. Moreover P' avoids k-periodicity.*

Proof. We apply Algorithm 9 with $e = 8k$ and $\mathcal{Q}$ consisting of all primitive strings in T^* of length in $[1 \,..\, 4k]$. If the resulting string P' satisfies $|P'| < 156k^3$, we return P'. By Lemmas 3.2.13 and 3.4.13, the string P' is $\mathsf{dyck}^w_{\leq k}$-equivalent to P and avoids k-periodicity. Thus, if $|P'| \leq$

Algorithm 16: Construct a string of length at most $156k^3$ that is $\mathsf{dyck}^w_{\leq k}$-equivalent to P.

```
DyckReduction(P, k):
    P' ← PeriodicityReduction(P, 8k, {Q ∈ T⁺ : |Q| ≤ 4k and Q is primitive});
    if |P'| ≥ 156k³ then return P'[0..78k³) · P'[|P'| − 78k³..|P'|);
    else return P';
```

$156k^3$, then the algorithm is correct. Otherwise, we return $P'[0\,..\,78k^3) \cdot P'[|P'| - 78k^3\,..\,|P'|)$. $P'[0\,..\,78k^3)$ and $P'[|P'| - 78k^3\,..\,|P'|)$ both avoid k-periodicity, so $P'[0\,..\,78k^3) \cdot P'[|P'| - 78k^3\,..\,|P'|)$ is $\mathsf{dyck}^w_{\leq k}$-equivalent to P by Lemma 3.4.15. Due to Lemma 3.2.13, the running time is linear (testing whether a primitive fragment belongs to $\mathcal{Q}$ simplifies to checking if its length does not exceed $4k$.) □

Theorem 3.4.19. *There exists a $O(n + k^5)$-time algorithm that, given a preprocessed string X and an integer $k \in \mathbb{Z}_+$, constructs strings X' of lengths at most $630k^4$ such that $\mathsf{dyck}^w_{\leq k}(X) = \mathsf{dyck}^w_{\leq k}(X')$ holds for every skewmetric weight function w.*

Proof. Our procedure is implemented as Algorithm 17. First, if X is already of length at most $630k^4$, then we return X as it is. If $\mathsf{dyck}(X) > k$, we return strings a^{k+1}, where $a \in \Sigma$ is an arbitrary character. If $\mathsf{dyck}(X) \leq k$, we construct a Dyck language alignment $\mathcal{M} \in \mathsf{M}(X)$ of (unweighted) cost at most k. We then build the output string X' using $\mathcal{M}$ as follows: scan X from left to right, if the the scanned character $X[i]$ is edited by $\mathcal{M}$ we append it to X' (here $\texttt{Order}(i, j) = 1$ if $j > i$ otherwise it is 0). Otherwise $X[i]$ is matched under $\mathcal{M}$. If $X[i] \in T$, we proceed with scanning the following characters to identify the maximal fragment $P = X[i\,..\,j) \in T^*$ such that there is a fragment $X(i'\,..\,j']$ where $\mathsf{dyck}(X[i\,..\,j)X(i'\,..\,j']) = 0$ and $\mathcal{M}$ matches $X[i\,..\,j)$ with $X(i'\,..\,j']$. Next we apply the reduction of Lemma 3.4.18 on P and append the

Algorithm 17: Construct strings X' of length at most $630k^4$ such that $\mathsf{dyck}^{w}_{\leq k}(X) = \mathsf{dyck}^{w}_{\leq k}(X')$

```
DyckKernel(X, k):
    if |X| ≤ 630k^4 then return (X);
    if dyck(X) > k then return (a^{k+1}) for some a ∈ Σ;
    Let M ∈ M(X) be a dyck language alignment satisfying dyck_M(X) ≤ k;
    X', P, Q ← ε;
    for i ← 0 to n − 1 do
        if M(i) = ⊥ or dyck(X[i]X[M(i)]) · Order(i, M(i)) = 1 or
          dyck(X[M(i)]X[i]) · Order(M(i), i) = 1 then
            X' ← X' · X[i]
        else if X[i] ∈ T and M(i + 1) = M(i) − 1 and
          dyck(X[i + 1]X[M(i) − 1]) = 0 then
            P ← P · X[i]
        else if X[i] ∈ T then
            P ← P · X[i];
            P ← DyckReduction(P, k);
            X' ← X' · P;
            P ← ε;
        else if X[i] ∈ T̄ and M(i + 1) = M(i) − 1 and
          dyck(X[M(i) − 1]X[i + 1]) = 0 then
            Q ← Q · X[i]
        else
            Q ← Q · X[i];
            Q ← the reverse of DyckReduction(reverse of Q, k);
            X' ← X' · Q;
            Q ← ε;
    return (X')
```

reduced string to X'. Otherwise $X[i] \in \overline{T}$. Here also we proceed with scanning the following characters to identify the maximal fragment $Q = X[i\,..\,j) \in \overline{T}^*$ such that there is a fragment $X(i'\,..\,j']$ where $\mathsf{dyck}(X(i'\,..\,j']X[i\,..\,j)) = 0$ and $\mathcal{M}$ matches $X[i\,..\,j)$ with $X(i'\,..\,j']$. Next we consider the $\overline{Q}$ (note $\overline{Q} \in T^*$), apply the reduction of Lemma 3.4.18 on $\overline{Q}$ and append the reverse complement of the reduced string to X'.

Let us now prove that the resulting string X' satisfies $\mathsf{dyck}^w_{\leq k}(X) = \mathsf{dyck}^w_{\leq k}(X')$. This is trivial when the algorithm returns X in Algorithm 17. If $\mathsf{dyck}(X) > k$, then $\mathsf{dyck}_{\leq k}(X) = \mathsf{dyck}_{\leq k}(a^{k+1}) = \infty$ and thus also $\mathsf{dyck}^w_{\leq k}(X) = \mathsf{dyck}^w_{\leq k}(a^{k+1}) = \infty$ because the weighted Dyck edit distance with a normalized weight function is at least as large as the unweighted Dyck edit distance. In the remaining case we assume $\mathsf{dyck}(X) \leq k$. Let $S = \{P_1, \ldots, P_\ell\}$ be the set of fragments from T^* that are ever generated and processed using `DyckReduction()` routine at Line 13. Similarly let $\overline{S} = \{Q_1, \ldots, Q_{\ell'}\}$ be the set of fragments from $\overline{T}^*$ that are ever generated and whose reveres complements are processed using `DyckReduction()` routine at Line 20. By construction it is trivial to follow that (i) the fragments are disjoint; (ii) for all $i \in [0\,..\,n)$, if $X[i] \in T$ and is not edited by $\mathcal{M}$, then there exist some P_j such that $X[i] \in P_j$ and if $X[i] \in \overline{T}$ and is not edited by $\mathcal{M}$, then there exist some Q_j such that $X[i] \in Q_j$; (iii) The fragments are maximal in a sense that if $P_i = X[a\,..\,b) \in S$, then either $\mathcal{M}(b) \neq \mathcal{M}(b-1) - 1$ or $X[b] \neq \overline{X[\mathcal{M}(b)]}$ and same holds for the fragments in $\overline{S}$. Next we prove for each $P_i = X[a\,..\,b) \in S$, $\exists Q_j = X(c\,..\,d] \in \overline{S}$, such that $|P_i| = |Q_j|$ and for all $k \in [0\,..\,|P_i|)$, $(a+k, d-k) \in \mathcal{M}$ and $X[a+k] = \overline{X[d-k]}$. For this we first claim that $\mathcal{M}(b-1)$ is a starting index of some $Q_j \in \overline{S}$. As otherwise $\mathcal{M}(b) = \mathcal{M}(b-1)-1$ and $X[b] \neq \overline{X[\mathcal{M}(b)]}$; this contradicts the maximality of P_i. Further by construction for all $k \in [a\,..\,b)$, $X[\mathcal{M}(k)] \in Q_j$. Finally we argue $\mathcal{M}(a)$ is a ending index of Q_j. As otherwise $\mathcal{M}(a-1) = \mathcal{M}(a) + 1$ and this contradicts the fact that a is the starting index of some segment

from S. Similarly we can show for each $Q_j \in \overline{S}$ there is a corresponding match $P_i \in S$ and this provides an one to one correspondence between a pair of fragments from S and $\overline{S}$. Thus for a fragment $P_i \in S$ let $\mathcal{M}(P_i)$ represents the corresponding matched fragments from $\overline{S}$ and we can represent $S \cup \overline{S} = \cup_{i\in[\ell]}(P_i, \mathcal{M}(P_i))$. Following Fact 3.4.11, $P_i, \mathcal{M}(P_i)$ are k-synchronized. Next in the algorithm for each pair $(P_i, \mathcal{M}(P_i))$ we add strings $\texttt{DyckReduction}(P_i)$ representing P_i and $\overline{\texttt{DyckReduction}(P_i)}$ (note $\overline{\mathcal{M}(P_i)} = P_i$) representing $\mathcal{M}(P_i)$ to X'. Following the fact that every character that is not contained in a fragment from $S \cup \overline{S}$ is edited by $\mathcal{M}$ and thus copied to X' directly, by applying Lemma 3.4.18 repeatedly for every pair $(P_i, \mathcal{M}(P_i))$, we claim $\mathsf{dyck}^w_{\le k}(X) = \mathsf{dyck}^w_{\le k}(X')$.

Next, we show that the returned string is of length at most $630k^4$. This is clear when the algorithm terminates at Line 2 or 3. Otherwise, we create a string X', to which we directly copy the characters that are edited by $\mathcal{M}$. However there are at most $2k$ characters that $\mathcal{M}$ deletes or substitutes. Next we identify maximal fragments $P = X[i\,..\,j) \in T^*$ such that there is another fragment $X(i'\,..\,j'] \in \overline{T}^*$ that is matched with P by $\mathcal{M}$. The maximality of P and the preprocessing of X ensure that at least one of $X[j]$ and $X[i']$ is edited by $\mathcal{M}$, We call these characters the boundary characters for P. Notice for any two distinct fragments $P, P' \in T^*$, the the boundary characters are different and by construction P, P' are disjoint. As there are at most $2k$ characters that $\mathcal{M}$ edits, we conclude there can be at most $2k$ fragments over T^*, that our algorithm can construct. For each such fragment following the reduction of Lemma 3.4.18, we add a substring of length $156k^3$ to X'. Thus the total length of all the substrings is $312k^4$. Similarly we can argue for the fragments $Q \in \overline{T}^*$. Thus we can bound the total length of X' by $2 \cdot 312k^4 + 2k < 630k^4$.

It remains to analyze the complexity of our procedure. We use the algorithm [Fried et al., 2022] to check whether $\mathsf{dyck}(X) \le k$ and, if so, construct the alignment $\mathcal{M}$. This costs $\mathcal{O}(n +$

k^5) time. Next we perform a single left to right scan of X. Throughout, all the conditions in the *if/else* statements can be checked in $O(1)$ time. Moreover any character is passed to the `DyckReduction()` routine at most twice. Thus following Lemma 3.4.18, given X and $\mathcal{M}$, X' can be constructed in linear time. □

Proof of Theorem 3.1.3. We first preprocess X in linear time following the steps described in Section 3.4.2.1 to build strings X' such that $\mathsf{dyck}^w_{\leq k}(X') = \mathsf{dyck}^w_{\leq k}(X)$. Next we apply Theorem 3.4.19 on X', to build strings X'' of length $\mathcal{O}(k^4)$ such that $\mathsf{dyck}^w_{\leq k}(X'') = \mathsf{dyck}^w_{\leq k}(X')$. This takes time $O(n + k^5)$. Lastly if $X = a^{k+1}$ (this can be checked in time $O(k)$) output the distance is $> k$. Otherwise we compute $\mathsf{dyck}^w_{\leq k}(X'')$ using the dynamic program algorithm from [Myers, 1995] in time $O(k^{12})$. Thus the total running time is $O(n + k^{12})$. □

3.5 Deferred Proofs from Section 3.2

In the following, we give the missing proofs of facts from Section 3.2.

Fact 3.2.5. *If w is a quasimetric on $\bar{\Sigma}$, then ed^w is a quasimetric on Σ^*. In this case, $\mathsf{ed}^w(X, Y)$ can be equivalently defined as the minimum cost of a sequence of edits transforming X into Y.*

Proof. Consider arbitrary strings $X, Y, Z \in \Sigma^*$ as well as alignments $\mathcal{A} = (x_t, y_t)_{t=0}^{m} \in \mathsf{A}(X, Y)$ and $\mathcal{B} = (\hat{y}_t, \hat{z}_t)_{t=0}^{\hat{m}} \in \mathsf{A}(Y, Z)$. We construct a *product alignment* $\mathcal{A} \otimes \mathcal{B} \in \mathsf{A}(X, Z)$ such that $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X, Z) \leq \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z)$. Let us denote $\mathcal{A}' = (x_t, y_t)_{t=0}^{m-1}$ and $\mathcal{B}' = (\hat{y}_t, \hat{z}_t)_{t=0}^{\hat{m}-1}$, as well as $X' = X[0\,..\,|X| - 1)$ if $X \neq \varepsilon$, $Y' = Y[0\,..\,|Y| - 1)$ if $Y \neq \varepsilon$, and $Z' = Z[0\,..\,|Z| - 1)$ if $Z \neq \varepsilon$. We proceed by induction on $m + \hat{m}$ and consider several cases based on how $\mathcal{A}$ and $\mathcal{B}$ handle the trailing characters of X, Y, and Z.

1. $m = \hat{m} = 0$. In this case, $X = Y = Z = \varepsilon$, and we define $\mathcal{A} \otimes \mathcal{B} := (0, 0)$. Trivially, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X, Z) = 0 = \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z)$.

2. $(x_{m-1}, y_{m-1}) = (|X| - 1, |Y|)$, that is, $\mathcal{A}$ deletes $X[|X| - 1]$. In this case, $\mathcal{A}' \in \mathsf{A}(X', Y)$, and we define $\mathcal{A} \otimes \mathcal{B} := (\mathcal{A}' \otimes \mathcal{B}) \odot (|X|, |Z|)$, where $\odot$ denotes concatenation, so that $\mathcal{A} \otimes \mathcal{B}$ deletes $X[|X| - 1]$. By the induction hypothesis, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X, Z) = \mathsf{ed}^w_{\mathcal{A}'\otimes\mathcal{B}}(X', Z) + w(X[|X|-1], \varepsilon) \leq \mathsf{ed}^w_{\mathcal{A}'}(X', Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z) + w(X[|X|-1], \varepsilon) = \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z)$.

3. $(\hat{y}_{\hat{m}-1}, \hat{z}_{\hat{m}-1}) = (|Y| - 1, |Z|)$, that is, $\mathcal{B}$ inserts $Z[|Z| - 1]$. In this case, $\mathcal{B}' \in \mathsf{A}(Y, Z')$, and we define $\mathcal{A} \otimes \mathcal{B} := (\mathcal{A} \otimes \mathcal{B}') \odot (|X|, |Z|)$ so that $\mathcal{A} \otimes \mathcal{B}$ inserts $Z[|Z| - 1]$. By the induction hypothesis, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X, Z) = \mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}'}(X, Z') + w(\varepsilon, Z[|Z| - 1]) \leq \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}'}(Y, Z') + w(\varepsilon, Z[|Z| - 1]) = \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z)$.

4. $(x_{m-1}, y_{m-1}) = (|X|, |Y| - 1)$ and $(\hat{y}_{\hat{m}-1}, \hat{z}_{\hat{m}-1}) = (|Y| - 1, |Z|)$, that is, $\mathcal{A}$ inserts $Y[|Y| - 1]$ and $\mathcal{B}$ deletes $Y[|Y| - 1]$. In this case, $\mathcal{A}' \in \mathsf{A}(X, Y')$ and $\mathcal{B}' \in \mathsf{A}(Y', Z)$, and we define $\mathcal{A} \otimes \mathcal{B} := \mathcal{A}' \otimes \mathcal{B}'$. By the induction hypothesis, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X, Z) = \mathsf{ed}^w_{\mathcal{A}'\otimes\mathcal{B}'}(X, Z) \leq \mathsf{ed}^w_{\mathcal{A}'}(X, Y') + \mathsf{ed}^w_{\mathcal{B}'}(Y', Z) \leq \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z)$.

5. $(x_{m-1}, y_{m-1}) = (|X|, |Y| - 1)$ and $(\hat{y}_{\hat{m}-1}, \hat{z}_{\hat{m}-1}) = (|Y| - 1, |Z| - 1)$, that is, $\mathcal{A}$ inserts $Y[|Y| - 1]$ and $\mathcal{B}$ aligns $Y[|Y| - 1]$ with $Z[|Z| - 1]$. In this case, $\mathcal{A}' \in \mathsf{A}(X, Y')$ and $\mathcal{B}' \in \mathsf{A}(Y', Z')$, and we define $\mathcal{A} \otimes \mathcal{B} := (\mathcal{A}' \otimes \mathcal{B}') \odot (|X|, |Z|)$ so that $\mathcal{A} \otimes \mathcal{B}$ inserts $Z[|Z|-1]$. By the induction hypothesis, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X, Z) = \mathsf{ed}^w_{\mathcal{A}'\otimes\mathcal{B}'}(X, Z') + w(\varepsilon, Z[|Z|-1]) \leq \mathsf{ed}^w_{\mathcal{A}'}(X, Y') + \mathsf{ed}^w_{\mathcal{B}'}(Y', Z') + w(\varepsilon, Y[|Y| - 1]) + w(Y[|Y| - 1], Z[|Z| - 1]) = \mathsf{ed}^w_{\mathcal{A}}(X, Y) + \mathsf{ed}^w_{\mathcal{B}}(Y, Z)$.

6. $(x_{m-1}, y_{m-1}) = (|X| - 1, |Y| - 1)$ and $(\hat{y}_{\hat{m}-1}, \hat{z}_{\hat{m}-1}) = (|Y| - 1, |Z|)$, that is, $\mathcal{A}$ aligns $X[|X| - 1]$ with $Y[|Y| - 1]$ and $\mathcal{B}$ deletes $Y[|Y| - 1]$. In this case, $\mathcal{A}' \in \mathsf{A}(X', Y')$ and $\mathcal{B}' \in \mathsf{A}(Y', Z)$, and we define $\mathcal{A} \otimes \mathcal{B} := (\mathcal{A}' \otimes \mathcal{B}') \odot (|X|, |Z|)$ so that $\mathcal{A} \otimes \mathcal{B}$ deletes

$X[|X|-1]$. By the induction hypothesis, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X,Z) = \mathsf{ed}^w_{\mathcal{A}'\otimes\mathcal{B}'}(X',Z) + w(X[|X|-1],\varepsilon) \leq \mathsf{ed}^w_{\mathcal{A}'}(X',Y') + \mathsf{ed}^w_{\mathcal{B}'}(Y',Z) + w(X[|X|-1],Y[|Y|-1]) + w(Y[|Y|-1],\varepsilon) = \mathsf{ed}^w_{\mathcal{A}}(X,Y) + \mathsf{ed}^w_{\mathcal{B}}(Y,Z)$.

7. $(x_{m-1},y_{m-1}) = (|X|-1,|Y|-1)$ and $(\hat{y}_{\hat{m}-1},\hat{z}_{\hat{m}-1}) = (|Y|-1,|Z|-1)$, that is, $\mathcal{A}$ aligns $X[|X|-1]$ with $Y[|Y|-1]$ and $\mathcal{B}$ aligns $Y[|Y|-1]$ with $Z[|Z|-1]$. In this case, $\mathcal{A}' \in \mathsf{A}(X',Y')$ and $\mathcal{B}' \in \mathsf{A}(Y',Z')$, and we define $\mathcal{A}\otimes\mathcal{B} := (\mathcal{A}'\otimes\mathcal{B}')\odot(|X|,|Z|)$ so that $\mathcal{A}\otimes\mathcal{B}$ aligns $X[|X|-1]$ with $Z[|Z|-1]$. By the induction hypothesis, $\mathsf{ed}^w_{\mathcal{A}\otimes\mathcal{B}}(X,Z) = \mathsf{ed}^w_{\mathcal{A}'\otimes\mathcal{B}'}(X',Z') + w(X[|X|-1],Z[|Z|-1]) \leq \mathsf{ed}^w_{\mathcal{A}'}(X',Y') + \mathsf{ed}^w_{\mathcal{B}'}(Y',Z) + w(X[|X|-1],Y[|Y|-1]) + w(Y[|Y|-1],Z[|Z|-1]) = \mathsf{ed}^w_{\mathcal{A}}(X,Y) + \mathsf{ed}^w_{\mathcal{B}}(Y,Z)$.

It is easy to check that the above cases cover all the possibilities. In particular, Case 2 covers the case of $\hat{m} = 0 < m$ whereas Case 3 covers the case of $m = 0 < \hat{m}$. We also remark that Cases 2 and 3 are sometimes both applicable; by convention, we then follow Case 2. Finally, we note that Cases 5–7 rely on the assumption that w satisfies the triangle inequality. This completes the proof of the first part of the fact.

To show that $\mathsf{ed}^w(X,Y)$ can be equivalently defined as the minimum cost of a sequence of edits transforming X into Y, we first consider each of the operations in a minimum alignment $\mathcal{A}$ of X and Y individually to build a sequence of edits S from $\mathcal{A}$. We iterate through all pairs (x_t,y_t) of $\mathcal{A}$ from right to left starting with $t = m-1$, stopping after $t = 0$ has been handled, and building S according to the definition of alignments:

1. If $(x_t,y_t) = (x_{t+1}-1,y_{t+1}-1)$ and $X[x_t] = Y[y_t]$, we add nothing to S.
2. If $(x_t,y_t) = (x_{t+1}-1,y_{t+1}-1)$ and $X[x_t] \neq Y[y_t]$, we add a substitution of $X[x_t]$ with $Y[y_t]$ to S.

3. If $(x_t, y_t) = (x_{t+1} - 1, y_{t+1})$, we add a deletion of $X[x_t]$ to S.

4. If $(x_t, y_t) = (x_{t+1}, y_{t+1} - 1)$, we add an insertion of $Y[y_t]$ at position x_t in X to S.

In all cases, we decrement t by 1. Clearly the resulting sequence of edits has the same cost as $\mathcal{A}$, and by the definition of alignments, S transforms X into Y. We now consider a minimum sequence of edits S that transforms X to Y and build an alignment $\mathcal{A} \in \mathsf{A}(X, Y)$ from S such that $\mathsf{ed}^w_{\mathcal{A}}(X, Y) \leq cost(S)$ (we let $cost(S)$ denote the total cost of edits by S). We use notation $\mathcal{A}', X', Y'$ as before, and proceed by induction to construct $\mathcal{A}$:

1. If $X[|X|-1]$ is deleted by S and a character is inserted at the end of X, then $\mathcal{A}' \in \mathsf{A}(X', Y')$ and we set $\mathcal{A} = \mathcal{A}' \odot (|X|, |Y|)$. We note that the inserted character c may be substituted to $Y[|Y| - 1]$. We let S' be the sequence S without the insertion, deletion, and if possible substitution on the last character of X. By the induction hypothesis and triangle inequality, $\mathsf{ed}_{\mathcal{A}}(X, Y) = \mathsf{ed}_{\mathcal{A}'}(X', Y') + w(X[|X| - 1], Y[|Y| - 1]) \leq \mathsf{ed}_{\mathcal{A}'}(X', Y') + w(X[|X| - 1], \varepsilon) + w(\varepsilon, Y[|Y| - 1]) \leq \mathsf{ed}_{\mathcal{A}'}(X', Y') + w(X[|X| - 1], \varepsilon) + w(\varepsilon, c), + w(c, Y[|Y| - 1]) \leq cost(S') + w(X[|X| - 1], \varepsilon) + w(\varepsilon, c), + w(c, Y[|Y| - 1]) = cost(S)$.
2. If $X[|X|-1]$ is deleted by S and no character is inserted at the end of X, then $\mathcal{A}' \in \mathsf{A}(X', Y)$ and we set $\mathcal{A} = \mathcal{A}' \odot (|X|, |Y|)$. We let S' be the sequence S without the deletion of $X[|X| - 1]$. By the induction hypothesis, $\mathsf{ed}_{\mathcal{A}}(X, Y) = \mathsf{ed}_{\mathcal{A}'}(X', Y) + w(X[|X| - 1], \varepsilon) \leq cost(S') + w(X[|X| - 1], \varepsilon) = cost(S)$.
3. If a character is inserted at the end of X, then $\mathcal{A}' \in \mathsf{A}(X, Y')$ and we set $\mathcal{A} = \mathcal{A}' odot(|X|, |Y|)$. We let S' be the sequence S without this insertion. By the induction hypothesis, $\mathsf{ed}_{\mathcal{A}}(X, Y) = \mathsf{ed}_{\mathcal{A}'}(X, Y') + w(\varepsilon, Y[|Y| - 1], \varepsilon) \leq cost(S') + w(\varepsilon, Y[|Y| - 1]) = cost(S)$.
4. If $X[|X|-1]$ is substituted by S, then $\mathcal{A}' \in \mathsf{A}(X', Y')$ and we set $\mathcal{A} = \mathcal{A}' \odot (|X|, |Y|)$. We let

S' be the sequence of S without any substitutions of $X[|X|-1]$ and let C be an ordered list of characters substituted by S at $X[|X|-1]$. Then, by the induction hypothesis, $\mathsf{ed}_{\mathcal{A}}(X,Y) = \mathsf{ed}_{\mathcal{A}'}(X',Y') + w(X[|X|-1], Y[|Y|-1]) \leq \mathsf{ed}_{\mathcal{A}'}(X',Y') + \sum_{i=0}^{|C|-1} w(c_i, c_{i+1}) \leq cost(S') + \sum_{i=0}^{|C|-1} w(c_i, c_{i+1}) = cost(S)$.

By induction, we can see that there exists an alignment $\mathcal{A}$ with cost at most that of S.

□

Fact 3.2.6. *Consider a string X and its fragment $X[i\,..\,j)$. Then, for every quasimetric w, we have* $\mathsf{ed}^w(X, X[i\,..\,j)) = \mathsf{ed}^w(X[0\,..\,i) \cdot X[j\,..\,|X|), \varepsilon)$.

Proof. The unique alignment in $\mathsf{A}(X[0\,..\,i) \cdot X[j\,..\,|X|), \varepsilon)$ deletes all characters, and therefore $\mathsf{ed}^w(X[0\,..\,i) \cdot X[j\,..\,|X|), \varepsilon) = \sum_{u \in [0\,..\,i) \cup [j\,..\,|X|)} w(X[u], \varepsilon)$.

Next, consider an alignment $\mathcal{A} \in \mathsf{A}(X, X[i\,..\,j))$ that deletes $X[0\,..\,i)$, matches $X[i\,..\,j)$ perfectly, and deletes $X[j\,..\,|X|)$. Hence, $\mathsf{ed}^w(X, X[i\,..\,j)) \leq \mathsf{ed}^w_{\mathcal{A}}(X, X[i\,..\,j)) = \sum_{u \in [0\,..\,i) \cup [j\,..\,|X|)} w(X[u], \varepsilon) = \mathsf{ed}^w(X[0\,..\,i) \cdot X[j\,..\,|X|), \varepsilon)$.

Now, let $Y = X[i\,..\,j)$ and consider an arbitrary alignment $\mathcal{B} \in \mathsf{A}(X,Y)$. For each $u \in [0\,..\,i) \cup [j\,..\,|X|)$, we recursively define a sequence $(c_{u,t})_{t=0}^{m_u}$ so that $c_{u,0} = u$, the alignment $\mathcal{B}$ aligns $X[c_{u,t}]$ to $Y[c_{u,t+1} - i] = X[c_{u,t+1}]$ for $t \in [0\,..\,m_u)$, and $\mathcal{B}$ deletes $X[c_{u,m_u}]$. By construction, the sequences $(c_{u,t})_{t=0}^{m_u}$ are finite and each position in X belongs to at most one such sequence. Moreover, since w is quasimetric, $w(X[c_{u,t}], \varepsilon) \leq w(X[c_{u,t}], X[c_{u,t+1}]) + w(X[c_{u,t+1}], \varepsilon)$ holds for every $t \in [0\,..\,m_u)$, and thus $w(X[c_{u,0}], \varepsilon) \leq w(X[c_{u,m_u}], \varepsilon) + \sum_{t \in [0\,..\,m_u)} w(X[c_{u,t}], X[c_{u,t+1}])$. Since $\mathcal{B}$ deletes $X[c_{u,m_u}]$ and $X[c_{u,t}] \sim_{\mathcal{B}} Y[c_{u,t+1} - i] = X[c_{u,t+1}]$ holds for $t \in [0\,..\,m_u)$, this

yields

$$\begin{aligned}
\mathsf{ed}^w_{\mathcal{B}}(X,Y) &\geq \sum_{u\in[0..i)..[j\cup|X|)} \left(w(X[c_{u,m_u}],\varepsilon) + \sum_{t\in[0..m_u)} w(X[c_{u,t}],X[c_{u,t+1}]) \right) \\
&\geq \sum_{u\in[0..i)..[j\cup|X|)} w(X[u],\varepsilon) \\
&= \mathsf{ed}^w(X[0..i)\cdot X[j..|X|),\varepsilon).
\end{aligned}$$

Since $\mathcal{B}$ was chosen arbitrarily, we conclude that $\mathsf{ed}^w(X,Y) \geq \mathsf{ed}^w(X[0..i)\cdot X[j..|X|),\varepsilon)$. $\square$

Part II

Dynamic Tree and Dyck Distance

Chapter 4: Dynamic Dyck and Tree Edit Distance: Decompositions and Reductions to String Edit Distance

4.1 Introduction

The *Dyck language* represents the set of well-balanced sequences of parentheses and is one of the most fundamental context-free languages. Early work by Chomsky and Schützenberger [Chomsky and Schützenberger, 1963] showed that a nondeterministic version of Dyck is equivalent to the entire set of context-free grammars. Since then, the Dyck language has not only played a pivotal role in formal language theory but has also found numerous applications in other fields. For example, the popular *RNA folding* problem in bioinformatics requires a similar balance condition between various nucleotides [Nussinov and Jacobson, 1980, Bringmann et al., 2019, Das et al., 2022b] and the modern-day programming languages HTML, XML, and JSON maintain balanced sequences of parentheses to determine the structure of stored information. Most of these applications also require correcting errors, and thus determining how far the underlying data semantics deviate from the well-balanced property. This is formulated as the *Dyck edit distance* and has been studied since the early seventies [Aho and Peterson, 1972]. The Dyck edit distance counts the minimum number of insertions, deletions, and substitutions needed for a given parenthesis string to become well balanced, that is to become a member of the Dyck language.

Furthermore, in many of the above applications, the parenthesis strings being considered change frequently (e.g., mutations in RNA sequences, edits in programming text, etc.) and the corresponding Dyck edit distance must be recomputed many times over. This motivates the study of *dynamic Dyck edit distance* where the goal is to maintain the Dyck edit distance quickly over such updates.

Along with Dyck edit distance, we consider a fundamental measure of distances over trees: *tree edit distance*. The tree edit distance involves comparing two ordered, rooted, and labeled trees to determine the minimum number of node insertions, deletions, and relabelings required to transform one tree into another. Originally introduced by Selkow [Selkow, 1977], tree edit distance has found a wide range of applications including in computational biology (e.g., in analyzing RNA molecules, whose secondary structure is depicted as a rooted tree) [Bille, 2005, Gusfield, 1997, Harel and Tarjan, 1984, Shapiro and Zhang, 1990], image analysis [Bunke and Shearer, 1998], and compiler optimization [Demaine et al., 2010], as well as in NoSQL big databases like MongoDB [MongoDB, 2022]. Like in Dyck edit distance, the change in underlying data requires updating the tree edit distance, and the goal of *dynamic tree edit distance* is to maintain these updates quickly.

In this work, we initiate the study of dynamic Dyck and tree edit distance problems. In this setting, the underlying sequence goes through dynamic insertions, deletions, and substitutions. The goal is to maintain Dyck edit distance or tree edit distance fast with such updates. The dynamic variants are particularly important for practical applications, such as efficiently tracking and updating changes in structured documents or correcting compilation errors. However, even the fastest known algorithms in the static setting for these problems are quite slow. The static Dyck edit distance problem admits a simple folklore $\mathcal{O}(n^3)$-time dynamic programming solution which was improved in a series of papers [Bringmann et al., 2019, Williams and Xu, 2020, Chi

et al., 2022a, Chi et al., 2022b] to $\tilde{O}(n^{\frac{3+\omega}{2}})$[1] where ω is the exponent of fast matrix multiplication, $\omega \leq 2.371339$ from [Alman et al., 2024]. For tree edit distance, Mao [Mao, 2021] gave the first subcubic algorithm with a runtime of $\mathcal{O}(n^{2.9546})$ for tree edit distance, which was then slightly improved by Dürr to 2.9148 [Dürr, 2023]. While this improvement came after a long series of works [Tai, 1979, Zhang and Shasha, 1989, Klein, 1998, Demaine et al., 2010], it also uses fast matrix multiplication. For both of these problems, the best combinatorial algorithms that do not use fast matrix multiplication stands at $O(n^3)$, and fine-grained complexity states that this running time is conditionally optimal [Abboud et al., 2018, Bringmann et al., 2020].

To obtain faster static running time, a series of work have studied designing approximation algorithms [Saha, 2014, Saha, 2015, Das et al., 2022b, Koucký and Saks, 2023, Boroujeni et al., 2019, Seddighin and Seddighin, 2022]. If we concentrate our attention to nearly-linear time algorithms, that is algorithms running in $O(n^{1+\epsilon})$ time for any $\epsilon > 0$, the best known approximation factor known for the Dyck edit distance is $O(\log n)$ in $O(n^{1+\epsilon})$ time for any constant $\epsilon > 0$ and $O(2^{\sqrt{\log n \log\log n}})$ in $O(n^{1+o(1)})$ time [2], whereas for the tree edit distance, it is $O(\sqrt{n})$ in $\tilde{O}(n)$ time [Boroujeni et al., 2019]. This raises the following question.

Can we obtain a dynamic algorithm for the Dyck and Tree edit distance problems that have update times of n^ϵ, $\epsilon > 0$, and achieve approximation factors that are close to known near-linear time static algorithms for these problems?

Results. In this paper, we make significant progress towards resolving the above question. We show for the Dyck edit distance, if we allow $n^{o(1)}$ update time, then a $O(2^{\sqrt{\log n \log\log n}})$ approxima-

[1] $\tilde{O}(n)$ notation is commonly used to hide poly $\log n$ factors.

[2] These bounds come from [Saha, 2014, Koucký and Saks, 2023] utilizing the best bounds known for the string edit distance problem in the desired time bounds [Andoni and Onak, 2012, Andoni and Nosatzki, 2020].

tion is possible. On the other hand, if we let the update time to be n^ϵ, for any constant $\epsilon > 0$, then we can achieve an approximation factor of $(\log n)^{O(\frac{1}{\epsilon})}$.

Theorem 4.1.1. (Informal) *There exists a randomized dynamic algorithm, that maintains a poly* $\log n$*-approximation of Dyck edit distance (correctly with high probability against an oblivious adversary) for an (initially empty) string* X *of length at most* n *undergoing edits. The expected amortized update time of the algorithm is* $n^{O(\epsilon)}$ *per update for any constant* $\epsilon > 0$*. Moreover, if we allow an approximation factor of* $O(2^{\sqrt{\log n \log\log n}})$*, then an expected amortized update time of* $n^{o(1)}$ *is possible.*

For tree edit distance, we get the following result, which achieves an $\mathcal{O}(\sqrt{n \log n})$-factor approximation in the worst case (since the maintained value can be capped by n without loss of generality). However, when $\mathsf{ted}(\mathcal{F}, \mathcal{G})$ is small (less than $\sqrt{n/\log n}$), it achieves a significantly better approximation ratio. Moreover, the update time is only polylogarithmic.

Theorem 4.1.2. *There exists a deterministic dynamic algorithm that, for (initially empty) forests* $\mathcal{F}$ *and* $\mathcal{G}$ *of size at most* n *undergoing edits, maintains a value* $\widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G})$ *such that* $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq \widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G}) \leq \mathcal{O}(\mathsf{ted}(\mathcal{F}, \mathcal{G})^2 \log n)$*. The algorithm supports updates in* $\mathcal{O}(\log^{3+o(1)} n)$ *worst-case time.*

Reductions and Decompositions to String Edit Distance. At the core of two of our main results lie new dynamic reductions and decomposition methods to string edit distance. Simplistically speaking, we show an instance of a Dyck edit distance problem can be decomposed into multiple disjoint string edit distance instances in such a way that it is possible to maintain this decomposition fast with dynamic updates, and we do not lose much in the approximation factor. In the static setting, Saha [Saha, 2014] gave the first such decomposition method for Dyck edit distance that loses

a $\log n$ factor in approximation. While her method was randomized, Koucký and Saks [Koucký and Saks, 2023] simplified the reduction and made it deterministic. In this paper, one of our main contributions is to implement a novel decomposition inspired by these reductions under dynamic updates.

For tree edit distance, we also take a similar approach. We show there exists a reduction from tree edit distance to a *single* instance of string edit distance which loses only an $O(\sqrt{n})$ factor in the approximation. This significantly improves the best-known such result by Akutsu, Fukagawa, and Takasu [Akutsu et al., 2010b] that provides an $O(n^{3/4})$-approximate reduction, and only for trees that have constant maximum degree. Not only do we improve the approximation factor in the static setting, we also show how this reduction can be maintained dynamically. Using this novel tree to string reduction, we are able to obtain the following dynamic result.

Theorem 4.1.3. (Informal) *There exists a randomized dynamic algorithm, that maintains an $O(n^{\frac{1}{2}+o(1)})$-approximation of tree edit distance between $\mathcal{F}$ and $\mathcal{G}$ (correctly with high probability against an oblivious adversary) for (initially empty) forests $\mathcal{F}, \mathcal{G}$ of size at most n undergoing edits. The expected amortized update time of the algorithm is $n^{o(1)}$ per edit.*

The above approach is both simple and powerful as it enables us to use the known dynamic approximation algorithms for string edit distance [Kociumaka et al., 2023] as a black box.

Compared to dynamic graph algorithms, the development in dynamic string algorithms is relatively recent [Chen et al., 2013b, Charalampopoulos et al., 2020b, Kociumaka et al., 2023, Bringmann et al., 2024]. We add several important tools to this growing field of research. As part of our Dyck and tree reductions to string edit distance, we propose a new dynamic heavy-light decomposition (see Section 4.2.2.3) method where the heavy paths do not need to be split

or merged, and therefore can be maintained efficiently over updates. Given the significance of heavy-light decomposition in the literature of strings, we believe our approach will be useful for many other applications as well.

Improved Static Approximation Algorithm for Tree Edit Distance Our results on tree edit distance not only give the first nontrivial algorithm in the dynamic setting, it also beats the best approximation bound known for tree edit distance in linear time. Our approximation algorithm from Theorem 4.1.2 determines whether a given tree edit distance instance has a distance of at most k or at least $k^2 \log n$ for some integer k and input size n. Therefore, as long as $\mathsf{ted}(\mathcal{F}, \mathcal{G}) < \sqrt{\frac{n}{\log n}}$, it beats the state-of-the-art linear-time algorithm by achieving the $O(\sqrt{n})$ factor approximation [Boroujeni et al., 2019].

It is to be noted that k vs k^2 gap problem has been studied extensively for string edit distance [Landau and Vishkin, 1988, Goldenberg et al., 2019, Kociumaka and Saha, 2020], and for Dyck edit distance [Saha, 2014, Koucký and Saks, 2023]. Unfortunately, none of the prior decomposition approaches for string problems in this regime extends to tree problems. We provide a new string decomposition approach that not only can be extended to the tree setting with better approximation factor and runtime than the previous state-of-the-art for tree edit distance, but can be easily computed in the dynamic setting as well.

4.1.1 Roadmap

In the remainder of this section, we give a technical overview of our reductions to string edit distance and our novel tree edit distance approximation algorithm. In Section 4.2, we discuss relevant preliminaries for our later technical sections. In Section 4.3, we give our dynamic Dyck

approximation algorithm and proof of its runtime and correctness. In Section 4.4, we give our new tree to string reduction in both the static and dynamic setting. In Section 4.5, we give our state-of-the-art tree edit distance approximation algorithm with proof of correctness and its adaptation to the dynamic setting. Sections 4.6 and 4.7 provide some additional results for exact dynamic Dyck edit distance using existing techniques from prior work.

4.1.2 Technical Overview

4.1.2.1 Dynamic Dyck Approximation Algorithm.

We start with a technical overview of our first algorithm that approximates Dyck edit distance in the dynamic setting by reducing the Dyck edit distance instance to a set of dynamic string edit distance instances. In the static setting, similar reductions were studied [Saha, 2014, Koucký and Saks, 2023], where the core idea was to decompose the Dyck edit distance instance into multiple string edit distance instances. Our new dynamic reduction begins with a similar decomposition concept but employs a new technique that diverges significantly from previous approaches. As [Koucký and Saks, 2023] directly decomposes the input string, even a single dynamic edit in their approach can affect multiple string edit distance instances, making it challenging for efficient dynamic maintenance (we discuss this further in the following paragraphs).

We introduce a new reduction approach that, rather than directly decomposing the input Dyck edit distance instance, first constructs a tree representation of the Dyck edit distance problem. This tree is then decomposed into paths using a carefully designed heavy-light decomposition, which we subsequently convert back into strings. These strings serve as string edit distance instances, allowing us to approximate the Dyck edit distance of the original parenthesis sequence. Our heavy-

light decomposition is specifically tailored to enable polylogarithmic maintenance time, resulting in a sub-polynomial update time. While the steps of our new reduction and the static reduction are completely distinct, we are able to use one of the core analysis techniques from the static reduction to prove that our new reduction achieves a logarithmic approximation factor statically and a sub-polynomial approximation factor dynamically.

We now provide a brief overview of the reduction and analysis framework of [Koucký and Saks, 2023] and discuss the challenges of maintaining the reduction dynamically. Following this, we introduce our new dynamic reduction and provide an overview of the technical details and analysis for our main Dyck edit distance result.

Reduction from Dyck to String Edit Distance in the Static Setting [Koucký and Saks, 2023]. Given an input string X, the Dyck to string reduction of [Koucký and Saks, 2023] begins by constructing a collection $\mathcal{C}(X)$ of strings corresponding to subsequences of X such that each string contains a sequence of opening parentheses followed by a sequence of closing parentheses of equal length. Each string in $\mathcal{C}(X)$ corresponds to an instance of the string edit distance problem, and the resulting string edit distance instances are used to compute an approximation for the Dyck edit distance of X.

The construction of $\mathcal{C}(X)$ begins by partitioning X according to its *LR-decomposition* where each part of the LR-decomposition is a maximal-length disjoint substring composed of a sequence of opening parentheses followed by a sequence of closing parentheses, called *LR-segments*. For each LR-segment, either a prefix of opening parentheses or a suffix of closing parentheses is trimmed so that the remaining string has an equal number of opening and closing parentheses. The resulting substring of X is added to $\mathcal{C}(X)$ and removed from X for the remainder of the

Iteration	X_1	$\mathcal{C}(X_1)$
1	(([()([]]](()(()))]]	(], ([]], (), (())
2	(([]()]]	(], ([]], (), (()), [], ()
3	(()]	(], ([]], (), (()), [], (), (()]

Table 4.1: The above table contains an example of the construction routine for $\mathcal{C}(X_1)$ given string X_1. The red and blue parenthesis coloring separate the LR-segments of X_1 in each iteration.

Iteration	X_2	$\mathcal{C}(X_2)$
1	(([(([]]](()(()))]]	(([]]], (), (())
2	(([()]]	(([]]], (), (()), ([()]]
3	(	(([]]], (), (()), ([()]]

Table 4.2: The above table contains an example of the construction routine for $\mathcal{C}(X_2)$ given string X_2. The red and blue parenthesis coloring separate the LR-segments of X_2 in each iteration. X_1 and X_2 differ by a single parenthesis, $X_1[4]$ is not present in X_2. However, $\mathcal{C}(X_1)$ and $\mathcal{C}(X_2)$ differ in over half of their strings.

construction algorithm. After each LR-segment is processed, the input string X is comprised of exactly the trimmed prefixes and suffixes. This process is then repeated iteratively until only a sequence of opening or a sequence of closing parentheses remains in X. See Table 4.1 for an example of the construction of $\mathcal{C}(X_1)$ for a string X_1.

The collection $\mathcal{C}(X)$ constructed as above corresponds to a set of string edit distance instances that are used to approximate the Dyck edit distance of X. For each LR-segment in $\mathcal{C}(X)$, recall that the segment contains a sequence of opening parentheses followed by a sequence of closing parentheses. Each segment is split in half, the closing parenthesis sequence is reversed, and each closing parenthesis is replaced with an opening parenthesis of the same type.[3] The sum of the edit distances between each pair of halves for each string of the LR-decomposition yields an $\mathcal{O}(\log n)$-approximation of the Dyck edit distance for the original input string X.

The core concept behind the LR-decomposition comes from a natural observation. Note

[3] e.g., segment "(([[(()))])" would become pair "(([[((", "([(({" .

that the Dyck edit distance can be defined as the number of deletions or substitutions needed to balance a parenthesis string. In a balanced parenthesis string, every parenthesis has a "match": the parenthesis with opposite parity (open or closed), of the matching type and matching height. Here the height of a parenthesis is defined as the difference between the number of opening and closing parentheses preceding it, see Figure 4.1a. Consider an opening parenthesis followed immediately by a closing parenthesis contained in a parenthesis string. If these parentheses do not match after all edits in an optimal sequence, one of the two parentheses must have been deleted. Instead, the sequence could be modified to match these two parentheses, which may require a substitution to align their types and an additional deletion for a later parenthesis that was intended to match with one of them. This adjustment would require 2 edits instead of the original 1 to handle this pair of parentheses.

This simple idea can be extended to LR-segments of X that contain an equal number of opening and closing parentheses. If d of the opening parentheses in an LR-segment are not matched to the following sequence of d closing parentheses after performing an optimal sequence of edits, then at least d deletions must take place on this LR-segment. These d deletions can be replaced with d substitutions with an additional d deletions of the later parentheses to obtain a 2-approximation for the number of edits required to balance the parentheses in this LR-segment. By computing this approximation for each segment of the LR-decomposition of X, a 2-approximation of the number of edits needed to balance all characters contained in these segments is found excluding a prefix of opening parentheses or a suffix of closing parentheses per segment. The same steps are repeated iteratively on the string comprised of the remaining prefixes and suffixes until no more LR-segments are contained in X. Since over half of the string is removed in each iteration, there are $\mathcal{O}(\log n)$ iterations. Determining the substitutions and deletions needed for each LR-segment

can be done by computing the string edit distance between the two halves of the LR-segment. Thus, the LR-decomposition provides a way to achieve a $2 \cdot \mathcal{O}(\log n)$-approximation for the Dyck edit distance.

Reduction from Dyck to String Edit Distance in the Dynamic Setting. The key challenge in implementing this reduction dynamically is maintaining $\mathcal{C}(X)$ as the input string X undergoes updates. Reconstructing the collection of strings after each update following the above construction method is too time-consuming. Even a single pass to parse the entire string to compute the LR-decomposition, and then to trim prefixes and suffixes would require linear time in the number of LR-segments, which could be $\mathcal{O}(n)$ in the worst case.

Furthermore, a single parenthesis insertion or deletion in X can change a super-constant number of strings in $\mathcal{C}(X)$, see Table 4.2 for an example of how deleting a single parenthesis greatly changes $\mathcal{C}(X)$. A key challenge, therefore, lies in structuring $\mathcal{C}(X)$ to efficiently identify and update only the affected strings. To address these challenges, rather than constructing a collection of subsequences of X directly, we first represent X with a tree T_X. We then re-frame the reduction by representing X through a collection of paths within this tree T_X.

To build a tree T_X from X, we begin by pairing together the opening parentheses and closing parentheses of equal height, which we refer to as *twins*. For each opening parenthesis, its twin is the closest closing parenthesis to its right that has the same height. Note, that his procedure defines a one-to-one mapping between opening and closing parentheses in X. Each pair of twins corresponds to a node in the tree. For an opening parenthesis at index i in X, referred to as $X[i]$, its corresponding node in the tree is the child of the node that corresponds to the nearest opening parenthesis $X[j]$ with $j < i$ and of height exactly 1 less than that of $X[i]$. Any opening

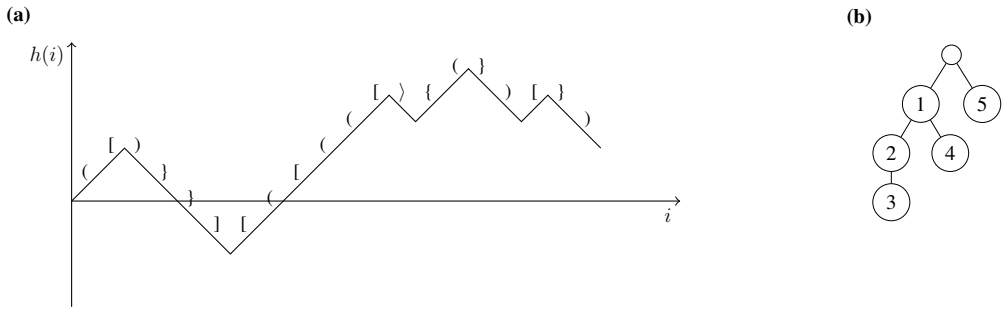


Figure 4.1: (a) Example of a graph of heights of parentheses in a given input string from [Fried et al., 2024]. (b) The tree representation T_X of parenthesis string $X = (_1\,(_2\,(_3\,)_3\,)_2\,(_4\,)_4\,)_1\,(_5\,)_5$ where matching indices in the subscripts of the parentheses indicate pairs of twins. The path containing nodes labeled 1, 2, 3 corresponds to string $(_1\,(_2\,(_3\,)_3\,)_2\,)_1$.

parentheses with no parent represent the root of their own tree. In this tree representation of X, each path corresponds to an LR-segment of X by arranging the sequence of opening parentheses, followed by the sequence of closing parentheses, that corresponds to the nodes appearing in the path. See Figure 4.1b for an example of T_X.

To reduce Dyck to string edit distance with our tree representation of X, we first perform a heavy-light decomposition on T_X. We recall the definition of heavy-light decomposition here.

Definition 4.1.4 (Heavy-Light Decomposition)**.** Given a forest $\mathcal{F}$, a heavy-light decomposition of $\mathcal{F}$ assigns a label *heavy* or *light* to each node of $\mathcal{F}$. For a given node $u \in \mathcal{F}$, a child node v of u is *heavy* if $\lfloor \lg |\mathcal{F}(v)| \rfloor = \lfloor \lg |\mathcal{F}(u)| \rfloor$, and *light* otherwise. A node with no parent is light by default. Here, for a node $u \in \mathcal{F}$, $\mathcal{F}(u)$ denotes the subtree of $\mathcal{F}$ rooted at vertex u , and $\lfloor \lg |\mathcal{F}(u)| \rfloor$ denotes the *heavy depth* of u in $\mathcal{F}$. A set of heavy nodes in $\mathcal{F}$ is called a *heavy path* if it forms a path in $\mathcal{F}$.

As in a typical heavy-light decomposition, every node in a forest has at most one heavy child. We also make one other useful observation that is folklore for heavy-light decompositions.

Observation 4.1.5. *Given a forest $\mathcal{F}$ with $|\mathcal{F}| = n$, any root-to-leaf path includes at most $\mathcal{O}(\log n)$ light nodes.*

Once the heavy-light decomposition is obtained, we partition the tree into paths based on the heavy-light status of each node, ensuring that each path consists solely of heavy nodes except for the starting node, which is a light node. Since each path of the tree corresponds to a sequence of opening and closing parentheses of matching height, the core principle of the reduction from [Koucký and Saks, 2023] still holds; for each string corresponding to these heavy node paths, we can find a 2-approximation for the number of edits needed to balance the parentheses in this string using the corresponding string edit distance estimation. We compute the string edit distance between the opening and closing parenthesis sequences for each path, and as a heavy-light decomposition ensures each root-to-leaf path has at most $\mathcal{O}(\log n)$ light nodes, the 2-approximation factor only compounds additively $\mathcal{O}(\log n)$ times. Thus, the re-framed Dyck to string reduction using the tree still provides an $\mathcal{O}(\log n)$-approximation.

New Dynamic Heavy-light Decomposition. With our new tree representation, we have shifted the challenge of the dynamic approach from maintaining $\mathcal{C}(X)$ to maintaining the heavy-light decomposition of the tree representation of X in the dynamic setting. In a traditional heavy-light decomposition, which typically designates the child with the largest subtree as heavy while categorizing the remaining children as light, a single dynamic edit to the tree can result in many nodes changing their type from heavy to light and vice versa. By defining our heavy-light decomposition based on the logarithm of the size of the subtrees, a dynamic update to X creates a structured impact on the nodes of the tree, which our algorithm utilizes. This approach ensures that any root-to-leaf path contains only $\mathcal{O}(\log n)$ light nodes, which is consistent with the conventional heavy-light decomposition. We observe that when a parenthesis is inserted or deleted at index i, all the parentheses appearing at an index greater than i have their height changed by 1. Further,

any pair of twin parentheses, where both indices are greater than i, have their height shifted and thus, these pairs are unaffected. Thus, the only pairs of twins, with opening parenthesis appearing at an index $j_1 < i$ and closing parenthesis appearing at an index $j_2 > i$ are affected and need to be handled. These are exactly the ancestor nodes of the node corresponding to $X[i]$. Since any root-to-leaf path can contain at most $\mathcal{O}(\log n)$ heavy paths, we only need to update a logarithmic number of heavy paths in the path decomposition of T_X for each edit. Moreover, since the heights of the parentheses only change by 1, we are primarily concerned with a shift of just 1 node between adjacent affected heavy paths, specifically the first or last nodes of these heavy paths. These are specifically the nodes that may change their status from heavy to light or vice versa. For each of these first and last nodes of affected heavy paths, we recompute their heavy-light status by comparing the size of their subtree to that of their parent's subtree. Since we only need to check $\mathcal{O}(1)$ starting and ending nodes per heavy path across $\mathcal{O}(\log n)$ heavy paths, we may maintain the entire heavy-light decomposition in polylogarithmic time.

When a dynamic update occurs on X, we can identify the set of heavy paths that contain the ancestor nodes of the node corresponding to the inserted or deleted parenthesis. Subsequently, we can adjust the first and last nodes of these heavy paths as needed, all within logarithmic time. For all the affected paths, we then perform $\mathcal{O}(1)$ edits on the corresponding string edit distance instances according to these shifts. By utilizing a dynamic string edit distance algorithm that can handle dynamic insertions and deletions in sub-polynomial time, we recompute the distances for these updated instances and then update our Dyck edit distance approximation with these new distances.

We highlight that our proposed dynamic heavy-light decomposition introduces significantly new ideas, allowing heavy paths to remain intact without needing to be split or merged. Traditional

dynamic heavy-light decompositions, such as those in link-cut trees, lack this property and could not be applied in our reduction (since the dynamic string edit distance approximation does not support splits and concatenations). Given the heavy-light decomposition's broad utility, we believe our approach has potential for further applications and could be of independent interest.

4.1.2.2 Dynamic Approximate Tree to String Edit Distance

We now concentrate our attention on an overview of the $\tilde{O}(\sqrt{n})$-approximation algorithm for dynamic tree edit distance. Our method begins with a new $\tilde{O}(\sqrt{n})$-approximate reduction from tree to string edit distance in the dynamic setting. This reduction enables us to use the $\mathcal{O}(b \log_b n)$-approximation dynamic string edit distance algorithm where $2 \leq b \leq n$ is an integer parameter we may set, which has an update time of $\mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$ [Kociumaka et al., 2023] to achieve our final result. We start by discussing the new reduction in the static setting and then outline the additional techniques required to implement it dynamically.

Reduction from Tree to String Edit Distance in the Static Setting. In the static setting, the best-known reduction from string to tree edit distance, developed by Akutsu, Fukagawa, and Takasu [Akutsu et al., 2010b], provides an $\tilde{O}(n^{3/4})$ approximation. Despite a decade passing, no improvement has been proposed on this bound and moreover, the reduction only works for constant degree trees. Until now, whether or not there exists a reduction of tree to string edit distance for unbounded degree trees that has an approximation loss of $o(n)$ remained an open question. We therefore not only significantly improve the prior bound and provide a reduction offering an approximation of $\tilde{O}(\sqrt{n})$, we show that it works for all trees and can be maintained even when the trees go through changes over time. In [Akutsu et al., 2010b], the authors convert two rooted

labeled trees $\mathcal{T}$ and $\mathcal{T}'$ into strings by labeling the edges based on specific structural properties: a node v is defined as "special" if the subtree rooted at v is small while the subtree rooted at its parent is large. Labels are assigned to edges based on the presence of special children. Each non-special edge (u, v) is labeled by v, while each special edge receives a unique label that incorporates information about the subtrees of v's special children. After labeling, the trees $\mathcal{T}$ and $\mathcal{T}'$ are converted to strings S and S' through an Euler tour. The next step is to compute an optimal string alignment $\mathcal{A}$ between these strings. However, converting $\mathcal{A}$ to a valid tree alignment results in an approximation loss of $\tilde{O}(n^{3/4})$.

Our algorithm is inspired by a similar framework but introduces a new approach from the outset, beginning with the label generation step. Instead of assigning labels to edges, our method assigns labels to nodes using a heavy-light decomposition, rather than the heavy subtree approach in [Akutsu et al., 2010b]. Incorporating heavy-light decomposition in label generation gives us more flexibility in exploiting the structural properties of the Euler tour representation, allowing the optimal string alignment to be transformed directly into a valid tree alignment with an improved approximation guarantee. Our string-to-tree alignment step is also entirely new and simpler than that in [Akutsu et al., 2010b]. Further, the cost analysis of this step ensures a $\tilde{O}(\sqrt{n})$ approximation and introduces several foundational techniques and concepts, serving as a core contribution of our work.

Parentheses representation. The first step in our reduction translates a tree edit distance instance into a string edit distance instance. This begins by performing a heavy-light decomposition (Definition 4.1.4) on the input trees $\mathcal{T}$ and $\mathcal{T}'$. Specifically, we define a node v as light if it does not share the same heavy depth as its parent u. Next, each node v is assigned a label that encodes

information about the subtree rooted at v, excluding its heavy child and the subtree rooted at that heavy child, which is represented by a dummy character. Finally, we take an Euler tour of each tree using these new labels, resulting in strings that serve as instances for string edit distance. Formally, let $\mathcal{F}$ be a tree with root node r. Then, the string representation of $\mathcal{T}$ is defined recursively, as $\mathsf{P}(\mathcal{T}) = (_{\lambda(r)}\mathsf{P}(\mathcal{T} \setminus r))_{\lambda(r)}$[4]. Here $\lambda(r)$ represents the label of node r, and $\mathsf{P}(\mathcal{T} \setminus r)$ is the forest resulting from deleting node r from $\mathcal{T}$. For a given node u in $\mathcal{F}$, let $o(u)$ and $c(u)$ represent the index of the opening and closing parentheses, respectively, corresponding to u in $\mathsf{P}(\mathcal{F})$.

String to tree alignment. The second step begins by computing an optimal alignment $\mathcal{A}$ between $\mathsf{P}(\mathcal{T})$ and $\mathsf{P}(\mathcal{T}')$. A key observation here is that, under $\mathcal{A}$, a match between a node $u \in \mathcal{T}$ and a node $u' \in \mathcal{T}'$ satisfies the tree alignment constraints if $\mathcal{A}$ aligns both $\mathsf{P}(\mathcal{T})[o(u)]$ with $\mathsf{P}(\mathcal{T}')[o(u')]$ and $\mathsf{P}(\mathcal{T})[c(u)]$ with $\mathsf{P}(\mathcal{T}')[c(u')]$. Nodes that do not meet this condition are referred to as misaligned and thus violate the tree alignment constraints.

For a node $u \in \mathcal{T}$, misalignments can be categorized as follows: (i) $\mathcal{A}$ aligns only one of the parentheses corresponding to u while deleting the other, making u *partially deleted*; (ii) $\mathcal{A}$ aligns $o(u)$ with $o(v_1)$ and $c(u)$ with $c(v_2)$, where v_1 and v_2 lie on the same root-to-leaf path in $\mathcal{T}'$, classifying u as a *single-branch* misaligned node; (iii) similar to (ii), but v_1 and v_2 appear on different root-to-leaf paths in $\mathcal{T}'$, making u a *multi-branch* misaligned node.

Next, we discuss how our algorithm TreeAlign transforms $\mathcal{A}$ to eliminate all misaligned nodes, resulting in a valid tree alignment. If a misaligned node u is either light, partially deleted, or a multi-branch misaligned node, we update $\mathcal{A}$ by removing the matches related to u. For a heavy single-branch misaligned node, more complex processing is required. One common structure in

[4]For a forest $\mathcal{F}$, the string representation is defined as the concatenation of the string representations of the trees it contains.

single-branch mis-aligned nodes involves a sequence of single-branch nodes alternating between $\mathcal{T}$ and $\mathcal{T}'$ where each node aligns only one of its parentheses with the previous node in the sequence and its other parenthesis with the next node. We formally define this as a *chain*. To handle a heavy single-branch misaligned node u, we modify $\mathcal{A}$ by removing the matches associated with u if u is not part of a chain, if it appears near either end of a chain, or if there is a large subtree (size $\geq \sqrt{n}$) rooted at u. Otherwise, we adjust $\mathcal{A}$ to align the subtree rooted at u (including u) with the subtree rooted at v (including v), where $v \in \mathcal{T}'$ is the node following u in the chain sequence, while maintaining $\mathcal{A}$ as a valid alignment. It becomes fairly intuitive that after these modifications, no nodes remain misaligned under $\mathcal{A}$, allowing $\mathcal{A}$ to be interpreted as a valid tree alignment between $\mathcal{T}$ and $\mathcal{T}'$.

Cost analysis. We now prove that performing TreeAlign increases the number of edits in alignment $\mathcal{A}$ by at most a factor of $\tilde{\mathcal{O}}(\sqrt{n})$. Before, we delve into details, we present example instances of tree edit distance that shed light on the functioning of our reduction and give an intuitive idea of why we achieve a $\sqrt{n}$ approximation factor.

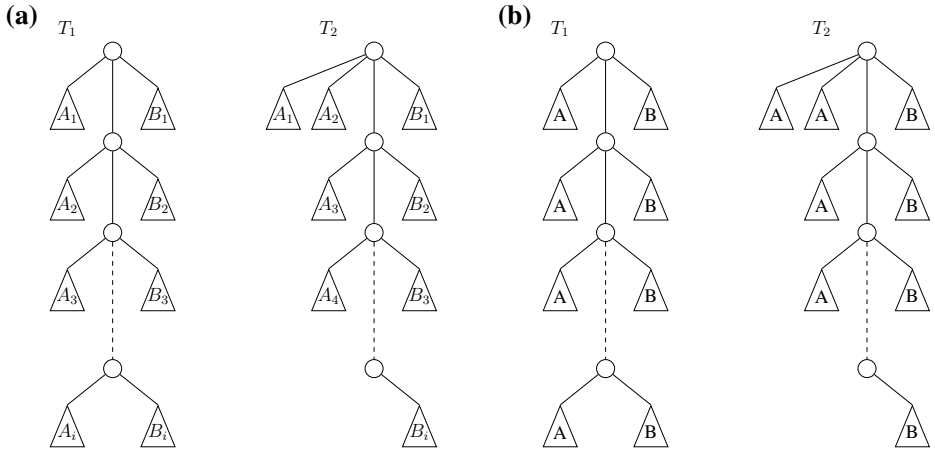


Figure 4.2: Two instances the tree edit distance problem. (a) Trees T_1 and T_2 with $A_i \neq A_j$, $B_i \neq B_j$ for $i \neq j$. (b) Trees T_1 and T_2 with repeating subtrees A and B to the left and right, respectively, of the center path.

Examples illustrating the Reduction. In Figure 4.2a, both trees T_1 and T_2 have subtrees labeled $A_1, A_2, \ldots, A_i, B_1, B_2, \ldots, B_i$, while Figure 4.2b presents subtrees simply labeled as A and B (without indexing). We refer to the central path through the tree as the *spine*. Furthermore, we assume that for all $i \neq j$, $A_i \neq A_j$ and $B_i \neq B_j$.

First, let's study the pair of trees depicted in Figure 4.2a. Ignoring any special labeling, the conventional Euler tour string representations of T_1 and T_2 are given as follows: T_1 is represented as $(A_1(A_2(A_3 \ldots B_3)B_2)B_1)$, and T_2 as $(A_1A_2(A_3(A_4(\ldots)B_3)B_2)B_1)$, where each pair of parentheses "(", ")" corresponds to a node along the spines of the trees. It is evident that only a few deletions are necessary to make these two strings similar. However, the only way to achieve similarity between T_1 and T_2 is either by deleting all the nodes in the spines or by removing all the subtrees attached to the spines. If the spine is quite long, for instance, if $i = \mathcal{O}(n)$, then the tree edit distance is $\mathcal{O}(n)$, whereas the string edit distance is $\mathcal{O}(1)$. Figure 4.2a serves as a representative example of the challenges we tackle using our label embedding technique. By embedding subtree information in the labels of the nodes, the resulting Euler tour strings for T_1 and T_2 are respectively $({}_{A_1,B_1}A_1({}_{A_2,B_2}A_2 \ldots$ and $({}_{A_1,A_2,B_1}A_1A_2({}_{A_3,B_2}A_3 \ldots$. Thus, for these strings, any optimal sequence of edits must remove all parentheses corresponding to the nodes on the spine, as each one carries a distinct label after the embedding process. Consequently, the gap between the string and tree edit distances reduces to a mere $\mathcal{O}(1)$.

However, we can still encounter a gap of $\mathcal{O}(\sqrt{n})$ in certain cases. For instance, if we modify the pair of trees to have more periodicity, as shown in Figure 4.2b, the resulting strings are $({}_{A,B}A({}_{A,B}A \ldots$ and $({}_{A,A,B}AA({}_{A,B}A({}_{A,B}\ldots$. In this scenario, only $\mathcal{O}(1)$ deletions are necessary to make these strings equal. Conversely, to make T_1 and T_2 identical trees, we have two options: either delete all the nodes in the spine, as previously mentioned or remove a few copies of A and B.

It is important to note that in Figure 4.2a, we were required to delete all A_i and B_i, whereas, in the periodic trees, we may only need to delete $\mathcal{O}(1)$ copies of them. If $|A|, |B| < \sqrt{n}$, then deleting $\mathcal{O}(1)$ copies of A and B will require only $\mathcal{O}(\sqrt{n})$ deletions. On the other hand, if $|A|, |B| \geq \sqrt{n}$, we know that the length of the spine can be at most $n/|A| = \mathcal{O}(\sqrt{n})$. Thus, deleting the spine nodes will also result in $\mathcal{O}(\sqrt{n})$ deletions. In both scenarios, the string edit distance remains $\mathcal{O}(1)$, while the tree edit distance is $\mathcal{O}(\sqrt{n})$. In Section 4.4.1, we formally establish that this is the maximum gap attainable through our reduction. We show that for any optimal string alignment between the Euler tour strings generated by our reduction, we can construct a corresponding tree alignment for the original input trees with at most $\tilde{\mathcal{O}}(\sqrt{n})$ additional deletions.

In the remainder of the cost analysis overview, we give more details that helps to establish the formal bound in Section 4.4.1.

If a node u is partially deleted, since the string alignment $\mathcal{A}$ already incurs a cost of 1 for deleting one of its parentheses, we can modify $\mathcal{A}$ to delete the other parenthesis as well, thereby only doubling the cost.

Next, we address the processing of misaligned nodes that appear in a chain. We make two key observations. First, by definition, any chain consists of either all light nodes or all heavy nodes. Second, for every maximal chain, since $\mathcal{A}$ aligns the opening and closing parentheses of each node with at least a one-position shift, it incurs at least one deletion. Thus, the cost of $\mathcal{A}$ is at least one for every maximal chain. We begin by considering chains of light nodes. Since the length of such a chain is at most $\mathcal{O}(\log n)$, deleting all light nodes may increase the cost by at most a factor of $\mathcal{O}(\log n)$.

We now analyze the cost of handling a chain that consists solely of heavy nodes. First, consider a heavy node u for which the subtree rooted at u, excluding the subtree of its heavy child

if one exists, is large (i.e., of size $\geq \sqrt{n}$). The number of such heavy nodes is bounded by $\sqrt{n}$. Therefore, removing all these nodes (i.e., their corresponding matches from $\mathcal{A}$) incurs a cost of at most $\mathcal{O}(\sqrt{n})$. Since each chain contributes at least one deletion under $\mathcal{A}$, the overall increase in cost is at most a factor of $\mathcal{O}(\sqrt{n})$. Therefore, we now need to address the heavy nodes that have small subtrees, excluding the subtree of its heavy child (if it exists), rooted at them. If such a node u does not appear close to either end of the chain, we modify $\mathcal{A}$ to align the subtree rooted at u (including u) with the subtree rooted at v (including v), where $v \in \mathcal{T}'$ is the node that follows u in the chain sequence. This adjustment maintains the same cost. Alternatively, for a constant number of nodes that root small subtrees and are located at either end of a chain, we delete the subtree that is rooted at those nodes. Since this is performed only a constant number of times for each chain, and the size of each subtree is at most $\sqrt{n}$, the total cost incurred remains $\mathcal{O}(\sqrt{n})$. Furthermore, as each chain contributes at least one deletion under $\mathcal{A}$, the overall increase in cost is again bounded by a factor of $\mathcal{O}(\sqrt{n})$.

We note that the threshold used to distinguish between large and small subtrees is crucial in determining the approximation guarantee. For instance, if we set a higher threshold, the number of nodes with large subtrees will decrease, but we will have to delete more nodes associated with small subtrees located close to either end of the chain, which will raise the overall cost. On the other hand, if we adopt a lower threshold, the number of nodes with large subtrees will increase, again contributing to a higher total cost. This analysis supports our decision to use a threshold of $\sqrt{n}$ and explains the rationale behind the approximation factor.

Finally, we analyze the cost of handling misaligned nodes that do not belong to a chain. Note these are essentially the multi-branch misaligned nodes. We represent these nodes using a concept called an *extended chain*. An extended chain is a sequence of nodes (possibly multi-

branch) alternating between $\mathcal{T}$ and $\mathcal{T}'$, where, under $\mathcal{A}$, each node aligns only one of its parentheses with the previous node in the sequence and its other parenthesis with the next node. We observe that an extended chain contains at most one multi-branch misaligned node and one partially deleted node. Since $\mathcal{A}$ incurs at least a cost of one for each partially deleted node, deleting the multi-branch misaligned nodes increases the overall cost by at most a constant factor. We remark that the formal analysis includes many additional details and must address more complex scenarios, such as cases where multiple chains or extended chains overlap. Further, instead of focusing on ordered trees, we consider a more general case where the input may consist of a pair of ordered forests, with each forest being an ordered collection of trees.

Reduction from Tree to String Edit Distance in the Dynamic Setting. To implement our reduction dynamically, the primary challenge is updating the label of each node following an edit operation to keep the Euler tour string representation up-to-date. Since our label encodings rely on the heavy-light decomposition of the input trees, the first step is to maintain this decomposition dynamically. We achieve this using a strategy similar to that used for dynamic heavy-light decomposition in the Dyck edit distance algorithm (Theorem 4.3.7), which enables a logarithmic update time.

Next, we observe that a dynamic edit (such as insertion, deletion, or relabeling of a node) may require adding, removing, or updating a pair of parentheses in the Euler tour for the affected node. The main challenge, however, is that any ancestor nodes containing the modified parentheses in their labels will also need corresponding label updates. In our string reduction, it is specifically the parents of light nodes who encode information about the subtrees rooted at these light nodes within their labels. Therefore, any update within these light subtrees must also be reflected in the label of their parent. Consequently, for any edit, the only affected nodes are those that serve as

parents to light nodes along the path from the root to the edited node. Since any root-to-leaf path in the heavy-light decomposition contains at most $\mathcal{O}(\log n)$ light nodes, only $\mathcal{O}(\log n)$ nodes need their labels updated for each edit. After identifying changes in the heavy-light decomposition, we apply split and concatenate operations from the dynamic strings data structure [Gawrychowski et al., 2018] to update the labels of each affected node in $\mathcal{O}(\log n)$ time. Therefore, the total update time for maintaining the Euler tour representation is bounded by polylog(n).

In addition to the updates mentioned above, there are further updates required due to nodes outside the root-to-edited-node path that may switch from light to heavy or vice versa. We note that only the nodes whose status has shifted between heavy and light, along with their parent nodes, need label modifications. Specifically, when a node changes from light to heavy, the label of its parent must substitute the child's subtree with a dummy character. Conversely, if a node transitions from heavy to light, we must replace the dummy character in the parent's label with the encoding of the light child's subtree. In a conventional heavy-light decomposition, an arbitrary number of nodes can change their status between heavy and light with each update. However, as we discussed while analyzing the heavy-light decomposition for Dyck edit distance, our specialized decomposition has the property that only a logarithmic number of nodes can have their statuses switched. Since each switch requires updating the labels of only two nodes, the total number of nodes needing label updates is bounded by $\mathcal{O}(\log n)$. Consequently, using the dynamic strings data structure allows us to perform all label updates, and thus update the Euler tour representation, in a total update time of polylog(n).

As in the final step of our approximate dynamic Dyck edit distance algorithm, we similarly pass the updated parenthesis representations of each tree to the dynamic string edit distance al-

gorithm. This algorithm then manages dynamic insertions and deletions of characters within the parenthesis string, producing an updated alignment in sub-polynomial update time.

4.1.2.3 Improved Static and Dynamic Tree Edit Distance

We end with a technical overview for our $\mathcal{O}(k^2 \log n)$-approximate static and dynamic tree edit distance algorithm where k is an upper bound on the distance and n is the input size. Our algorithm actually uses a decomposition of the parenthesis string representation of the input forests to distinguish between the two cases of whether the distance of the input instance is at most k or at least $\tilde{\mathcal{O}}(k^2)$. Our algorithm works straightforwardly in linear time up to logarithmic factors in the static setting. We show that using a clever combination of the dynamic string data structure of [Gawrychowski et al., 2018] and the dynamic tree query data structure of [Navarro and Sadakane, 2014], we can implement our novel decomposition algorithm in the dynamic setting with amortized $\tilde{\mathcal{O}}(1)$-update time.

Tree Decomposition in the Static Setting. We begin with a brief description of our novel tree edit distance approximation algorithm for distance bound k. Given forests $\mathcal{F}$ and $\mathcal{G}$, the algorithm maintains a decomposition S of $\mathsf{P}(F)$, where recall $\mathsf{P}(\mathcal{F})$ is the parenthesis representation of $\mathcal{F}$ (the string formed by concatenating the string representations of each tree in $\mathcal{F}$). Initially, S is just the entire string $\mathsf{P}(\mathcal{F})$.

We use the following routine for a limited number of iterations to find our approximation. First, we remove the first part σ in S. If σ has size at most $\mathcal{O}(k)$, we do nothing and move on to the next part of S. If instead the size of σ is larger than our threshold, we check if there is a part in $\mathcal{G}$ that exactly matches σ and has the same starting index up to a left or right shift of length k. If

such a match is found, we remove σ from S. If a match is not found, we partition σ into a constant number of pieces with size a constant fraction of the previous size of σ and add these pieces to S. We repeat these steps for $\mathcal{O}(k \log n)$ iterations, and if S is empty after the last iteration, we output $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq k$, otherwise $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \geq \mathcal{O}(k^2 \log n)$.

To analyze the correctness of our algorithm, we observe that we can view the parts contained in S at any timestep of the algorithm as a tree $\mathcal{T}$. Each node of the tree corresponds to a single part contained in S. The root of $\mathcal{T}$ is the node corresponding to the entire forest $\mathsf{P}(\mathcal{F})$. If a part σ is broken into a constant number of smaller parts $\sigma_1, \sigma_2, \ldots, \sigma_M$ in the last case of our algorithm, then the node $v_\sigma \in \mathcal{T}$ corresponding to σ will be the parent of the nodes $v_{\sigma_1}, v_{\sigma_2}, \ldots, v_{\sigma_M} \in \mathcal{T}$ corresponding to $\sigma_1, \sigma_2, \ldots, \sigma_M$. Each level of the tree contains each parenthesis of $\mathsf{P}(\mathcal{F})$ at most once since at any timestep, S cannot contain the same parenthesis in two different parts. Since $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq k$, there are at most k edits made to $\mathsf{P}(\mathcal{F})$ and furthermore, at most k nodes on each level of the tree whose corresponding part will not find matches in $\mathcal{G}$. Therefore, the total number of nodes in $\mathcal{T}$ cannot ever exceed $\mathcal{O}(k \cdot h)$ where h is the height of the tree. Since we either match or partition each $\sigma \in S$ into pieces of a constant fraction of the previous size, h must be $\mathcal{O}(\log n)$. Thus, if $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq k$, we are guaranteed to find S empty after $\mathcal{O}(k \cdot h) = \mathcal{O}(k \log n)$ iterations, which matches our algorithm. Moreover, since our algorithm matches all parts of S exactly except those with size at most $\mathcal{O}(k)$, the set of matched elements actually defines an alignment between $\mathcal{F}$ and $\mathcal{G}$ with $\mathcal{O}(k^2 \log n)$ deletions needed for non-matched elements.

Tree Decomposition in the Dynamic Setting. Implementing the above algorithm in the dynamic setting introduces two main challenges. First, how we partition each unmatched part of set S into a constant number of pieces with reduced size is straightforward in the static setting, i.e., using a

tree centroid decomposition to guide our partitioning. In the dynamic setting, however, we are not aware of any such quickly computable decomposition for trees. Second, we need a fast way to find matches for each part in S. The latter challenge we discuss in more detail after explaining how we perform our partitioning, as the partitioning method determines the types of parts of $\mathsf{P}(\mathcal{F})$ that will be contained in S.

Our partitioning method will take as input a *subforest* $\mathsf{P}(F)[i\,..\,j)$, a balanced parenthesis substring of $\mathsf{P}(\mathcal{F})$, or a *context* $(\mathsf{P}(\mathcal{F})[i_1\,..\,j_1), \mathsf{P}(\mathcal{F})[i_2\,..\,j_2))$. A context is a subforest of $\mathcal{F}$ excluding the subtree of some node in this subforest. More formally, both $\mathsf{P}(\mathcal{F})[i_1\,..\,j_2)$ and $\mathsf{P}(\mathcal{F})[j_2\,..\,i_2)$ must be a subforest of $\mathcal{F}$. See Figure 4.3 for an example.

To partition a subforest piece σ, we find a node u in σ such that its subtree has size at most $|\sigma|/2$ and its parent u' (if it exists) has subtree size at least $|\sigma|/2$. Furthermore, we choose u such that it is centered in σ, i.e., $o(u) \leq \lceil |\sigma|/2 \rceil$, $c(u) \geq \lceil |\sigma|/2 \rceil$. We then divide the subforest into six subforest and context parts around this node. See Figure 4.4 for a picture of the six parts. To see that each of the six parts has size at most $|\sigma|/2$, we first observe that we chose u to be a central node in σ, so any of the parts fully contained to the left or right of u must have size at most $|\sigma|/2$. For the remaining two parts, we observe that u and its parent u' are chosen such that the subtree of u has size at most $|\sigma|/2$ and the set of remaining nodes has size at most $|\sigma| - |\sigma|/2 = |\sigma|/2$, respectively.

Partitioning a context piece σ is a bit more difficult and results in two cases. In the first case, if there is a node u in σ satisfying the same constraints as in the case for subforests, we may perform the same decomposition as we did before. However, it is possible all valid nodes u satisfying these constraints are exactly in the gap of the context. Therefore, in the second case, no such u exists. Let v be the node in σ whose subtree is excluded from σ as per the definition of

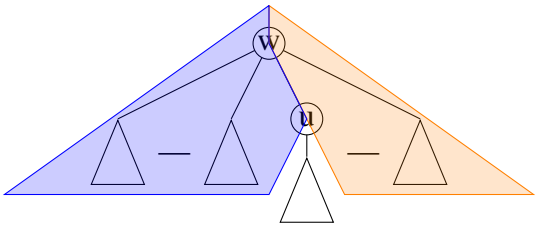


Figure 4.3: An example of a context $(\mathsf{P}(F)[o(w)\,..\,o(u)], \mathsf{P}(F)[c(u)\,..\,c(w)])$ for the subtree of node w in a forest $\mathcal{F}$ formed by excluding the subtree of node u. The left part of the context, depicted in blue, is $\mathsf{P}(F)[o(w)\,..\,o(u)]$ and the right part of the context, depicted in orange, is $\mathsf{P}(F)[c(u)\,..\,c(w)]$.

contexts. The subtree of v in σ must already be size $|\sigma|/2$ or else we would fall into the first case. We set the first part of our partition to be all nodes of σ excluding v and its subtree. Then, we note that at most one of the left half of the children of v or the right half of the children of v must have a combined size of greater than $|\sigma|/2$. For the smaller half, we set it to be its own part. For the remaining half of the subtree of v, this half is itself a subforest, and so we partition the remaining subforest into 6 more pieces as we did for subforest pieces before. This results in at most 8 total pieces of size at most $|\sigma|/2$.

For both subforest and context partitioning, in order to check for a node u in σ satisfying our size constraints, we perform a binary search on a root-to-leaf path in σ aided by the dynamic tree query data structure of [Navarro and Sadakane, 2014] (see Theorem 4.5.2 for the specific set of queries). The tree query data structure supports $\mathcal{O}(\log n)$-time dynamic insertions and deletions and answers queries in time $\mathcal{O}(\log n)$. Thus, our partitioning can be done in time $\mathcal{O}(\log^2 n)$ per part in S.

Lastly, we describe our efficient matching routine for subforests and contexts for the matching step of our algorithm. We utilize the dynamic strings data structure of [Gawrychowski et al., 2018] to answer IPM, *internal pattern matching*, queries in $\mathcal{O}(\log n)$-time each, formally given in [Kempa and Kociumaka, 2022]. Given a pattern P and text T, $\mathsf{IPM}(P, T)$ reports all occur-

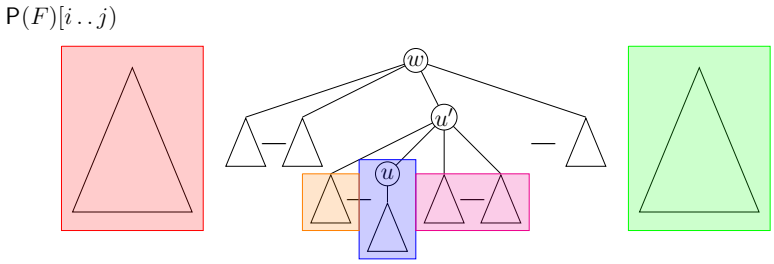


Figure 4.4: An example of partitioning a forest piece $\mathsf{P}(F)[i\,..\,j)$. The six parts as defined in the proof of Lemma 4.5.5 are colored 1 - blue, 2 - orange, 3 - magenta, 4 - uncolored, 5 - red, 6 - green.

rences of P in T. For $|T| < 2|P|$, all occurrences are reported as a single arithmetic progression of starting positions of P.

By definition, subforests are a subtring of $\mathcal{F}$, so a single IPM query straightforwardly finds a match for a given subforest σ of $\mathcal{F}$ if one exists. Contexts are defined as two substrings in $\mathcal{F}$ that correspond to a balanced set of parentheses when combined, and so, we can use two IPM queries to find the set of matches in $\mathcal{G}$ for each substring individually. The remaining difficulty is to find some pair of matches for each part that correspond to a single valid context in $\mathcal{G}$. Again using the data structure of [Navarro and Sadakane, 2014], we may rewrite the arithmetic progression of starting indices from each IPM query as instead an arithmetic progression of depths. See Figure 4.5 for an example. The intersections of these arithmetic progressions corresponds to potential context matches. We prove that we can check precisely the smallest depth intersection of the two depth progressions, and if this intersection is a valid context, we output this match, otherwise there is no match.

Since both our partitioning and our matching routines can be done in poly-logarithmic time dynamically, our entire dynamic algorithm can be performed in $\tilde{\mathcal{O}}(k)$-time. We may amortize this time cost to obtain $\tilde{\mathcal{O}}(1)$-time per update, see Section 4.5 for details.

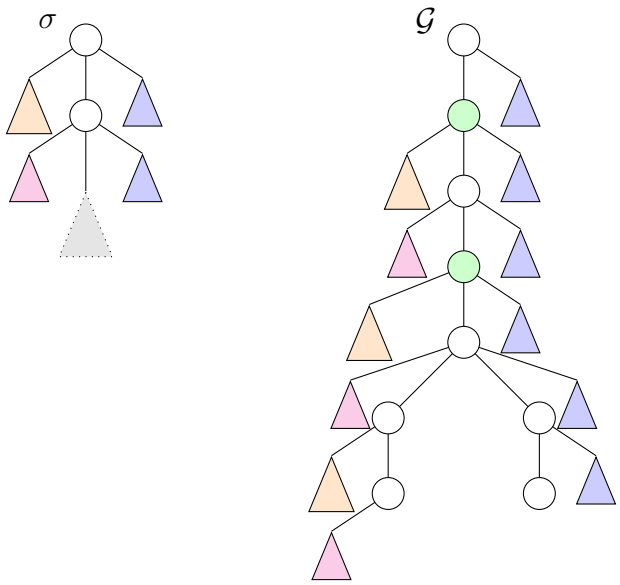


Figure 4.5: An example of a context σ and forest $\mathcal{G}$ for which $\mathsf{HasMatch}(\sigma) = \mathsf{true}$. The arithmetic progression of depths for the left and right part of σ are $f_1(i) = 2i + 1$, $f_2(i) = i$, respectively. Green colored nodes represent root nodes of valid matching contexts corresponding to intersections in f_1 and f_2.

Further Results. We provide further new results for Dynamic Dyck edit distance, specifically when we are allowed higher update times, and when we want to compute the Dyck edit distance exactly. Specifically, our result is a dynamic algorithm with sub-linear update time that computes an $\mathcal{O}(\log n)$-approximation of Dyck edit distance. For a parenthesis string X, our algorithm uses two separate subroutines to efficiently handle the cases when the Dyck edit distance of X, $\mathsf{ded}(X)$, is large ($\Omega(\sqrt{n})$) and small ($\mathcal{O}(\sqrt{n})$). For large distance, we may straightforwardly recompute the Dyck to string edit distance reduction of [Saha, 2014, Koucký and Saks, 2023] after a number of dynamic edits occur that is proportional to the distance. We can use a string edit distance algorithm on the resulting instances to obtain the following result. See Subsection 4.6.1 for details.

Theorem 4.1.6. *There exists a randomized dynamic algorithm that, for a fixed parameter $\epsilon \in \mathbb{R}_+$, maintains an $\mathcal{O}(f(\epsilon) \log |X|)$-approximation of $\mathsf{ded}(X)$ (correctly with high probability against an adaptive adversary) for a string $X \in \Sigma^*$ undergoing edits. The algorithm has amortized update*

time $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon}/\mathsf{ded}(X))$ *per edit and pre-processing time* $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon})$.

When $\mathsf{ded}(X)$ is small, maintaining the same approximation guarantee requires more frequent recomputations due to amortization constraints. We show how these recomputations can be done efficiently using the dynamic strings data structure of [Gawrychowski et al., 2018] that we equip with a few specific additional operations useful for our algorithm. For this case we obtain the following. See Subsection 4.6.2 for details.

Theorem 4.1.7. *There exists a randomized dynamic algorithm that maintains an* $\mathcal{O}(\log(\mathsf{ded}(X)))$*-approximation of* $\mathsf{ded}(X)$ *(correctly with high probability against an adaptive adversary) for a string* $X \in \Sigma^*$ *undergoing edits. This algorithm has amortized update time* $\tilde{\mathcal{O}}(\mathsf{ded}(X)+1)$ *per edit and pre-processing time of* $\tilde{\mathcal{O}}(|X|)$.

We combine the two above results to obtain the following theorem.

Theorem 4.1.8. *There exists a randomized algorithm that, for a fixed parameter* $\epsilon \in \mathbb{R}_+$*, maintains an* $\mathcal{O}(f(\epsilon)\log\mathsf{ded}(X))$*-approximation of* $\mathsf{ded}(X)$ *(correctly with high probability against an adaptive adversary) for a string* $X \in \Sigma^*$ *undergoing edits. This algorithm has amortized update time* $\tilde{\mathcal{O}}(|X|^{.5+\epsilon})$ *per edit and pre-processing time of* $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon})$.

In addition to approximation algorithms, we design an exact algorithm for the bounded-distance regime, where the Dyck edit distance is upper bounded by k. This algorithm achieves an update time of $\tilde{\mathcal{O}}((1+\mathsf{ded}(X))^5)$ per edit. Consequently, when k is small, this approach beats the traditional cubic-time solution for Dyck edit distance in the dynamic setting. To obtain this result, we adapt the Dyck edit distance algorithm of [Fried et al., 2024]. The algorithm of [Fried et al., 2024] separates parentheses of the input string into trapezoids and clusters, and we show

this decomposition can be maintained in the dynamic setting without much additional work. See Section 4.7 for details.

Theorem 4.1.9. *There exists a dynamic algorithm that maintains* $\mathsf{ded}(X)$ *(correctly with high probability against an adaptive adversary) for a string* $X \in \Sigma^*$ *undergoing edits, with update time* $\tilde{\mathcal{O}}((1+\mathsf{ded}(X))^5)$ *per edit. The algorithm has a pre-processing time* $\tilde{\mathcal{O}}(|X| + \mathsf{ded}(X)^5)$.

4.1.3 Related Works

String Edit Distance: One of the most fundamental problems in computer science is *string edit distance*, with roots dating back to the 1960s [Levenshtein, 1965, Needleman and Wunsch, 1970, Wagner and Fischer, 1974]. Given two strings, the objective of string edit distance is to find the minimum number of edit operations (insertions, deletions, and substitutions) required to transform one string into the other. An algorithm by Landau and Vishkin [Landau and Vishkin, 1988] builds upon insights by Ukkonen [Ukkonen, 1985] and Myers [Myers, 1986] to achieve this task in $\mathcal{O}(n+k^2)$ time by leveraging suffix trees and an elegant approach combining greedy and dynamic programming component. This algorithm has linear running time in the length of the strings n as long as the edit distance k is at most $\mathcal{O}(\sqrt{n})$. For larger values of k, the study of approximation algorithms for edit distance has been extensive [Landau et al., 1998, Indyk, 2001, Bar-Yossef et al., 2004, Batu et al., 2006, Andoni and Onak, 2012, Andoni et al., 2010], particularly in recent years [Haeupler et al., 2019, Chakraborty et al., 2020, Goldenberg et al., 2020, Koucký and Saks, 2020, Brakensiek and Rubinstein, 2020, Boroujeni et al., 2021, Kuszmaul, 2019]. Andoni and Nosatzki [Andoni and Nosatzki, 2020] obatin the current best bound, presenting a constant-factor approximation algorithm running in $\mathcal{O}(n^{1+\epsilon})$-time for any constant $\epsilon > 0$. Recently, Das, Gilbert,

Hajiaghayi, Kociumaka, and Saha [Das et al., 2023] proposed an $\mathcal{O}(n + k^5)$-time algorithm for weighted string edit distance in which a weight function determines the cost of edits, further improved to $\tilde{\mathcal{O}}(n+\sqrt{nk^3})$ by Cassis, Kociumaka, and Wellnitz [Cassis et al., 2023]. This latter paper also shows that any polynomial-factor improvement of this runtime (for any $\sqrt{n} \leq k \leq n$) would violate the All-Pairs Shortest Paths Hypothesis (i.e., yield an All-Pairs Shortest Paths algorithm with truly sub-cubic runtime).

Dyck Edit Distance: In this paper, we delve deeply into the variant of edit distance known as *Dyck edit distance*, a special case of general language edit distance [Aho and Peterson, 1972, Myers, 1995, Saha, 2017, Bringmann et al., 2019], with various practical applications such as fixing hierarchical data files, notably XML and JSON files [Harrison, 1978, Kozen, 1997]. For a parenthesis string of length n, the Dyck edit distance is the minimum number of edits (character insertions, deletions, and substitutions) required to make the string well-balanced. Various algorithms have been developed for both exact [Bringmann et al., 2019, Chi et al., 2022a, Dürr, 2023] and approximate [Saha, 2014, Das et al., 2022b, Koucký and Saks, 2023] versions of this problem. Finding the exact Dyck edit distance is at least as hard as Boolean matrix multiplication [Abboud et al., 2018]. Recent studies have also explored the bounded-distance Dyck edit problem in which an upper bound k is known for the Dyck edit distance of a string. Backurs and Onak [Backurs and Onak, 2016] initially presented an algorithm with a runtime of $\mathcal{O}(n + k^{16})$, later improved to $\mathcal{O}(n+k^5)$ [Fried et al., 2024], and further improved to $\mathcal{O}(n+k^{4.5442})$ through the use of fast matrix multiplication [Fried et al., 2024, Dürr, 2023]. However, except for the $\mathcal{O}(n^3)$-time exact algorithm for language edit distance [Myers, 1995], these results do not extend to the weighted setting. When the weight function is anti-symmetric and satisfies the triangle inequality, an $\mathcal{O}(n + k^{12})$-time al-

gorithm is known [Das et al., 2023] for the weighted setting.

Tree Edit Distance: The *tree edit distance* problem, initially introduced by Selkow [Selkow, 1977], is another generalization of edit distance aiming to compute a measure of dissimilarity between two rooted ordered trees with node labels. Given two trees, the tree edit distance is the minimum number of insertions, deletions, and relabelings needed to transform one tree into the other. This problem finds applications in diverse fields such as compiler optimization [Demaine et al., 2010], structured data analysis [Chawathe, 1999, Buneman et al., 2003, Ferragina et al., 2009], image analysis [Bunke and Shearer, 1998], and computational biology [Harel and Tarjan, 1984, Shapiro and Zhang, 1990, Gusfield, 1997, Landau et al., 1998, Bille, 2005]. Currently the best algorithm for finding exact tree edit distance has $\mathcal{O}(n^{2.9149})$ runtime due to Dürr [Dürr, 2023], following a series of improvements from $\mathcal{O}(n^6)$ [Tai, 1979], $\mathcal{O}(n^4)$ [Zhang and Shasha, 1989], $\mathcal{O}(n^3 \log n)$ [Klein, 1998], $\mathcal{O}(n^3)$ [Demaine et al., 2010], and $\mathcal{O}(n^{2.9546})$ [Mao, 2021]. Furthermore, a $(1+\epsilon)$-approximation algorithm for tree edit distance with a running time of $\tilde{\mathcal{O}}(n^2)$ was presented by Boroujeni, Ghodsi, Hajiaghayi, and Seddighin [Boroujeni et al., 2019]. Recently, Seddighin and Seddighin [Seddighin and Seddighin, 2022] introduced an $\mathcal{O}(n^{1.99})$-time $(3+\epsilon)$-approximation algorithm for tree edit distance. For the case when the tree edit distance doe not exceed a threshold k, Akmal and Jin [Akmal and Jin, 2021] presented an $\tilde{\mathcal{O}}(nk^2)$-time algorithm (improving upon an $\mathcal{O}(nk^3)$-time solution [Touzet, 2005]), whereas Das, Gilbert, Hajiaghayi, Kociumaka, and Saha [Das et al., 2023] achieved a runtime of $\tilde{\mathcal{O}}(n + k^7)$ algorithm (improving upon an $\mathcal{O}(n + k^{15})$-time solution [Das et al., 2022a]). For weighted tree edit distance, the fastest algorithm, by Demaine, Mozes, Rossman, and Weimann [Demaine et al., 2010], operates in $\mathcal{O}(n^3)$ time, matching the conditional lower bound of Bringmann, Gawrychowski, Mozes,

and Weimann [Bringmann et al., 2020]. However, when the weighted tree edit distance k is small and the weight function satisfies the triangle inequality, the $\mathcal{O}(n + k^{15})$-time algorithm of [Das et al., 2023] may outperform the $\mathcal{O}(n^3)$ upper bound.

Dynamic Edit Distance: Dynamic algorithms address real-world scenarios where data evolves rapidly, necessitating efficient updates to maintain solutions. Over recent decades, substantial progress has been made in understanding fundamental problems in graphs and sequences within the dynamic setting. Notable examples include maximum matchings in graphs [Sankowski, 2007, Gupta and Peng, 2013, Bhattacharya et al., 2023], connectivity [Nanongkai and Saranurak, 2017, Chuzhoy et al., 2020], maximum flow [Chen et al., 2020, van den Brand et al., 2023], minimum spanning trees [Nanongkai et al., 2017, Chuzhoy et al., 2020], clustering [Charikar et al., 2004, Henzinger and Kale, 2020, Bateni et al., 2023], diameter estimation [van den Brand and Nanongkai, 2019], independent set [Assadi et al., 2018], pattern matching [Alstrup et al., 2000, Gawrychowski et al., 2018, Charalampopoulos et al., 2020c, Clifford et al., 2022], lossless compression [Nishimoto et al., 2020], string similarity [Alstrup et al., 2000, Charalampopoulos et al., 2020a], longest increasing subsequence [Mitzenmacher and Seddighin, 2020, Kociumaka and Seddighin, 2021, Gawrychowski and Janczewski, 2021], suffix arrays [Kempa and Kociumaka, 2022], and others.

Various dynamic models with differing complexities have been explored, including incremental, decremental, and fully-dynamic models, supporting insertion, deletion, or both [Bhattacharya and Kulkarni, 2020, Gutenberg et al., 2020, Karczmarz et al., 2022, van den Brand et al., 2023, Bernstein, 2016, Henzinger et al., 2016, Bernstein et al., 2020b, Bernstein et al., 2020a, Demetrescu and Italiano, 2004, Gupta and Peng, 2013, Roditty and Zwick, 2016, Hanauer et al., 2022].

The evolution of dynamic edit distance algorithms initially tackled simpler variants of the problem, with early focus on updates limited to string endpoints [Landau et al., 1998, Kim and Park, 2004, Ishida et al., 2005, Tiskin, 2008]. Notably, Tiskin's linear-time algorithm [Tiskin, 2008] accommodates edit distance maintenance with character insertions or deletions at either endpoint of the strings. Recently, Charalampopoulos, Kociumaka, and Mozes [Charalampopoulos et al., 2020b] presented an $\tilde{O}(n)$-update-time algorithm for the general dynamic edit distance problem, allowing updates anywhere within the strings, with the caveat that SETH prohibits sublinear update time even in the most restrictive scenarios. A very recent preprint of Gorbachev and Kociumaka [Gorbachev and Kociumaka, 2024] circumvents this lower bound and achieves $\tilde{O}(k)$ time per update, where k is the edit distance. Kociumaka, Mukherjee, and Saha presented the first approximate dynamic edit distance algorithm that approximates edit distance in $n^{o(1)}$ time within a $n^{o(1)}$ approximation factor [Kociumaka et al., 2023]. Improving the approximation to constant in sub-polynomial update time remains a fascinating open question.

In addition to dynamic edit distance algorithms, dynamic data structures for string problems have been extensively studied. These structures maintain dynamic strings while accommodating queries such as equality testing, longest common prefix, pattern matching, and more [Mehlhorn et al., 1997, Alstrup et al., 2000, Gawrychowski et al., 2018, Charalampopoulos et al., 2020c, Clifford et al., 2022, Gawrychowski et al., 2018, Nishimoto et al., 2020, Kempa and Kociumaka, 2022, Charalampopoulos et al., 2020a, Chen et al., 2013a, Mitzenmacher and Seddighin, 2020, Kociumaka and Seddighin, 2021, Gawrychowski and Janczewski, 2021, Amir et al., 2019]. Some works also consider split and concatenate operations, enabling functionalities beyond character edits, such as cut-paste and copy-paste operations [Mehlhorn et al., 1997, Alstrup et al., 2000, Gawrychowski et al., 2018]. Meanwhile, significant strides have been made in fully dy-

namic graph algorithms, addressing central problems like spanning forests, transitive closure, shortest paths, maximum matching, maximal independent set, maximal matching, set cover, vertex cover and others [Nanongkai and Saranurak, 2017, Chuzhoy et al., 2020, Nanongkai et al., 2017, Sankowski, 2004, van den Brand et al., 2019, Karczmarz et al., 2022, Demetrescu and Italiano, 2004, van den Brand and Nanongkai, 2019, Sankowski, 2007, Gupta and Peng, 2013, Assadi et al., 2018, Baswana et al., 2011, Solomon, 2016, Bhattacharya et al., 2019, Abboud et al., 2019, Bhattacharya and Kulkarni, 2019] (see also survey [Hanauer et al., 2022] for recent advances in fully dynamic graph algorithms).

4.2 Preliminaries

Given a string $X = x_0x_1 \dots x_{n-1}$, we denote x_i as $X[i]$ and substrings $x_ix_{i+1} \dots x_{j-1}$ as $X[i \,..\, j-1] = X(i-1 \,..\, j)$. $X[i \,..]$ denotes $X[i \,..\, |X|)$, the *suffix* of X starting at i. Given an interval $I \subseteq [0 \,..\, n)$, $X[I]$ is the substring of X of all characters in X with indices that fall in interval I. Any substring of X starting with the first character of X is a *prefix* of X, i.e. $X[0 \,..\, i)$ for any $i \in [0 \,..\, n]$. We let $\overline{X}$ represent the reverse string of X, i.e. $x_{n-1}x_{n-2} \dots x_0$. For two strings X, Y, $X \cdot Y$ is the concatenation of X and Y. For a sequence of strings $X_1, X_2, \dots, X_k$, $\bigodot_{i=1}^{k} X_i = X_1 \cdot X_2 \cdot \ldots \cdot X_k$. We denote $\mathsf{LCP}(X, Y)$ as the length of the longest common prefix of X and Y. Given two strings X and Y, the *edit distance* $\mathsf{ed}(X, Y)$ is the minimum number of insertions, deletions, and substitutions needed to transform X into Y.

4.2.1 Parenthesis Strings

For a parenthesis string X, let the *transpose* of X, denoted by $T(X)$, be the string corresponding to $\overline{X}$ with each opening parenthesis replaced by a closing parenthesis of the same type and each closing parenthesis replaced by an opening parenthesis of the same type. For example, if $X =$ "(([)[", then $T(X) =$ "](])". We define the height of a character in X as the height of the index at which it occurs, where the height of an index $i \in [0\,..\,n]$, denoted $h_X(i)$, is defined as the difference in the number of opening and closing parentheses in $X[0\,..\,i)$. A parenthesis string is *monotone* if it is composed of only opening parentheses or only closing parentheses. For a string X, $\mathsf{LMP}(X)$ denotes the longest monotone prefix of X.

A *Dyck* sequence is a sequence of opening and closing parentheses that is well-parenthesized. We denote Dyck as the set of all Dyck sequences.

Definition 4.2.1. Consider a parenthesis string X. Indices $0 \le i < j < |X|$ are called *twins* if $h_X(i) = h_X(j+1)$ and $h_X(j') > h_X(i)$ holds for all $j' \in (i..j]$.[5]

We write $\mathsf{twin}_X(i) := j$ and $\mathsf{twin}_X(j) := i$ if i and j are twins.[6]

The following observation is easy to see from the above definition.

Observation 4.2.2. *The set of twins is* non-crossing*, i.e. given twins $i < i'$ and $j < j'$ with $i \neq j, j'$ and $i' = \mathsf{twin}_X(i), j' = \mathsf{twin}_X(j)$, it cannot be that $i < j < i' < j'$.*

In this work, we consider the problem of Dyck edit distance, defined as follows.

[5] Note that $X[i]$ is an opening parenthesis since $h_X(i+1) > h_X(i)$, and $X[j]$ is a closing parenthesis since $h_X(j) > h_X(j+1)$.

[6] If the parenthesis string X is clear from the context, we omit the subscript X at $h_X(\cdot)$ and $\mathsf{twin}_X(\cdot)$.

Definition 4.2.3 (Dyck edit distance)**.** Given a parenthesis string X, the *Dyck edit distance* of X, denoted as $\mathsf{ded}(X)$, is the minimum number of character deletions, insertions, and substitutions needed to place X in Dyck.

4.2.2 Tree Edit Distance Preliminaries

4.2.2.1 Alignments

Recall that given two strings X and Y, the edit distance $\mathsf{ed}(X, Y)$ is the minimum number of insertions, deletions, and substitutions needed to transform X into Y. Edit distance may be equivalently defined via the minimum cost *alignment* $\mathcal{A} : X \to Y$.

Definition 4.2.4 (Alignments)**.** A sequence $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ is an *alignment* of a string $X \in \Sigma^*$ onto a string $Y \in \Sigma^*$, denoted $\mathcal{A} : X \to Y$, if $(x_0, y_0) = (0, 0)$, $(x_m, y_m) = (|X|, |Y|)$, and $(x_{t+1}, y_{t+1}) \in \{(x_t + 1, y_t + 1), (x_t + 1, y_t), (x_t, y_t + 1)\}$ for $t \in [0\,..\,m)$.

Given an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m} : X \to Y$, for every $t \in [0\,..\,m)$:

- If $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t)$, we say that $\mathcal{A}$ *deletes* $X[x_t]$.
- If $(x_{t+1}, y_{t+1}) = (x_t, y_t + 1)$, we say that $\mathcal{A}$ *deletes* $Y[y_t]$.
- If $(x_{t+1}, y_{t+1}) = (x_t + 1, y_t + 1)$, we say that $\mathcal{A}$ *aligns* $X[x_t]$ and $Y[y_t]$, denoted $X[x_t] \sim_{\mathcal{A}} Y[y_t]$. If additionally $X[x_t] = Y[y_t]$, we say that $\mathcal{A}$ *matches* $X[x_t]$ and $Y[y_t]$, denoted $X[x_t] \simeq_{\mathcal{A}} Y[y_t]$. Otherwise, we say that $\mathcal{A}$ *substitutes* $X[x_t]$ for $Y[y_t]$.

The *cost* of an edit distance alignment $\mathcal{A}$ is the total number characters that $\mathcal{A}$ deletes, inserts, or substitutes. We denote the cost by $\mathsf{ed}_{\mathcal{A}}(X, Y)$. The cost of an alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ is at least its *width*, defined as $\max_{t=0}^{m} |x_t - y_t|$. Observe that $\mathsf{ed}(X, Y)$ can be defined as the minimum

cost of an alignment of X and Y, that is, $\mathsf{ed}(X,Y) = \min_{\mathcal{A}:X\to Y} \mathsf{ed}_{\mathcal{A}}(X,Y)$. An alignment of X and Y is *optimal* if its cost is equal to $\mathsf{ed}(X,Y)$. We often consider a restricted version of edit distance, the *deletion-only edit distance*, which is the minimum cost of an alignment of X and Y in which only deletions and matches are allowed (no substitutions are allowed), denoted $\mathsf{ded}(X,Y)$. It is easy to see that $\mathsf{ded}(X,Y) \le 2\cdot\mathsf{ed}(X,Y)$ as any substitution $(x_t,y_t),(x_t+1,y_t+1)$ can be replaced with two deletions $(x_t,y_t),(x_t+1,y_t),(x_t+1,y_t+1)$. We define analogous deletion-only versions for Dyck and Tree edit distances as well, denoted $\mathsf{ded}_{\mathsf{d}}$ and $\mathsf{ted}_{\mathsf{d}}$, respectively.

We say that a set $M \subseteq \mathbb{Z}_{\ge 0}\times\mathbb{Z}_{\ge 0}$ is ***non-crossing*** if there are no distinct pairs $(x,y),(x',y') \in M$ with $x \le x'$ and $y \ge y'$. Observe that, for every alignment $\mathcal{A}$ of X,Y, the set $\{(x,y)\in[0\,..\,|X|)\times[0\,..\,|Y|) : X[x]\sim_{\mathcal{A}} Y[y]\}$ is non-crossing.

4.2.2.2 Forests

A *tree* $\mathcal{T}=(V,E,\lambda)$ is a connected, directed, acyclic graph with node set V, edge set E, and node labeling $\lambda : V\to\Sigma$ for some alphabet Σ. For a node u, we say $u\in\mathcal{T}$ when u is in the node set V of $\mathcal{T}$. Given nodes $u,v\in V$ and edge $(u,v)\in E$, we call u the *parent* of v and v the *child* of u. For a given node $v\in V$, let $\mathsf{child}(v)$ denote the set of children of v. A node $v\in V$ is an *internal* node if it has at least one child and is a *leaf* node otherwise. Let P denote the unique path from a node $v\in V$ to the root of r of $\mathcal{T}$ (inclusive). Every node u in P is an *ancestor* of v and v is a *descendant* of u. A path from the root of T to a leaf node is called a *branch* of T. In this work, we consider *ordered* trees in which the children of each node have a left-to-right ordering. Let $\mathcal{T}(v)$ denote the subtree of $\mathcal{T}$ rooted at v. We say that $|\mathcal{T}|$ is the *size* of $\mathcal{T}$ and is equal to $|V|$. A *forest* is a ordered set of trees.

Given two forests $\mathcal{F} = (V_{\mathcal{F}}, E_{\mathcal{F}}, \lambda_{\mathcal{F}})$ and $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}}, \lambda_{\mathcal{G}})$, the *tree edit distance* $\mathsf{ted}(\mathcal{F}, \mathcal{G})$ is the minimum number of node insertions, node deletions, and node label substitutions needed to transform $\mathcal{F}$ into $\mathcal{G}$. In a forest $\mathcal{F}$, a *node deletion* of a node $v \in \mathcal{F}$ with parent u and children $w_1, w_2, \ldots, w_\ell$ removes v from $\mathcal{F}$ and sets the parent of $w_1, w_2, \ldots, w_\ell$ to u. If v has no parent, then each of $w_1, w_2, \ldots, w_\ell$ become root nodes of its own tree (its respective subtree) in $\mathcal{F}$. In a forest $\mathcal{F}$ containing node u with children $w_1, w_2, \ldots, w_\ell$, a *node insertion* adds a new node v as the child of some node $u \in \mathcal{F}$ such that u is to the right of w_i, to the left of w_{j+1}, and u is set to be the parent of $w_{i+1}, w_{i+2}, \ldots, w_j$ for any $0 \leq i < j \leq \ell$. If $i = 0$ or $j = \ell$, then u will have no left or right sibling, respectively, after insertion since there is no w_0 or $w_{\ell+1}$ node. Furthermore, if a node v is inserted as a new root of $\mathcal{F}$, v becomes the child of the "virtual root" of $\mathcal{F}$, the parent of all root nodes in $\mathcal{F}$, and so v will become the parent of a subset of the root nodes of $\mathcal{F}$.

Given a forest $\mathcal{F}$, a folklore mapping of $\mathcal{F}$ to a string, called the *parenthesis representation* of $\mathcal{F}$, $\mathsf{P}(\mathcal{F})$ is defined as follows:

Let $\mathcal{F}$ be comprised of trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$ with root nodes $r_1, r_2, \ldots, r_m$, respectively, where trees are in order from left to right such that $\mathcal{T}_1$ is the leftmost tree in $\mathcal{F}$. Then, $\mathsf{P}(\mathcal{F}) = \bigodot_{i=1}^{m} \texttt{(}_{\lambda(r_i)} \mathsf{P}(\mathcal{T}_i \setminus r_i) \texttt{)}_{\lambda(r_i)}$. Note that in the above definition, we use parentheses `(` and `)` to distinguish the parentheses contained in the string from the normal algebraic use of parentheses. Furthermore, the subscript of such a parentheses $\texttt{(}_i$ denotes a unique type of parentheses that is only equal to other parentheses with the same subscript, which we often call the *label* of the parenthesis. For a given node u in $\mathcal{F}$, let $o_P(u)$ and $c_P(u)$ represent the index of the opening and closing parentheses, respectively, corresponding to u in $\mathsf{P}(\mathcal{F})$. We call $o_P(u)$ the *twin* of $c_P(u)$, and vice versa.

It is easy to see inductively that the substring of $\mathsf{P}(\mathcal{T})$ between $o_P(u)$ and $c_P(u)$ for a node u

in tree $\mathcal{T}$ is exactly the parenthesis representation of the subtree of u, i.e. $\mathsf{P}(\mathcal{T})[o_P(u)\,..\,c_P(u)] = \mathsf{P}(\mathcal{T}(u))$. This implies the following useful observation.

Observation 4.2.5. *Given a forest $\mathcal{F}$ with nodes u, v. u is an ancestor of v if and only if $o_P(u) < o_P(v) < c_P(v) < c_P(u)$.*

Definition 4.2.6 (Tree Alignments)**.** We say that an alignment $\mathcal{A} : \mathsf{P}(\mathcal{F}) \to \mathsf{P}(\mathcal{G})$ is a *tree alignment* of forests $\mathcal{F}$ and $\mathcal{G}$, denoted by $\mathcal{A} : \mathcal{F} \to \mathcal{G}$ if the following *consistency* conditions are satisfied for each $u \in V_{\mathcal{F}}$:

- either $\mathcal{A}$ deletes both $\mathsf{P}(\mathcal{F})[o_P(u)]$ and $\mathsf{P}(\mathcal{F})[c_P(u)]$, or
- there exists $v \in V_{\mathcal{G}}$ such that $\mathsf{P}(\mathcal{F})[o_P(u)] \sim_{\mathcal{A}} \mathsf{P}(\mathcal{G})[o_P(v)]$ and $\mathsf{P}(\mathcal{F})[c_P(u)] \sim_{\mathcal{A}} \mathsf{P}(\mathcal{G})[c_P(v)]$.

Note that $\mathsf{ted}(\mathcal{F}, \mathcal{G})$ can be defined as the minimum cost tree alignment of $\mathsf{P}(\mathcal{F})$ and $\mathsf{P}(\mathcal{G})$, i.e., $2 \cdot \mathsf{ted}(\mathcal{F}, \mathcal{G}) = \min_{\mathcal{A} \in \mathsf{TA}(\mathcal{F}, \mathcal{G})} \mathsf{ted}_{\mathcal{A}}(\mathcal{F}, \mathcal{G})$.

4.2.2.3 Heavy-Light Decompositions

Given a forest $\mathcal{F}$, a heavy-light decomposition of $\mathcal{F}$ assigns a label *heavy* or *light* to each node of $\mathcal{F}$. For a given node $u \in \mathcal{F}$, a child node v of u is *heavy* if $\lfloor \lg |\mathcal{F}(v)| \rfloor = \lfloor \lg |\mathcal{F}(u)| \rfloor$, and *light* otherwise. A node with no parent is light by default. Here, for a node $u \in \mathcal{F}$, $\mathcal{F}(u)$ denotes the subtree of $\mathcal{F}$ rooted at vertex u , and $\lfloor \lg |\mathcal{F}(u)| \rfloor$ denotes the *heavy depth* of u in $\mathcal{F}$. A set of heavy nodes in $\mathcal{F}$ is called a *heavy path* if it forms a connected path in $\mathcal{F}$. The following useful observations are folklore:

Observation 4.2.7. *Given a forest $\mathcal{F}$ with $|\mathcal{F}| = n$, any root-to-leaf path includes at most $\mathcal{O}(\log n)$ light nodes.*

Proof. A node v is light if it does not have the same heavy depth as its parent u. For nodes u and v such that v is in the subtree of u, note that the heavy depth of v is at most the heavy depth of u since $|\mathcal{F}(v)| < |\mathcal{F}(u)|$. Therefore, for any root-to-leaf path, the sequence of heavy depth values corresponding to each node on the path is non-increasing. Since the maximum heavy depth value is $\lfloor \lg |\mathcal{F}| \rfloor = \lfloor \lg |n| \rfloor$ and the minimum heavy depth value is 0, there are at most $\lfloor \lg |n| \rfloor + 1 = \mathcal{O}(\log n)$ pairs of nodes u and v where u is the parent of v in a root-to-leaf path such that the heavy depth of v is smaller than the heavy depth of u. Therefore, there are at most $\mathcal{O}(\log n)$ light nodes in a root-to-leaf path. □

Observation 4.2.8. *Given a forest $\mathcal{F}$, every node $u \in \mathcal{F}$ has at most one child node that is heavy.*

Proof. Assume for sake of contradiction that a node u has at least two heavy children v_1, v_2. Let $2^{k+1} > |\mathcal{F}(u)| \geq 2^k$ for some $k \in \mathbb{Z}^+$. Then, $\lfloor \lg |\mathcal{F}(v_1)| \rfloor = \lfloor \lg |\mathcal{F}(v_2)| \rfloor = k$, and so,

$$|\mathcal{F}(v_1)| + |\mathcal{F}(v_2)| \geq 2^{\lfloor \lg |\mathcal{F}(v_1)| \rfloor} + 2^{\lfloor \lg |\mathcal{F}(v_2)| \rfloor} = 2 * 2^k = 2^{k+1} > |\mathcal{F}(u)|. \tag{4.1}$$

Since v_1 and v_2 are both children of u, however, $|\mathcal{F}(u)| \geq |\mathcal{F}(v_1)| + |\mathcal{F}(v_2)|$, which is a contradiction with Inequality 4.1. □

4.3 Approximate Dynamic Dyck Edit Distance

In this section, we prove our main result for dynamic Dyck edit distance, a sub-polynomial approximation with sub-polynomial update time for dynamic edits. Specifically, for parameters $2 \leq b \leq n$ and a dynamic parenthesis string X of length at most n, our algorithm achieves an $\mathcal{O}(b \log_b n \log n)$-approximation and supports updates in time $\mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$. We show that

using a tree representation of parenthesis string X determined by the heights of the parentheses in X, the heavy-light decomposition of the tree provides a novel natural reduction from Dyck to string edit distance. Each update to X will cause at most $\mathcal{O}(\log n)$ updates to the heavy paths and the corresponding string edit distance instances. We then use a dynamic approximation of [Kociumaka et al., 2023] to maintain an approximate string edit distance for each heavy path.

Definition 4.3.1 (Reduction tree)**.** Given a parenthesis string X, the *height tree* $\mathcal{T}_X$ is defined via pairs of twins in X.

Each node in $\mathcal{T}_X$ corresponds to an opening parenthesis $X[i]$ and the corresponding closing parenthesis $X[\mathsf{twin}(i)]$. For any closing parentheses $X[i]$ with $X[\mathsf{twin}(i)]$ undefined (there is no earlier opening parenthesis twin in X), we add an opening parenthesis of an unused type to the start of X, which we refer to as a *dummy* opening parenthesis. The parent of the node corresponding to $X[i]$ is the node corresponding to $X[j]$ where $j = \max\{j' \in [0\,..\,i) : h(j') = h(i) - 1\}$ if such a j exists. We add an additional node to be the root of the tree and set the root to be the parent of all nodes with no parents according to our above construction.

Given a node u, we let i_u represent the index of the corresponding opening parenthesis of u. We say u *contains* $X[i_u]$. Note these indices are adjusted after inserting the dummy opening parentheses.

A useful observation regarding the height tree is that, since the parent u of a given node v must have a smaller index and smaller height, then, in fact, any node w is a descendant of u if and only if the indices i_w and $\mathsf{twin}(i_w)$ lie in the range $[i_u, \mathsf{twin}(i_u)]$. Using this observation, we can efficiently determine whether a node is heavy or light in $\mathcal{O}(\log(n))$ time by maintaining a self-balancing binary tree built on top of X that returns the indices corresponding to any given node of

$\mathcal{T}_X$.

Observation 4.3.2. *For every two nodes u, v in height tree $\mathcal{T}_X$, the node v is a (proper) descendant of u if and only if $i_u < i_v < \mathsf{twin}(i_v) < \mathsf{twin}(i_u)$.*

Proof. First, let v be a proper descendant of u and assume for contradiction we have $i_u < \mathsf{twin}(i_u) < i_v < \mathsf{twin}(i_v)$ or $i_v < \mathsf{twin}(i_v) < i_u < \mathsf{twin}(i_u)$ (other orderings contradict the non-crossing properties of twins). We begin with the case when $i_v < \mathsf{twin}(i_v) < i_u < \mathsf{twin}(i_u)$. Consider the path from u to v. Observe that by Definition 4.3.1, each parent-child pair in this path must have the opening parenthesis corresponding to the parent to the left of the opening parenthesis corresponding to the child. Therefore, $i_u < i_v$, which contradicts our assumption.

Now, assume v is a proper descendant of u with $i_u < \mathsf{twin}(i_u) < i_v < \mathsf{twin}(i_v)$. Consider the path $w_0, w_1, w_2, \ldots, w_\ell$ in $\mathcal{T}_X$ from u to v with $w_0 = u$ and $w_\ell = v$. By Definition 4.3.1, $h(i_{w_i}) < h(i_{w_{i+1}})$ for all $0 \leq i < \ell$ and so, $h(i_u) < h(i_v)$. Furthermore by our definition of parents, u must be the closest opening parenthesis to i_v with height $h(i_u)$ satisfying $i_u < i_v$. By the definition of twins, we know that $h(\mathsf{twin}(i_u) + 1) = h(i_u)$. Since height only increments by 1 per opening parenthesis and decrements by 1 per closing parenthesis in the parenthesis string and $h(i_v) > h(i_u)$, there must be some opening parenthesis at index $x > \mathsf{twin}(i_u)$ (which may be $\mathsf{twin}(i_u) + 1$) such that $h(x) = h(i_u)$. This contradicts that i_u is the closest opening parenthesis to i_v satisfying $i_u < i_v$ with height $h(i_u)$. Thus, if v is a proper descendant of u, we must have that $i_u < i_v < \mathsf{twin}(i_v) < \mathsf{twin}(i_u)$.

Now, assume we have $i_u < i_v < \mathsf{twin}(i_v) < \mathsf{twin}(i_u)$. Let u' be the ancestor of v with height $h(i_{u'}) = h(i_u)$; by Definition 4.3.1, u' is the node corresponding to the closest opening parenthesis to i_v with height $h(i_u)$ and satisfying $i_{u'} < i_v$. If $u' = u$, we are done. Otherwise, by

our proof above and the non-crossing property of twins, we know that $i_u < i_{u'} < i_v < \mathsf{twin}(i_v) < \mathsf{twin}(i_{u'}) < \mathsf{twin}(i_u)$. However, this contradicts the definition of twins for i_u and $\mathsf{twin}(i_u)$, since $\mathsf{twin}(i_{u'})$ satisfies $h(\mathsf{twin}(i_{u'}) + 1) = h(i_u)$ and is closer to i_u than $\mathsf{twin}(i_u)$. Thus, u must be an ancestor of v. □

We decompose $\mathcal{T}_X$ into a heavy-light decomposition. Recall that in our heavy-light decomposition, for a given node $u \in \mathcal{T}_X$, a child node v of u is *heavy* if $\lfloor \lg |\mathcal{T}_X(v)| \rfloor = \lfloor \lg |\mathcal{T}_X(u)| \rfloor$, and *light* otherwise. Note that u has at most one heavy child by Observation 4.2.8.

A *heavy* path of $\mathcal{T}_X$ is a path beginning with a light node u and the longest path of descendants of u which are all heavy nodes (some heavy paths may consist of a single light node). Note in Section 4.2, we defined heavy paths to exclude light nodes, but in the remainder of this section we assume that light nodes are included as defined above. Given a heavy path $P = (u_1, u_2, u_3, \ldots, u_\ell)$, the *heavy string* corresponding to P is the string $X[i_{u_1}] \cdot X[i_{u_2}] \cdot \ldots \cdot X[i_{u_\ell}] \cdot X[\mathsf{twin}(i_{u_\ell})] \cdot \ldots \cdot X[\mathsf{twin}(i_{u_1})]$. We denote the set of heavy paths of tree $\mathcal{T}_X$ as $\mathcal{H}_{\mathcal{T}_X}$, and we note that $\mathcal{H}_{\mathcal{T}_X}$ partitions $\mathcal{T}_X$. We denote $\mathcal{S}_X$ as the set of heavy strings of $\mathcal{X}$, which partitions X. We assume each heavy path in $\mathcal{H}_{\mathcal{T}_X}$ stores a pointer to its head node, tail node, and the corresponding heavy string in $\mathcal{S}_X$. We also assume each heavy path knows its length (which may be calculated during pre-processing and maintained in the head nodes of each path for constant-time access). We will maintain these dynamically, and we do not explicitly store $T_{\mathcal{X}}$.

We give an $\mathcal{O}(\log^4 n)$-time algorithm for maintaining $\mathcal{S}_X$ when X undergoes dynamic edits. Furthermore, we show that there are at most $\mathcal{O}(\log n)$ changes to the set of heavy path strings, and therefore only $\mathcal{O}(\log n)$ changes to the string edit distance instances we obtain from our set of heavy paths. To show this, we partition the set of nodes in $\mathcal{T}_X$ into three types based on their

location relative to the update: Type I nodes whose corresponding opening and closing parenthesis come before the edit, Type II nodes whose corresponding opening parenthesis comes before the edit and whose corresponding closing parenthesis comes after the edit, and Type III nodes whose corresponding parentheses come after the edit. It is easy to see that the subtrees of Type I and Type III nodes remain completely the same since the heights of all parentheses in these subtrees are shifted by the same amount by the dynamic edit. As for Type II nodes, it turns out that these are exactly the set of nodes on the path from the root of $\mathcal{T}_X$ to the node containing the edited parenthesis. As such, we know there are at most $\mathcal{O}(\log n)$ light nodes on this path by Observation 4.2.7.

Through a careful case analysis of the interactions between Type II nodes and their parents and children, we show that any heavy node that becomes light must have a neighboring light Type II node that may be charged for updating the heavy path. Since there are $\mathcal{O}(\log n)$ light Type II nodes and only one child of a Type II node may be heavy, there are at most $\mathcal{O}(\log n)$ edits to $\mathcal{S}_X$ per edit to X. To identify these changes, we look at the light Type II nodes of $\mathcal{T}_X$ and check all the nearby nodes to see if any have changed from light to heavy or heavy to light. When checking to see if a node is light or heavy, we use Observation 4.3.2 as well as a segment tree to answer range queries of the form (i, h) where we want to know the index of the next parenthesis after index i whose height is h. To maintain this segment tree efficiently subject to updates, we use a common technique called lazy propagation. With these range queries, we can find the size of a node's subtree quickly and check whether they are heavy or light. While the lemma is significant, we leave the proof to the end of this section, in Subsection 4.3.1 since it is a long case-by-case analysis and introduces a few additional algorithmic tools, e.g. the range queries discussed above.

Lemma 4.3.3. *For a string X with $n = |X|$, $\mathcal{H}_{\mathcal{T}_X}$ and $\mathcal{S}_X$ undergoes $\mathcal{O}(\log n)$ edits, which can be*

identified and maintained in $\mathcal{O}(\log^5 n)$ time, when X undergoes a dynamic insertion or deletion.

Proof. See Subsection 4.3.1 for details. □

We now prove that the heavy string set provides a good approximation of $\mathsf{ded}(X)$ for a parenthesis string X. Recall that $\mathsf{ded_d}(X)$ denotes the deletion-only Dyck edit distance where substitutions and insertions are not allowed, and that $\mathsf{ded}(X) \leq \mathsf{ded_d}(X) \leq 2 \cdot \mathsf{ded}(X)$.

Lemma 4.3.4. *Given a parenthesis string X and its heavy string set $\mathcal{S}_X$,*

$$\mathsf{ded_d}(X) \leq \sum_{s \in \mathcal{S}_X} \mathsf{ded_d}(s) \leq 2(\lg(|X|)+1)\mathsf{ded_d}(X).$$

Proof. Given a string $s \in \mathcal{S}_X$, let $P_s \in \mathcal{H}_{\mathcal{T}_X}$ be the corresponding heavy path in $\mathcal{T}_X$ of s and let r_s represent the head node of P_s (which is a light node). We define the *depth* of s to be $\lfloor \lg |\mathcal{T}_X(r_s)| \rfloor$. We partition the strings of $\mathcal{S}_X$ according to their depths. We let $M_i = \{M_1^i, M_2^i, \ldots, M_{\ell_i}^i\}$ represent the set of heavy path strings whose depth is i for $0 \leq i < h$, where $h \leq \lfloor \lg |X| \rfloor + 1$. For example, M_0 is the set of single-node heavy paths at the leaves of T. For $0 \leq i \leq h$, we define X_i to be the string of X with $\bigcup_{i'=0}^{i-1} \bigcup_{j=0}^{\ell_{i'}} M_j^{i'}$ removed. We will prove that the following holds for every $0 \leq i < h$:

$$\mathsf{ded_d}(X_{i+1}) \leq \mathsf{ded_d}(X_i) \leq \mathsf{ded_d}(X_{i+1}) + \sum_{j=1}^{\ell_i} \mathsf{ded_d}(M_j^i) \qquad \text{and} \qquad \sum_{j=1}^{\ell_i} \mathsf{ded_d}(M_j^i) \leq 2\mathsf{ded_d}(X_i).$$

Let us first consider an optimal deletion-only edit sequence OPT_i for X_i. For each $1 \leq j \leq \ell_i$, let D_j^i be the number of deletions in M_j^i made by OPT_i and N_j^i be the number of parentheses in M_j^i that are matched to parentheses outside of M_j^i. We note that, since M_j^i is a balanced LR-segment, whenever a parenthesis of M_j^i is matched to a parenthesis outside of this segment, we

must delete a parenthesis in the other half of M_j^i since a matching must be non-crossing. Therefore,

$$N_j^i \le D_j^i. \tag{4.2}$$

Now, we define $\hat{M}_0^i, \hat{M}_1^i, \ldots, \hat{M}_{\ell_i}^i$ as the remaining substrings of X_i between each M_j^i, so that $X_i = \hat{M}_0^i \cdot M_1^i \cdot \hat{M}_1^i \cdot M_2^i \cdot \ldots \cdot M_{\ell_i}^i \cdot \hat{M}_{\ell_i}^i$. We define $\hat{D}_j^i$ as the number of deletions in $\hat{M}_j^i$ made by OPT_i and $\hat{N}_j^i$ as the number of parentheses in $\hat{M}_j^i$ that are matched to a parenthesis in some $M_{j'}^i$. We note that by definition, $\sum_{j=0}^{\ell_i} \hat{N}_j^i = \sum_{j=1}^{\ell_i} N_j^i$, and furthermore,

$$\mathsf{ded_d}(X_i) = \sum_{j=0}^{\ell_i} \hat{D}_j^i + \sum_{j=1}^{\ell_i} D_j^i.$$

Now, we consider $\mathsf{ded_d}(X_{i+1})$, which does not need to delete any parentheses from M_i since all of these segments are removed but may need to delete parentheses of X_{i+1} which were matched to parentheses in M_i. Therefore,

$$\begin{aligned}
\mathsf{ded_d}(X_{i+1}) &\le \mathsf{ded_d}(X_i) + \sum_{j=0}^{\ell_i} \hat{N}_j^i - \sum_{j=1}^{\ell_i} D_j^i \\
&= \mathsf{ded_d}(X_i) + \sum_{j=1}^{\ell_i} \left(N_j^i - D_j^i\right) \\
&\le \mathsf{ded_d}(X_i) + \sum_{j=1}^{\ell_i} \left(D_j^i - D_j^i\right) = \mathsf{ded_d}(X_i)
\end{aligned}$$

where the second inequality follows from (4.2). Furthermore, (4.2) yields

$$\sum_{j=1}^{\ell_i} \mathsf{ded_d}(M_j^i) \le \sum_{j=1}^{\ell_i} \left(D_j^i + N_j^i\right) \le \sum_{j=1}^{\ell_i} 2D_j^i \le 2\mathsf{ded_d}(X_i).$$

It remains to prove that $\mathsf{ded_d}(X_i) \leq \mathsf{ded_d}(X_{i+1}) + \sum_{j=1}^{\ell_i} \mathsf{ded_d}(M_j^i)$. For this, note that each string M_j^i is a substring of X_i and X_i is obtained from X_{i+1} by removing all these substrings M_j^i. If $Y, Z \in \mathsf{Dyck}$, then also $Y[0\,..\,y) \cdot Z \cdot Y[y\,..\,|Y|) \in \mathsf{Dyck}$ for every $y \in [0\,..\,|Y|]$; hence, in order to place X_i in Dyck, it suffices to independently perform the deletions that place X_{i+1} and all strings M_j^i in Dyck.

Finally, observe that

$$\begin{aligned}\sum_{s\in\mathcal{S}_X} \mathsf{ded_d}(s) = \sum_{i=0}^{h-1}\sum_{j=1}^{\ell_i} \mathsf{ded_d}(M_j^i) &\leq \sum_{i=0}^{h-1} 2\mathsf{ded_d}(X_i)\\ &\leq \sum_{i=0}^{h-1} 2\mathsf{ded_d}(X_0)\\ &= 2h \cdot \mathsf{ded_d}(X)\\ &\leq 2(\lg(|X|)+1)\mathsf{ded_d}(X).\end{aligned}$$

and

$$\begin{aligned}\sum_{s\in\mathcal{S}_X} \mathsf{ded_d}(s) = \sum_{i=0}^{h-1}\sum_{j=1}^{\ell_i} \mathsf{ded_d}(M_j^i) &\geq \sum_{i=0}^{h-1} (\mathsf{ded_d}(X_i) - \mathsf{ded_d}(X_{i+1}))\\ &= \mathsf{ded_d}(X_0) - \mathsf{ded_d}(X_h) = \mathsf{ded_d}(X) - \mathsf{ded_d}(\varepsilon) = \mathsf{ded_d}(X).\end{aligned}$$

This completes the proof. □

Combining Lemma 4.3.4, Observation 4.6.1, and the facts that $\mathsf{ded}(X) \leq \mathsf{ded_d}(X) \leq 2\mathsf{ded}(X)$ and $\mathsf{ed}(Y,Z) \leq \mathsf{ed_d}(Y,Z) \leq 2\mathsf{ed}(Y,Z)$, we immediately obtain the following result.

Corollary 4.3.5. *Given a parenthesis string X and its heavy string set $\mathcal{S}_X$,*

$$\mathsf{ded}(X) \leq 2 \sum_{s \in \mathcal{S}_X} \mathsf{ed}(\mathcal{L}(s), T(\mathcal{R}(s))) \leq 8(\lg(|X|) + 1)\mathsf{ded}(X).$$

We now state the dynamic string edit distance result of [Kociumaka et al., 2023], which we utilize to quickly solve each string edit distance instance we obtain from our reduction.

Theorem 4.3.6 ([Kociumaka et al., 2023] Theorem 7.10)**.** *Given integers $2 \leq b \leq n$, there exists a randomized dynamic algorithm that maintains an $\mathcal{O}(b \log_b n)$-approximation of $\mathsf{ed}(X, Y)$ (correctly with high probability against an oblivious adversary) for (initially empty) strings X, Y of length at most n undergoing edits. The expected update time of the algorithm is $\mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$.*

Our dynamic approximation with a trade-off between approximation factor and update time follows from Lemma 4.3.3, Corollary 4.3.5, and Theorems 4.3.6.

Theorem 4.3.7. *Given integers $2 \leq b \leq n$, there exists a randomized dynamic algorithm, that maintains an $\mathcal{O}(b \log_b n \log n)$-approximation of $\mathsf{ded}(X)$ (correctly with high probability against an oblivious adversary) for an (initially empty) string X of length at most n undergoing edits. The expected update time of the algorithm is $\mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$ per edit.*

Proof. Note, by Lemma 4.3.3, $\mathcal{S}(X)$ requires $\mathcal{O}(\log |X|)$ edits per dynamic edit to X and can be maintained in $\mathcal{O}(\log^4 n)$-time. By Theorem 4.3.6, these edits may be handled in time $\mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$ each, and so in total take expected time $\mathcal{O}(\log^4 n + \log n \cdot b^2 (\log n)^{\mathcal{O}(\log_b n)}) = \mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$ to maintain. We use the sum of the approximations for each heavy path string to obtain our approximation of $\mathsf{ded}(X)$, which may be updated by subtracting the $\mathcal{O}(\log n)$ affected

paths' old approximations and adding their new approximations after recomputing them. By Theorem 4.3.6 and Corollary 4.3.5, the approximation factor will be $\mathcal{O}(b \log_b n) \cdot 8(\lg(n) + 1) = \mathcal{O}(b \log_b n \log n)$. □

Observe that when b is set to $2^{\Theta(\sqrt{\log n \log\log n})}$, the approximation ratio becomes $\mathcal{O}(\log n \cdot 2^{\mathcal{O}(\sqrt{\log n \log\log n})})$ and the update time becomes $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$. Setting $b = \log^{O(\frac{1}{\epsilon})} n$, we get $(\log n)^{O(\frac{1}{\epsilon})}$-factor approximation with $O(n^{\epsilon})$ update time for any constant $\epsilon > 0$.

Although the dependence on parameter b in the approximation factor and update time may be somewhat high in the above theorem, both quantities match the state-of-the-art for dynamic string edit distance (up to an extra $\mathcal{O}(\log n)$ factor present in the approximation only). The Dyck edit distance is at least as hard to approximate as the string edit distance, so this relationship is expected: any significant improvements to the approximation or update time for dynamic Dyck edit distance would therefore automatically improve the state-of-the-art for the dynamic string edit distance, and vice versa.

4.3.1 Dynamic Heavy-Light Decomposition

In the remainder of this section, we describe how the set of heavy paths of a height tree for a parenthesis string X can be maintained in $\mathcal{O}(\log^5 n)$ time. Before we show the proof of how the heavy paths of $\mathcal{T}_X$ are maintained, we develop a few useful tools.

First, we discuss *minimum height range queries* $\mathsf{range}(i, h)$ that return the maximum range starting at i in X such that all parentheses in this range have heights greater than h. To answer them in $\mathcal{O}(\log^2 n)$ time, we use a well-known technique called lazy propagation to maintain a balanced binary tree with height information on top of X that we may use to quickly determine the answer

to such queries.

Lemma 4.3.8. *A parenthesis string X can be dynamically maintained in $\mathcal{O}(\log |X|)$ time per update so that, given $i, h \in [0\,..\,|X|)$, the range $\mathsf{range}(i, h)$ can be computed in $\mathcal{O}(\log^2 |X|)$ time.*

Proof. See Appendix 4.8.1 for details. □

Additionally, we also wish to know, for a given index i_v, its twin index $\mathsf{twin}(i_v)$. We use a self-balancing binary tree structure to do so, similar to the construction of $\hat{X}$ in the proof of Lemma 4.6.8.

Lemma 4.3.9. *A parenthesis string X can be dynamically maintained in $\mathcal{O}(\log |X|)$ time per update so that, given any parenthesis $X[i_v]$ corresponding to node $v \in \mathcal{T}_X$, the indices i_v and $\mathsf{twin}(i_v)$ can be computed in $\mathcal{O}(\log |X|)$ and $\mathcal{O}(\log^2 |X|)$ time, respectively. Furthermore, $|\mathcal{T}_X(v)|$ may be computed in $\mathcal{O}(\log^2 |X|)$ time.*

Proof. To answer index queries of a node $v \in \mathcal{T}_X$, we utilize a binary tree B built on top of X. Each node $x \in B$ corresponds to a substring of X, denoted S_x. Each leaf node of B corresponds to a single character of X and leaves are assigned characters in sorted order according to X such that the leftmost leaf corresponds to $X[0]$. Each internal node x of B with children x_L, x_R corresponds to the concatenation $S_{x_L} \cdot S_{x_R}$. Therefore, each level of B partitions X. In addition to this partition, each node $x \in B$ stores the size of its substring $|S_x|$ (note that x does not store the substring S_x itself, as this would require too much time to maintain). The tree B can be constructed straightforwardly in time $\mathcal{O}(|X|)$. When X undergoes a dynamic edit, we only need to update the nodes whose corresponding substrings are affected. When the height of B is $\mathcal{O}(\log |X|)$, there are only $\mathcal{O}(\log |X|)$ such nodes. Leaf nodes do not require any work to maintain, and internal nodes may

be reconstructed in constant time from the information in their children (the length of the substring S_x at node x is the sum of the lengths of the substrings S_{x_L} and S_{x_R} at the two children of x). We may use a self-balancing tree such as an AVL tree to maintain the height of $\mathcal{O}(\log |X|)$ while still only affecting $\mathcal{O}(\log |X|)$ nodes per update.

Now, we show how to use B to answer index queries. We assume that each node $v \in \mathcal{T}_X$ stores a pointer to the leaf of B corresponding to the character $X[i_v]$. To find the index i_v, we begin at the leaf corresponding to $X[i_v]$ and we traverse B upwards while maintaining a count. Whenever the node we are at is the left child of its parent we do not increase the count. Whenever the node we are at is the right child of its parent, this means that the left child corresponds to a substring of X lying to the left of $X[i_v]$ and so we add the size of this substring, stored in that left child, to the count. We do this until we reach the root node so that the count becomes the number of characters to the left of $X[i_v]$, which is i_v. Since the height of B is $\mathcal{O}(\log |X|)$ and each traversal to a parent node takes constant time, these queries only take $\mathcal{O}(\log |X|)$ time.

To get the index of $\mathsf{twin}(i_v)$, we perform a minimum height range query $\mathsf{range}(i_v + 1, h + 1)$ which takes $\mathcal{O}(\log^2 |X|)$ time by Lemma 4.3.8. Then, we may compute $|\mathcal{T}_X(v)| = (1 + \mathsf{twin}(i_v) - i_v)/2$ □

With the two tools above, we can efficiently determine if a given node $v \in \mathcal{T}_X$ is heavy or not. The subtree of the heavy child of u (if any) is at least half the size of the subtree of u. Therefore, we may make a binary search using minimum range height queries to identify the heavy child.

Lemma 4.3.10. *Given a parenthesis string X and the data structure of Lemma 4.3.8, for a parenthesis $X[i_v]$ corresponding to node $v \in \mathcal{T}_X$, we may find $i_u, \mathsf{twin}(i_u)$ where u is the heavy child of $v \in \mathcal{T}_X$, if it exists, in $\mathcal{O}(\log^3 n)$ time.*

Proof. We first obtain i_v and $\mathsf{twin}(i_v)$ using Lemma 4.3.9, and let $d = \mathsf{twin}(i_v) - i_v$. Additionally, we find $h(i_v)$ by computing the total length of all heavy paths from v to the root of $\mathcal{X}_T$ (recall each heavy path stores its length). We observe that any heavy child u of v must have $|\mathcal{T}_X(u)| \geq |\mathcal{T}_X(v)|/2$. Therefore, we perform $\mathsf{range}(i_v + d/4, h(i_v))$ since $h(i_u)$ must be $h(i_v) + 1$. If the range query returns an index after $i_v + d/2$, we may determine that $i_u \in (i_v \,..\, i_v + d/4)$, otherwise $i_u \in [i_v + d/4 \,..\, i_v + d/2]$ if it exists. We set $d' = d/2$ and search the respective half by setting our starting point to either $i_v + d'/4$ again or $i_v + d/4 + d'/4$. We then continue until we find the exact starting index of i_u if it exists, or find that no such index exists.

Once we determine i_u, we may use Lemma 4.3.9 to determine if u is indeed a heavy child of v. □

Now we prove that the heavy-light decomposition of $\mathcal{T}_X$ can be maintained in poly-logarithmic update time when X undergoes dynamic edits.

Proof of Lemma 4.3.3. Let Y be the string X after a dynamic update. We discuss deletions and insertions separately. First, let us assume that the dynamic update is a deletion of $X[j]$ so that $Y = X[0 \,..\, j) \cdot X(j \,..\, |X|)$. Let $\mathcal{T}_X$ and $\mathcal{T}_Y$ be the respective height trees of X and Y. We define the following function $f : [0 \,..\, |X|] \to [0 \,..\, |Y|]$ mapping positions in X to positions in Y:

$$f(i) = \begin{cases} i & \text{if } i \in [0 \,..\, j), \\ i - 1 & \text{if } i \in [j \,..\, |X|]. \end{cases}$$

Observe that $Y[f(i)] = X[i]$ holds for every $i \in [0 \,..\, |X|) \setminus \{j\}$. We also lift f to a mapping between the nodes of $\mathcal{T}_X$ and of $\mathcal{T}_X$ so that if a node $u \in \mathcal{T}_X$ contains an opening parenthesis $X[i]$

with $i \neq j$, then its image in $\mathcal{T}_Y$ is the node $f(u)$ that contains the corresponding opening parenthesis $Y[f(i)]$. If $X[j]$ is an opening parenthesis, then f is undefined ($\perp$) for the node containing $X[j]$. Furthermore, we classify nodes $u \in \mathcal{T}_X$ according to their relationship with j.

We say that a node $u \in \mathcal{T}_X$ is of *Type I* if $i_u < \mathsf{twin}_X(i_u) < j$, of *Type II* if $i_u < j \leq \mathsf{twin}_X(i_u)$, and of *Type III* if $j < i_u < \mathsf{twin}_X(i_u)$. Note that if $X[j]$ is an opening parenthesis, the node u with $i_u = j$ is not assigned to any type. This node will be deleted, so it is not useful to group with other nodes. We now discuss some relevant properties of each type of node.

Claim 4.3.11. *For all Type I or Type III nodes $u \in \mathcal{T}_X$, we have $\mathcal{T}_X(u) = \mathcal{T}_Y(f(u))$.*

Proof. First, assume u is a Type I node. By definition, $i_u < \mathsf{twin}_X(i_u) < j$. Then by Observation 4.3.2, every node $v \in \mathcal{T}_X(u)$ satisfies $i_v < \mathsf{twin}_X(i_v) < \mathsf{twin}_X(i_u) < j$, and thus it is also a Type I node. Furthermore, the heights within $[0\,..\,j)$ do not change, i.e., $h_Y(f(i)) = h_X(i)$ holds for all $i \in [0\,..\,j)$. Therefore, for any two type I nodes $v, w \in \mathcal{T}_X$, we have that w is the parent of v if and only if $f(w)$ is the parent of $f(v)$. Thus, $\mathcal{T}_X(u) = \mathcal{T}_X(f(u))$.

The proof of when u is a Type III node follows similarly. By definition, $j < i_u < \mathsf{twin}_X(i_u)$, and every node $v \in \mathcal{T}_X(u)$ satisfies $j < i_u < i_v < \mathsf{twin}_X(i_u)$, and thus it is also a Type III node. Furthermore, the heights within $[j\,..\,|X|]$ are all incremented, that is, $h_Y(f(i)) = h_X(i) + 1$ for all $i \in [j\,..\,|X|]$ if $X[j]$ is closing, or all decremented, that is, $h_Y(f(i)) = h_X(i) - 1$ for all $i \in [j\,..\,|X|]$ if $X[j]$ is opening. Therefore, for any two type III nodes $v, w \in \mathcal{T}_X$, we have that w is the parent of v if and only if $f(w)$ is the parent of $f(v)$. Thus, $\mathcal{T}_X(u) = \mathcal{T}_X(f(u))$. □

By Claim 4.3.11, the structure of Type I nodes and Type III is essentially intact by the mapping f. We therefore turn our attention to Type II nodes.

Claim 4.3.12. *Every Type II node in $\mathcal{T}_X$ has exactly one Type II child, except for the following node with no Type II children:*

- *if $X[j]$ is closing, the node $u \in \mathcal{T}_X$ with $\mathsf{twin}_X(i_u) = j$;*
- *if $X[j]$ is opening, the parent of the node u with $i_u = j$.*

Furthermore, for any two Type II nodes $v, w \in \mathcal{T}_X$, we have that w is the parent of v if and only if $f(w)$ is the parent of $f(v)$.

Proof. First, consider a node $u \in \mathcal{T}_X$ and its two children w, w' with w located to the left of w'. Then, $i_w < \mathsf{twin}_X(i_w) < i_{w'} < \mathsf{twin}_X(i_{w'})$. If both w, w' were Type II nodes, we would have $j \leq \mathsf{twin}_X(i_w) < i_{w'} < j$, a contradiction. Thus, every node $u \in \mathcal{T}_X$ has at most one Type II child.

Now, we consider the case when $X[j]$ is closing. First, if $\mathsf{twin}_X(i_u) = j$, all descendant nodes v of u satisfy $i_v < \mathsf{twin}_X(i_v) < \mathsf{twin}_X(i_u) = j$ and are Type I nodes. So, in this case, u has no Type II children as claimed. Now when $i_u < j < \mathsf{twin}_X(i_u)$, let us assume u has no Type II children. Therefore, all children v satisfy either $\mathsf{twin}_X(i_v) < j$ or $j < i_v$. Since, twin pairs are non-crossing by Observation 4.2.2, the Type II node w with $i_w < \mathsf{twin}_X(i_w) = j$ must be a descendant of v since $i_u < j < \mathsf{twin}_X(i_u)$. However, w cannot be a child of any Type I and Type III node, again due to twins' non-crossing property. Therefore, u must have at least one Type II child.

Now, let us assume that $u \in \mathcal{T}_X$ is a Type II node and $X[j]$ is opening. If u is the parent of node v with $i_v = j$, note that any other Type II node w must satisfy $i_w < j < \mathsf{twin}_X(i_w)$ and cannot be a sibling of v. Thus, u has no Type II children in this case. If u is not the parent of v, it must still be an ancestor of v and, by the same argument as above, u must have at least one Type II child.

To prove the final property of the statement, take Type II nodes v, w. By the definition of the mapping function, $f(v) \in \mathcal{T}_Y$ is the node containing $Y[f(i_v)]$ and $f(w) \in \mathcal{T}_Y$ is the node containing $Y[f(i_w)]$. Since $i_v, i_w < j$ and the heights within $[0\,..\,j)$ do not change, i.e., $h_Y(f(i)) = h_X(i)$ holds for all $i \in [0\,..\,j)$, we conclude that w is the parent of v if and only if $f(w)$ is the parent of $f(v)$. □

Claim 4.3.13. *Consider a Type II node $u \in \mathcal{T}_X$ and its image $f(u) \in \mathcal{T}_Y$. If $X[j]$ is closing, then $\mathsf{twin}_Y(f(i_u)) = f(\mathsf{twin}_X(i_w))$, where w is the parent of u. If the parent of u is the root of $\mathcal{T}_X$, then $\mathsf{twin}_Y(f(i_u))$ is undefined. If $X[j]$ is opening, then $\mathsf{twin}_Y(f(i_u)) = f(\mathsf{twin}_X(i_v))$, where v is the Type II child of u, or $\mathsf{twin}_Y(i_v) = f(\mathsf{twin}_X(j))$ if no such child exists.*

Proof. First, suppose that $X[j]$ is closing and w is the parent of u where w is not the root of $\mathcal{T}_X$. Note that $\mathsf{twin}_X(i_w) = \min\{r \geq i_w : h_X(r+1) = h_X(i_w)\}$ and $\mathsf{twin}_X(i_u) = \min\{r \geq i_u : h_X(r+1) = h_X(i_u)\}$. Moreover, $h_X(i_u) = h_X(i_w) + 1$. Consequently, $h_X(r) > h_X(i_u)$ for $r \in (i_u\,..\,\mathsf{twin}_X(i_u)]$, $h_X(r) \geq h_X(i_u)$ for $r \in (i_w\,..\,\mathsf{twin}_X(i_w)]$, and $h_X(\mathsf{twin}_X(i_w)+1) = h_X(i_u) - 1$. Informally, we know that all parentheses between i_u and $\mathsf{twin}_{\mathcal{X}}(i_u)$ have height greater than $h(i_u)$, and moreover, the height of i_w and its twin $\mathsf{twin}(i_w)+1$ is $h(i_u)-1$. After deleting $X[j]$, the height of all parentheses after index j are increased by 1, and so the height of $\mathsf{twin}_X(i_w)$ matches $h(i_u)$ in $\mathcal{T}_Y$ and is the nearest closing parenthesis with matching height to the right of i_u, i.e., it is the twin of i_u. Formally put, since $i_w \leq i_u < j \leq \mathsf{twin}_X(i_u) < \mathsf{twin}_X(i_w)$, as well as $h_Y(f(r)) = h_X(r)$ for $r \in [0\,..\,j]$ and $h_Y(f(r)) = h_X(r) + 1$ for $r \in (j\,..\,|X|]$, we have $h_Y(q) > h_X(i_u) = h_Y(f(i_u))$ for $q \in (f(i_u)\,..\,f(\mathsf{twin}_X(i_w))]$ and $h_Y(f(\mathsf{twin}_X(i_w)) + 1) = h_Y(f(\mathsf{twin}_X(i_w) + 1)) = h_X(\mathsf{twin}_X(i_w)) + 1 = h_X(i_u) = h_Y(f(i_u))$. Therefore, $\mathsf{twin}_Y(f(i_u)) = f(\mathsf{twin}_X(i_w))$.

If the parent of u is the virtual root added to $\mathcal{T}_X$ that does not correspond to any parenthesis

of X, then note that $h_X(i_u) > h_X(r)$ for all $r \in (i_u \,..\, |X|)$ and so, $h_Y(f(i_u)) > h_Y(q)$ for all $q \in (f(i_u) \,..\, |X|)$. Thus, $\mathsf{twin}_Y(f(i_u))$ is undefined in this case.

Now, suppose that $X[j]$ is opening and w is either a Type II child of u or the deleted node. Note that $\mathsf{twin}_X(i_w) = \min\{r \geq i_w : h_X(r+1) = h_X(i_w)\}$ and $\mathsf{twin}_X(i_u) = \min\{r \geq i_u : h_X(r+1) = h_X(i_u)\}$. Moreover, $h_X(i_u) = h_X(i_w) - 1$. Consequently, $h_X(r) > h_X(i_u)$ for $r \in (i_u \,..\, \mathsf{twin}_X(i_u)]$, $h_X(r) > h_X(i_u) + 1$ for $r \in (i_w \,..\, \mathsf{twin}_X(i_w)]$, and $h_X(\mathsf{twin}_X(i_w) + 1) = h_X(i_u) + 1$. Since $i_u \leq i_w < j \leq \mathsf{twin}_X(i_w) < \mathsf{twin}_X(i_u)$, as well as $h_Y(f(r)) = h_X(r)$ for $r \in [0 \,..\, j]$ and $h_Y(f(r)) = h_X(r) + 1$ for $r \in (j \,..\, |X|]$, we have $h_Y(q) > h_X(i_u) = h_Y(f(i_u))$ for $q \in (f(i_u) \,..\, f(\mathsf{twin}_X(i_w))]$ and $h_Y(f(\mathsf{twin}_X(i_w)) + 1) = h_Y(f(\mathsf{twin}_X(i_w) + 1)) = h_X(\mathsf{twin}_X(i_w)) - 1 = h_X(i_u) = h_Y(f(i_u))$. Therefore, $\mathsf{twin}_Y(f(i_u)) = f(\mathsf{twin}_X(i_w))$. □

The following two claims specify how Type I and Type III nodes with Type II parents may move from $\mathcal{T}_X$ to $\mathcal{T}_Y$. We also consider the children of the deleted node (if $X[j]$ is opening); all these children are of Type III.

Claim 4.3.14. *If $u \in \mathcal{T}_X$ is a Type I node with a Type II parent w, then $f(w)$ is the parent of $f(u)$.*

Proof. Note that since the heights of opening parentheses corresponding to Type I and Type II nodes do not change, the parent-child relationships between these nodes also do not change. □

Claim 4.3.15. *Consider a Type III node $u \in \mathcal{T}_X$ with a Type II or deleted parent w. If $X[j]$ is closing, then the parent of $f(u)$ is $f(w')$, where w' is the Type II child of w.*[7] *If $X[j]$ is opening, then the parent of $f(u)$ is $f(w')$, where w' is the parent of w.*

Proof. Now, suppose that $X[j]$ is closing. Observe that the heights of opening parentheses corresponding to Type III nodes increase by one whereas the heights of opening parentheses corre-

[7] If w does not have a Type II child, then $\mathsf{twin}_X(i_w) = j$ and all children of w are of Type I.

sponding to Type II nodes stay the same. If a Type III node u has a Type II parent w and w' is the type II child of w, then $h_X(i_u) = h_X(i_w) + 1 = h_X(i_{w'})$ and $h_Y(f(i_u)) = h_X(i_u) + 1 = h_X(i_{w'}) + 1 = h_Y(f(i_{w'})) + 1$. Furthermore, $\mathsf{twin}_X(i_{w'}) = \min\{r \geq i_{w'} : h_X(r+1) = h_X(i_{w'})\}$ and $\mathsf{twin}_X(i_w) = \min\{r \geq i_w : h_X(r+1) = h_X(i_w)\}$. For all $q \in (i_{w'}, i_u)$, we have that $h_Y(q) = h_X(q) + 1 > h_X(i_w) = h_X(i_u) - 2 = h_Y(f(i_u)) - 1$. Thus, $i_{w'}$ is the maximum index of a parenthesis with $i_{w'} < i_j$ and $h_Y(f(i_{w'})) = h_Y(f(i_u)) - 1$.

The analysis is analogous if $X[j]$ is opening: then, the heights of opening parentheses corresponding to Type III nodes decrease instead. If w is a child of the root of $\mathcal{T}_X$ that does not correspond to any parenthesis in X, then there would have no opening parenthesis that satisfies the parent condition for $f(u)$ and so, in this case, $f(u)$ becomes a child of the root.

□

Next, we discuss how an algorithm can identify the changes to the heavy paths in $\mathcal{T}_X$. For the following analysis, define $k_x = \lfloor \lg |\mathcal{T}_X(x)| \rfloor$ and $k'_x = \lfloor \lg |\mathcal{T}_Y(f(x))| \rfloor$ for a node $x \in \mathcal{T}_X$. The following claim implies that a node $x \in \mathcal{T}_X$ needs to change its heavy path only if $k'_x \neq k_x$.

Claim 4.3.16. *Let $u, v \in \mathcal{T}_X$ be nodes such that $k_u = k'_u = k'_v = k_v$. Then, u is an ancestor of v if and only if $f(u)$ is an ancestor of $f(v)$.*

Proof. The characterization of the parent-child relation specified within the claims above leaves very few cases where the ancestor-descendant relation does not need to be preserved by f. If $X[j]$ is closing, this is only possible when u is a Type II node and v is a descendant of its Type III sibling, in which case $f(v)$ is a descendant of $f(u)$. In that case, however, $\mathcal{T}_Y(f(u))$ contains the image under f of all the nodes in $\mathcal{T}_X(u)$ and $\mathcal{T}_X(v)$, so $|\mathcal{T}_Y(f(u))| \geq |\mathcal{T}_X(u)| + |\mathcal{T}_X(v)| \geq 2^{k_u} + 2^{k_v} \geq 2^{k_u+1}$, so $k'_u \geq k_u + 1$ contradicts our assumption. If $X[j]$ is closing, this is only possible when u is a Type

II node and v is a descendant of its Type III child, in which case $f(v)$ is a descendant of a sibling of $f(u)$. In that case, however, $\mathcal{T}_X(u)$ contains the pre-image under f of all the nodes in $\mathcal{T}_Y(f(u))$ and $\mathcal{T}_Y(f(v))$, so $|\mathcal{T}_X(u)| \geq |\mathcal{T}_Y(f(u))| + |\mathcal{T}_Y(f(v))| \geq 2^{k'_u} + 2^{k'_v} \geq 2^{k'_u+1}$, so $k_u > k'_u + 1$ contradicts our assumption. □

Thus, it suffices to iterate over all nodes $x \in \mathcal{T}_X$ with $k'_x \neq k_x$ and move them from one heavy path to another. By Claim 4.3.11, we only have to concern ourselves with Type II nodes (and the deleted node). We use the following claim to characterize them.

Claim 4.3.17. *Consider a Type II node v with parent u. If $X[j]$ is closing, then $k_v \leq k'_v \leq k_u \leq k'_u$. If $X[j]$ is opening, then $k'_v \leq k_v \leq k'_u \leq k_u$.*

Proof. Suppose that $X[j]$ is opening. By Claims 4.3.14 and 4.3.15, we have $\mathcal{T}_Y(f(v)) = f(\mathcal{T}_X(v) \cup \{w \in \mathcal{T}_X : w \text{ is a descendant of a Type III sibling of } v\})$. Thus, $|\mathcal{T}_X(v)| \leq |\mathcal{T}_Y(f(v))|$. The subtree $\mathcal{T}_X(u)$ additionally contains u and the descendants of Type III siblings of v, so $|\mathcal{T}_Y(f(v))| < |\mathcal{T}_X(u)|$. The inequality $|\mathcal{T}_X(u)| \leq |\mathcal{T}_Y(f(u))|$ is analogous to its counterpart for v.

Suppose that $X[j]$ is closing. By Claims 4.3.14 and 4.3.15, we have $\mathcal{T}_Y(f(u)) = f(\mathcal{T}_X(u) \setminus \{w \in \mathcal{T}_X : w \text{ is a descendant of a Type III sibling of } v \text{ or represents } X[j]\})$. Thus, $|\mathcal{T}_Y(f(u))| < |\mathcal{T}_X(u)|$. The subtree $\mathcal{T}_X(v)$ additionally contains the node w representing $X[j]$ but misses u and the descendants of Type I siblings of v, so $|\mathcal{T}_X(v)| \leq |\mathcal{T}_Y(f(u))|$. The inequality $|\mathcal{T}_Y(f(v))| < |\mathcal{T}_X(v)|$ is analogous to its counterpart for u. □

We are ready to characterize the affected nodes. First, suppose that $X[j]$ is closing. By Claim 4.3.17, $k_v \leq k'_v \leq k_u \leq k'_u$ holds for every Type II node v with parent u. If $k_v \neq k'_v$, then $k_v < k'_v \leq k_u$, so v is a light node. Consequently, our algorithm inspects the light Type II nodes v (in the bottom-up order) and filters out those for which $k_v = k'_v$. We remove every remaining node

v from its original heavy path and insert its image $f(v)$ into the updated heavy path. In order to identify the latter path, we use Lemma 4.3.10 to find the heavy child of $f(v)$. If there is none but the parent u of v satisfies $k'_v = k_u = k'_u$, then we insert $f(v)$ to the heavy path of u. For all other nodes v, we have $k'_v < k'_u$, so $f(v)$ is light and belongs to a singleton heavy path.

Next, suppose that $X[j]$ is opening. In this case, the deleted node containing $X[j]$ needs to be removed from its heavy path. Let us focus on the other nodes. By Claim 4.3.17, we have that $k'_v \leq k_v \leq k'_u \leq k_u$ for every Type II node v with parent u. If $k_u \neq k'_u$, then $k'_v \leq k_v \leq k'_u < k_u$, so v is a light node. Consequently, our algorithm inspects the parents u of the light Type II nodes (as well as the parent of the deleted node, all in the bottom-up order) and filters out nodes u with $k_u = k'_u$. We remove every remaining node u from its original heavy path and insert its image $f(u)$ into the updated heavy path. In order to identify the latter path, we use Lemma 4.3.10 to find the heavy child of $f(u)$. If there is none, then the parent w of u satisfies $k'_u < k_u \leq k'_w$, so $f(u)$ is light and belongs to a singleton heavy path.

In all cases, by Observation 4.3.2, there are $\mathcal{O}(\log n)$ Type II nodes to inspect, and they can be found by iterating over the heavy paths containing the node corresponding to $X[j]$ and its ancestors in $\mathcal{T}_X$. All the changes described above can be implemented in $\mathcal{O}(\log^4 n)$ time since we utilize Lemma 4.3.10 $\mathcal{O}(\log n)$ times. Thus in all cases, we may perform the changes to the heavy path set in $\mathcal{O}(\log^4 n)$ time and there are at most $\mathcal{O}(\log n)$ such changes. This gives us our total time of $\mathcal{O}(\log^5 n)$.

Note that when deleting an opening parenthesis, we have to delete the node corresponding to that parenthesis. We may straightforwardly remove this node's parenthesis from its heavy path in $\mathcal{T}_X$ to fix the heavy path set. Additionally, a closing parenthesis previously matched at the root node may now no longer have a match. To fix this, we add another dummy opening parenthesis

and a new node to the root of $\mathcal{T}_Y$.

We note that a dynamic insertion of an opening parenthesis is symmetric in the case of deleting a closing parenthesis with one difference. The algorithm must create a new node for the new opening parenthesis, which we may add to the set of heavy paths using Lemma 4.3.10 in $\mathcal{O}(\log^3 n)$ time. We also may perform an additional clean-up step for both a dynamic insertion of an opening parenthesis and dynamic deletion of a closing parenthesis, since it is possible that a dummy opening parenthesis no longer has a matching closing parenthesis. This must be the root node of $\mathcal{T}_Y$, and so we may directly check if this is the case in the root node and delete it if so. Similarly, a dynamic insertion of a closing parenthesis is symmetric to the deletion of an opening parenthesis except we do not have to add any node for the inserted parenthesis. As before, we may need to add a dummy node to the root of the tree which can be done in $\mathcal{O}(\log^3 n)$ time using Lemma 4.3.10. □

4.4 Static and Dynamic Tree to String Edit Distance

In this section, we discuss the related problem of *tree edit distance* and provide the first approximation algorithm for tree edit distance in the dynamic setting. Given two labeled trees as input, the goal of tree edit distance is to find the minimum number of edits needed to transform one tree into the other. To do so, we utilize a novel reduction technique, which converts an instance of tree edit distance to an instance of string edit distance. Via a Euler tour of the trees and a heavy-light decomposition, the algorithm builds a string with special labeling where subtrees of a node are embedded into its label. In total, we are able to give an $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$-approximation algorithm in the dynamic setting for tree edit distance with sub-polynomial update time. We begin by

defining the reduction and discussing structural properties that arise in the reduction, and then in Subsection 4.4.3, we show how to find a tree alignment using a string alignment obtained via our reduction. We analyze the obtained tree alignment in Subsections 4.4.4 and 4.4.5, and finally, in Subsection 4.4.6, we show how to maintain our reduction in the dynamic setting.

4.4.1 $\tilde{\mathcal{O}}(\sqrt{n})$-approximate Reduction to String Edit Distance

In order to utilize string edit distance to compute an approximation to the tree edit distance of two given forests, we utilize a modification of the parenthesis representation based on the heavy-light decomposition of $\mathcal{F}$, which we denote $\mathcal{P}(\mathcal{F})$ for a forest $\mathcal{F}$.

Recall that $\mathcal{F}(v)$ is the subtree of $\mathcal{F}$ rooted at vertex v. We define a new subtree of $\mathcal{F}$ rooted at vertex v denoted by $\mathcal{F}'(v)$ as follows: if w is a heavy child of v, replace w and the entire subtree rooted at w with a single node having a label #, that is not used by any other node in $\mathcal{F}$. We call $\mathcal{F}'(v)$ the *light* subtree of v. We define $\mathcal{P}(\mathcal{F})$ as follows:

Definition 4.4.1. Let $\mathcal{F}$ be comprised of trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$ with root nodes $r_1, r_2, \ldots, r_m$, respectively, where the trees are in order from left to right such that $\mathcal{T}_1$ is the leftmost tree in $\mathcal{F}$. Let $\mathsf{h}(v)$ for $v \in \mathcal{F}$ be 1 if v is a heavy node in $\mathcal{F}$ and 0 if v is a light node in $\mathcal{F}$, and let $\mathsf{hd}(v) = \lfloor \lg |\mathcal{F}(v)| \rfloor$ denote the heavy depth of v in forest $\mathcal{F}$.

$$\mathcal{P}(\mathcal{T}_i, \mathsf{h}, \mathsf{hd}) = \begin{cases} (_{\mathcal{F}'(r_i)} \cdot \mathcal{P}(\mathcal{T}_i \setminus r_i, \mathsf{h}, \mathsf{hd}) \cdot)_{\mathcal{F}'(r_i)} & \text{if } h(r_i) = 1, \\ (_{\mathcal{F}'(r_i),\mathsf{hd}(r_i)} \cdot \mathcal{P}(\mathcal{T}_i \setminus r_i, \mathsf{h}, \mathsf{hd}) \cdot)_{\mathcal{F}'(r_i),\mathsf{hd}(r_i)} & \text{if } h(r_i) = 0. \end{cases}$$

Then, $\mathcal{P}(\mathcal{F}) = \mathcal{P}(\mathcal{F}, \mathsf{h}, \mathsf{hd}) = \bigodot_{i=1}^{m} \mathcal{P}(\mathcal{T}_i, \mathsf{h}, \mathsf{hd})$.

Moving forward, for notational simplicity we replace $o_{\mathcal{P}}(u)$ with $o(u)$ and $c_{\mathcal{P}}(u)$ with $c(u)$.

In the above definition, the entire light sub-tree of a node is embedded in the label of the parentheses corresponding to that node. In addition, parentheses corresponding to light nodes have the heavy depth of their corresponding node embedded in the label as well. From our previous observations of the heavy-light decomposition, we know there are at most $\mathcal{O}(\log n)$ light node ancestors of any node in forest $\mathcal{F}$ or $\mathcal{G}$. Furthermore, each node u appears in the label of $o(v)$ and $c(v)$ for a node v precisely when u lies in the light subtree of v. Therefore, any optimal tree alignment on the parenthesis representation $\mathsf{P}(\mathcal{F}), \mathsf{P}(\mathcal{G})$ has a corresponding optimal tree alignment on the modified parentheses representation $\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G})$ with at most an additional multiplicative $\mathcal{O}(\log n)$ number of edits. We now briefly prove this formally.

Observation 4.4.2. *Given a forest $\mathcal{F}$ and any node u in $\mathcal{F}$, u appears in the label of at most $\mathcal{O}(\log n)$ parentheses in $\mathcal{P}(\mathcal{F})$.*

Proof. By Definition 4.4.1, any ancestor node a of u such that u is a descendant of a light node child of a will contain u in the label of $o(a), c(a)$. Since there are at most $\mathcal{O}(\log n)$ light node ancestors of u by Observation 4.2.7, then there are at most $\mathcal{O}(\log n)$ parentheses in $\mathcal{P}(\mathcal{F})$ whose labels contain u. □

Observation 4.4.3. *Given forests $\mathcal{F}, \mathcal{G}$ and alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, if $u \simeq_{\mathcal{A}} v$ for nodes u and v in F and G, respectively, then u and v must either both be light nodes or both be heavy nodes.*

Proof. Note that in the construction of $\mathcal{P}(F)$, for a light node u we include $\lfloor \lg |\mathcal{F}'(u)| \rfloor$ in the label of $\mathcal{P}(\mathcal{F})[o(u)]$ and $\mathcal{P}(\mathcal{F})[c(u)]$. On the other hand, a heavy node v in $\mathcal{G}$ does not include

$\lfloor \lg |\mathcal{G}'(v)| \rfloor$. Thus, $\mathcal{P}(\mathcal{F})[o(u)] \neq \mathcal{P}(\mathcal{G})[o(v)]$ nor $\mathcal{P}(\mathcal{F})[c(u)] \neq \mathcal{P}(\mathcal{G})[c(v)]$, and so $\mathcal{A}$ cannot match the parentheses corresponding to u with the parentheses corresponding to v. □

Lemma 4.4.4. *Given forests* $\mathcal{F}, \mathcal{G}$, $\mathsf{ed}(\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G})) \leq \mathcal{O}(\log(n)) \cdot \mathsf{ted}(\mathcal{F}, \mathcal{G})$

Proof. Consider an optimal sequence of edits S which transforms $\mathcal{F}$ into $\mathcal{G}$. Such a sequence defines a corresponding optimal sequence of edits S on $\mathsf{P}(\mathcal{F}), \mathsf{P}(\mathcal{G})$ (note this is the parenthesis representation of $\mathcal{F}$ and $\mathcal{G}$, not to be confused with the modified version $\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G})$) where each node deletion, insertion, and substitution on a node u in $\mathcal{F}$ or $\mathcal{G}$ corresponds to a deletion or substitution, respectively, of the corresponding parentheses $o(u), c(u)$ in $\mathsf{P}(\mathcal{F})$ and $\mathsf{P}(\mathcal{G})$ such that $\mathsf{ed}(\mathsf{P}(\mathcal{F}), \mathsf{P}(\mathcal{G})) = 2|S| = 2 \cdot \mathsf{ted}(\mathcal{F}, \mathcal{G})$. Let $\mathcal{F}_i$ be the forest after the first i edits of S on $\mathcal{F}$. Note that $\mathsf{ed}(\mathsf{P}(\mathcal{F}), \mathsf{P}(\mathcal{G})) = \sum_{i=0}^{|S|-1} \mathsf{ed}(\mathsf{P}(\mathcal{F}_i), \mathsf{P}(\mathcal{F}_{i+1}))$. Now, we define a sequence of edits S' which transforms $\mathcal{P}(F)$ into $\mathcal{P}(G)$ (the modified representations). We note that $\mathsf{ed}(\mathcal{P}(F_i), \mathcal{P}(F_{i+1})) \leq \mathcal{O}(\log n) \cdot \mathsf{ed}(P(F_i), P(G_i))$ since every node of $\mathcal{F}$ or $\mathcal{G}$ is contained in the label of at most $\mathcal{O}(\log n)$ parentheses by Observation 4.4.2, which means any edit to a node may require an additional $\mathcal{O}(\log n)$ substitutions to update these parentheses. So, we need at most an additional $\mathcal{O}(\log n)$ label substitutions per edit of S on $\mathcal{P}(\mathcal{F}_i)$ to reach $\mathcal{P}(\mathcal{F}_{i+1})$. Thus, we have the following inequalities

$$\begin{aligned}\mathsf{ed}(\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G})) &\leq \sum_{i=0}^{|S|-1} \mathsf{ed}(\mathcal{P}(\mathcal{F}_i), \mathcal{P}(\mathcal{F}_{i+1})) \leq \sum_{i=0}^{|S|-1} \mathcal{O}(\log n) \cdot \mathsf{ed}(\mathsf{P}(\mathcal{F}_i), \mathsf{P}(\mathcal{F}_{i+1})) \\ &= \mathcal{O}(\log n) \sum_{i=0}^{|S|-1} \mathsf{ed}(\mathsf{P}(\mathcal{F}_i), \mathsf{P}(\mathcal{F}_{i+1})) = \mathcal{O}(\log n) \cdot \mathsf{ted}(\mathcal{F}, \mathcal{G}).\end{aligned}$$

□

Figure 4.6: Example alignments between two parenthesis representations of forests $\mathcal{F}$ and $\mathcal{G}$. Parentheses with lines connecting them are aligned, and parentheses with no lines connecting them to another are deleted. (left) The orange pair of parentheses correspond to a partially deleted node. The blue pair of parentheses correspond to a single-branch mis-aligned node. The green pair of parentheses correspond to a multi-branch mis-aligned node. (right) The shown alignment contains a chain of nodes in $\mathcal{F}$ and $\mathcal{G}$ corresponding to the blue parentheses.

4.4.2 String to Tree Alignment

Given forests $\mathcal{F}, \mathcal{G}$ along with their modified parenthesis representations $\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G})$, the primary issue with using a string alignment on the modified representations to approximate $\mathsf{ted}(\mathcal{F}, \mathcal{G})$ is that an optimal string alignment between $\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G})$ may not adhere to the tree alignment constraints as stated in Definition 4.2.6.

Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, we say that a node u in $\mathcal{F}$ (or in $\mathcal{G}$) is *tree-aligned* if $\mathcal{A}$ aligns both $o(u)$ and $c(u)$ to $o(v)$ and $c(v)$, respectively, for some node $v \in G$. Otherwise, u is *mis-aligned*. Mis-aligned nodes are precisely those which violate the tree alignment constraints. There are three categories of mis-aligned nodes we consider in this work. A node $u \in \mathcal{F}$ is *partially deleted* if $\mathcal{A}$ aligns only one of $o(u)$ or $c(u)$ while $\mathcal{A}$ deletes the remaining twin. Node u is a *single-branch* mis-aligned node if $o(u) \sim_{\mathcal{A}} o(v_1)$ and $c(u) \sim_{\mathcal{A}} c(v_2)$ for $v_1, v_2 \in \mathcal{G}$ such that $v_1 \neq v_2$ and v_1 and v_2 are on the same branch of $\mathcal{G}$, e.g., $o(v_1) < o(v_2) < c(v_2) < c(v_1)$. Node u is a *multi-branch* mis-aligned node if $o(u) \sim_{\mathcal{A}} o(v_1)$ and $c(u) \sim_{\mathcal{A}} c(v_2)$ for $v_1, v_2 \in \mathcal{G}$ such that v_1 and v_2 are not on the same branch of $\mathcal{G}$, e.g., $o(v_1) < c(v_1) < o(v_2) < c(v_2)$. See Figure 4.6 for examples of each type of mis-aligned node.

One common structure in string alignments that violate the tree alignment constraints in-

volves a sequence of single-branch mis-aligned nodes such that each node in the sequence has only one of its corresponding parenthesis aligned to the previous node in the sequence and its other corresponding parentheses aligned to the next node in the sequence. We define them formally as follows.

Definition 4.4.5 (Chains)**.** Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, a *chain* C is a maximal-length sequence of at least 3 nodes alternating between $\mathcal{F}$ and $\mathcal{G}$ where all nodes in C from $\mathcal{F}$ lie on a single branch and all nodes in C from $\mathcal{G}$ lie on a single branch such that $o(C[i]) < o(C[i+2]) < c(C[i+2]) < c(C[i])$ for all $i \in [0, |C| - 2]$ and either

1. For all even i such that $C[i] \in \mathcal{F}$, if $i < |C| - 1$, $o(C[i]) \simeq_{\mathcal{A}} o(C[i+1])$ and if $i > 0$, $c(C[i]) \simeq_{\mathcal{A}} c(C[i-1])$. We call this an *opening* chain. OR
2. For all even i such that $C[i] \in \mathcal{F}$, if $i < |C| - 1$, $c(C[i]) \simeq_{\mathcal{A}} c(C[i+1])$ and if $i > 0$, $o(C[i]) \simeq_{\mathcal{A}} o(C[i-1])$. We call this a *closing* chain.

Note in the above definition, we require that the parentheses in each case are not just aligned, but actually match (i.e., their labels are equal). Figure 4.6 gives a simple example of a chain. As a result of requiring matches, an important property of a chain is that the light subtree of every node in a chain is equal.

Observation 4.4.6. *For any $0 \leq i, j < |C|$ for a chain C of deletion-only alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ where $C[i]$ is a node in $\mathcal{F}$ and $C[j]$ is a node in $\mathcal{G}$, $\mathcal{F}'(C[i]) = \mathcal{G}'(C[j])$. Similarly, if $C[i]$ is a node in $\mathcal{F}$ and $C[j]$ is a node in $\mathcal{F}$, $\mathcal{F}'(C[i]) = \mathcal{F}'(C[j])$.*

Proof. This observation follows from induction. Consider the first case of the observation statement, when $C[i]$ is a node in $\mathcal{F}$ and $C[j]$ is a node in $\mathcal{G}$. Without loss of generality let $i < j$ and $C[0] \in \mathcal{F}, C[1] \in \mathcal{G}$.

For the base case, by Definition 4.4.5, $o(C[0]) \simeq_{\mathcal{A}} o(C[1])$ or $c(C[0]) \simeq_{\mathcal{A}} c(C[1])$ and so, $\mathcal{P}(\mathcal{F})[o(C[0])] = \mathcal{P}(\mathcal{G})[o(C[1])]$ or $\mathcal{P}(\mathcal{F})[c(C[0])] = \mathcal{P}(\mathcal{G})[c(C[1])]$. Thus,by the definition of $\mathcal{P}(\mathcal{F})$, Definition 4.4.1, it must be that $\mathcal{F}'(C[0]) = \mathcal{G}'(C[1])$ since the light subtree of a node $u \in \mathcal{F}$ is embedded in $\mathcal{P}(\mathcal{F})[o(u)]$ and $\mathcal{P}(\mathcal{F})[c(u)]$.

Now assume the statement is true for nodes in the chain up to index $i-1$. By Definition 4.4.1, the label of a node $u \in \mathcal{P}(\mathcal{F})$ ($\mathcal{P}(\mathcal{G})$) contains $\mathcal{F}'(u)$ ($\mathcal{G}'(u)$). Furthermore, by Definition 4.4.5, $o(C[i]) \simeq_{\mathcal{A}} o(C[i+1])$ or $c(C[i]) \simeq_{\mathcal{A}} c(C[i+1])$ and so, $\mathcal{F}'(C[i]) = \mathcal{G}'(C[i+1])$. Similarly, $\mathcal{G}'(C[i+1]) = \mathcal{F}'(C[i+2])$. Thus, by induction, $\mathcal{F}'(C[i]) = \mathcal{G}'(C[j])$.

The remaining case when $C[i], C[j] \in \mathcal{F}$ follows analogously. □

As a result of the above observation, mis-aligned nodes may lead to a significant gap between the string edit distance and tree edit distance of $\mathsf{P}(\mathcal{F})$ and $\mathsf{P}(\mathcal{G})$. Chains allow for a fluidity in string alignments that often help minimize the number of edits required at the cost of violating tree alignment constraints. See Figure 4.7 for an example of a pair of forests whose parentheses representations have a string edit distance of $\mathcal{O}(1)$ but tree edit distance of $\mathcal{O}(\sqrt{n})$. By identifying and fixing mis-aligned nodes in chains, we show how to construct a tree alignment from any string alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ at the cost of an additional multiplicative factor of $\tilde{\mathcal{O}}(\sqrt{n})$ edits to match the gap up to logarithmic factors.

4.4.3 String to Tree Alignment Construction

4.4.3.1 Tree-Aligning a Mis-aligned Heavy Node

Given a minimum-cost deletion-only string alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, we first discuss how to fix a single mis-aligned heavy node of $\mathcal{F}$. There are two main cases: first, if a mis-aligned

Figure 4.7: Pictured above are a string alignment (left) and a tree alignment (right) for the parentheses representations of a forest $\mathcal{F}$ and $\mathcal{G}$. Parentheses with lines connecting them are aligned, and parentheses with no lines connecting them to another are deleted. X represents a substring of length $\sqrt{n}/4$. Using a chain, an optimal string alignment only requires two deletions of parentheses highlighted in red. On the other hand, an optimal tree alignment must delete two copies of substring X to satisfy the constraints of Definition 4.2.6. The difference in the optimal string edit distance and tree edit distance is $\mathcal{O}(|X| - 2) = \mathcal{O}(\sqrt{n})$.

heavy node u is part of a chain, we aim to modify $\mathcal{A}$ to match both $o(u)$ and $c(u)$ to the next node $v \in \mathcal{G}$ in the chain sequence, so that u and v are tree-aligned. Furthermore, we modify $\mathcal{A}$ to match the light subtrees of u and v together as well. Recall one of the main constraints of an alignment is that for alignment $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$, $(x_{t+1}, y_{t+1}) \in (x_t + 1, y_t + 1), (x_t + 1, y_t), (x_t, y_t + 1)$ for all $t \in [0, m)$. After matching the light subtrees of u and v, we remove any pairs that would violate this constraint and add deletions of any affected parentheses outside of these light subtrees. Second, if a mis-aligned heavy node u is not part of a chain, we simply delete $o(u)$, $c(u)$ and the parentheses in $\mathcal{P}(\mathcal{G})$, if any, that were previously matched to them by $\mathcal{A}$. We now define the two cases formally. For two nodes v, v' in a forest $\mathcal{F}$, we let $w(v, v')$ denote $|o(v) - o(v')| + |c(v) - c(v')|$. Let u be a mis-aligned node in a maximal-length heavy path H of $\mathcal{F}$ where recall a heavy path is a set of heavy nodes in $\mathcal{F}$ that form a connected path in $\mathcal{F}$. We refer to the following subroutine as $\mathsf{TreeAlign}(\mathcal{A}, u)$, which we define for a only a single node u and utilize iteratively, discussed subsequently in Subsection 4.4.3.2.

$\mathsf{TreeAlign}(\mathcal{A}, u)$:

1. If there exists a chain C such that

- $C[k] = u$ for some $k \in [0\,..\,|C| - 5]$,
- $w(C[k], C[k+2]) < \sqrt{n}$,
- $w(C[k+2], C[k+4]) < \sqrt{n}$,
- $C[k+2], C[k+4] \in H$

then modify $\mathcal{A}$ to match the light subtree of u with the light subtree of $C[k+1]$.

2. Otherwise, modify $\mathcal{A}$ to delete node u.

In the above routine, we use two short-hand phrases "match the light subtree of u with the light subtree of $C[k+1]$" and "delete node u" when discussing modifications to $\mathcal{A}$. We now rigorously define these two steps. Recall an alignment matches parentheses of strings X and y at indices i, j, respectively, when it contains pairs $(i, j), (i+1, j+1)$ and deletes a parenthesis at index i in X when it contains pairs $(i, j), (i+1, j)$ with a symmetric case for deletion in Y.

Delete node u: To "*delete a node u*" for a node $u \in \mathcal{F}$, any matches of $o(u)$ and $c(u)$ are replaced with deletions in alignment $\mathcal{A}$. For example, if $o(u) \simeq_{\mathcal{A}} o(v)$ for $u \in \mathcal{F}, v \in G$ in a deletion-only alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, then we replace pairs $(o(u), o(v)), (o(u)+1, o(v)+1)$ with pairs $(o(u), o(v)), (o(u)+1, o(v)), (o(u)+1, o(v)+1)$. Note that this deletes both $o(u)$ and $o(v)$. If instead $o(u)$ is not matched, we make no changes to $\mathcal{A}$. We do the analogous modifications of $\mathcal{A}$ for $c(u)$.

Match light subtrees of u and v: To "*match the light subtree of u with the light subtree of v*" for nodes $u \in \mathcal{F}, v \in G$ such that u and v are consecutive nodes in a chain C of $\mathcal{A}$, $\mathcal{A}$ is modified to match all parentheses in $\mathcal{P}(\mathcal{F}')(u)$ to those in $\mathcal{P}(\mathcal{G}')(v)$ and delete any parentheses of $\mathcal{F}$ or $\mathcal{G}$ who no longer have a valid match in $\mathcal{A}$ according to the alignment definition, Definition 4.2.4. This modification is much more involved than deleting a node. First, we observe that the constraints of

case 1 in the above routine require that u is not the last node in heavy path H since $C[k+2] \in H$, and so u has some heavy child u'. Furthermore, since $v = C[k+1]$, by Observation 4.4.6, $\mathcal{P}(\mathcal{G}')(v) = \mathcal{P}(\mathcal{F}')(u)$ and so, v must have a heavy child v'. Every chain must either be an opening chain or a closing chain such that either $o(u) \simeq_{\mathcal{A}} o(v)$ or $c(u) \simeq_{\mathcal{A}} c(v)$, respectively. Without loss of generality, we assume that $c(u) \simeq_{\mathcal{A}} c(v)$, and so pairs $(c(u), c(v)), (c(u)+1, c(v)+1)$ are in $\mathcal{A}$.

We begin our modifications of $\mathcal{A}$ to match $\mathcal{P}(\mathcal{F})(c(u'), c(u)]$ to $\mathcal{P}(\mathcal{G})(c(v'), c(v)]$. Figure 4.8 provides a depiction of the changes we will make to $\mathcal{A}$ for these substrings, which we describe in detail as follows. In order to maintain alignment constraints as per Definition 4.2.4, before we can add anything, we first remove any previous pairs of $\mathcal{A}$ containing an index corresponding to a parenthesis in $\mathcal{P}(\mathcal{F})(c(u'), c(u)]$ or $\mathcal{P}(\mathcal{G})(c(v'), c(v)]$. Let $(i_{\mathcal{F}}, i_{\mathcal{G}}) \in A$ be the rightmost pair in $\mathcal{A}$ such that $i_{\mathcal{F}} \leq c(u')$ and $i_{\mathcal{G}} \leq c(v')$. By the definition of alignments, it must be the case that at least one of $i_{\mathcal{F}} = c(u')$ or $i_{\mathcal{G}} = c(v')$ is true; without loss of generality, let $i_{\mathcal{F}} = c(u')$. We remove all pairs of $\mathcal{A}$ between pairs $(c(u'), i_{\mathcal{G}})$ and $(c(u)+1, c(v)+1)$, exclusive. We insert a matching sequence $(c(u'), c(v')), (c(u')+1, c(v')+1), (c(u')+2, c(v')+2), \ldots, (c(u), c(v))$ into $\mathcal{A}$ at the position before pair $(c(u)+1, c(v)+1)$. We then insert a deletion sequence $(c(u'), i_{\mathcal{G}}+1), (c(u'), i_{\mathcal{G}}+2), \ldots, (c(u'), c(v')-1)$ into $\mathcal{A}$ at the position after pair $(c(u'), i_{\mathcal{G}})$. This completes the matching of $\mathcal{P}(\mathcal{F})(c(u'), c(u)]$ to $\mathcal{P}(\mathcal{G})(c(v'), c(v)]$ and deletion of any additional parentheses needed to maintain alignment constraints.

We now finish matching the light subtrees of u and v by modifying $\mathcal{A}$ to match $\mathcal{P}(\mathcal{F})[o(u), o(u'))$ and $\mathcal{P}(\mathcal{G})[o(v), o(v'))$. See Figure 4.9 for a depiction of these modifications to $\mathcal{A}$ that we describe formally as follows. Let $(o(u), i_{\mathcal{G}})$ be the first pair in $\mathcal{A}$ with left index $o(u)$ and let $(i_{\mathcal{F}}, o(v'))$ be the first pair in $\mathcal{A}$ with right index $o(v')$. Observe that $o(v) \simeq_{\mathcal{A}} o(C[k+2])$ with $o(u) < o(C[k+2])$ by our earlier assumption of C. Since $o(u) < o(C[k+2])$, $o(v') > o(v)$

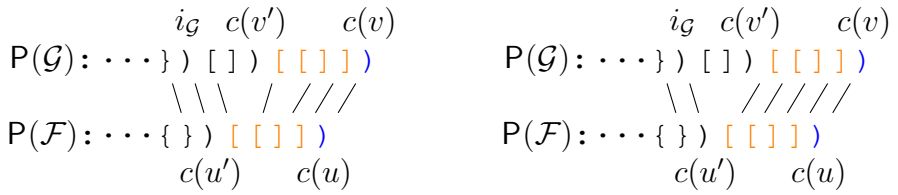


Figure 4.8: Example of an alignment $\mathcal{A}$ (left) and the result of $\mathsf{TreeAlign}(\mathcal{A}, u)$ (right) for a node u in Case 1 of the TreeAlign routine. Parentheses with lines connecting them are aligned, and parentheses with no lines connecting them to another are deleted. For a chain C, $u = C[k]$ corresponds to the rightmost (blue) parenthesis of $\mathcal{P}(\mathcal{F})$ and $v = C[k+1]$ corresponds to the rightmost (blue) parenthesis in $\mathcal{P}(\mathcal{G})$. After $\mathsf{TreeAlign}(\mathcal{A}, u)$, $o(u) \simeq_{\mathcal{A}} o(v)$ and the rest of the light subtrees of u and v, depicted by the orange parentheses, are matched as well. The parentheses previously aligned with the light subtree of u, which correspond to range $[i_{\mathcal{G}}, c(v))$ are now deleted.

and $(o(C[k+2], o(v)) \in \mathcal{A}$, by the non-crossing property of alignments, $i_G \leq o(v)$ and $i_{\mathcal{F}} \geq o(u')$. We remove all pairs of $\mathcal{A}$ between $(o(u), i_{\mathcal{G}})$ and $(i_{\mathcal{F}}, o(v'))$, exclusive. We insert deletion sequence $(o(u), i_{\mathcal{G}} + 1), (o(u), i_{\mathcal{G}} + 2), \ldots, (o(u), o(v))$ into $\mathcal{A}$ at the position after pair $(o(u), i_{\mathcal{G}})$. Next, we insert matching sequence $(o(u) + 1, o(v) + 1), (o(u) + 2, o(v) + 2), \ldots, (o(u'), o(v'))$ into $\mathcal{A}$ at the position after pair $(o(u), o(v))$. Finally, we insert deletion sequence $(o(u') + 1, o(v')), (o(u') + 2, o(v')), \ldots, (i_{\mathcal{F}} - 1, o(v'))$ into $\mathcal{A}$ at the position after pair $(o(u'), o(v'))$. The above steps modify $\mathcal{A}$ to match substrings $\mathcal{P}(\mathcal{F})[o(u), o(u'))$ to $\mathcal{P}(\mathcal{G})[o(v), o(v'))$ while maintaining alignment constraints.

4.4.3.2 Tree-Aligning All Heavy Nodes

We can use the TreeAlign subroutine iteratively to fix all mis-aligned heavy nodes of $\mathcal{A}$ in $\mathcal{F}$. First, let $\mathcal{H} = \{H_1, H_2, \ldots, H_\ell\}$ be the set of all maximal-length heavy paths $H_i = (u^i_1, u^i_2, \ldots, u^i_{m_i})$ in $\mathcal{F}$, which partitions the set of heavy nodes in $\mathcal{F}$. Observe that for a heavy path $H_i \in \mathcal{H}$, any two nodes $u^i_{j_1}, u^i_{j_2} \in H_i$ must have the same heavy depth according to the definition of heavy nodes. We order the paths in $\mathcal{H}$ according to the heavy depths of their nodes

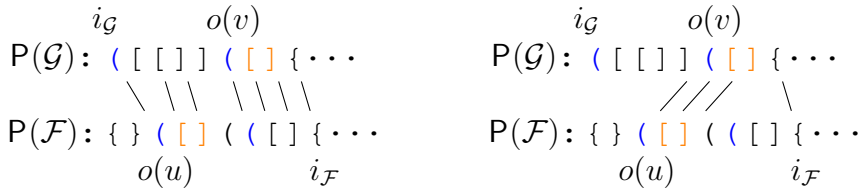


Figure 4.9: Example of an alignment $\mathcal{A}$ (left) and the result of $\mathsf{TreeAlign}(\mathcal{A}, u)$ (right) for a node u in Case 1 of the TreeAlign routine. Parentheses with lines connecting them are aligned, and parentheses with no lines connecting them to another are deleted. For a chain C, $u = C[k]$ corresponds to the leftmost (blue) parenthesis of $\mathcal{P}(\mathcal{F})$ and $v = C[k+1]$ corresponds to the rightmost (blue) parenthesis in $\mathcal{P}(\mathcal{G})$. After $\mathsf{TreeAlign}(\mathcal{A}, u)$, $o(u) \simeq_{\mathcal{A}} o(v)$ and the rest of the light subtrees of u and v, depicted by the orange parentheses, are matched as well. The parentheses previously aligned with the light subtrees of u and v, which correspond to ranges $[o(u'), i_{\mathcal{F}})$ and $[i_{\mathcal{G}}, o(v))$ are now deleted (where u' is the heavy child of u, corresponding to the black opening parenthesis ' (' in $\mathcal{F}$).

from smallest heavy depth to largest heavy depth. We begin with H_1, the heavy path with smallest heavy depth, and consider each node of H_1 one at a time. Let $H_1 = (u_1^1, u_2^1, \ldots, u_{m_1}^1)$ be ordered from highest level node to lowest level node in $\mathcal{F}$, i.e., u_i^1 is an ancestor of all u_j^1 for $i \leq j$. For u_i^1 for $i = 1$ to m_1, if u_i^1 is mis-aligned, we call $\mathsf{TreeAlign}(u_i^1, \mathcal{A})$. Otherwise, we make no changes to $\mathcal{A}$ and move on to the next heavy node. After handling H_1, we perform TreeAlign on the heavy nodes of H_2, then H_3, etc., until all heavy nodes have been handled.

4.4.3.3 Aligning Light Nodes

We have not yet dealt with mis-aligned light nodes. It turns out any mis-aligned light node can be deleted without increasing the number of edits by more than a poly-logarithmic factor. Therefore, for any mis-aligned light node $u \in \mathcal{F}$, we modify $\mathcal{A}$ to delete node u. The order of these modifications does not matter, so long as we do this for all such mis-aligned light nodes. We let $\mathsf{TreeAlign}(\mathcal{A})$ denote the alignment created after performing $\mathsf{TreeAlign}(\mathcal{A}, u)$ on all heavy nodes $u \in \mathcal{F}$ and deleting all mis-aligned light nodes as described above. Now, we give a short

proof that TreeAlign($\mathcal{A}$) is indeed a tree alignment followed by a case analysis proving that the new alignment only increases the number of deletions by a factor of $\tilde{\mathcal{O}}(\sqrt{n})$.

4.4.4 Tree Alignment Constraint Analysis

First, we show that for alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ and node $u \in \mathcal{F}$, TreeAlign$(\mathcal{A}, u)$ does in fact yield a sequence of pairs satisfying the alignment definition.

Lemma 4.4.7. *Given a deletion-only alignment $\mathcal{A}$ and a mis-aligned node $u \in \mathcal{F}$, sequence $\mathcal{A}$ after performing* TreeAlign(A, u) *is an alignment.*

Proof. By Definition 4.2.4, we must make sure that $(0, 0), (|\mathcal{P}(\mathcal{F})|, |\mathcal{P}(\mathcal{G})| \in \mathcal{A}$ and $(x_{t_1}, y_{t+1}) \in (x_t + 1, y_t + 1), (x_t + 1, y_t), (x_t, y_t + 1)$ for all $t \in [0, m)$ where $\mathcal{A} = (x_t, y_t)_{t=0}^{m}$ after modifying $\mathcal{A}$. Note that in both cases of TreeAlign$(\mathcal{A}, u)$, we never replace the first or last pair of $\mathcal{A}$, so we can focus on the third constraint as the first two are trivially true. It is easy to see that both cases of TreeAlign follow this constraint, but in the following, we show this explicitly for the sake of completion. If u is handled in Case 2 of TreeAlign, it is deleted. In this case pairs $(o(u), o(v)), (o(u)+1, j+1)$ are replaced with $(o(u), o(v)), (o(u)+1, o(v)), (o(u)+1, o(v)+1)$ or $(c(u), c(v)), (c(u)+1, c(v)+1)$ are replaced with $(c(u), c(v)), (c(u)+1, c(v)), (c(u)+1, c(v)+1)$. The only change to $\mathcal{A}$ are new pairs $(o(u) + 1, o(v))$ and $(c(u) + 1, c(v))$, which both satisfy the alignment constraint.

If u is handled in Case 1 of TreeAlign, its light subtree is matched to the light subtree of a node v where $C[k] = u, C[k + 1] = v$ for some chain C. As in case 1, we denote u' as the heavy child of u and v' as the heavy child of v. We assume without loss of generality that $c(u) \simeq_{\mathcal{A}} c(v)$ and that the rightmost pair $(i_{\mathcal{F}}, i_{\mathcal{G}})$ in $\mathcal{A}$ satisfying $i_{\mathcal{F}} \leq c(u'), i_{\mathcal{G}} \leq c(v')$ has $i_{\mathcal{G}} = c(v')$ (see the

previous "**Match light subtrees**" section for discussion of these assumptions). First, the TreeAlign routine replaces the sequence of pairs between $(c(u'), i_{\mathcal{G}})$ and $(c(u)+1, c(v)+1)$ with the sequence $(c(u'), i_{\mathcal{G}}+1), (c(u'), i_{\mathcal{G}}+2), \ldots, (c(u'), c(v')-1), (c(u'), c(v')), (c(u')-1, c(v')-1), (c(u')-2, c(v')-2), \ldots, (c(u), c(v))$. Thus, each index increments by at most one between every pair in the new alignment $\mathcal{A}$ after these modifications and so, the alignment constraint is satisfied. Second, the TreeAlign routine replaces the sequence of pairs between $(o(u), i_{\mathcal{G}})$ and $(i_{\mathcal{F}}, o(v'))$ (where $(o(u), i_{\mathcal{G}})$ is the first pair in $\mathcal{A}$ with left index $o(u)$ and $(i_{\mathcal{F}}, o(v'))$ is the first pair in $\mathcal{A}$ with right index $o(v')$) with sequence $(o(u), i_{\mathcal{G}}+1), (o(u), i_{\mathcal{G}}+2), \ldots, (o(u), o(v)), (o(u)+1, o(v)+1), (o(u)+2, o(v)+2), \ldots, (o(u'), o(v')), (o(u')+1, o(v')), (o(u')+2, o(v')), \ldots, (i_{\mathcal{F}}-1, o(v'))$. Thus, again each index increments by at most one between every pair in the new alignment $\mathcal{A}$ after these modifications, and so $\mathcal{A}$ remains an alignment in either case of TreeAlign.

□

Recall that $\mathsf{TreeAlign}(\mathcal{A})$ denotes the alignment $\mathcal{A}$ after performing $\mathsf{TreeAlign}(\mathcal{A}, u)$ on all mis-aligned heavy nodes and deleting all mis-aligned light nodes. Next, we show that in addition to $\mathsf{TreeAlign}(\mathcal{A})$ satisfying the alignment requirements, it is also a tree alignment. To this end, we show that there are no nodes of $\mathcal{F}$ mis-aligned by $\mathsf{TreeAlign}(\mathcal{A})$. A useful observation about alignments is that if there is a mis-aligned node of $\mathcal{G}$, then there must be a mis-aligned node of $\mathcal{F}$. Therefore, if there are no mis-aligned nodes of $\mathcal{F}$, then all nodes of $\mathcal{F}$ and $\mathcal{G}$ are tree aligned by $\mathsf{TreeAlign}(\mathcal{A})$. The proof of this observation is as follows.

Observation 4.4.8. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, if $u \in \mathcal{G}$ is mis-aligned by $\mathcal{A}$, then there must be some $v \in \mathcal{F}$ that is also mis-aligned.*

Proof. Let $u \in \mathcal{G}$ be any mis-aligned node by $\mathcal{A}$. By definition of mis-aligned nodes, at least

one of $o(u)$ or $c(u)$ must be matched by $\mathcal{A}$. If $o(u) \simeq_{\mathcal{A}} o(v)$ for some $v \in \mathcal{F}$, by definition of mis-aligned nodes, $c(u)$ is either deleted or matched to a different parenthesis $c(v')$ for $v' \neq v$. Therefore, $c(v)$ does not match to $c(u)$ by $\mathcal{A}$, and so, v is also mis-aligned. A symmetric argument can be shown for when $c(u) \simeq_{\mathcal{A}} c(v)$ and $o(u)$ is deleted or matched to a different parenthesis $c(v')$ for $v' \neq v$. □

As a pre-cursor to the final lemma of the section, we show that when running TreeAlign on a node u in a heavy path H, any other node in H previously aligned by TreeAlign is unaffected by the modifications made during the alignment of u.

Lemma 4.4.9. *Given forests $\mathcal{F}$,$\mathcal{G}$ and a deletion-only alignment $\mathcal{A} : \mathcal{F} \to \mathcal{G}$ and heavy path $H = \{u_1, u_2, \ldots, u_m\}$ in $\mathcal{F}$ where $o(u_i) < o(u_j) < c(u_j) < c(u_i)$ for all $i < j$, if $u_j \in H$ is tree-aligned with node $v \in \mathcal{G}$ by $\mathcal{A}$, then u_j will still be tree-aligned with v by $\mathcal{A}$ after performing* $\mathsf{TreeAlign}(A, u_i)$.

Proof. First, if u_i is tree-aligned itself by $\mathcal{A}$, then $\mathsf{TreeAlign}(A, u_i)$ does not make any changes to $\mathcal{A}$. If u_i is mis-aligned and falls into Case 2 of $\mathsf{TreeAlign}(A, u_i)$, then trivially the statement is true since the only nodes affected by the changes to $\mathcal{A}$ are u_i and the node in $\mathcal{G}$ it is matched to.

If u_i is mis-aligned and falls into case 1, then we consider the two steps taken by $\mathsf{TreeAlign}(A, u_i)$ and how they affect a node $u_j \in H$ tree-aligned with $v \in \mathcal{G}$ by $\mathcal{A}$. We will argue that in fact these steps do not affect the alignment pairs for u_j and v at all, and if they did, the non-crossing property of alignments would be violated.

By the constraints of case 1, u_i is contained in a chain C such that $C[k] = u_i$ where $k \leq |C| - 5$ and $C[k+2] \in \mathcal{F}$. Since there is another node in H after u_i, u_i must have heavy child u_{i+1}. Let v_i denote $C[k+1]$, by Observation 4.4.6, v_i must have heavy child v_{i+1}. Without

loss of generality, we assume $c(u_i) \simeq_{\mathcal{A}} c(v_i)$, and let $(i_{\mathcal{F}}, i_{\mathcal{G}}) \in \mathcal{A}$ denote the rightmost pair in $\mathcal{A}$ with $i_{\mathcal{F}} \leq c(u_{i+1}), i_{\mathcal{G}} \leq c(v_{i+1})$. The first step of the TreeAlign routine modifies all pairs between $(i_{\mathcal{F}}, i_{\mathcal{G}})$ and $(c(u_i)+1, c(v_i)+1)$, exclusive. Refer to Figure 4.8 for an example of these modifications.

First, we note that either $i_{\mathcal{F}} = c(u_{i+1})$ or $i_{\mathcal{G}} = c(v_{i+1})$ by their definition. In either case, we claim that $o(u_j) < i_{\mathcal{F}}$. In the former case, this is trivially true since the opening parentheses corresponding to nodes in H have smaller index in $\mathcal{P}(\mathcal{F})$ than the closing parentheses corresponding to these same nodes. In the latter case, the claim is true by the non-crossing property of alignments since $(i_{\mathcal{F}}, c(v_{i+1}))$ and $(c(C[k+2]), c(C[k+3]))$ are in $\mathcal{A}$ with $c(C[k+3]) < c(v_{i+1})$, so $o(u_j) < c(C[k+3]) < i_{\mathcal{F}}$. Therefore, $o(u_j)$ is unaffected by the modifications of the first step of TreeAlign.

If $j < i$, then u_j is one of the ancestors of u_i with $c(u_j) > c(u_i)$, and so $c(u_j)$ is unaffected by the modifications of the first step as well in this case.

If $j > i$, then $c(u_j)$ is affected only if $i_{\mathcal{F}} < c(u_j) \leq c(u_i)$. In this case, since u_j is a descendant of u_i, the heavy child u_{i+1} of u_i must also be affected by these changes, i.e., it must be that $i_{\mathcal{F}} < c(u_{i+1})$. As mentioned previously, either $i_{\mathcal{F}} = c(u_{i+1})$ or $i_{\mathcal{G}} = c(v_{i+1})$ and since $i_{\mathcal{F}} < c(u_{i+1})$, it must be that $i_{\mathcal{G}} = c(v_{i+1})$. Recall that $v \in \mathcal{G}$ denotes the node tree-aligned with u_j by $\mathcal{A}$. By the non-crossing property of alignments, since pairs $(i_{\mathcal{F}}, c(v_{i+1})), (c(u_i), c(v_i))$ are in $\mathcal{A}$ and $i_{\mathcal{F}} < c(u_j) < c(u_i)$, it must be that $c(v_{i+1}) < c(v) < c(v_i)$. Furthermore, since v_{i+1} is the heavy child of v_i, then this implies that v must lie in the light subtree of v_i, i.e., $c(v_{i+1}) < o(v) < c(v) < c(v_i)$. However, this leads to a contradiction: the non-crossing property is violated by $(o(u_j), o(v))$ and $(i_{\mathcal{F}}, c(v_{i+1}))$ since $o(u_j) < i_{\mathcal{F}}$ but $o(v) > c(v_{i+1})$. Thus, it must be that $i_{\mathcal{F}} > c(u_j)$ and so u_j is unaffected by these changes.

Now, let $(o(u_i), i_{\mathcal{G}})$ be the first pair in $\mathcal{A}$ with left index $o(u_i)$ and let $(i_{\mathcal{F}}, o(v_{i+1}))$ be the first pair in $\mathcal{A}$ with right index $o(v_{i+1})$. The second step of the TreeAlign routine modifies all pairs between $(o(u_i), i_{\mathcal{G}})$ and $(i_{\mathcal{F}}, o(v_{i+1}))$, exclusive (see Figure 4.9 for example). As in the first step, if $j < i$, then u_j is not affected by these changes since $o(u_j) < o(u_i)$ and $c(u_j) > i_{\mathcal{F}}$. If $j > i$, the alignment of $o(u_j)$ could be affected if $o(u_{i+1}) \leq o(u_j) < i_{\mathcal{F}}$. By an analogous argument to the first step, this would imply that v is in the light subtree of $o(v_i)$, i.e., $o(v_i) < o(v) < c(v) < o(v_{i+1})$. However, this would violate the non-crossing property of $\mathcal{A}$ due to pairs $(c(u_j), c(v)), (i_{\mathcal{F}}, o(v_{i+1})) \in \mathcal{A}$ with $c(u_j) > i_{\mathcal{F}}$ and $c(v) < o(v_{i+1})$. Thus, u_j is not modified by either step of $\mathsf{TreeAlign}(\mathcal{A}, u_i)$. □

We now give an inductive argument to prove that all nodes of $\mathcal{F}$ are tree aligned. The main challenge of the proof comes from Case 1 of the $\mathsf{TreeAlign}(\mathcal{A}, u)$ routine, which affects the alignment of heavy nodes beyond just u. The previous lemma already shows that no other node in the same heavy path as u is affected, but we must also make sure any heavy nodes not in this heavy path are also still tree-aligned after this routine. In the modifications of this case, we were careful to add deletions for all affected parentheses, so in fact even for nodes in other heavy paths, these modifications maintain that nodes of $\mathcal{F}$ are still tree-aligned. We discuss this formally as follows.

Lemma 4.4.10. *Given an alignment $\mathcal{A}$, $\mathsf{TreeAlign}(\mathcal{A})$ is a tree alignment.*

Proof. Let $\mathcal{H} = \{H_1, H_2, \ldots, H_\ell\}$ be the set of all maximal-length heavy paths in order from smallest node heavy depth to largest node heavy depth. We prove that all heavy nodes in $\mathsf{TreeAlign}(\mathcal{A})$ are tree-aligned by induction on i where i is the index of the current heavy path H_i whose mis-aligned nodes are currently being processed by the TreeAlign routine.

As the base case, we consider $\mathsf{TreeAlign}(\mathcal{A}, u_i^1)$ for some u_i^1 in $H_1 = (u_1^1, u_2^1, \ldots, u_{m_1}^1)$ where

the nodes of H_1 are ordered from highest level to lowest level in $\mathcal{F}$ such that u_i^1 is the ancestor of all u_j^1 with $j > i$. We show that after performing $\mathsf{TreeAlign}(\mathcal{A}, u_i^1)$ for all i from 1 to m_1, all nodes in H_1 are tree-aligned. First, if u_i^1 does not satisfy the conditions of case 1, $\mathcal{A}$ is modified to delete both $o(u_i^1)$ and $c(u_i^1)$. Clearly, u_i^1 will be tree-aligned after this step and no other node alignment will be affected. Instead, if u_i^1 falls into case 1, then it is contained in a chain C such that $C[k] = u_i^1$ and $\mathcal{A}$ is modified so that u_i^1 is tree-aligned to $v = C[k+1] \in \mathcal{G}$. By Lemma 4.4.9, all other nodes of H that are already tree-aligned will remain tree-aligned, and so after all nodes of H_1 are passed to $\mathsf{TreeAlign}$, all heavy nodes in H_1 will be tree-aligned.

Now, we assume that all heavy nodes in heavy paths up to path H_{z-1} are tree-aligned, and we want to show that all heavy nodes in heavy paths up to H_z are tree-aligned after performing $\mathsf{TreeAlign}(\mathcal{A}, u_i^z)$ for all i from 1 to m_z. By the same argument as the base case, all misaligned heavy nodes u_i^z of H_z will be tree-aligned after performing either case 1 or case 2 of $\mathsf{TreeAlign}(\mathcal{A}, u_i^z)$. We now must confirm that all heavy nodes from H_j for $j < z$ are also still tree-aligned after performing these steps. If u_i^z falls into case 2 of the $\mathsf{TreeAlign}(\mathcal{A}, u_i^z)$ routine, it is deleted and no heavy nodes of any prior paths are affected, so we only need to consider case 1.

If u_i^z falls into case 1, it is contained in a chain C such that $C[k] = u_i^z$ and will be matched to $v = C[k+1] \in \mathcal{G}$. As in the base case, assume without loss of generality that $c(u_i^z) \simeq_{\mathcal{A}} c(v)$ and let $(i_{\mathcal{F}}, i_{\mathcal{G}})$ denote the rightmost pair in $\mathcal{A}$ satisfying $i_{\mathcal{F}} \leq c(u_{i+1}^z), i_{\mathcal{G}} \leq c(v')$ where v' is the heavy child of v. First, the $\mathsf{TreeAlign}$ routine replaces the sequence of pairs between $(i_{\mathcal{F}}, i_{\mathcal{G}})$ and $(c(u_i^z)+1, c(v)+1)$ with a sequence of matches between the light subtrees u_i^z and v and then deletions of all other parentheses corresponding to indices of the removed pairs.

If $i_{\mathcal{F}} = c(u_{i+1}^z)$, then only nodes $u' \in \mathcal{F}$ with $c(u_{i+1}^z) < o(u') < c(u') < c(u_i^z)$ are affected by these modifications. These are exactly the nodes in the light subtree of u_i^z and are therefore

matched and tree-aligned to corresponding nodes in the light subtree of v.

If $i_{\mathcal{F}} < c(u^z_{i+1})$, then $i_G = c(v')$. In this case, there may exist some $u' \in \mathcal{F}$ that lie outside of the light subtree of u^z_i and are affected by modifications to $\mathcal{A}$ $\mathsf{TreeAlign}(\mathcal{A}, u^z_i)$ as long as u' is tree-aligned with a node in the light subtree of v before modifications. Note that for any such u', $o(u^z_i) < o(u') < c(u') < c(u^z_i)$, and so u' must either be in H_z or a path $H_{z'}$ for $z' < z$. Since we have already dealt with nodes in H_z, we only concern ourselves with the latter case. By the induction hypothesis these nodes are already tree-aligned. Therefore, both $o(u')$ and $c(u')$ must be deleted or aligned to parentheses in the light subtree of v before $\mathsf{TreeAlign}$ modifies $\mathcal{A}$, and so, after modifications, these parentheses will both be deleted and u' will still be tree-aligned.

Now let $(o(u^z_i), i_{\mathcal{G}})$ be the first pair in $\mathcal{A}$ with left index $o(u^z_i)$ and $(i_{\mathcal{F}}, o(v'))$ be the first pair in $\mathcal{A}$ with right index $o(v')$. Second, the $\mathsf{TreeAlign}$ routine replaces the sequence of pairs between $(o(u^z_i), i_{\mathcal{G}})$ and $(i_{\mathcal{F}}, o(v'))$ with a matching of the parentheses corresponding to the light subtrees of u^z_i and v and then deletions of all other parentheses corresponding to removed pairs. By an analogous argument to the previous case, any node u' affected by these modifications must be in a heavy path $H_{z'}$ for $z' < z$, and so by the induction hypothesis, after modifications, u' is tree-aligned to a node in the light subtree of v or both $o(u')$ and $o(v')$ are now deleted.

By induction, all heavy nodes in $\mathcal{F}$ will be tree-aligned in $\mathsf{TreeAlign}(\mathcal{A})$. Recall that $\mathcal{A}$ is modified to delete mis-aligned light nodes of $\mathcal{F}$, so trivially, there will be no mis-aligned light nodes in $\mathcal{F}$ in the final alignment. By Observation 4.4.8, all nodes in $\mathcal{F}$ and $\mathcal{G}$ are tree-aligned, and so $\mathsf{TreeAlign}(\mathcal{A})$ is a tree alignment □

4.4.5 Alignment Cost Analysis

We now prove that performing TreeAlign only increases the number of edits in alignment $\mathcal{A}$ by a $\tilde{\mathcal{O}}(\sqrt{n})$ multiplicative factor. The main challenge is arguing that additional deletions from case 1 of the TreeAlign routine do not blow up the cost of the alignment. Any node falling into this case must be in a chain, and to manage deletion costs, we first consider the end points the chain. Since we define chains to include nodes in only a single root-to-leaf path of each forest, chains must have finite length and the last node of a chain must be a partially deleted node. Therefore, we will typically charge deletions incurred by the TreeAlign to the deletions already present at the end points of chains. Moreover, we can extend this idea further to sequences that include multi-branch mis-aligned nodes as well. We define them as follows.

Definition 4.4.11 (Extended chain)**.** Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$, an *extended chain* $\mathcal{C}$ is a maximal-length sequence of at least 3 nodes alternating from $\mathcal{F}$ and $\mathcal{G}$ where

1. For all even i such that $\mathcal{C}[i] \in \mathcal{F}$, if $i < |\mathcal{C}| - 1$, $o(\mathcal{C}[i]) \simeq_{\mathcal{A}} o(\mathcal{C}[i+1])$ and if $i > 0$, $c(\mathcal{C}[i]) \simeq_{\mathcal{A}} c(\mathcal{C}[i-1])$. OR
2. For all even i such that $\mathcal{C}[i] \in \mathcal{F}$, if $i < |\mathcal{C}| - 1$, $c(\mathcal{C}[i]) \simeq_{\mathcal{A}} c(\mathcal{C}[i+1])$ and if $i > 0$, $o(\mathcal{C}[i]) \simeq_{\mathcal{A}} o(\mathcal{C}[i-1])$.

Note the difference between the above definition and Definition 4.4.5 is we no longer require the mis-aligned nodes of an extended chain to be on a single branch of $\mathcal{F}$ or $\mathcal{G}$. This allows us to follow maximal-length sequences of aligned parentheses corresponding to mis-aligned nodes. Similarly to chains, we show that every extended chain ends in a partially deleted node. We now prove this formally for chains and then extended chains.

Lemma 4.4.12. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ and a chain C in $\mathcal{A}$, let $u = C[i]$ for any $i > 2$. If there exists a node $v \neq C[i-1]$ such that either $o(u) \simeq_{\mathcal{A}} o(v)$ or $c(u) \simeq_{\mathcal{A}} c(v)$, then $v = C[i+1]$.*

Proof. Without loss of generality let $u \in \mathcal{F}$, $v \in \mathcal{G}$. Let $u' = C[i-2] \in \mathcal{F}, v' = C[i-1] \in \mathcal{G}$. We know by Definition 4.4.5 that $o(u') < o(u) < c(u) < c(u')$ and without loss of generality let $o(u') \simeq_{\mathcal{A}} o(v'), c(u') \simeq_{\mathcal{A}} c(v)$, and so, $o(u) \simeq_{\mathcal{A}} o(v)$ since $v \neq v'$ as given. We know that pairs $(o(u'), o(v')), (o(u), o(v)), (c(u'), c(v))$ are in $\mathcal{A}$ and since $o(u') < o(u) < c(u')$, then $o(v') < o(v) < c(v)$ by the non-crossing property of alignments. Therefore, v satisfies the requirements of Definition 4.4.5, and $v = C[i+1]$. □

Corollary 4.4.13. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ and a chain C in $\mathcal{A}$, $C[|C|-1]$ is a partially deleted node.*

Proof. From Lemma 4.4.12, if C is finite, the last node u of a chain must have either $o(u)$ or $c(u)$ not aligned by $\mathcal{A}$, i.e. u is a partially deleted node. Note that since there are a finite number of nodes, C can only be infinite if it contains the same node more than once. Assume for contradiction this is the case; let $C[i] = C[j]$ with $i < j$. Since $C[i] = C[j]$, then $o(C[i]) = o(C[j])$, which violates the requirement that $o(C[i]) < o(C[j])$ by the definition of chains, Definition 4.4.5. Therefore, every chain is finite and must reach a partially deleted node. □

Now, we turn our attention back to extended chains. Every extended chain has a very predictable structure; each extended chain contains at most 1 multi-branch mis-aligned node and at most 2 chains. We now prove this formally.

Lemma 4.4.14. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ and an extended chain $\mathcal{C}$ in $\mathcal{A}$, $\mathcal{C}$ contains at most 1 multi-branch mis-aligned node and at most 2 chains.*

Proof. Let $\mathcal{C}$ be an extended chain with multi-branch mis-aligned node $u_1 = \mathcal{C}[i]$ for any $2 < i < |\mathcal{C}| - 2$. Without loss of generality, let $u_1 \in \mathcal{F}$ and $v_1 = \mathcal{C}[i-1] \in_{\mathcal{G}}, v_2 = \mathcal{C}[i+1] \in \mathcal{G}$. By Definition 4.4.11, without loss of generality let $o(u_1) \simeq_{\mathcal{A}} o(v_1)$ and $c(u_1) \simeq_{\mathcal{A}} c(v_2)$. We know by the definition of multi-branch mis-aligned nodes, $o(v_1) < c(v_1) < o(v_2) < c(v_2)$. Consider the next node $u_2 = \mathcal{C}[i+2]$. We know that $o(v_2) \simeq_{\mathcal{A}} o(u_2)$. Furthermore, since pairs $(o(u_1), o(v_1)), (c(u_1), c(v_2))$ are in $\mathcal{A}$ and $o(v_1) < o(v_2) < c(v_2)$, then $o(u_1) < o(u_2) < c(u_1)$. Therefore, by Definition 4.4.5, u_1, v_2, u_2 form a chain. By Lemma 4.4.12, any node $\mathcal{C}[j]$ for $i < j < |\mathcal{C}| - 1$ must also be part of the chain and $\mathcal{C}[|\mathcal{C}| - 1]$ must be a partially deleted node. By a symmetric argument, it must also be the case that $\mathcal{C}[i], \mathcal{C}[i-1], \ldots, \mathcal{C}[1]$ form a chain and $\mathcal{C}[1]$ is a partially deleted node. □

Since we know that extended chains only contain a single multi-branch mis-aligned node and can only continue as long as the chains contained in them, we know that eventually they must end in a partial deletion as chains do.

Corollary 4.4.15. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ and an extended chain $\mathcal{C}$ in $\mathcal{A}$, then $\mathcal{C}[|\mathcal{C}| - 1]$ is a partially-deleted node.*

Proof. By Lemma 4.4.14, every extended chain has at most 2 chains and at most 1 multi-branch mis-aligned node. By Corollary 4.4.13, chains must end in a partially deleted node. Thus, extended chains are finite, and by Definition 4.4.11, the last node will have either $o(\mathcal{C}[|\mathcal{C}|-1])$ or $c(\mathcal{C}[|\mathcal{C}|-1])$ deleted. □

We prove a few more useful properties of chains. First, we show that every chain only intersects with at most $\mathcal{O}(\log n)$ heavy paths. We want to map deletions incurred by the TreeAlign

routine to deletions already present in $\mathcal{A}$, and since TreeAlign handles each heavy path of $\mathcal{F}$ iteratively, we do this mapping from the perspective of nodes in heavy paths. Therefore, this fact allows us to consider nodes in a chain on a per path basis at the cost of only an additional $\mathcal{O}(\log n)$ factor.

Lemma 4.4.16. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ with a chain C, let $S = \{H_1, H_2, \ldots, H_m\}$ be the set of heavy paths such that for all $1 \leq i \leq m$, there exists $u \in C$ with $u \in H_i$. Then $|S| = \mathcal{O}(\log n)$ where $n = |F| + |G|$.*

Proof. Observe by Definition 4.4.5, for any $i \in [0, |C| - 2]$, we must have that $o(C[i]) < o(C[i + 2]) < c(C[i + 2]) < c(C[i])$. By Observation 4.2.5, this implies that all nodes in a chain are in an ancestor-descendant relationship. Thus, the nodes of $\mathcal{F}$ in C lie on a single root-to-leaf path of $\mathcal{F}$. By Observation 4.2.7, these nodes can be in at most $\mathcal{O}(\log n)$ heavy paths. □

Another important property about chains is that since they have some periodicity in their light subtrees, we know that any two chains that overlap on the same heavy path follow a similar structure. The lemma and its proof are very notation-heavy, but the idea is very simple. See Figure 4.10 for an example of two chains lying on the same path and an example of why trying to change the alignment structure of only one of the chains leads to a violation of the non-crossing property of alignments. This structure is important because we can fix chains all in the same way without having to worry about how they are oriented in the alignment, and so when we fix a chain, we do not incur a blowup in the number of deletions from other overlapping chains.

Lemma 4.4.17. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ with chains C_1, C_2. For indices $0 \leq i_1 \leq |C_1| - 4, 0 \leq i_2 \leq |C_2| - 4$ with $C_1[i_1], C_2[i_2], C_1[i_1 + 2]$ in the same heavy path of either $\mathcal{F}$ or $\mathcal{G}$,*

Figure 4.10: (left) An example of two overlapping chains C_1, C_2 where C_1 includes the blue parentheses and C_2 includes the black brackets. This is the only possible alignment structure of two or more chains when $o[C_1(k)] < o[C_2(k)] < o[C_1(k+1)]$. (right) An example of two overlapping chains violating the non-crossing property.

- $o(C_1[i_1]) < o(C_2[i_2]) < o(C_1[i_1+2])$ *if and only if* $o(C_1[i_1+1]) < o(C_2[i_2+1]) < o(C_1[i_1+3])$,
- $c(C_1[i_1+2]) < c(C_2[i_2]) < c(C_1[i_1])$ *if and only if* $c(C_1[i_1+1]) < c(C_2[i_2+1]) < c(C_1[i_1+3])$.

Proof. First, note that $o(C_1[i_1]) < o(C_2[i_2]) < o(C_1[i_1+2])$ if and only if $c(C_1[i_1+2]) < c(C_2[i_2]) < c(C_1[i_1])$ since $C_1[i_1], C_2[i_2], C_1[i_1+2]$ lie on the same path of $\mathcal{F}$ or $\mathcal{G}$. By the definition of chains, either $o(C_1[i_1]) \simeq_{\mathcal{A}} o(C_1[i_1+1])$ or $c(C_1[i_1]) \simeq_{\mathcal{A}} c(C_1[i_1+1])$. Without loss of generality, we assume the former. Again by the definition of chains, $o(C_2[i_2]) \simeq_{\mathcal{A}} o(C_2[i_2+1])$ or $c(C_2[i_2]) \simeq_{\mathcal{A}} c(C_2[i_2+1])$. Consider the latter case where pair $(c(C_2[i_2]), c(C_2[i_2+1]))$ is in $\mathcal{A}$. Since $o(C_1[i_1]) \simeq_{\mathcal{A}} o(C_1[i_1+1])$, pairs $(c(C_1[i_1+2]), c(C_1[i_1+1])), (c(C_1[i_1+4]), c(C_1[i_1+3])$ must be in $\mathcal{A}$. By the non-crossing property, we must have that $c(C_1[i_1+4]) < c(C_1[i_1+2]) < c(C_2[i_2])$ and $c(C_1[i_1+3]) < c(C_1[i_1+1]) < c(C_2[i_2+1])$. Therefore, we also know that $o(C_2[i_2+1]) < o(C_1[i_1+1])$. Furthermore, pairs $(o(C_1[i_1]), o(C_1[i_1+1])), (o(C_1[i_1+2]), o(C_1[i_1+3])), (o(C_2[i_2+2], o(C_2[i_2+1])$ are in $\mathcal{A}$. By the non-crossing property, since $o(C_2[i_2+1]) < o(C_1[i_1+1]) < o(C_1[i_1+3])$, then $o(C_2[i_2+2]) < o(C_1[i_1]) < o(C_1[i_1+2])$. This implies that $o(C_2[i_2+2]) < o(C_1[i_1]) < o(C_2[i_2])$, however this is a contradiction with the definition of chains for C_2 since $o(C_2[i_2+2])$ must be greater than $o(C_2[i_2])$. Therefore, it must be that $o(C[i_2]) \simeq_{\mathcal{A}} o(C[i_2+1])$.

In this case, we know that pairs $(o(C_1[i_1]), o(C_1[i_1+1])), (o(C_1[i_1+2]), o(C_1[i_1+3])), (o(C_2[i_2], o(C_2[i_2+1])$ are in $\mathcal{A}$, and since $o(C_1[i_1]) < o(C_2[i_2]) < o(C_1[i_1+2])$, by the non-crossing property, we must have that $o(C_1[i_1+1]) < o(C_2[i_2+1]) < o(C_1[i_1+3])$. Note that this implies that $c(C_1[i_1+1]) < c(C_2[i_2+1]) < c(C_1[i_1+3])$ since $C_1[i_1+1], C_2[i_2+1], C_1[i_1+3]$ are on the same heavy path by the definition of chains and $C_1[i_1], C_2[i_2], C_1[i_1+2]$ lie on the same path. □

With the above lemma, we can prove another useful structural lemma. If the number of nodes in the light subtree of a chain node $C[i]$ is smaller than the light subtree of the next node in the chain $C[i+2]$, then there must be some mis-aligned node not in C in the light subtree of $C[i+2]$ on the same heavy path as the chain nodes. Furthermore, if this node is in a different chain C', it must be the first node. We write this formally in the following.

Lemma 4.4.18. *Given an alignment $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ with a chain C such that $w(C[i], C[i+2]) < \sqrt{n}, w(C[i+2], C[i+4]) \geq \sqrt{n}$ with $C[i], C[i+2] \in H$ for some heavy path H in $\mathcal{F}$. Then, there must be a mis-aligned node u in H with $o(C[i+2]) < o(u) < o(C[i+4])$. Furthermore, u must either be a multi-branch mis-aligned node, a partially-deleted node, or the first node of $\mathcal{F}$ in a chain $C' \neq C$.*

Proof. First, note that since $w(C[i], C[i+2]) < \sqrt{n}$ but $w(C[i+2], C[i+4]) \geq \sqrt{n}$, by Observation 4.4.6, since the light subtrees of $C[i]$ and $C[i+2]$ are the same, there must be some heavy node on path H between $C[i+2]$ and $C[i+4]$. If any node on H between $C[i+2]$ and $C[i+4]$ is a multi-branch mis-aligned node or partially deleted, we are done.

If all such nodes are in a chain, assume for contradiction that none are the first node of $\mathcal{F}$ in their respective chain. For any such node in chain C', i.e., $C'[j]$ for some $j \geq 3$, we have that by

Lemma 4.4.17, $C'[j-2]$ must be on H between $C[i], C[i+2]$. Since the the light subtrees of $C'[j]$ and $C'[j-2]$ are the same by Observation 4.4.6, if all nodes in chains were not the first node of $\mathcal{F}$ in their chain, then we have that $w(C[i], C[i+2]) \geq w(C[i+2], C[i+4]) \geq \sqrt{n}$, which is a contradiction with our assumption that $w(C[i], C[i+2]) < \sqrt{n}$. Thus, one of the nodes between $C[i+2]$ and $C[i+4]$ in a chain must be the first node in $\mathcal{F}$ in this chain. □

Finally, we aim to prove the main lemma of the section. Let $\mathcal{A} : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{G})$ be an optimal alignment and let $\mathcal{A}'$ denote $\mathsf{TreeAlign}(\mathcal{A})$. Let $D_{\mathcal{A}}$ be the deleted parentheses of $\mathcal{A}$ and let $D'_{\mathcal{A}}$ be the deleted parentheses of $\mathsf{TreeAlign}(\mathcal{A}, u)$. For a mis-aligned node u, let $\mathcal{C}(u)$ correspond to the extended chain containing u. For an extended chain $\mathcal{C}(u)$, let $d_{\mathcal{C}(u)}$ denote the deleted parenthesis corresponding to $\mathcal{C}[|\mathcal{C}| - 1]$. We define a mapping $\gamma : D'_{\mathcal{A}} \to D_{\mathcal{A}}$ and show that the pre-image $\gamma^{-1}(d)$ of any parenthesis $d \in D_{\mathcal{A}}$ is at most of size $\mathcal{O}(\sqrt{n} \log n)$.

To define our mapping, we consider the deletions incurred by $\mathsf{TreeAlign}(\mathcal{A}, u)$ for each heavy node u in $\mathcal{F}$. Recall that if u is tree-aligned, nothing occurs, so we only are concerned with mis-aligned nodes. Consider a mis-aligned node u in a heavy path H and let $\mathcal{C}$ denote the extended chain containing u if one exists.

Case 1: If u falls into case 1, then $u = C[k]$ for some chain C and integer k. The $\mathsf{TreeAlign}$ routine modifies $\mathcal{A}$ so that nodes in tree $\mathcal{F}(u)'$ are tree-aligned to nodes in tree ${}_{\mathcal{G}}(v)'$ where $v = C[k+1]$, deleting any other parentheses whose alignment would violate the non-crossing property after this modification. Let $\mathcal{P}(\mathcal{F})[a]$ be a parenthesis deleted in this step, let $u_a \in H$ be the heavy node in path H whose light subtree contains the node corresponding to $\mathcal{P}(\mathcal{F})[a]$, and let $\mathcal{C}(u_a)$ be the extended chain containing u_a. To define mapping γ for each such a, we break up this case into subcases to make sure we avoid mapping more than $\tilde{\mathcal{O}}(\sqrt{n})$ new deletions to a single deletion of

$\mathcal{A}$.

Before discussing the mappings, if u_a is node $C_a[k_a]$ in chain C_a with $C_a[k_a{+}2], C_a[k_a{+}4] \in H$, $w(C_a[k_a], C_a[k_a+2]) < \sqrt{n}, w(C_a[k_a+2], C_a[k_a+4]) < \sqrt{n}$, then u_a itself falls into case 1 of TreeAlign and will have its light subtree, including $\mathcal{P}(\mathcal{F})[a]$, later aligned by TreeAlign$(\mathcal{A}, u_a)$. As a result, we do not map a in this case.

a) If a is already deleted in $\mathcal{A}$, set $\gamma(a) = a$.

b) If u_a is a multi-branch mis-aligned node or a partially-deleted node, set $\gamma(a) = d_{\mathcal{C}(u_a)}$.

c) If $C[k-2] \notin H$ or $C[k+6] \notin H$, set $\gamma(a) = d_{\mathcal{C}(u)}$.

d) If u_a is a node $C_a[k_a]$ in chain C_a and either $C_a[k_a+2] \notin H$ or $C_a[k_a+4] \notin H$, then set $\gamma(a) = d_{\mathcal{C}(u_a)}$.

e) If $w(C[k_a+4], C[k_a+6]) \geq \sqrt{n}$, by Lemma 4.4.18, there is some $u' \in H$ such that u' is a mis-aligned node with $o(C[k_a+4]) < o(u') < o(C[k_a+6])$ such that u' is a multi-branch mis-aligned node or the first node of $\mathcal{F}$ in a chain. Let $\mathcal{C}(u')$ be the extended chain containing u'. Set $\gamma(a) = d_{\mathcal{C}(u')}$.

Case 2: For u in case 2, u must either be a multi-branch node, a single-branch node, or part of a chain not satisfying the requirements of case 1 in TreeAlign. We set $\gamma(o(u)) = \gamma(c(u)) = d_{\mathcal{C}(u)}$.

We now analyze the above mapping in the proof of the main lemma of the section.

Lemma 4.4.19. *Given forests $\mathcal{F}, \mathcal{G}$,* $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq \mathcal{O}(\log n \cdot \sqrt{n}) \cdot \mathsf{ed}(\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G}))$.

Proof. We consider each of the cases of mapping γ and show that the number of parentheses a mapped to any single $d \in D_{\mathcal{A}}$, i.e., $\gamma(a) = d$ is at most $\tilde{\mathcal{O}}(\sqrt{n})$ for each case.

Case 1(a): In this case, a is d, so trivially we only have a single element mapped to d from this case.

Case 1(b): If u_a is a multi-branch mis-aligned node or partially deleted node mapped to d in this case, then d must be in the same extended chain as u_a, which only contains a single multi-branch mis-aligned node by Lemma 4.4.14 and at most two partially deleted nodes by the definition of extended chains, Definition 4.4.11. Since $w(C[k], C[k+2]) < \sqrt{n}$, we know that less than $\sqrt{n}$ parentheses lie in the light subtree of u_a as by the definition of the parenthesis representation and Observation 4.2.5, the light subtree of $C[k]$ corresponds precisely to $\mathcal{P}(\mathcal{F})[o(C[k]), o(C[k+2]))$ and $\mathcal{P}(\mathcal{F})(c(C[k+2]), c(C[k])]$, and so only $\mathcal{O}(\sqrt{n})$ parentheses map to d from this case.

Case 1(c): In this case, u is the first node of chain C in H or the third-last node of chain C in H. In either case, since $w(C[k], C[k+2]) < \sqrt{n}$, then at most $\mathcal{O}(\sqrt{n})$ deletions are mapped to $d_{\mathcal{C}(u)}$ per heavy path. By Lemma 4.4.16, there are then only $\mathcal{O}(\log n \cdot \sqrt{n})$ total deletions mapped to $d_{\mathcal{C}(u)}$.

Case 1(d): In this case u_a is the second-last or third-last node of chain C_a in H. Similarly to Case 1(c), in either case, since $w(C[k], C[k+2]) < \sqrt{n}$, then at most $\mathcal{O}(\sqrt{n})$ deletions are mapped to $d_{\mathcal{C}(u_a)}$ per heavy path. By Lemma 4.4.16, there are then only $\mathcal{O}(\log n \cdot \sqrt{n})$ total deletions mapped to $d_{\mathcal{C}(u_a)}$.

Case 1(e): In this case, there is some mis-aligned node $u' \in H$ with $o(C[k_a + 4]) < o(u') < o(C[k_a + 6])$. Since $w(C[k_a], C[k_a + 4]) < 2\sqrt{n}$, at most $\mathcal{O}(\sqrt{n})$ deletions are mapped to $d_{\mathcal{C}(u')}$ in this step. By Lemma 4.4.18, u' is either a multi-branch mis-aligned node, partially deleted node, or first node in $\mathcal{F}$ of its chain. In all cases, these can only occur at most $\mathcal{O}(1)$ times per extended chain, and so $\mathcal{O}(\sqrt{n})$ total deletions are mapped to $d_{\mathcal{C}(u')}$ in this case.

Case 2: For u in case 2, u must either be a multi-branch node, a single-branch node, or part of a chain not satisfying the requirement of case 1. Deletions of $o(u)$ and $c(u)$ are assigned to $d_{\mathcal{C}(u)}$.

There can only be $\mathcal{O}(1)$ multi-branch or partially-deleted nodes mapped to $d_{\mathcal{C}}$ from case 2 by the Definition 4.4.11 and Lemma 4.4.14. If u is at position k in a chain C, it must be that either $C[k+2], C[k+4] \notin H$, $w(C[k], C[k+2]) \geq \sqrt{n}$, or $w(C[k+2], C[k+4]) \geq \sqrt{n}$. Note that the first case implies that u is the last or second-last node in H part of C, so this can only happen twice per heavy path that shares nodes with extended chain $\mathcal{C}$. By Lemma 4.4.16, there can only be $\mathcal{O}(\log n)$ deletions mapped to $d_{\mathcal{C}(u)}$ by this case. In the remaining two cases, either $w(C[k], C[k+2]) \geq \sqrt{n}$ or $w(C[k+2], C[k+4]) \geq \sqrt{n}$, which can only occur $\mathcal{O}(n/\sqrt{n}) = \mathcal{O}(\sqrt{n})$ times per chain, and so at most $\mathcal{O}(\sqrt{n})$ chain nodes are mapped to $d_{\mathcal{C}(u)}$ from case 2.

Finally, we briefly mention that any deleted light node in extended chain $\mathcal{C}$ may have its corresponding deletion mapped to $d_{\mathcal{C}}$. Since chains of light nodes can only be length $\mathcal{O}(\log n)$ by Observation 4.2.7, then there are at most $\mathcal{O}(\log n)$ light node deletions mapped to a single deletion of $D_{\mathcal{A}}$.

Thus, in all cases, at most $\mathcal{O}(\log n \cdot \sqrt{n})$ parentheses are mapped to any $d \in D_{\mathcal{A}}$, and therefore, $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq |D'_{\mathcal{A}}| \leq \mathcal{O}(\log n \cdot \sqrt{n}) \cdot |D_{\mathcal{A}}| = \mathcal{O}(\log n \cdot \sqrt{n}) \cdot \mathsf{ed}(\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G}))$.

□

4.4.6 Approximate Dynamic Tree Edit Distance

We now show that our novel tree to string reduction can be maintained dynamically and used in conjunction with the dynamic string edit distance algorithm of [Kociumaka et al., 2023] to obtain a dynamic approximation for tree edit distance with sub-polynomial update time. Our reduction relies on maintaining a heavy-light decomposition on the input trees as dynamic edits occur on the tree. Fortunately, we have already shown how to dynamically maintain our heavy-

light decomposition in our dynamic Dyck edit distance algortihm; observe that the relationship between input trees and their parentheses representation is precisely the same relationship between a parenthesis string and its reduction tree from Section 4.3. The only difference is that now dynamic edits come directly to the tree, which correspond to two edits on the parentheses string. With this in mind, we can directly use our earlier result that states that we can maintain the heavy-light paths of our decomposition in poly-logarithmic time. The main challenge of this section then is showing how we update the subtree embeddings in the labels of our reduction strings according to Definition 4.4.1. Recall that in our reduction strings, a node's label contains the subtree of all its light children. For any dynamic insertion or deletion, we therefore need to reflect the dynamic edit in the labels of $\mathcal{O}(\log n)$ ancestor nodes, one per light node. Additionally, for any nodes that change from heavy to light or light to heavy we need to adjust its parents label accordingly.

In order to reflect the dynamic edit in light node ancestor labels, we first need a way to compute the set of light node ancestors of the dynamic edit. We show that using a binary tree built on top of the parenthesis representation, we can do so.

Lemma 4.4.20. *For a forest $\mathcal{F}$, a data structure can be dynamically maintained to compute the light node ancestor set of node v for all $v \in \mathcal{F}$ in time $\mathcal{O}(\log^2 |\mathcal{F}|)$ with update time $\mathcal{O}(\log^2 |\mathcal{F}|)$ when a node of $\mathcal{F}$ changes from heavy to light or light to heavy. The data structure can be initialized in time $\tilde{\mathcal{O}}(|\mathcal{F}|)$.*

Proof. We begin by computing a heavy-light decomposition of $\mathcal{F}$. Recall that by Observation 4.2.5, ancestors of node v are precisely those nodes u with $o(u) < o(v) < c(v) < c(u)$. So, we show how to find the set of light nodes satisfying these conditions. To do so, we utilize a binary tree B built on top of the parenthesis representation of $\mathcal{F}$.

The leaves of B partition $\mathsf{P}(\mathcal{F})$ into individual characters so the ith leaf corresponds to the ith character in $\mathsf{P}(\mathcal{F})$ and each internal node v corresponds to the concatenation of the strings in its children, which we denote as S_v. Each node v of B stores the number of unmatched light node opening parentheses U_v^O and unmatched light node closing parentheses U_v^C in S_v. Note here that parenthesis $o(u)$ ($c(u)$) for a node u is unmatched in a substring if $c(u)$ ($o(u)$) is not contained in that substring. Observe that given node v with left child u and right child w, $U_v^O = \max\{U_u^O - U_w^C, 0\} + U_w^O$ and $U_v^C = \max\{U_w^C - U_u^O, 0\} + U_u^C$. It is easy to see that constructing B bottom-up can be done in $\mathcal{O}(|\mathcal{F}|)$-time with this fact. Furthermore, if a single parentheses is deleted from $\mathcal{F}$ or changed from light to heavy or heavy to light, updating the root-to-leaf path of the updated parenthesis can be done in time logarithmic of the height of tree, which can be maintained using a self-balancing tree, e.g., an AVL tree, while only affecting an additional factor of $\mathcal{O}(\log |\mathcal{F}|)$ nodes per update. Therefore, when a node u changes from heavy to light, we can rebuild the root-to-leaf path to the leaves corresponding to $o(u)$ and $c(u)$ in $\mathcal{O}(\log^2 |\mathcal{F}|)$ time.

Now, we discuss how to use B to find the set of light node ancestors of a node $v \in \mathcal{F}$. We start at the leaf corresponding to $o(v)$ and traverse upwards through parent-child links. As we traverse upwards, if we are at a node u, we will keep track of the number of unmatched closing parentheses to the left of $o(v)$ in S_u, which we denote as L_u. If we are currently at node $w \in B$ moving to parent node u, we may compute L_u from L_w as follows. If w is the left child of u and w' is the right child of u, then $L_u = L_w$. If w is the right child and w' is the left child of u, $L_u = \max\{L_w - U_{w'}^O, 0\}$.

We use the above values L_u to determine when there is a new unmatched light node parenthesis $o(v')$ in S_u to the left of $o(v)$, which implies that $o(v') < o(v) < c(v) < c(v')$, i.e., v' is a light node ancestor of v. If w, the previous node we moved to u from is the left child of u, we

know that there is no new such parenthesis $o(v')$ to the left of u in S_u that was not already present in S_w, so we do nothing. However, if w is the right child of u and w' denotes the left child of u, we know the number of new light node parentheses to the left of $o(v)$ in S_u is $U^O_{w'} - L_w$. If this value is positive, we mark the left child w' and continue traversing to the root.

For all marked nodes u, we traverse back down the tree from u to find the indices of the light node ancestors of v. Note that all parentheses in strings corresponding to marked nodes lie to the left of $o(v)$ by our marking algorithm. Therefore, we just need to determine the exact leaf nodes corresponding to opening parentheses whose match is to the right of $o(v)$. Let w be the left child of u and let w' be the right child of u. If $U^O_{w'}$ is greater than 0, we traverse right. If $U^O_w - U^C_{w'} > 0$, we traverse left. In either case, we know that there is an unmatched opening parenthesis corresponding to a light node ancestor of v. These two cases are not exclusive, we may do one or both traversals. When we reach the leaf nodes, we may take the index stored there and add its corresponding node in $\mathcal{F}$ to the set of light node ancestors. A single ancestor node requires traversing up and down the tree once each where the height of the tree is $\mathcal{O}(\log |\mathcal{F}|)$, and since there are $\mathcal{O}(\log |\mathcal{F}|)$ such light node ancestors, we may compute the entire set in time $\mathcal{O}(\log^2 |\mathcal{F}|)$. □

To dynamically maintain our subtree embeddings efficiently, we utilize a dynamic strings data structure of [Gawrychowski et al., 2018], which allows us to store and perform splitting and merging operations on a collection of strings. Additionally, each string in the structure is assigned a unique integer identifier (see [Gawrychowski et al., 2018] for details on how this is done).

We now state some necessary definitions from [Gawrychowski et al., 2018]. As mentioned, we will dynamically maintain this data structure on a collection of strings $\mathcal{X}$ that will be initially created during a pre-processing step. We define the following operations for the data structure:

Theorem 4.4.21 ([Gawrychowski et al., 2018] Thm. 6.12)**.** *A collection $\mathcal{X}$ of persistent strings of total length n can be dynamically maintained subject to the following operations:*

$\mathsf{add}(X)$ *for $X \in \Sigma^+$ results in $\mathcal{X} = \mathcal{X} \cup \{X\}$ and takes $\mathcal{O}(\log n + |X|)$ time.*

$\mathsf{concat}(X_1, X_2)$ *for $X_1, X_2 \in \mathcal{X}$ results in $\mathcal{X} := \mathcal{X} \cup \{X_1 \cdot X_2\}$ and takes $\mathcal{O}(\log n)$ time.*

$\mathsf{split}(X, k)$ *for $X \in \mathcal{X}, k \in [|X|]$ results in $\mathcal{X} := \mathcal{X} \cup \{X[0\,..\,k), X[k\,..\,|X|)\}$ and takes $\mathcal{O}(\log n)$ time.*

$\mathsf{LCP}(X_1, X_2)$ *for $X_1, X_2 \in \mathcal{X}$ computes the length of longest common prefix of X_1 and X_2 and takes $\mathcal{O}(1)$ time.*

The algorithms are Las-Vegas randomized and the running time bounds hold with high probability.

We now discuss how to maintain our label embeddings formally in the proof of the main theorem of this section.

Theorem 4.4.22. *Given integers $2 \leq b \leq n$, there exists a randomized dynamic algorithm, that maintains an $\mathcal{O}(b \log_b n \log^2 n \cdot \sqrt{n})$-approximation of $\mathsf{ted}(\mathcal{F}, \mathcal{G})$ (correctly with high probability against an oblivious adversary) for (initially empty) forests $\mathcal{F}, \mathcal{G}$ of size at most n undergoing edits. The expected update time of the algorithm is $\mathcal{O}(b^2 (\log n)^{\mathcal{O}(\log_b n)})$ per edit.*

Proof. Let $n = |\mathcal{F}|$. Without loss of generality let the dynamic update occur in $\mathcal{F}$. We will utilize a dynamic stings data structure instance $\mathcal{X}$ from Theorem 4.4.21 to store the parentheses representations of light subtrees for all nodes in $\mathcal{F}$ and $\mathcal{G}$, and we use the unique integer identifier assigned to each string in $\mathcal{X}$ as the label of each node in $\mathcal{P}(\mathcal{F})$.

First, we show how to maintain $\mathsf{P}(\mathcal{F})$ in $\mathcal{X}$ so that we may quickly find the parenthesis representation of the light subtree of any node in $\mathcal{F}$. If $u \in \mathcal{F}$ is deleted, then we may use $\mathsf{split}(\mathsf{P}(\mathcal{F}), o(u))$ and $\mathsf{split}(\mathsf{P}(\mathcal{F}), o(u) + 1)$ followed by $\mathsf{concat}(\mathsf{P}(F)[0\,..\,o(u) - 1], \mathsf{P}(\mathcal{F})[o(u) +$

$1\,..))$ to remove $\mathsf{P}(F)[o(u)]$ from $\mathsf{P}(\mathcal{F})$ in $\mathcal{X}$. We may then remove $\mathsf{P}(F)[c(u)]$ similarly. If $u \in \mathcal{F}$ is inserted with label $\lambda(u)$, then we may similarly perform $\mathsf{split}(\mathsf{P}(\mathcal{F}), o(u))$, $\mathsf{split}(\mathsf{P}(\mathcal{F}), o(u)+1)$ and then $\mathsf{concat}(\mathsf{P}(F)[0\,..\,o(u)-1], \lambda(u))$, $\mathsf{concat}(\mathsf{P}(F)[0\,..\,o(u)-1] \cdot \lambda(u), \mathsf{P}(F)[o(u)\,..])$ to insert $\lambda(u)$ at $\mathsf{P}(F)[o(u)]$. We may insert $\lambda(u)$ at $\mathsf{P}(F)[c(u)]$, similarly. By Theorem 4.4.21, these steps take time $\mathcal{O}(\log n)$ since we are using $\mathcal{O}(1)$ split and concat operations.

We now show how to maintain $\mathcal{P}(\mathcal{F}) \in \mathcal{X}$. For a dynamically deleted node $u \in \mathcal{F}$, we may remove it from $\mathcal{P}(\mathcal{F})$ analogously to deletions from $\mathsf{P}(\mathcal{F})$ described above. For a dynamically inserted node $u \in \mathcal{F}$, we need to find a unique integer identifier for its light subtree that we may use as its label in $\mathcal{P}(\mathcal{F})$. To find the parenthesis representation of the light subtree of u, we just take its subtree in $\mathcal{P}(F)$ and replace its heavy child v with a dummy character #. We perform $\mathsf{split}(\mathsf{P}(F), o(u))$ and then on the resulting string $\mathsf{P}(F)[o(u)\,..)$, perform $\mathsf{split}(\mathsf{P}(F)[o(u)\,..], o(v))$ and $\mathsf{split}(\mathsf{P}(F)[o(u)\,..\,n], c(v)+1)$. We obtain $\mathsf{P}(F)[o(u)\,..\,o(v))$ and $\mathsf{P}(F)[c(u)\,..\,c(v))$, which are the two strings corresponding to the light subtree of u. We then can concatenate the two substrings together with an additional # between them to obtain the parenthesis representation of the light subtree of u, which may done with two concat operations. If u is a light node, we also concatenate its heavy depth as well as per Definition 4.4.1. We let the unique integer identifier for the string corresponding to the light subtree of u represent the label of u in $\mathcal{P}(\mathcal{F})$ in $\mathcal{X}$. We then insert the resulting integer identifier at positions $o(u)$ and $c(u)$ in $\mathcal{P}(\mathcal{F})$ analogously to insertions into $\mathsf{P}(F)$. Note that the above steps again take time $\mathcal{O}(\log n)$.

Now, a dynamic edit also affects the labels of more than just the edited node itself in $\mathcal{P}(\mathcal{F})$. First, by Observation 4.4.2, we know that u appears in the label of $\mathcal{O}(\log n)$ parentheses in $\mathcal{P}(\mathcal{F})$, which are precisely those parentheses corresponding to parents of light node ancestors of u. By Lemma 4.4.20, we may obtain the set of light node ancestors of v and recompute the light subtree

label of its parent analogously to the dynamic insertion steps above.

Furthermore, some nodes may change from heavy to light and light to heavy during a dynamic edit. By Lemma 4.3.3, we may maintain the heavy-light decomposition of $\mathcal{F}$ and $\mathcal{G}$ in $\mathcal{O}(\log^5)$-time. For each such node u that changes from heavy to light or light to heavy, we update the data structure of Lemma 4.4.20 accordingly in $\mathcal{O}(\log^2 n)$-time by Lemma 4.4.20 and reconstruct the light subtree label of its parent and add the resulting label to $\mathcal{P}(\mathcal{F})$ in $\mathcal{X}$ as done above. In total, this can be done in $\mathcal{O}(\log^7 n)$ time, and so each update to $\mathcal{P}(\mathcal{F})$ can be maintained in poly-logarithmic time.

Finally, we use the dynamic string edit distance algorithm of Theorem 4.3.6 to compute $\mathsf{ed}(\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G}))$, which is our approximation of $\mathsf{ted}(\mathcal{P}(\mathcal{F}), \mathcal{P}(\mathcal{G}))$. The approximation factor follows from Theorem 4.3.6, Lemma 4.4.19, Lemma 4.4.4, and the update time is dominated by Theorem 4.3.6. □

4.5 k versus $k^2 \log n$ Approximate Tree Edit Distance

We begin by giving a new static approximation algorithm for tree edit distance. Given forests $|\mathcal{F}, \mathcal{G}$ that we wish to approximate the distance of, the following approximation algorithm maintains a collection of *pieces* of $\mathcal{F}$ and iteratively checks if each piece has a corresponding occurrence in $\mathsf{P}(\mathcal{G})$. A *piece* of $\mathsf{P}(\mathcal{F})$ may be either a *forest* $\mathsf{P}(\mathcal{F})[i\,..\,j)$ which is a balanced parenthesis string, or a *context* $(\mathsf{P}(\mathcal{F})[i_1\,..\,j_1), \mathsf{P}(\mathcal{F})[i_2\,..\,j_2))$ with $i_1 < j_1 < i_2 < j_2$ which is a pair of substrings such that $\mathsf{P}(\mathcal{F})[i_1\,..\,j_2)$ is a tree and $\mathsf{P}(\mathcal{F})[j_1\,..\,i_2)$ is itself a forest. A context may be thought of as a subtree of a node $w \in \mathcal{F}$ corresponding to indices $o(w) = i_1$ and $c(w) = j_2 - 1$ excluding, for some descendant u of w, a contiguous set of children of u and their subtrees. We call $\mathsf{P}(\mathcal{F})[i_1\,..\,j_1)$

the *left* part and $\mathsf{P}(\mathcal{F})[i_2\,..\,j_2)$ the *right* part. See Figure 4.3 for an example. For a given forest piece $\mathsf{P}(\mathcal{F})[i_1\,..\,j_1]$, we define $|\sigma| := j - i$, and for a given context piece $(\mathsf{P}(\mathcal{F})[i_1\,..\,j_1), \mathsf{P}(\mathcal{F})[i_2\,..\,j_2))$, we define $|\sigma| := j_1 - i_1 + j_2 - i_2$.

In each iteration of the approximation algorithm, a single piece of the collection is checked for a corresponding occurrence in $\mathsf{P}(\mathcal{G})$ up to a potential left or right shift in index of up to size k, and if an occurrence is found successfully, we remove that piece from the collection. If no occurrence is found, it is partitioned into a constant number of new pieces of size smaller by a constant factor, which are added back into the collection. At any step, if a considered piece of our collection is of size $\mathcal{O}(k)$, we may remove it from the collection and move on to the next piece. We will argue that only $k \log |\mathcal{F}|$ iterations of this procedure is enough to distinguish whether $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq O(k)$ or $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \geq \mathcal{O}(k^2 \log |\mathcal{F}|)$. Since we only consider $\mathcal{O}(k \log |\mathcal{F}|)$ pieces in total, we know that ignoring pieces of size $\mathcal{O}(k)$ only incurs potentially $\mathcal{O}(k^2 \log |\mathcal{F}|)$ total edits. If $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq k$, we will show that all k edits will fall into pieces of size at most $\mathcal{O}(k)$ after $\mathcal{O}(k \log |\mathcal{F}|)$ steps, and so at the end of the algorithm, the collection of pieces will be empty in this case. Furthermore, the set of pairs of matched pieces found by the algorithm can be complemented by a set of deletions for any unmatched pieces to form a valid tree alignment of distance $\mathcal{O}(k^2 \log n)$, which we discuss in the proof of Theorem 4.5.1 below. If the collection of pieces is non-empty, this implies instead that $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \geq k^2 \log |\mathcal{F}|$.

An important subroutine in our algorithm will be finding corresponding pieces in $\mathsf{P}(\mathcal{G})$ for each piece of $\mathsf{P}(\mathcal{F})$. We refer to this subroutine as `HasMatch` in the following pseudocode. For a given forest piece $\sigma = \mathsf{P}(F)[i^{\mathcal{F}}\,..\,j^{\mathcal{F}})$ of $\mathcal{F}$, $\texttt{HasMatch}(\sigma)$ returns true if there exists $i^{\mathcal{G}}\,..\,j^{\mathcal{G}}$ such that $\mathsf{P}(\mathcal{F})[i^{\mathcal{F}}\,..\,j^{\mathcal{F}}) = \mathsf{P}(\mathcal{G})[i^{\mathcal{G}}\,..\,j^{\mathcal{G}})$ and $|i_{\mathcal{F}} - i_{\mathcal{G}}| \leq k$, otherwise it returns false. For a given context piece $\sigma_{\mathcal{F}} = (\mathsf{P}(\mathcal{F})[i_1^{\mathcal{F}}\,..\,j_1^{\mathcal{F}})\,..\,\mathsf{P}(\mathcal{F})[i_2^{\mathcal{F}}\,..\,j_2^{\mathcal{F}}))$, $\texttt{HasMatch}(\sigma_{\mathcal{F}})$ with $j_1^{\mathcal{F}} -$

$i_1^{\mathcal{F}} \geq 2k$ and $j_2^{\mathcal{F}} - i_2^{\mathcal{F}} \geq 2k$ returns true if there exists indices $i_1^{\mathcal{G}}, i_2^{\mathcal{G}}, j_1^{\mathcal{G}}, j_2^{\mathcal{G}}$ such that $\sigma_{\mathcal{G}} = (\mathsf{P}(\mathcal{G})[i_1^{\mathcal{G}} \,..\, j_1^{\mathcal{G}}), \mathsf{P}(\mathcal{G})[i_2^{\mathcal{G}} \,..\, j_2^{\mathcal{G}}))$ is a context with $\sigma_{\mathcal{F}} = \sigma_{\mathcal{G}}$ and $|i_1^{\mathcal{F}} - i_1^{\mathcal{G}}|, |i_2^{\mathcal{F}} - i_2^{\mathcal{G}}| \leq k$, otherwise it returns false. For a context piece with only one of either $j_1^{\mathcal{F}} - i_1^{\mathcal{F}} \geq 2k$ or $j_2^{\mathcal{F}} - i_2^{\mathcal{F}} \geq 2k$, i.e. only the left or right part of $\sigma_{\mathcal{F}}$ has size at least $2k$, we return true if the large part has a match in $\mathcal{G}$ up to a shift of size k (and we ignore the smaller part since it does not affect the approximation factor).

In the following algorithm, we assume $M = O(1)$ and $C < 1$ is a constant.

Algorithm 18: $O(k^2 \log n)$-approximate Tree Edit Distance

```
S ← P(F) ;
for i = 0 to (C + 1)k log_C |F| do
    σ ← S[0];
    if |σ| ≤ 4k then
        Remove σ from S;
    else if HasMatch(σ) then
        Remove σ from S;
    else
        Partition σ into at most M pieces σ_1, σ_2, ..., σ_M of size at most C|σ|;
        Add σ_1, σ_2, ..., σ_M to S;
if |S| == 0 then
    Return Yes;
else
    Return No;
```

Theorem 4.5.1. *Given forests $\mathcal{F}$ and $\mathcal{G}$ such that either* $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq k$ *or* $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \geq k^2 \log |\mathcal{F}|$, *Algorithm 18 outputs* Yes *if* $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq k$ *and* No *if* $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \geq k^2 \log |\mathcal{F}|$.

Proof. First, we consider when $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq k$ and show that Algorithm 18 always outputs YES. Let $n = |\mathcal{F}|$. Consider a tree in which each node of the tree corresponds to one of the pieces processed by Algorithm 18. Any internal node corresponding to a piece σ is the parent of at most M children whose corresponding pieces $\sigma_1, \sigma_2, \ldots, \sigma_M$ partition σ and each have size at most

$C|\sigma|$. Any leaf node corresponds to a piece σ such that either $|\sigma| \leq 4k$ or $\texttt{HasMatch}(\sigma)$ is true. Note that each parenthesis in $\mathsf{P}(\mathcal{F})$ appears in the piece of at most one node per level in the tree. Since $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq k$, there at most k parentheses undergoing edits in either $\mathcal{F}$ or $\mathcal{G}$, and so, at each level of the tree, at most k pieces will be unmatched and potentially be partitioned further. Furthermore, the size of a piece at level i is at most $\frac{n}{C^i}$, so there are at most $\log_C \frac{n}{k}$ levels before all pieces are smaller than $4k$ and removed from S. In total, we have at most $k \cdot \log_C(\frac{n}{k})$ internal nodes and therefore, a total of at most $(C+1) \cdot k \log_C(\frac{n}{k})$ that nodes in the entire tree. In other words, after at most $(C+1) \cdot k \log_C(\frac{n}{k})$ pieces are processed, Algorithm 18 will terminate with S empty when $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq k$. Since our algorithm runs for exactly $(C+1)k \cdot \log_C n$ iterations, the algorithm will return Yes.

Now, we consider when Algorithm 18 outputs Yes, and we show that in this case, $\mathsf{ted}(|\mathcal{F}, \mathcal{G}) \leq \mathcal{O}(k^2 \log n)$. Even better, we will explicitly construct a sequence of pairs $\mathcal{A}$, which will be a tree alignment between $\mathsf{P}(\mathcal{F})$ and $\mathsf{P}(\mathcal{G})$ with cost $\mathcal{O}(k^2 \log n)$.

Observe that when Algorithm 18 outputs Yes, S must be empty, and so every parenthesis of $\mathcal{F}$ must have been part of a removed piece $\sigma_{\mathcal{F}}$ such that either $|\sigma_{\mathcal{F}}| \leq 2k$ or $\texttt{HasMatch}(\sigma_{\mathcal{F}})$ is true. If $\texttt{HasMatch}(\sigma_{\mathcal{F}})$ is true, there must be some $\sigma_{\mathcal{G}}$ that is a subforest or context of $\mathcal{G}$ such that $\sigma_G = \sigma_{\mathcal{F}}$ with starting index or indices matching that of $\sigma_{\mathcal{F}}$ up to a shift of k in either direction. We add a sequence of pairs to match $\sigma_{\mathcal{F}}$ and $\sigma_{\mathcal{G}}$, excluding the first and last $2k$ parentheses and their matches.

Formally, if $\sigma_{\mathcal{F}}$ is a forest $\mathsf{P}(\mathcal{F})[i^{\mathcal{F}} \,..\, i^{\mathcal{F}}]$ with match $\mathsf{P}(\mathcal{G})[i^{\mathcal{G}} \,..\, j^{\mathcal{G}}]$, we add $(i^{\mathcal{F}} + 2k, i^{\mathcal{G}} + 2k), (i^{\mathcal{F}} + 2k + 1, i^{\mathcal{G}} + 2k + 1), (i^{\mathcal{F}} + 2k + 2, i^{\mathcal{G}} + +2k2), \ldots, (j^{\mathcal{F}} - 2k, j^{\mathcal{G}} - 2k), (j^{\mathcal{F}} - 2k, j^{\mathcal{G}} - 2k)$ to $\mathcal{A}$. Note by Definition 4.2.6, we must careful that all nodes $u \in \mathcal{F}$ have both $o(u)$ and $c(u)$ either deleted or both aligned to some $o(v)$ and $c(v)$ for $v \in \mathcal{G}$. Since $\sigma_{\mathcal{F}}$ is a forest, there are no nodes

$u \in \mathcal{F}$ such that $i \leq o(u) < j < c(u)$, i.e., all parentheses in $\sigma_{\mathcal{F}}$ have their corresponding twin in $\sigma_{\mathcal{F}}$ as well. Thus, the only issue we must be careful about is if the opening parenthesis $o(u)$ of a node $u \in \mathcal{F}$ is in the first $2k$ parentheses of $\sigma_{\mathcal{F}}$ and therefore excluded from the added matching sequence, but the corresponding twin $c(u)$ is contained in the matching sequence. In this case, we may replace the pairs matching $c(u)$ with two deletions. Specifically, $(c(u), c(v)), (c(u)+1, c(v)+1)$ is replaced with $(c(u), c(v)), (c(u)+1, c(v)), (c(u)+1, c(v)+1)$, which deletes $c(u)$ and $c(v)$. Later, we will add deletions for $o(u)$ and $o(v)$ after adding all matching sequences to satisfy Definition 4.2.6. We make the analogous modifications for the symmetric case when $c(u)$ is in the last $2k$ parentheses of $\sigma_{\mathcal{F}}$ and excluded from the matching sequence but $o(u)$ is not excluded.

If $\sigma_{\mathcal{F}}$ is a context $(\mathsf{P}(\mathcal{F})[i_1^{\mathcal{F}} \,..\, j_1^{\mathcal{F}}], \mathsf{P}(\mathcal{F})[i_2^{\mathcal{F}}, j_2^{\mathcal{F}}])$ with match $(\mathsf{P}(G)[i_1^{\mathcal{G}} \,..\, j_1^{\mathcal{G}}], \mathsf{P}(G)[i_2^{\mathcal{G}} \,..\, j_2^{\mathcal{G}}])$, similarly we add $(i_1^{\mathcal{F}}+2k, i_1^{\mathcal{G}}+2k), (i_1^{\mathcal{F}}+2k+1, i_1^{\mathcal{G}}+2k+1), \ldots (j_1^{\mathcal{F}}-2k+1, j_1^{\mathcal{G}}-2k+1)$ to $\mathcal{A}$ if $j_1^{\mathcal{F}} - i_1^{\mathcal{F}} \leq 2k$ and $(i_2^{\mathcal{F}}+2k, i_2^{\mathcal{G}}+2k), (i_2^{\mathcal{F}}+2k+1, i_2^{\mathcal{G}}+2k+1), \ldots, (j_2^{\mathcal{F}}-2k+1, j_2^{\mathcal{G}}-2k+1)$. We additionally modify this sequence to add deletions for any matched parentheses whose twins are not matched as in the forest case above. Furthermore, recall that if the left or right part of $\sigma_{\mathcal{F}}$ has size less than $2k$, we do not find a match for it in the `HasMatch` routine. In this case, we add a matching sequence for only the part of $\sigma_{\mathcal{F}}$ which has size at least $2k$.

Observe that after our above steps, any left index x of a pair $(x, y) \in \mathcal{A}$ with $0 \leq x \leq |\mathcal{F}|$ is added to $\mathcal{A}$ at most once since when a piece of $\mathcal{F}$ has a match in $\mathcal{G}$, it is removed from S for the remainder of Algorithm 18. Any parentheses contained in a matched piece will never be matched in a future piece. Furthermore, any right index y of a pair $(x, y) \in \mathcal{A}$ is similarly added to $\mathcal{A}$ at most once. Consider a second pair $(\sigma_2^{\mathcal{F}}, \sigma_2^{\mathcal{G}})$ of matched pieces handled by the algorithm. First, since $\sigma_1^{\mathcal{F}}, \sigma_2^{\mathcal{F}}$ are both matched, they must have no overlap in $\mathsf{P}(\mathcal{F})$, and second, according to the definition of `HasMatch`, since we only allow shifts of up to k, their corresponding pieces in $\mathcal{G}$ can

only overlap in at most $2k$ characters. Since we are careful to skip the first and last $2k$ indices in our matching sequences added to $\mathcal{A}$, these right indices will never be added twice.

To finish our alignment $\mathcal{A}$, if $(0,0)$ or $(|\mathcal{F}|,|\mathcal{G}|)$ is not already included in $\mathcal{A}$, we add these pairs to the beginning or end of $\mathcal{A}$, respectively. Finally, we add deletions for any remaining unmatched indices of $\mathcal{F}$ and $\mathcal{G}$. Formally, for any adjacent pairs $(x_1,y_1),(x_2,y_2)$ in $\mathcal{A}$ with $x_2 - x_1 > 1$ ($y_2 - y_1 > 1$) we add sequence $(x_1+1,y_1,x_1+2,y_1),\ldots,(x_1,y_2)$ ($(x_1,y_1+1,x_1,y_1+2),\ldots,(x_1,y_2)$) to $\mathcal{A}$ between these pairs. Deletions of indices in $\mathcal{F}$ correspond exactly to the pieces σ with $|\sigma| \leq k$ and the $2k$ beginning and ending indices per matched piece. Since there are $(C+1)k\log_C n$ such pieces, we have at most $\mathcal{O}(k^2\log_C n)$ deletions in $\mathcal{F}$. By choosing $\mathcal{F}$ and $\mathcal{G}$ such that $|\mathcal{F}| \geq |\mathcal{G}|$, there similarly must be at most $\mathcal{G} - \mathcal{F} + \mathcal{O}(k^2\log_C n) \leq \mathcal{O}(k^2\log_C n)$ deletions of indices in $\mathcal{G}$. Thus, $\mathcal{A}$ is a tree alignment with cost $\mathcal{O}(k^2\log_C n)$. □

4.5.1 $\tilde{\mathcal{O}}(k)$-time implementation

We now discuss how Algorithm 18 can be implemented in $\tilde{\mathcal{O}}(k)$-time in the dynamic setting. The main two challenges we consider are: 1) the partitioning of a piece of forest $\mathcal{F}$ into $O(1)$ pieces with size at most a constant fraction of the original piece size 2) the `HasMatch` routine.

To perform some useful basic operations on trees in the dynamic setting, we utilize the data structure of [Navarro and Sadakane, 2014]. This dynamic tree data structure supports poly-logarithmic dynamic node insertion and deletion and can be used to answer a number of common query types that we list in the following. We assume, as in [Navarro and Sadakane, 2014], that forest $\mathcal{F}$ is stored in its parenthesis string form $\mathsf{P}(\mathcal{F})$ and dynamic updates occur on this string.

Theorem 4.5.2 (Theorem 2 and Table 1 of [Navarro and Sadakane, 2014])**.** *Given an ordered tree*

T with n nodes the following operations can be carried out with $2n + \mathcal{O}(n \log\log n / \log n)$ bits:

- $\mathsf{insert}(i, j)$*, insert a node at indices i and j in $\mathsf{P}(\mathcal{F})$ in time $\mathcal{O}(\log n / \log\log n)$.*
- $\mathsf{delete}(v)$*, delete a node v, in time $\mathcal{O}(\log n / \log\log n)$.*
- $\mathsf{depth}(v)$*, return the depth of node v in time $\mathcal{O}(\log n / \log\log n)$.*
- $\mathsf{LAQ}(v, d)$, level ancestor queries*, which returns the ancestor u of v with $depth(u) = depth(v) - d$ in time $\mathcal{O}(\log n)$.*
- $\mathsf{LCA}(u, v)$, lowest common ancestor*, which returns the lowest common ancestor of nodes u and v in time $\mathcal{O}(\log n / \log\log n)$.*
- $\mathsf{parent}(v)$*, which returns the parent of node v in time $\mathcal{O}(\log n / \log\log n)$.*

We will create an instance of the above data structure for each tree of forests $\mathcal{F}$ and $\mathcal{G}$ and use the insert and delete operations to maintain these instances when a dynamic update occurs in $\mathcal{F}$ and $\mathcal{G}$ at the cost of only $\mathcal{O}(\log n / \log\log n)$-time per update.

In addition to the above operations, for a forest $\mathcal{F}$ and node u, we will often want to know $|\mathcal{F}(u)|$, the size of the subtree rooted at u (in the parenthesis representation of $\mathcal{F}$). By Observation 4.2.5, it is easy to see that $\mathsf{P}(F)[o(u)\,..\,c(u)] = |\mathcal{F}(u)|$, and so we assume we have constant-time access to these sizes.

4.5.1.1 Partitioning Forest and Context Pieces

In order to partition a forest piece σ of S, we find a central node u contained in a tree $T \in \sigma$ such that the following four sets all have size at most $n/2$:

- siblings to the left of u and their subtrees,
- siblings to the right of u and their subtrees,

- the subtree of of u,
- all remaining nodes.

We must be careful with the fourth set, the remaining nodes, as they may not be either a forest nor context. To handle this, we further separate the remaining nodes into three more additional sets:

- the forest to the left of T,
- the forest to the right of T,
- the context formed of the remaining nodes of T not yet in a piece.

We will later show that each part of the decomposition has size at most $|\sigma|/2$, and show that with some additional work, a similar decomposition works for context pieces as well. We utilize the data structure of [Navarro and Sadakane, 2014] to answer two additional query types to help us find our key node u.

Lemma 4.5.3. *Given a forest $\mathcal{F}$ with $n = |\mathcal{F}|$, a subforest $\mathsf{P}(\mathcal{F})[i\,..\,j)$, and node v with $i \leq o(v) < c(v) < j$, the highest ancestor u of v satisfying $i \leq o(u) < c(u) < j$ can be found in time $\mathcal{O}(\log^2 n/\log\log n)$.*

Proof. To find u from node v, we utilize a binary search on the path from the root of the tree T containing v. First, using the data structure of Theorem 4.5.2, we get the depth d of v and perform a level ancestor query $\mathsf{LAQ}(v, d/2)$. Let v' be the returned node of this query and let v'' denote its parent if it exists.

If $o(v') < i$, v' is outside of the subforest $\mathsf{P}(F)[i\,..\,j]$ and so, we continue the search recursively moving back towards v, i.e., we would next query $\mathsf{LAQ}(v, 3d/4)$ in order to find an ancestor in the target subforest. Otherwise, we continue the search recursively moving towards the root of T. In this case, we would next query $\mathsf{LAQ}(v, d/4)$. If we always traverse towards the root and

eventually reach the root itself without stopping in one of our earlier cases, we return the root as the highest ancestor of v in $\mathsf{P}(F)[i, j]$. Otherwise, we must have found our target node on the root to v path and will return that node instead from one of the above cases.

Since we halve our search range every iteration, our algorithm will finish in $\mathcal{O}(\log n)$ iterations and by Theorem 4.5.2, each iteration takes time $\mathcal{O}(\log n / \log \log n)$ for a total time of $\mathcal{O}(\log^2 n / \log \log n)$. □

Lemma 4.5.4. *Given a forest $\mathcal{F}$ with $n = |\mathcal{F}|$, subforest $\mathsf{P}(F)[i \,..\, j)$, a threshold m, and node v with $i \leq o(v) < c(v) < j$, the highest ancestor u of v satisfying $|\mathcal{F}(u)| \leq m$ and $i \leq o(u) < c(u) < j$ can be found in time $\mathcal{O}(\log^2 n / \log \log n)$ if it exists.*

Proof. To find u from node v, we utilize a binary search on the path from the root of the tree T containing v. First, using the data structure of Theorem 4.5.2, we get the depth d of v and perform a level ancestor query $\mathsf{LAQ}(v, d/2)$. Let v' be the returned node of this query and let v'' denote its parent if it exists.

If $o(v') < i$, v' is outside of the subforest $\mathsf{P}(F)[i \,..\, j]$ and so, we continue the search recursively moving back towards v, i.e., we would next query $\mathsf{LAQ}(v, 3d/4)$ in order to find an ancestor in the target subforest. Similarly, if $|\mathcal{F}(v')| > m$, we continue the search recursively towards v since the size of the subtree of v' is too large. Instead, if $|\mathcal{F}(v')| \leq m$ and $o(v'') < i$, then we return v', the highest ancestor of v in $\mathsf{P}(F)[i, j)$. If $|\mathcal{F}(v')| \leq m$ and $|\mathcal{F}(v'')| > m$, we return v' as it satisfies the lemma requirements. Finally, $|\mathcal{F}(v')| \leq m$ and $|\mathcal{F}(v'')| \leq m$, we continue the search recursively moving towards the root of T. In this case, we would next query $\mathsf{LAQ}(v, d/4)$. If we always traverse towards the root and eventually reach the root itself without stopping in one of our earlier cases, we return the root as the highest ancestor of v in $\mathsf{P}(F)[i \,..\, j]$. Otherwise, we

must have found our target node on the root to v path and will return that node instead from one of the above cases. If we finish our search and do not find u, we return nothing.

Since we halve our search range every iteration, our algorithm will finish in $\mathcal{O}(\log n)$ iterations and by Theorem 4.5.2, each iteration takes time $\mathcal{O}(\log n/\log\log n)$ for a total time of $\mathcal{O}(\log^2 n/\log\log n)$. □

We now give the formal piece partition statement.

Lemma 4.5.5. *Given forest $\mathcal{F}$ and either forest piece $\sigma = \mathsf{P}(F)[i\,..\,j)$ or context piece $\sigma = (\mathsf{P}(F)[i_1\,..\,j_1), \mathsf{P}(F)[i_2\,..\,j_2))$, we may partition σ into at most 8 pieces each of size at most $\lceil|\sigma|/2\rceil$ in time $\mathcal{O}(\log^2 n/\log\log n)$.*

Proof. We begin with the treatment for forest pieces $\sigma = \mathsf{P}(F)[i\,..\,j)$. Let $m = \lceil\frac{j-i}{2}\rceil$. First, we check index $m+i$. We assume $\mathsf{P}(F)[m+i]$ is an opening parenthesis, otherwise the following steps may be done symmetrically in the other direction. Let v be the node of $\mathcal{F}$ corresponding to $\mathsf{P}(F)[m+i]$. Since v is the node corresponding to the midpoint of σ, by Lemmas 4.5.3 and 4.5.4, we may find ancestors w, the highest ancestor of v in $\mathsf{P}(F)[i\,..\,j)$ and u, the highest ancestor of v in $\mathsf{P}(F)[i\,..\,j)$ such that $|\mathcal{F}(u)| \leq m$ if it exists, in time $\mathcal{O}(\log^2 n/\log\log n)$. Additionally, by Theorem 4.5.2, we may find the parent u' of u if it exists in time $\mathcal{O}(\log n/\log\log n)$. We now list the six parts we partition σ into and then discuss their size. We utilize the following observation that since u is an ancestor of v, we have that $o(u) \leq o(v) = m < c(v) \leq c(v)$.

1. If u exists, the first part is $\mathsf{P}(F)[o(u)\,..\,c(u)]$, the subtree rooted at node u. By Lemma 4.5.4, this part must have size at most m.
2. If u exists and $w \neq u$, the second part is $\mathsf{P}(F)(o(u')\,..\,o(u))$, the forest composed of subtrees rooted at siblings of u to the left of u. Since $w \neq u$ and w is the highest ancestor of v

satisfying $i \leq o(w)$, then we have that $i \leq o(w) \leq o(u')$. Furthermore, we know that $o(u) \leq o(v) = m + i$, and so the size of this part is bounded by $o(u) - o(u') \leq (m + i) - i$ = m.

3. If u exists and $w \neq u$, the third part is $\mathsf{P}(F)(c(u)\,..\,c(u'))$, the forest composed of subtrees rooted at siblings of u to the right of u. By a symmetric argument as to the previous part, the size of this part is bounded by m.
4. If u exists and $w \neq u$, the fourth part is $(\mathsf{P}(F)[o(w)\,..\,o(u')], \mathsf{P}(F)[c(u')\,..\,c(w)])$, the context rooted at w excluding the subtree of u'. By Lemma 4.5.4, $|\mathcal{F}(u')| > m$ and so, this part can have size at most $j - i - m \leq m$.
5. The fifth part is $\mathsf{P}(F)[i\,..\,o(w))$, the forest composed of subtrees rooted at siblings of w to the left of w. By the same reasoning as the fourth part, the size of this part is bounded by m.
6. The sixth part is $\mathsf{P}(F)(c(w)\,..\,j)$, the forest composed of subtrees rooted at siblings of w to the right of w. By the same reasoning as the fourth part, the size of this part is bounded by m.

The above partitions $\mathsf{P}(F)[i\,..\,j)$ into six pieces of size at most m, which is half of the original forest piece, in time $\mathcal{O}(\log^2 n / \log\log n)$. See Figure 4.4 for an example of this partition scheme.

Now we show how to partition a context piece $\sigma = (\mathsf{P}(F)[i_1\,..\,j_1), \mathsf{P}(F)[i_2\,..\,j_2))$. Let $m = (j_1 - i_1 + j_2 - i_2)/2$. We begin by finding node v, the parent of the subforest $\mathsf{P}(F)[j_1\,..\,i_2)$, which we may find using Theorem 4.5.2 by querying the parent of the node corresponding to $\mathsf{P}(F)[j_1]$. We will have two cases for partitioning contexts.

First, if $|\mathcal{F}(v)| - (i_2 - j) \leq m$ then we repeat the same partitioning as for forest pieces

using nodes w and u from Lemmas 4.5.3 and 4.5.4 in relation to v. The main change is that the first part, subforest $\mathsf{P}(F)[o(u)\,..\,c(u)]$ is replaced with context $(\mathsf{P}(F)[o(u)\,..\,j_1), \mathsf{P}(F)[i_2\,..\,c(u)])$ as we must still exclude the missing subforest $\mathsf{P}(F)[j_1\,..\,i_2)$. Also, we note that by the definition of contexts, parts five and six will be empty since a context contains nodes only in a single tree, and so σ will not contain any siblings of the root of this tree. Therefore, we will have 4 pieces total of size m.

If instead $|\mathcal{F}(v)| - (i_2 - j_1) > m$, we cannot repeat the same partitioning as already the subtree of v is too large to be its own part. Observe that only one of $|(o(v)\,..\,j_1)| > m$ or $|[i_2\,..\,c(v))| > m$ may be true. Without loss of generality, let us assume the former, $|(o(v)\,..\,j_1)| > m$. In this case, we set $|[i_2\,..\,c(v))|$ to be its own forest part of size at most m. We additionally set $(\mathsf{P}(F)[i_1\,..\,o(v)], \mathsf{P}(F)[c(v)\,..\,j_2))$ to be a context part of size at most $(j_1 - i_1 + j_2 - i_2) - (|\mathcal{F}[v]| - (i_2 - j_1)) \leq 2m - m = m$. Finally, with the remaining forest $\mathsf{P}(F)(o(v)\,..\,j_1)$, we partition it into 6 pieces in the same way we did in the above case for forest pieces. Therefore, we get 8 pieces total of size at most m in time $\mathcal{O}(\log^2 n/\log\log n)$. □

4.5.1.2 The HasMatch Routine

We now show how, given a forest or context piece σ in $\mathcal{F}$, we may find a matching piece in $\mathcal{G}$ if one exists up to a k-index shift.

We primarily will rely on the data structure of [Kempa and Kociumaka, 2022], which supports updates and *internal pattern matching queries* (IPM) in $\mathcal{O}(\log^{1+o(1)} n)$ time each. Given a pattern P and a text T, $\mathsf{IPM}(P, T)$ reports all occurrences of P in T. For $|T| < 2|P|$, all occurrences are reported as a single arithmetic progression of starting positions. Since `HasMatch` is

only interested in finding matches up to a $2k$-shift in either direction, in order to limit our search, we use the split operation so that we may cut out exactly the set of indices we need to perform an IPM query on.

Finding matches in $\mathcal{G}$ for forest pieces of $\mathcal{F}$ is straightforward applications of the previous theorem. Finding matches in $\mathcal{G}$ for context pieces of $\mathcal{F}$ requires additional work due to being two separate parts, which we cannot find a match for with a single IPM query. Instead, we use IPM queries for each part of a context piece, which return two separate arithmetic progressions for the starting indices of matches for each part. Using depth queries from the data structure of Theorem 4.5.2, we may rewrite each progression as an arithmetic progressions of the depths of the starting indices of matches for each part. Then, we find where these depths are equal between each progression and check if at such an equal depth these matches correspond to a single context piece in $\mathcal{G}$. We give the statement formally as follows.

Lemma 4.5.6. *Given forests $\mathcal{F}, \mathcal{G}$, and a piece σ of $\mathcal{F}$, $\texttt{HasMatch}(\mathcal{F})$ can be computed in time $\mathcal{O}(\log(|\mathcal{F}| + |\mathcal{G}|))$.*

Proof. We utilize an instance $\mathcal{X}$ of the dynamic strings data structure of [Kempa and Kociumaka, 2022]. We first consider when a piece σ that we pass to the `HasMatch` routine is a forest piece $\mathsf{P}(\mathcal{F})[i\,..\,j)$. In order to limit our search in $\mathcal{G}$ to just the starting index range $[i-2k\,..\,i+2k]$, we first perform two split operations on $\mathsf{P}(\mathcal{G})$ to retrieve $\mathsf{P}(\mathcal{G})[i-2k\,..\,j+2k)$, the substring of $\mathcal{G}$ that may contain a match for σ with a shift of up to $2k$. Finally, we query $\mathsf{IPM}(\sigma, \mathsf{P}(\mathcal{G})[i-2k\,..\,j+2k))$ using $\mathcal{X}$ and, if there is a match, output $\texttt{HasMatch}(\sigma) = \mathsf{true}$. By Theorems 4.4.21 and [Kempa and Kociumaka, 2022], this takes time $\mathcal{O}(\log(|\mathcal{F}| + |\mathcal{G}|))$ with high probability.

Now, if a piece σ that we pass to the `HasMatch` routine is a context

$\langle \mathsf{P}(\mathcal{F})[i_1 \,..\, j_1), \mathsf{P}(\mathcal{F})[i_2 \,..\, j_2) \rangle$, we need to check the left and right parts of the context separately and then find if any matches for each part correspond to a single context[8]. Now, using the same steps as for forest pieces, we may find an arithmetic progression for all occurrences of each part in $\mathcal{G}$ separately with a shift of size up to k using the dynamic strings data structure $\mathcal{X}$ to perform split and IPM operations. We may have multiple matches for the left and right parts separately, and we must check if any pair of these matches forms a single context, i.e., the unmatched opening parentheses in the left part correspond to the same nodes as the unmatched closing parentheses in the right part.

Since the string $\mathsf{P}(\mathcal{F})[i_1 \,..\, j_1)$ is of length more than $4k$, its occurrences within $\mathsf{P}(\mathcal{G})[i_1 - 2k \,..\, j_1 + 2k)$ overlap, and their starting positions form an arithmetic progression $i_1 + \delta_{1,1}, i_1 + \delta_{1,2}, \ldots, i_1 + \delta_{1,s}$. Moreover, since $\mathsf{P}(\mathcal{F})[i_1 \,..\, j_1)$ does not contain any unmatched closing parenthesis, the same must be true about $\mathsf{P}(\mathcal{G})[i_1 + \delta_{1,1} \,..\, j_1 + \delta_{1,s})$. Consequently, all potential matches for the root of the context (the opening parenthesis at position $\mathsf{P}(F)[i_1]$) must lie on a single root-to-leaf path of $\mathcal{G}$. Furthermore, the depths of these potential matches also form an arithmetic progression (whose difference is the number of unmatched opening parentheses in $\mathsf{P}(G)[i_1 + \delta_{1,1} \,..\, i_1 + \delta_{1,2}) = \ldots = \mathsf{P}(G)[i_1 + \delta_{1,s-1} \,..\, i_1 + \delta_{1,s})$. A symmetric argument shows that the possible matches for the root of the context based on the exact occurrences of the right part $\mathsf{P}(\mathcal{F})[i_2 \,..\, j_2)$ within $\mathsf{P}(\mathcal{G})[i_2 - 2k \,..\, j_2 + 2k)$ also lie on a single root-to-leaf path at depths constituting an arithmetic progression.

Let $\{a_1 \cdot x + b_1 : x \in \{0, \ldots, c_1\}\}$ be the arithmetic progression of the depths for the set of returned left part matches and let $\{a_2 \cdot x + b_2 : x \in \{0, \ldots, c_2\}\}$ be the arithmetic progression of

[8] As per the description of `HasMatch`, if either part has size at most $4k$, we ignore it and treat the remaining part as a forest piece.

the depths for the right part matches, which we may obtain using depth queries in $\mathcal{O}(\log |\mathcal{G}|)$-time with the data structure of Theorem 4.5.2. We intersect the two arithmetic progressions and, if the intersection is empty, return false. If the intersection is non-empty, we identify its minimum element and retrieve the corresponding occurrences $\mathsf{P}(G)[i'_1 \mathinner{..} j'_1)$ and $\mathsf{P}(G)[i'_2 \mathinner{..} j'_2)$ of $\mathsf{P}(\mathcal{F})[i_1 \mathinner{..} j_1)$ and $\mathsf{P}(\mathcal{F})[i_2 \mathinner{..} j_2)$, respectively. We return true if and only if $\mathsf{P}(G)[i'_1 \mathinner{..} j'_1)$ is balanced.

In this case, $\langle \mathsf{P}(G)[i'_1 \mathinner{..} j'_1), \mathsf{P}(G)[i'_2 \mathinner{..} j'_2) \rangle$ is indeed a valid context: since $\mathsf{P}(G)[i'_1]$ and $\mathsf{P}(G)[j'_2 - 1]$ are opening and closing, respectively, and correspond to nodes at the same level, the unmatched parentheses from $\mathsf{P}(G)[i'_1 \mathinner{..} j'_1)$ and $\mathsf{P}(G)[i'_2 \mathinner{..} j'_2)$ are all matched within $\mathsf{P}(G)[i'_1 \mathinner{..} j'_2)$.

It remains to prove that we have not lost a valid match by greedily taking the minimum depth. For this, suppose that there was valid match $\langle \mathsf{P}(G)[i''_1 \mathinner{..} j''_1), \mathsf{P}(G)[i''_2 \mathinner{..} j''_2) \rangle$ at a larger depth. In this case, $\mathsf{P}(G)[j''_1 \mathinner{..} i''_2)$ is balanced, and so is $\mathsf{P}(G)[i''_1 \mathinner{..} j''_2)$. Then $\mathsf{P}(G)[i'_1 \mathinner{..} j'_1)$ is sandwiched between the two, and hence also balanced: any hypothetical unmatched parenthesis within $\mathsf{P}(G)[i'_1 \mathinner{..} j'_1)$ must be matched within $\mathsf{P}(G)[i''_1 \mathinner{..} j''_2)$ but, since $\langle \mathsf{P}(G)[i''_1 \mathinner{..} j''_1), \mathsf{P}(G)[i''_2 \mathinner{..} j''_2) \rangle$ is a context, the unmatched parentheses of $\mathsf{P}(G)[i''_1 \mathinner{..} j'_1)$ are matched within $\mathsf{P}(G)[i'_2 \mathinner{..} j''_2)$, and vice versa. □

4.5.2 Dynamic Approximation Algorithm

The results above can be interpreted as a dynamic data structure supporting the following two operations:

Updates that modify $\mathcal{F}$ or $\mathcal{G}$ using a given edit operation, supported in $\mathcal{O}(\log^{1+o(1)} n)$ time. These updates simply maintain the dynamic strings data structure [Kempa and Kociumaka, 2022] and the dynamic forest data structure of Theorem 4.5.2.

Queries that, given an integer k, return Yes if $\mathsf{ted}(\mathcal{F},\mathcal{G}) \leq k$ and only if $\mathsf{ted}(\mathcal{F},\mathcal{G}) \leq Ck^2 \log n$ (for some explicit constant C). The query procedure works in $\mathcal{O}(k \log^{3+o(1)} n)$ time.

We now show how to apply these two procedures in order to maintain an approximate tree edit distance $\widetilde{\mathsf{ted}}(\mathcal{F},\mathcal{G})$ that, all times, satisfies $\mathsf{ted}(\mathcal{F},\mathcal{G}) \leq \widetilde{\mathsf{ted}}(\mathcal{F},\mathcal{G}) \leq \mathcal{O}(\mathsf{ted}(\mathcal{F},\mathcal{G}) \log n)$, and takes $\mathcal{O}(\log^{3+o(1)} n)$ worst-case time per update.

We always ask queries in succession for subsequent thresholds $k = 1, 2, 4, \ldots$ until the answer is Yes. The threshold κ for which this condition happens satisfies $\kappa/2 \leq \mathsf{ted}(\mathcal{F},\mathcal{G}) \leq C\kappa^2 \log n$, and it is computed in time at most $\mathcal{O}(\kappa \log^{3+o(1)} n)$. We call this an *extended query procedure*.

Our approximation algorithm maintains the aforementioned dynamic data structure, $\mathcal{D}$, which stores forests $\mathcal{F}$ and $\mathcal{G}$. The updates are not immediately applied to $\mathcal{D}$, though, but stored in a buffer B (implemented as a simple queue). We occasionally run the extended query procedure on $\mathcal{D}$ in small $\mathcal{O}(\log^{3+o(1)} n)$-time steps so that a value κ is guaranteed to be returned in no more than $\lceil 0.1\kappa \rceil$ steps. Furthermore, we store the result κ of the last completed execution of the extended query procedure, and the value b representing the number of updates (applied to $\mathcal{D}$ or still in the buffer) since that execution has been launched.

Initially, $\mathcal{D}$ stores two empty forests, the buffer B is empty, no extended query procedure is underway, $\kappa = 0$ (which is the tree edit distance of empty strings), and $b = 0$. At each update, we perform the following steps:

1. Buffer the incoming update and increment b.
2. If there is no extended query procedure underway:
    - Apply the first $\min(|B|, 2)$ updates from the buffer.

- If the buffer is now empty, launch a new extended query procedure (without running it yet).

3. If there is an extended query procedure underway (possibly just launched):
   - Run the extended query procedure for one step.
   - If the procedure terminates, update κ and set $b := |B|$.
4. Return $C \cdot \kappa^2 \log n + b$.

Lemma 4.5.7. *The returned value* $\widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G})$ *satisfies* $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq \widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G}) \leq \mathcal{O}(\mathsf{ted}(\mathcal{F}, \mathcal{G})^2 \log n)$.

Proof. The lower bound for $\widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G})$ is easy: the forests $\mathcal{F}^*$ and $\mathcal{G}^*$ from the moment when we launched the previous extended query procedure satisfy $\mathsf{ted}(\mathcal{F}^*, \mathcal{G}^*) \leq C\kappa^2 \log n$ as well as $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \leq \mathsf{ted}(\mathcal{F}^*, \mathcal{G}^*) + b$. Combining these two inequalities, we get $\mathsf{ted}(\mathcal{F}, G) \leq C\kappa^2 \log n + b = \widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G})$.

As for the upper bound, let us first note that $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \geq \mathsf{ted}(\mathcal{F}^*, \mathcal{G}^*) - b \geq 0.5\kappa - b$. When the extended query procedure terminates with value κ, it has been running for at most $\lceil 0.1\kappa \rceil$ steps, and thus the buffer contains at most 0.1κ unprocessed updates, that is, we have $b \leq 0.1\kappa$ once the procedure terminates. These updates are applied in the double speed, and thus $b \leq 0.2\kappa$ once the next extended procedure is launched.

Consequently, if the update algorithm returns $\widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G})$ when no extended query algorithm is underway or one has already been running for $s \leq 0.05\kappa$ steps, then $b \leq 0.25\kappa$, and hence $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \geq 0.5\kappa - 0.25\kappa = 0.25\kappa$. Thus, $\widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G}) = C\kappa^2 \log n + b \leq C\kappa^2 \log n + 0.25\kappa \leq 16C\mathsf{ted}(\mathcal{F}, \mathcal{G})^2 \log n + \mathsf{ted}(\mathcal{F}, \mathcal{G})$.

Otherwise, an extended query procedure is underway running for $s \geq 0.05\kappa$ steps already,

and then $b \leq 0.2\kappa + s \leq 5s$. Moreover, we know that a value $\kappa' \geq 10s$ will be returned by the procedure underway, and the instance $\mathcal{F}', \mathcal{G}'$ from its launch time satisfies $\mathsf{ted}(\mathcal{F}', \mathcal{G}') \geq \kappa'/2 = 5s$. Since s updates have been requested since the launch, we have $\mathsf{ted}(\mathcal{F}, \mathcal{G}) \geq \mathsf{ted}(\mathcal{F}', \mathcal{G}') - s \geq 4s$. Thus, $\widetilde{\mathsf{ted}}(\mathcal{F}, \mathcal{G}) = C\kappa^2 \log n + b \leq 400Cs^2 \log n + 5s \leq 25C\mathsf{ted}(\mathcal{F}, \mathcal{G})^2 \log n + 1.25\mathsf{ted}(\mathcal{F}, \mathcal{G})$. □

In order to derive Theorem 4.1.2, it suffices to observe that the procedure that we run at each update takes $\mathcal{O}(\log^{3+o(1)} n)$ time, dominating by the single step of the extended query algorithm.

4.6 Further Results on Approximating Dynamic Dyck Edit Distance

In this section, we present two algorithms that each return good approximate solutions for different regimes of Dyck edit distance. The first algorithm provides an approximation factor of $\mathcal{O}(f(\epsilon) \log |X|)$ in $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon}/\mathsf{ded}(X))$ update time where $f(\epsilon)$ is a function dependent only on a fixed parameter ϵ. The second algorithm gives an $\mathcal{O}(\log |X|)$-approximations in $\tilde{\mathcal{O}}(\mathsf{ded}(X) + 1)$ amortized update time. Thus, the first algorithm is preferable for $\mathsf{ded}(X) = \Omega(\sqrt{n})$ while the second algorithm is preferable for $\mathsf{ded} = \mathcal{O}(\sqrt{n})$. These algorithms can be combined to create a logarithmic-approximation algorithm for dynamic Dyck edit distance for any input string.

In this section, we will often consider approximate Dyck edit distance with deletions only. For a parenthesis string X, $\mathsf{ded_d}(X)$ refers to the *deletion-only* Dyck edit distance. To see why considering only deletions is still satisfactory for our purposes, observe that any substitution in a minimum sequence of edits occurs so that two parentheses can be matched together, and instead, we may replace the substitution by deleting the formerly matched pair of parentheses. Since we already lose a factor of $\mathcal{O}(\log n)$ in our approximation algorithms, we may handle doubling the

number of edits needed by disallowing substitutions without increasing our asymptotic approximation factor. As for insertions, any minimal sequence of edits with insertions and deletions has a corresponding sequence of edits with deletions only with the same number of total edits. This is because whenever a parenthesis is inserted into X to be matched with a pre-existing parenthesis in X, we can instead forego the insertion and simply delete the existing parenthesis in X to satisfy the requirements of a Dyck string.

4.6.1 Large Dyck Edit Distance

We first consider the case when the Dyck edit distance of the input string $X \in \Sigma^n$ is large, that is $k = \mathsf{ded}_{\mathsf{d}}(X)$ satisfies $k \geq \sqrt{n}$. Since k is so large and a single edit changes $\mathsf{ded}_{\mathsf{d}}(X)$ by at most ± 1, at a constant-factor loss in the approximation ratio, it suffices to update our approximation of $k = \mathsf{ded}_{\mathsf{d}}(X)$ every $\Theta(k)$ updates. As a result, we can afford to use a static approximation algorithm because its large cost amortizes over the $\Theta(k)$ updates.

The first tool we use is an $\mathcal{O}(\log n)$-approximate reduction from Dyck edit distance to string edit distance from [Koucký and Saks, 2023]. This technique is a large part of several of our later algorithms. We give some necessary definitions to understand this reduction. A string Y is called an *LR-string* if it can be decomposed into a prefix $\mathcal{L}(Y)$ consisting of opening parentheses and a suffix $\mathcal{R}(Y)$ consisting of closing parentheses, i.e. $Y = \mathcal{L}(Y) \cdot \mathcal{R}(Y)$. A *maximal LR-segment* of a string X is an LR-string that is a substring of X and cannot be extended to a larger LR-string. Given a string X, its maximal LR-strings determine a unique partition of X into disjoint substrings, the *LR-decomposition* of X.

We observe that finding the deletion-only Dyck edit distance of an LR-string X is equivalent

to finding the minimum number of edits needed to transform the maximal-length sequence of opening parentheses in X into the transpose of the maximal-length sequence of closing parentheses in X.

Observation 4.6.1. *Every LR-string X satisfies* $\mathsf{ded_d}(X) = \mathsf{ed_d}(\mathcal{L}(X), T(\mathcal{R}(X)))$.

Proof. Let X' be a longest subsequence of X such that $X' \in \mathsf{Dyck}$. Since X is an LR-string, X' must also be an LR-string and, furthermore, since $X' \in \mathsf{Dyck}$, it must be that $\mathcal{L}(X') = T(\mathcal{R}(X'))$ is a common subsequence of $\mathcal{L}(X)$ and $T(\mathcal{R}(X))$. Therefore, $\mathsf{ded_d}(X) = |X| - |X'| = |\mathcal{L}(X)| + |T(\mathcal{R}(X))| - 2|\mathcal{L}(X')| \leq \mathsf{ed_d}(\mathcal{L}(X), T(\mathcal{R}(X)))$. Similarly, let Z be a longest common subsequence of $\mathcal{L}(X)$ and $T(\mathcal{R}(X))$. Note that $Z \cdot T(Z)$ belongs to Dyck and is a subsequence of X. Consequently, $\mathsf{ed_d}(\mathcal{L}(X), T(\mathcal{R}(X)) = |\mathcal{L}(X)| + |T(\mathcal{R}(X))| - 2|Z| = |X| - |Z \cdot T(Z)| \leq \mathsf{ded_d}(X)$. □

We now state the reduction.

Theorem 4.6.2 (Theorem 7.1, [Koucký and Saks, 2023])**.** *There is a deterministic algorithm that, given a string X, in $\tilde{\mathcal{O}}(|X|)$ time outputs a collection $\mathcal{C}(X)$ of LR-strings such that:*

1. $\sum_{Y \in \mathcal{C}(X)} |Y| = |X|$, *and*
2. $\mathsf{ded_d}(X) \leq \sum_{Y \in \mathcal{C}(X)} \mathsf{ded_d}(Y) \leq (3 + 2\lg(\mathsf{ded_d}(X)))\, \mathsf{ded_d}(X)$.

The preceding theorem, for a parenthesis string X, outputs a collection $\mathcal{C}(X)$ of LR-strings which may be used to find a logarithmic-factor approximation of the Dyck edit distance of X. We will then use string edit distance algorithms to find the Dyck edit distance of each such LR-string according to Observation 4.6.1. The particular string edit distance approach we utilize will be as fast as possible while making sure not to add any super-logarithmic factors to the approximation ratio.

Theorem 4.6.3 ([Andoni and Nosatzki, 2020, Theorem 1.1]). *There exists a function $f : \mathbb{R}_+ \to \mathbb{R}_+$ and a randomized algorithm that, given two strings $X, Y \in \Sigma^{\leq n}$ and a parameter $\epsilon \in \mathbb{R}_+$, computes an $f(\epsilon)$-factor approximation of $\mathsf{ed_d}(X, Y)$ in $\mathcal{O}_\epsilon(n^{1+\epsilon})$ time correctly with high probability.*[9]

With the above techniques, the main result of this subsection follows.

Theorem 4.1.6. *There exists a randomized dynamic algorithm that, for a fixed parameter $\epsilon \in \mathbb{R}_+$, maintains an $\mathcal{O}(f(\epsilon) \log |X|)$-approximation of $\mathsf{ded}(X)$ (correctly with high probability against an adaptive adversary) for a string $X \in \Sigma^*$ undergoing edits. The algorithm has amortized update time $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon}/\mathsf{ded}(X))$ per edit and pre-processing time $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon})$.*

Proof. The algorithm proceeds in epochs. Each epoch starts with finding the collection $\mathcal{C}(X)$ using the algorithm of Theorem 4.6.2. Recall that each string $Y \in \mathcal{C}(X)$ is an LR-string and $\mathsf{ded_d}(Y) = \mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y)))$ holds by Observation 4.6.1. For each $Y \in \mathcal{C}(X)$, the second step is to approximate $\mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y)))$ using the algorithm of Theorem 4.6.3. Let a_Y represent the approximated edit distance found for each $Y \in \mathcal{C}(X)$; we are guaranteed that $\mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y))) \leq a_Y \leq f(\epsilon)\mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y)))$. We use $a := 2\sum_{Y\in\mathcal{C}(X)} a_Y$ as the approximation of $\mathsf{ded_d}(X)$ for the duration of the epoch, which lasts for $m := \lfloor a/(4f(\epsilon)(3+2\lg a))\rfloor$ dynamic edits. At the beginning of the epoch,

$$\begin{aligned} a = 2\sum_{Y\in\mathcal{C}(X)} a_Y &\geq 2\sum_{Y\in\mathcal{C}(X)} \mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y))) \\ &= 2\sum_{Y\in\mathcal{C}(X)} \mathsf{ded_d}(Y) \end{aligned}$$

[9] As discussed in [Andoni and Nosatzki, 2020], the value $f(\epsilon)$ is doubly exponential in $1/\epsilon$ and independent of any other variables. The subscript ϵ in $\tilde{\mathcal{O}}_\epsilon$ denotes hidden ϵ factors.

$$\geq 2 \cdot \mathsf{ded_d}(X) \geq 2 \cdot \mathsf{ded_d}(X)$$

and

$$\begin{aligned} a = 2 \sum_{Y \in \mathcal{C}(X)} a_Y &\leq 2 \sum_{Y \in \mathcal{C}(X)} f(\epsilon) \mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y))) \\ &= 2f(\epsilon) \sum_{Y \in \mathcal{C}(X)} \mathsf{ded_d}(Y) \\ &\leq 2f(\epsilon)(3 + 2\lg(\mathsf{ded_d}(X)))\mathsf{ded_d}(X) \\ &\leq 2f(\epsilon)(3 + 2\lg a)\mathsf{ded_d}(X) \\ &\leq 4f(\epsilon)(3 + 2\lg a)\mathsf{ded_d}(X), \end{aligned}$$

where, in both chains, the first inequality follows from Theorem 4.6.3 and the second inequality follows from Theorem 4.6.2.

Throughout the epoch, the value $\mathsf{ded_d}(X)$ may change by at most $\pm m$, so we still have $a \geq \frac{a}{2} + m \geq \mathsf{ded_d}(X) - m + m = \mathsf{ded_d}(X)$ and $a = 2a - a \leq 8f(\epsilon)(3 + 2\lg a)(\mathsf{ded_d}(X) + m) - a \leq 8f(\epsilon)(3 + 2\lg a)\mathsf{ded_d}(X) + 8f(\epsilon)(3 + 2\lg a)m - a \leq 8f(\epsilon)(3 + 2\lg a)\mathsf{ded_d}(X)$, so the value a remains an $\mathcal{O}(f(\epsilon)\log n)$-approximation of $\mathsf{ded_d}(X)$.

It remains to analyze the running time. The application of Theorem 4.6.2 takes $\tilde{\mathcal{O}}(|X|)$ time. The application of Theorem 4.6.3 takes $\mathcal{O}_\epsilon(|Y|^{1+\epsilon})$ time for each $Y \in \mathcal{C}(X)$. Across all $Y \in \mathcal{C}(X)$, this cost is proportional to

$$\sum_{Y \in \mathcal{C}(X)} |Y|^{1+\epsilon} \leq \sum_{Y \in \mathcal{C}(X)} |Y| \cdot |X|^{\epsilon} = |X|^{1+\epsilon}.$$

Furthermore, summing up each approximation for the edit distance of all strings in the LR-

decomposition of X takes at most $\mathcal{O}(|X|)$ time. Consequently, the total cost of each epoch is $\mathcal{O}_\epsilon(|X|^{1+\epsilon})$. Across $m = \Omega(\mathsf{ded}_\mathsf{d}(X)/\log|X|)$ updates within the epoch, this yields $\tilde{\mathcal{O}}(|X|^{1+\epsilon}/\mathsf{ded}_\mathsf{d}(X))$ amortized time per update. □

4.6.2 Small Dyck Edit Distance

To complement the previous dynamic approximate Dyck edit distance algorithm, we consider the case when the Dyck edit distance of an input string is small. Specifically, given an input string X with $|X| = n$, let $k = \mathsf{ded}_\mathsf{d}(X)$; we aim to match our approximation ratio and update time from the large edit distance case of the previous section when $k = \mathcal{O}(\sqrt{n})$ instead. To this end, we pre-process X such that all neighboring parentheses which may be matched together are removed. We denote the remaining string as $\hat{X}$, which has the same Dyck edit distance as X but is composed of $\mathcal{O}(k)$ maximal LR-segments instead of potentially $\mathcal{O}(n)$ segments. We are careful to maintain $\hat{X}$ as X is updated, potentially removing any new pairs of neighboring matching parentheses or even reintroducing any previously removed parentheses if their neighbor no longer matches after an update. We again use the Dyck to string edit distance reduction to find a collection of LR-segments that we use to approximate the Dyck edit distance of X. Then, we introduce a string edit distance algorithm of [Cole and Hariharan, 2002] to compute the edit distance of each segment in time $\mathcal{O}(k^2)$ each, which we then sum to find our approximation for $\mathsf{ded}_\mathsf{d}(X)$. In each step of the preceding process, we will rely on the dynamic strings data structure ([Gawrychowski et al., 2018]) to build a balanced binary tree on top of our input string X which allows us to efficiently construct, store, and query information of $\hat{X}$.

In addition to the listed operations in Theorem 4.4.21, we augment the data structure with

two additional operations to efficiently maintain $\hat{X}$ and the Dyck to string edit distance reduction. First, for any string $X \in \mathcal{X}$, the transpose of X, $T(X)$, may be queried and will also be stored in $\mathcal{X}$. Second, for any string $X \in \mathcal{X}$, we will answer $\mathsf{LMP}(X)$ queries (recall $\mathsf{LMP}(X)$ is the longest prefix of X composed of only opening or only closing parentheses) in $\mathcal{O}(\log n)$ time.

Lemma 4.6.4. *A collection $\mathcal{X}$ of parenthesis strings of total length n can be dynamically maintained subject to the following queries in addition to the operations of Theorem 4.4.21:*

$\mathsf{addT}(X)$ *for $X \in \mathcal{X}$ results in $\mathcal{X} = \mathcal{X} \cup \{T(X)\}$ and takes $\mathcal{O}(1)$ time,*

$\mathsf{LMP}(X)$ *for $X \in \mathcal{X}$ computes the length of the longest monotone prefix of X (consisting of opening or closing parentheses only) and takes $\mathcal{O}(\log n)$ time.*

Proof. Internally, we use Theorem 4.4.21 to maintain two collections of strings $\mathcal{X}_T = \mathcal{X} \cup \{T(X) : X \in \mathcal{X}\}$ and $\mathcal{X}_{\mathbf{O}} = \{\mathbf{O}(X) : X \in \mathcal{X}_T\}$, where $\mathbf{O}(X)$ is the *outline* of X, defined as a binary of string of length $|X|$ such that $\mathbf{O}(X)[i] = 0$ if $X[i]$ is an opening parenthesis and $\mathbf{O}(X)[i] = 1$ if $X[i]$ is a closing parenthesis for all $i \in [|X|]$.

Let us first describe how to maintain $\mathcal{X}_T$. For every operation $\mathsf{add}(X)$ on $\mathcal{X}$, we compute $T(X)$ in $\mathcal{O}(|X|)$ time and use both $\mathsf{add}(X)$ and $\mathsf{add}(T(X))$ on $\mathcal{X}_T$. For every operation $\mathsf{split}(X, k)$ on $\mathcal{X}$, we perform both $\mathsf{split}(X, k)$ and $\mathsf{split}(T(X), |X| - k)$ on $\mathcal{X}_T$ so that, besides $X[0\,..\,k)$ and $X[k\,..\,|X|)$, also $T(X[0\,..\,k)) = T(X)[|X| - k\,..\,|X|)$ and $T(X[k\,..\,|X|)) = T(X)[0\,..\,|X| - k)$ are also added to $\mathcal{X}_T$. For every operation $\mathsf{concat}(X_1, X_2)$ on $\mathcal{X}$, we perform both $\mathsf{concat}(X_1, X_2)$ and $\mathsf{concat}(T(X_2), T(X_1))$ on $\mathcal{X}_T$ so that, besides $X_1 \cdot X_2$, also $T(X_1 \cdot X_2) = T(X_2) \cdot T(X_1)$ is added to $\mathcal{X}_T$. The $\mathsf{LCP}(X_1, X_2)$ operation on $\mathcal{X}$ is simply performed on $\mathcal{X}_T$. It is easy to see that, with the above implementations, the asymptotic update and query times match those of Theorem 4.4.21. Moreover, the $\mathsf{addT}(X)$ operation on $\mathcal{X}$ does not require any update on $\mathcal{X}_T$: since

we already have $T(X) \in \mathcal{X}_T$, it suffices to return the handle to $T(X)$.

Next, we focus on maintaining $\mathcal{X}_{\mathbf{O}}$. When we perform $\mathsf{add}(X)$ on $\mathcal{X}_T$, we also straightforwardly compute $\mathbf{O}(X)$ in $\mathcal{O}(n)$-time and perform $\mathsf{add}(\mathbf{O}(X))$ on $\mathcal{X}_{\mathbf{O}}$. When we perform $\mathsf{split}(X, k)$ or $\mathsf{concat}(X_1, X_2)$ on $\mathcal{X}_T$, we also perform $\mathsf{split}(\mathbf{O}(X), k)$ or $\mathsf{concat}(\mathbf{O}(X_1), \mathbf{O}(X_2))$ on $\mathcal{X}_{\mathbf{O}}$, respectively. Again, it is easy to see that, with the above implementations, the asymptotic update and query times match those of Theorem 4.4.21. It remains to implement the $\mathsf{LMP}(X)$ query on $\mathcal{X}$. For this, we observe that

$$\mathsf{LMP}(X) = \mathsf{LCP}(\mathbf{O}(X), \mathbf{O}(X)[1\,..\,|X|)) + 1.$$

Consequently, we may perform $\mathsf{split}(\mathbf{O}(X), 1)$ in $\mathcal{O}(\log n)$-time and then using the above LCP query to find $\mathsf{LMP}(X)$ in $\mathcal{O}(\log n)$ total time. To make sure that $\mathcal{X}_O$ does not grow too much, we remove the outputs of $\mathsf{split}(\mathbf{O}(X), 1)$ as soon as the LCP query is completed. □

[Charalampopoulos et al., 2020c] and [Cole and Hariharan, 2002] show that an oracle that answers LCP queries in logarithmic time is sufficient to efficiently compute the string edit distance between two strings. These papers utilize a slightly modified implementation of the classic dynamic programming solution of [Landau and Vishkin, 1989] for string edit distance that only relies on LCP queries. We may utilize the dynamic strings data structure to answer these queries and solve an instance of string edit distance in quadratic runtime with respect to the edit distance between the two strings. Since in this section we are considering when the edit distance is at most $\mathcal{O}(\sqrt{n})$, we can afford a quadratic blow-up (and some additional log factors).

Lemma 4.6.5 ([Charalampopoulos et al., 2020c] Lemma 6.2, [Cole and Hariharan, 2002], Sec-

tion 5). *Given strings X_1, X_2, $k \in \mathbf{Z}^+$, and an oracle that answers* LCP *queries of any pair of substrings of X_1, X_2 in $\mathcal{O}(\log n)$ time, $\min\{\mathsf{ed}_\mathsf{d}(X_1, X_2), k\}$ can be computed in $\tilde{\mathcal{O}}(k^2)$ time.*

Proof. As noted in [Charalampopoulos et al., 2020c], the algorithm of Section 5 [Cole and Hariharan, 2002] can be utilized to find $\mathsf{ed}_\mathsf{d}(X_1, X_2)$ with $\mathcal{O}(k^2)$ LCP queries. Therefore, if each LCP query may be answered in $\mathcal{O}(\log n)$ time, we may compute $\mathsf{ed}_\mathsf{d}(X_1, X_2)$ in total time $\mathcal{O}(k^2 \log n)$. □

As a result of the preceding lemma, it turns out that the main challenge of this section is efficiently maintaining the dynamic reduction of Dyck to string edit distance, rather than solving the string edit distance instances themselves. In order to limit how many instances of string edit distance we must compute, we first pre-process X to find and store $\hat{X}$ in a collection $\mathcal{X}$ of Lemma 4.6.4.

Definition 4.6.6. Given a parenthesis string X, we define $\hat{X}$ to be the parenthesis string resulting from removing every neighboring pair of an opening parenthesis followed by a matching closing parenthesis until no more removals can be performed.

An important detail about $\hat{X}$ is that it contains no balanced substrings since we remove all neighboring matching parentheses from X. Any sequence of opening parentheses followed by closing parentheses in $\hat{X}$ requires at least 1 edit to balance. Therefore, $\mathcal{C}(\hat{X})$, which is a collection of LR-segments by Theorem 4.6.2, only contains at most $\mathcal{O}(k)$ segments. This allows us to limit the number of string edit distance instances that we need to solve to $\mathcal{O}(k)$ each time we recompute our Dyck edit distance approximation via the reduction to string edit distance.

Lemma 4.6.7 ([Backurs and Onak, 2016] Claim 35)**.** *Given a parenthesis string X with $\mathsf{ded}_\mathsf{d}(X) \leq k$, there are $\mathcal{O}(k)$ maximal LR-segments in $\hat{X}$.*

An important step of constructing $\mathcal{C}(\hat{X})$ is finding sequences of opening parentheses followed by equal-length sequences of closing parentheses. We construct $\mathbf{O}(\hat{X})$ to quickly find such neighboring sequences since we do not care about the type of parentheses in this step, only whether they are opening or closing.

Lemma 4.6.8. *For a parenthesis string X with $n = |X|$, the string $\hat{X}$ can be maintained with $\tilde{\mathcal{O}}(n)$ pre-processing time and $\mathcal{O}(\log^2 n)$ update time when X undergoes dynamic insertions, deletions, and substitutions.*

Proof. First, we describe how to construct and maintain $\hat{X}$. We are going to build a binary tree where the leaves of the tree partition X into individual characters such that each leaf contains a single parenthesis of X. Each parent of a leaf node will concatenate the characters in its children, removing any pairs of matching parenthesis. We will continue this process level by level: for any internal node, we concatenate the substrings in its children and combine them, removing any new matching neighboring parentheses. We set the height of the binary tree such that the root node will contain $\hat{X}$ since we made sure to remove matching neighboring parentheses at each level of the tree.

To build the above binary tree, we use a dynamically maintained collection of strings. First, we let $\mathcal{X}$ be an empty collection of strings. Let $L = \mathcal{O}(\log n)$ be the number of levels in the binary tree we wish to build. We denote each node of the binary tree as v_I^ℓ where $\ell \in [L]$ is the level of the node and $I \subseteq [0\,..\,|X| - 1]$ is the interval such that v_I^ℓ contains a pointer to $\hat{X}_I \in \mathcal{X}$ where $X_I = X[I]$ and I is the union of the intervals of the children of v_I^ℓ. Handling each leaf node v_I^L is straightforward since I contains just a single index and so $X_I = \hat{X}_I$ since there can be no pairs of parentheses in any of the intervals at the leaf level. We sort the leaves from left-to-

right in increasing index such that the leftmost leaf is $v^L_{\{0\}}$ and the rightmost leaf is $v^L_{\{|X|-1\}}$. We additionally perform $\mathsf{add}(X[i])$ to $\mathcal{X}$ and store a pointer to $X[i]$ in leaf $v^L_{\{i\}}$ for all $i \in [|X| - 1]$. Now we show how to continue building the binary tree bottom-up using $\mathcal{X}$. We assume for any level of the tree, the intervals of the nodes at that level form a non-overlapping partition of $[|X|]$ and are in sorted order. It is easy to see this will be the case inductively since the leaf level already partitions $[|X|]$ in sorted order, and we defined the interval of a node to be the union of the intervals of its children. For a given node v^ℓ_I with $0 \leq \ell \leq L$, we wish to add $\hat{X}_I$ to $\mathcal{X}$ and store pointers to these substrings at this node. We may assume its children $v^{\ell+1}_{I_1}$ and $v^{\ell+1}_{I_2}$ have pointers to $\hat{X_{I_1}} \in \mathcal{X}$ and $\hat{X_{I_2}} \in \mathcal{X}$, respectively and $I = I_1 \cup I_2$ as per our construction. To construct $\hat{X}_I$, we want to concatenate $\hat{X_{I_1}}, \hat{X_{I_2}}$ and remove any neighboring matching parentheses. By Definition 4.6.6, there must not be any neighboring matching parentheses within either $\hat{X_{I_1}}$ or $\hat{X_{I_2}}$; we only need to check parentheses that are matches between the suffix of $\hat{X_{I_1}}$ and prefix of $\hat{X_{I_2}}$. To do so, we take $m = \min(\mathsf{LCP}(T(\hat{X_{I_1}}), \hat{X_{I_2}}), \mathsf{LMP}(\hat{X_{I_2}}))$, which tells us exactly how long the sequence of matching neighboring parentheses between the two substrings is. Note that we only want to find the number of opening parentheses in the suffix of $\hat{X_{I_1}}$ that match with the closing parentheses in the prefix of $\hat{X_{I_2}}$, and we must be careful not to include any closing parentheses of $\hat{X_{I_1}}$ or any opening parentheses of $\hat{X_{I_2}}$ by making sure the largest value we can return is $\mathsf{LMP}(\hat{X_{I_2}})$. We then perform $\mathsf{split}(\hat{X_{I_1}}, |\hat{X_{I_1}}| - m)$ and $\mathsf{split}(\hat{X_{I_2}}, m)$, and finally we concatenate the resulting substrings $\mathsf{concat}(\hat{X_{I_1}}[0\,..\,|\hat{X_{I_1}}| - m]), \hat{X_{I_2}}[m\,..])$ to add $\hat{X}_I$ to $\mathcal{X}$ and store a pointer to $\hat{X}_I$ in node v^ℓ_I. We will continue concatenating substrings of each node's intervals in this way level by level until we reach the root and obtain $\hat{X}$.

First, we note that there are $\mathcal{O}(\log n)$ levels of the binary tree, and therefore $\mathcal{O}(n)$ total nodes in the binary tree. Each node requires 1 LCP query, 1 LMP query, 2 split operations, and 1 concat

operation to construct the substring for its interval. Therefore, each node takes time $\mathcal{O}(\log n)$ to process by Theorem 4.4.21 and Lemma 4.6.4, and so the total pre-processing time to build $\hat{X}$ is $\mathcal{O}(n \log n)$.

When X gets updated, we need only update nodes whose corresponding substring intervals are affected by the update. When the height of the binary tree is $\mathcal{O}(\log n)$, there are only $\mathcal{O}(\log n)$ such nodes. Since we used a constant number of split, add, and concat operations each taking time $\mathcal{O}(\log n)$, reconstructing our affected paths may take time $\mathcal{O}(\log^2 n)$. Since deletions and insertions may cause the tree to become unbalanced, instead of a standard binary tree we may use a self-balancing tree such as an AVL tree instead to maintain the height of $\mathcal{O}(\log n)$ while still affecting $\mathcal{O}(\log n)$ nodes per update. □

We now have all the ingredients to prove the main theorem of this subsection.

Theorem 4.1.7. *There exists a randomized dynamic algorithm that maintains an $\mathcal{O}(\log(\mathsf{ded}(X)))$-approximation of $\mathsf{ded}(X)$ (correctly with high probability against an adaptive adversary) for a string $X \in \Sigma^*$ undergoing edits. This algorithm has amortized update time $\tilde{\mathcal{O}}(\mathsf{ded}(X) + 1)$ per edit and pre-processing time of $\tilde{\mathcal{O}}(|X|)$.*

Proof. We will dynamically maintain a collection of strings $\mathcal{X}$ according to Theorem 4.4.21, Lemma 4.6.4 and construct $\hat{X}$ according to Lemma 4.6.8. Using these tools, we build our reduction to string edit distance in time $\tilde{\mathcal{O}}(k)$ where $k = \mathsf{ded}_{\mathsf{d}}(X)$. Normally the reduction takes $\tilde{\mathcal{O}}(n)$ time according to Theorem 4.6.2, but we show how to reduce computation time utilizing our dynamic strings collection in the following claim.

Claim 4.6.9. *Given a parenthesis string X belonging to a dynamic collection $\mathcal{X}$ of Lemma 4.6.4, one can implement the Dyck-to-string reduction of Theorem 4.6.2 in time $\tilde{\mathcal{O}}(s)$, where s is the*

number of maximal LR-segments of X, with each string in $\mathcal{C}(X)$ reported as a handle to an element of $\mathcal{X}$.

Proof. We first recall the algorithm of [Koucký and Saks, 2023] for a parenthesis string X, which constructs $\mathcal{C}(X)$ satisfying Theorem 4.6.2 as follows:

1. Construct the LR-decomposition $\mathcal{D}$ of X.
2. If the first (last) segment of $\mathcal{D}$ only contains closing (opening) parentheses, add this segment to $\mathcal{C}(X)$.
3. For each remaining segment Y in $\mathcal{D}$, let Y' be the maximal-length substring of Y composed of a sequence of opening parentheses followed by an equal-length sequence of closing parentheses. Add Y' to $\mathcal{C}(X)$.
4. Let X' be the remainder of the string of X after removing all such substrings Y' from the previous step. If X' is not empty, repeat steps 1–3 on X'.

Note that in step 3, we fully remove either all the opening parentheses or all the closing parentheses in each segment, and therefore, we reduce the number of segments by at least half in each round of the algorithm. Thus, steps 1–3 may be repeated at most $\mathcal{O}(\log n)$ times. Unfortunately, without utilizing our dynamic string collection, each such repetition of steps 1–3 may take time $\mathcal{O}(|X|)$ to construct the LR-decomposition of X and handle each segment. We now explain how to leverage $\mathcal{X}$ to avoid this linear-time blowup.

Since the LR-decomposition of X consists of $\mathcal{O}(s)$ segments handled in steps 2–3 and we repeat steps 1–3 at most $\mathcal{O}(\log n)$ times, then the total reduction will take $\tilde{\mathcal{O}}(s)$ time if each segment can be processed in $\tilde{\mathcal{O}}(1)$ time.

Recall that via Lemma 4.6.4, we may find the length of maximal prefixes of strings in $\mathcal{X}$

composed of only opening or closing parentheses in $\mathcal{O}(\log n)$-time. Let $\ell_1 = \mathsf{LMP}(X)$, we then perform $\mathsf{split}(X, \ell_1)$ to isolate the first maximal sequence of opening or closing parentheses in X. If $X[0]$ is a closing parenthesis, then we add $X[0\,..\,\ell_1)$ to $\mathcal{C}(X)$ as per step 2. If $X[0]$ is an opening parenthesis, we find the length of the following sequence of closing parentheses $\ell_2 = \mathsf{LMP}(X(\ell_1\,..])$. Define $m = \min(\ell_1, \ell_2)$, we then add $X[\ell_1 - m\,..\,\ell_1 + m)$ to $\mathcal{C}(X)$, which is the maximal substring of the first LR-segment in X composed of an equal number of opening parentheses and following closing parentheses. If $m = \ell_1$, then there are more closing parentheses in this segment, and so, we use split to add the remaining fragment $Y_1 = X(\ell_1 + m\,..\,\ell_1 + \ell_2]$ to dynamic string collection $\mathcal{X}$. If $m = \ell_2$, we instead use split to add the remaining fragment $Y_1 = X[0\,..\,\ell_1 - m)$ to $\mathcal{X}$. This marks the end of the processing for the first LR-segment, and we now repeat the same steps on $X(\ell_1 + \ell_2\,..]$ to handle the remaining segments. When we find the new remaining fragment Y_2 for the second segment, we perform $\mathsf{concat}(Y_1, Y_2)$ and continue as before. We process each segment, querying $\mathcal{X}$ to determine the length of the maximal sequence of opening and closing parentheses in each segment of the LR-decomposition. Then, we find the remaining fragment and concatenate it to all previous fragments. After processing all segments, we will have concatenated all remaining fragments of LR-segments not added to $\mathcal{C}(X)$, and we may recurse on the resulting concatenation as per step 4.

Each segment of the LR-decomposition requires a constant number of split, concat operations and LCP, LMP queries each of which takes at most $\mathcal{O}(\log n)$ time by Theorem 4.4.21 and Lemma 4.6.4. There are $\mathcal{O}(s)$ segments per LR-decomposition and $\mathcal{O}(\log n)$ levels of recursion. Therefore, in total computing $\mathcal{C}(X)$ takes time $\tilde{\mathcal{O}}(s)$. □

Since the runtime of Claim 4.6.9 depends on the number of segments in the LR-

decomposition of X, instead of applying the reduction directly to X, we apply the reduction to $\hat{X}$, which has $\mathcal{O}(k)$ segments in its LR-decomposition by Lemma 4.6.7. This is possible feasible because $\mathsf{ded}(X) = \mathsf{ded}(\hat{X})$ and, by Lemma 4.6.8, $\hat{X}$ can be maintained in $\mathcal{X}$. Once we have our set of LR-strings $\mathcal{C}(\hat{X})$, we may solve the corresponding string edit distance instances according to Observation 4.6.1. We will utilize Lemma 4.6.5 to solve all instances in total time $\tilde{\mathcal{O}}(k^2)$. Note that one condition of Lemma 4.6.5 is that we may answer LCP queries in $\mathcal{O}(\log n)$ time. Note that Claim 4.6.9 ensures that $Y \in \mathcal{X}$ for all $Y \in \mathcal{C}(\hat{X})$. Therefore, for any substrings Y_1, Y_2 of Y we wish to query, we only need a constant number of split operations on $\mathcal{X}$ to add Y_1 and Y_2 to $\mathcal{X}$ and then perform our LCP query. By Theorem 4.4.21, each split takes $\mathcal{O}(\log n)$ time and each LCP query takes $\mathcal{O}(1)$, so we indeed satisfy the conditions of Lemma 4.6.5. Lemma 4.6.5 gives an exact solution for string edit distance[10], and so by summing the string edit distance of each $Y \in \mathcal{C}(\hat{X})$, we obtain an $\mathcal{O}(\log n)$-approximation for $\mathsf{ded}_{\mathsf{d}}(\hat{X})$, and thereby also $\mathsf{ded}(X)$, by Theorem 4.6.2.

Our algorithm proceeds in epochs. Each epoch starts by computing $\hat{X}$ according to Lemma 4.6.8 and the collection $\mathcal{C}(\hat{X})$ using the algorithm of Claim 4.6.9. For each $Y \in \mathcal{C}(\hat{X})$, the next step is to compute $\mathsf{ded}_{\mathsf{d}}(Y) = \mathsf{ed}_{\mathsf{d}}(\mathcal{L}(Y), T(\mathcal{R}(Y)))$ using the exact algorithm of Lemma 4.6.5. Let a_Y denote $\mathsf{ded}_{\mathsf{d}}(Y)$ for each $Y \in \mathcal{C}(\hat{X})$. We now discuss the approximation factor in a similar analysis to the Proof of Theorem 4.1.6. We use $a = 2\sum_{Y \in \mathcal{C}(Y)} a_Y$ as our approximation of $\mathsf{ded}_{\mathsf{d}}(X)$ for the duration of the epoch, which lasts for $m := \lfloor a/(4(3 + 2\lg a)) \rfloor$ dynamic edits. At the beginning of the epoch,

$$a = 2 \sum_{Y \in \mathcal{C}(\hat{X})} a_Y = 2 \sum_{Y \in \mathcal{C}(\hat{X})} \mathsf{ed}_{\mathsf{d}}(\mathcal{L}(Y), T(\mathcal{R}(Y)))$$

[10] If we do not know k, we may perform a binary search for the value of k at the cost of an additional $\mathcal{O}(\log k)$ factor.

$$= 2 \sum_{Y \in \mathcal{C}(\hat{X})} \mathsf{ded_d}(Y)$$

$$\geq 2 \cdot \mathsf{ded_d}(X) \geq 2 \cdot \mathsf{ded_d}(X)$$

and

$$a = 2 \sum_{Y \in \mathcal{C}(\hat{X})} a_Y = 2 \sum_{Y \in \mathcal{C}(\hat{X})} \mathsf{ed_d}(\mathcal{L}(Y), T(\mathcal{R}(Y)))$$

$$= 2 \sum_{Y \in \mathcal{C}(\hat{X})} \mathsf{ded_d}(Y)$$

$$\leq 2(3 + 2\lg(\mathsf{ded_d}(X)))\mathsf{ded_d}(X)$$

$$\leq 2(3 + 2\lg a)\mathsf{ded_d}(X)$$

$$\leq 4(3 + 2\lg a)\mathsf{ded_d}(X),$$

Throughout the epoch, the value $\mathsf{ded_d}(X)$ may change by at most $\pm m$, so we still have $a \geq \frac{a}{2} + m \geq \mathsf{ded_d}(X) - m + m = \mathsf{ded_d}(X)$ and $a = 2a - a \leq 8(3 + 2\lg a)(\mathsf{ded_d}(X) + m) - a \leq 8(3 + 2\lg a)\mathsf{ded_d}(X) + 8(3 + 2\lg a)m - a \leq 8(3 + 2\lg a)\mathsf{ded_d}(X)$, so the value a remains an $\mathcal{O}(\log n)$-approximation of $\mathsf{ded_d}(X)$.

For each dynamic edit to X, we must update $\hat{X}$ and $\mathcal{C}(\hat{X})$, which takes time $\tilde{\mathcal{O}}(\log n + k)$ according to Lemma 4.6.8 and Claim 4.6.9. We only recompute a using the $\mathcal{O}(k^2)$ string edit distance algorithm of Lemma 4.6.5 once per epoch that contains $m = \Omega(k/\log n)$ edits; this yields $\tilde{\mathcal{O}}(\log n + k + k^2 \log n/k) = \tilde{\mathcal{O}}(1 + k)$ amortized time per update. □

4.6.3 Combined Algorithm

We may combine the algorithms of Theorem 4.1.6 and Theorem 4.1.7 and transition between the two routines according to our current approximation of Dyck edit distance for the parenthesis string.

Theorem 4.1.8. *There exists a randomized algorithm that, for a fixed parameter $\epsilon \in \mathbb{R}_+$, maintains an $\mathcal{O}(f(\epsilon)\log \mathsf{ded}(X))$-approximation of $\mathsf{ded}(X)$ (correctly with high probability against an adaptive adversary) for a string $X \in \Sigma^*$ undergoing edits. This algorithm has amortized update time $\tilde{\mathcal{O}}(|X|^{.5+\epsilon})$ per edit and pre-processing time of $\tilde{\mathcal{O}}_\epsilon(|X|^{1+\epsilon})$.*

4.7 Exact Dynamic k-Dyck Edit Distance

In this section, we give an algorithm for exact dynamic Dyck edit distance by keeping track of a unique decomposition of parentheses strings in trapezoidal segments and the remaining clusters [Fried et al., 2024]. The algorithm supports dynamic updates in time $\mathcal{O}((1+\mathsf{ded}(X))^5)$. To build this decomposition, we consider the graph of the heights of all the parentheses in a parenthesis string, as shown in Figure 4.1a, where recall the height of an index is the difference in the number of preceding closing parentheses from the number of opening parentheses. We then use this graph to find natural trapezoids, defined formally in the following definition, and build a tree structure on the resulting components, which we now state.

Definition 4.7.1 (Trapezoid, [Fried et al., 2024, Definition 3.2])**.** Given a parenthesis string X, a *trapezoid* (a,b,c,d) is a quadruple such that

1. $h(a) = h(d)$,

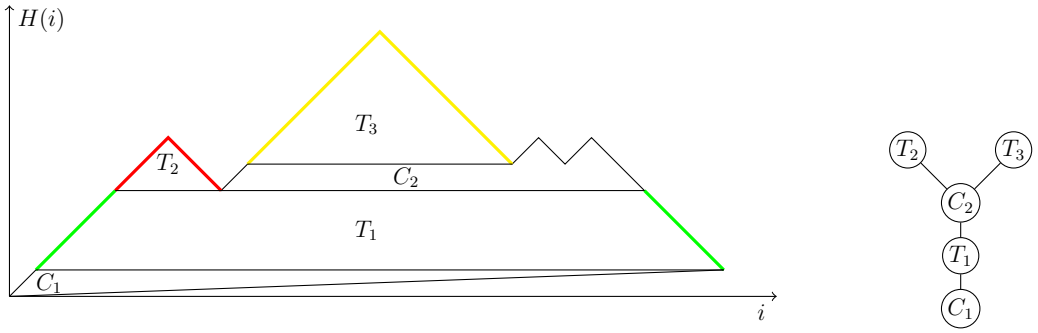


Figure 4.11: Example of a trapezoid and cluster decomposition (left) and corresponding decomposition (right) from [Fried et al., 2024]. Note that colored edges depict tall maximal trapezoids while black edges depict clusters.

2. $b - a = h(b) - h(a) = h(c) - h(d) = d - c$, and
3. $h(i) \geq h(b) = h(c)$ for all $i \in [b\,..\,c]$.

A trapezoid is *maximal* if it cannot be extended, i.e., neither $(a-1, b, c, d+1)$ nor $(a, b+1, c-1, d)$ is a trapezoid. A trapezoid is *tall* if $b - a \geq 2k$, where $k = \mathsf{ded}(X)$[11].

Definition 4.7.2 (Cluster, [Fried et al., 2024, Definition 3.5])**.** Consider a cycle on $n+1$ vertices $[0\,..\,n]$ with edges $\{(i, i+1) | i \in [0\,..\,n)\} \cup \{(n, 0)\}$. Given a parenthesis string X, for each tall maximal trapezoid (a, b, c, d) of X, remove all edges $(i, i+1)$ for $i \in [a\,..\,b) \cup [c\,..\,d)$, remove vertices $(a\,..\,b) \cup (c\,..\,d)$, and add edges (a, d), (b, c). A *cluster* is a connected component of the resulting graph.

The set of tall maximal trapezoids and clusters partition the indices of X, and there exists a natural tree, the *decomposition tree* such that each node of the tree corresponds to a tall maximal trapezoid or a cluster. The cluster containing indices 0 and n is the root of the tree. Given a cluster C and trapezoid $T = (a, b, c, d)$, C is the parent of T if $a, d \in C$, and T is the parent of C if

[11]Since k is initially unknown, we may use an exponential search for k by running the algorithm on k set to $1, 2, 4, 8, \ldots$ until it succeeds with the nearest power of 2 larger than $\mathsf{ded}(X)$. This only adds a factor of $\mathcal{O}(\log k)$ to the pre-processing of the data structure. When dynamic edits occur, we may use our computed value of $\mathsf{ded}(X) + 2$ before the edit as our upper bound since Dyck edit distance can only change by 2 in either direction per dynamic edit.

$b, c \in C$. See Figure 4.11 for an example of such a decomposition.

The key component of the trapezoid and cluster decomposition is that the Dyck edit distance of $\hat{X}$ (recall $\hat{X}$ is the parenthesis string obtained by removing all neighboring pairs of matching parentheses from X iteratively until no such pairs remain and has the same Dyck edit distance as X) may be computed efficiently when k is small using this tree. There are at most $\mathcal{O}(k)$ trapezoids and $\mathcal{O}(k^2)$ indices of $\hat{X}$ in clusters. Each trapezoid may be processed in time $\tilde{\mathcal{O}}(k^3)$ with access to an oracle that answers LCP queries in $\mathcal{O}(1)$-time, and each cluster processed in time $\mathcal{O}(k^5)$. The main difficulty in the dynamic setting is constructing the trapezoid and cluster decomposition and maintaining the decomposition tree efficiently. The static setting requires a linear time construction, but we may utilize the LR-decomposition, the partition of $\hat{X}$ into maximal-length substrings of opening parentheses followed by closing parentheses, first defined in Section 4.6.1 to compute this in polynomial time dependent on k. Similarly to Section 3, we utilize the dynamic strings data structure to build the LR-decomposition, and we augment the data structure to store additional information about X that we wish to access later.

Lemma 4.7.3. *Given a parenthesis string X with $k = \mathsf{ded}(X)$ and a dynamically maintained collection of strings X with $\hat{X} \in \mathcal{X}$, the LR-decomposition $\mathcal{D}$ of $\hat{X}$ can be computed in time $\mathcal{O}(k \log n)$. Furthermore, $\mathcal{D}$ may constructed such that given a segment $S \in \mathcal{D}$, $|\mathcal{L}(S)|, |\mathcal{R}(S)|$ and for any index $i \in [|X| - 1]$, $h(i)$ may be found in $\mathcal{O}(\log k)$ time.*

Proof. Let $\ell_1 = \mathsf{LMP}(\hat{X})$, where recall $\mathsf{LMP}(\hat{X})$ is the maximal-length prefix of $\hat{X}$ composed of only opening or only closing parentheses. Next, we perform $\mathsf{split}(\hat{X}, \ell_1 - 1)$ to separate this prefix from $\hat{X}$. If $\hat{X}[0]$ is a closing parenthesis, we add $S_1 = \hat{X}[0 \,..\, \ell_1)$ to $\mathcal{D}$. Otherwise, we let $\ell_2 = \mathsf{LMP}(\hat{X}[m \,..])$ and perform $\mathsf{split}(\hat{X}, \ell_1 + \ell_2 - 1)$ and then add $S_1 = \hat{X}[0 \,..\, \ell_1 + \ell_2)$ to $\mathcal{D}$. We

continue repeating the same steps for the remaining string $X' = X[\ell_1 + \ell_2 ..]$ until X' is empty after splitting the last segment. By Lemma 4.6.7, there are $\mathcal{O}(k)$ segments in $\mathcal{O}(\hat{X})$. Each segment may be added to $\mathcal{D}$ using a constant number of split and LMP queries. By Theorem 4.4.21, each split takes time $\mathcal{O}(\log n)$ and by Lemma 4.6.4, each LMP query takes time $\mathcal{O}(\log n)$. Therefore, in total, constructing $\mathcal{D}$ takes time $\mathcal{O}(k \log n)$.

Now, during construction of $\mathcal{D}$, when finding a segment $S_i \in \mathcal{D}$, we observe that the values ℓ_1 and ℓ_2 correspond to lengths $|\mathcal{L}(S_i)|, |\mathcal{R}(S_i)|$, respectively. Furthermore, we may easily store the heights of the first index $\alpha_i^{\mathcal{L}}$ in $\mathcal{L}(S_i)$ and the first index $\alpha_i^{\mathcal{R}}$ in $\mathcal{R}(S_i)$ by taking $h(\alpha_i^{\mathcal{L}}) = \sum_{j=0}^{i-1} |\mathcal{L}(S_j)| - |\mathcal{R}(S_j)|$ and $\alpha_i^{\mathcal{R}} = |\mathcal{L}(S_i)| + \sum_{j=0}^{i-1} |\mathcal{L}(S_j)| - |\mathcal{R}(S_j)|$, respectively. Then, for any index $i \in [|X|]$, we may first use a binary search through $\mathcal{D}$ to identify which segment $S_i \in \mathcal{D}$ that i belongs to. Then, if $X[i] \in \mathcal{L}(S_i)$, we find that $h(i) = h(\alpha_i^{\mathcal{L}}) + i - \alpha_i^{\mathcal{L}}$. Similarly, if $X[i] \in \mathcal{R}(S_i)$, then $h(i) = h(\alpha_i^{\mathcal{R}}) + \alpha_i^{\mathcal{R}} - i$. Overall, this takes $\mathcal{O}(\log k)$ time per height query due to the binary search required. □

We now give the construction of the set of tall maximal trapezoids of $\hat{X}$.

Lemma 4.7.4. *Given a parenthesis string X and a dynamically maintained collection of strings $\mathcal{X}$ with $\hat{X} \in \mathcal{X}$, the set of tall maximal trapezoids of $\hat{X}$ can be computed in time $\mathcal{O}(k^2 \log n)$.*

Proof. First, we compute the LR-decomposition $\mathcal{D}$ of $\hat{X}$ in time $\mathcal{O}(k \log n)$ according to Lemma 4.7.3.

We assume as per their construction that the segments $S_1, S_2, \ldots, S_m$ of $\mathcal{D}$ are sorted according to their starting indices, e.g. $X[0] \in S_1$. For each segment S_i, let $\alpha_i^{\mathcal{L}}$ and $\beta_i^{\mathcal{L}}$ be the first and last index of $\mathcal{L}(S_i)$, respectively, and define $\alpha_i^{\mathcal{R}}$ and $\beta_i^{\mathcal{R}}$ similarly for $\mathcal{R}(S_i)$. For each segment S_i, we observe that any index j of $\mathcal{L}(S_i)$ will be part of a trapezoid with some index in $\mathcal{R}(S_j)$

for some $j \geq i$ if S_j is the nearest segment to the right of S_i such that it contains an index of height exactly 1 more than $h(j)$. Therefore, we iterate through $\mathcal{D}$ from left-to-right to match every opening parenthesis in a segment of $\mathcal{D}$ to the closing parenthesis of a later (or the same) segment with the correct corresponding height.

We consider an index unmatched if it has not yet been included in a trapezoid by our construction algorithm. When considering two segments S_i, S_j with $i \leq j$, we first take the highest height of its unmatched opening parentheses and the lowest height of closing parentheses in S_j. We note that as we are considering segments left-to-right, S_j must be able to match all unmatched opening parentheses with heights between the highest height of S_i and the lowest height of S_j. Therefore, we take this difference as the height of our trapezoid, taking care to not exceed the number of unmatched opening parentheses in S_i. If the computed trapezoid height is larger than $2k$, we add it to the set of tall trapezoids, otherwise we move onto the next segment $\mathcal{R}(S_{j+1})$ to consider matching with $\mathcal{L}(S_i)$. In either case, we update how many opening parentheses have been matched so far in $\mathcal{L}(S_i)$. The following is our construction algorithm, which we do for each S_i for i from 1 to m:

1. $matched \leftarrow 0, \mathcal{T} \leftarrow \emptyset$
2. For each S_j for j from i to m:
   (a) Let $d \leftarrow h(\beta_i^{\mathcal{L}} - matched) - h(\beta_j^{\mathcal{R}})$, $d' \leftarrow |\mathcal{L}(S_i)| - matched$
   (b) If $d' \geq d \geq 2k$, then $\mathcal{T} \leftarrow \mathcal{T} \cup (\beta_i^{\mathcal{L}} - matched - d, \beta_i^{\mathcal{L}} - matched, \beta_j^{\mathcal{R}} - d, \beta_j^{\mathcal{R}})$
   (c) If $d > d' \geq 2k$, then $\mathcal{T} \leftarrow \mathcal{T} \cup (\alpha_i^{\mathcal{L}}, \beta_i^{\mathcal{L}} - matched, \beta_j^{\mathcal{R}} - d, \beta_j^{\mathcal{R}} - d + d')$
   (d) If $h(\beta_i^{\mathcal{L}} - matched) \geq h(\beta_j^{\mathcal{R}})$, then set $matched \leftarrow \min(matched + h(\beta_i^{\mathcal{L}} - matched) - h(\beta_j^{\mathcal{R}}) + 1, |\mathcal{L}(S_i)|)$.

Note that there are $m \leq k$ segments, so there are k^2 repetitions of step 2 for our construction of $\mathcal{T}$. By Lemma 4.7.3, we may answer height queries in time $\mathcal{O}(\log n)$, and therefore, the total time of construction for $\mathcal{T}$ is $\mathcal{O}(k^2 \log n)$. □

Lemma 4.7.5. *Given a parenthesis string X, its LR-decomposition $\mathcal{D}$, and its corresponding set of tall maximal trapezoids $\mathcal{T}$, the set of clusters of $\hat{X}$ and the decomposition tree of X can be computed in time $\tilde{\mathcal{O}}(k^2)$.*

Proof. We describe the procedure for computing each cluster. Recall that each cluster is a connected component of the graph of parenthesis heights in X with each edge of tall maximal trapezoids (a, b, c, d) removed and edges (a, d) and (b, c) inserted. We assume the segments in $\mathcal{D}$ are in increasing sorted order according to the starting indices of the segments. Let S_i be the first segment of $\mathcal{D}$ with a substring contained in a trapezoid $T = (a, b, c, d) \in \mathcal{T}$. We add all previous segments $S_j, j < i$ to our first cluster C_1 as well as the fragment of S_i before index a. Then, we find the segment S_j containing index d, and continue in a similar manner. Let the $S_{i'}$ be the next closest segment $i' \geq j$ contained in a tall maximal trapezoid $T' = (a', b', c', d') \neq T$. We add the fragment of S_j after index d and before trapezoid T' to cluster C_1 as well as all segments between S_j and $S_{i'}$. Additionally, we add the fragment of $S_{i'}$ before index a'. We then repeat these steps until we reach the end of $\hat{X}$. C_1 is the parent of any trapezoids traversed in this step.

We continue in a similar manner to construct the remaining connected components. To find the next cluster C_m, we find the first index u of X not contained in a previous cluster C_ℓ, $\ell < m$, and find the segment $S_i \in \mathcal{D}$ containing u (for example, for C_2 this index will be b of trapezoid T from our construction of C_1). We add every index in the remainder of S_i and every following segment of $\mathcal{D}$ until we reach a trapezoid $T = (a, b, c, d)$. If a is the first index of T that we reach,

then we continue adding indices starting at d as we did in the first step and set T as the child of C_m. If instead we reached index c (e.g., in the case we started our connected component at index b earlier), we have found the entire connected component, set T as the parent of C_m, and move onto the next cluster.

There are at most $\mathcal{O}(k)$ clusters we need compute, and for each, we may have to check each segment of $\mathcal{D}$, where $|\mathcal{D}| = \mathcal{O}(k)$ by Lemma 4.6.7. We may sort the set of trapezoids are sorted according to their starting indices in $\mathcal{O}(k \log k)$-time, and so we may use a binary search whenever we need to find the nearest trapezoid after an a given index of $\hat{X}$. Therefore, the total time of construction will be $\tilde{\mathcal{O}}(k^2)$,

□

Theorem 4.1.9. *There exists a dynamic algorithm that maintains* $\mathsf{ded}(X)$ *(correctly with high probability against an adaptive adversary) for a string* $X \in \Sigma^*$ *undergoing edits, with update time* $\tilde{\mathcal{O}}((1 + \mathsf{ded}(X))^5)$ *per edit. The algorithm has a pre-processing time* $\tilde{\mathcal{O}}(|X| + \mathsf{ded}(X)^5)$.

Proof. The subroutine processing a tall trapezoid is given $\mathsf{ded}_{\le k}(X[i\,..\,j))$ for all $i, j \in [b - 2k\,..\,b] \cup [c - 2k\,..\,c]$, and the output is $\mathsf{ded}_{\le k}(X[i\,..\,j))$ for $i, j \in [a\,..\,a + 2k] \cup [d - 2k\,..\,d]$. For $(i, j) \in [a\,..\,b]^2 \cup [c\,..\,d]^2$, we have $\mathsf{ded}_{\le k}(X[i\,..\,j)) = \lceil (j - i)/2 \rceil$ because all parentheses in $X[i\,..\,j)$ are all opening or all closing. Furthermore, [Fried et al., 2024, Lemma 3.4] claims that $(i, j) \in ([a\,..\,b] \times [c\,..\,d]) \setminus ([b-2k\,..\,b] \times [c\,..\,c+2k])$ and $v \in [0\,..\,k]$, we have $\mathsf{ded}(X[i\,..\,j)) \le v$ if and only if $L_v[(i + j) - (b + c)] \ge j$, where L_v is computed by the following algorithm, which assumes that all out-of-bounds are initialized entries have value $-\infty$. We also denote $\delta_+ = \max(\delta, 0)$ and $\delta_- = \min(\delta, 0)$.

This algorithm can be implemented in $\mathcal{O}(k^2 \log n)$ time assuming that X is stored in a data

Algorithm 19: Processing a trapezoid (a, b, c, d)

for $v := 0$ **to** k **do**
  for $\delta := -2v$ **to** $2v$ **do**
    $L'_v[\delta] := \min(d - 2k + \delta_+,$
      $\max(L_{v-1}[\delta - 2] + 2, L_{v-1}[\delta - 1] + 1, L_{v-1}[\delta] + 1, L_{v-1}[\delta + 1], L_{v-1}[\delta + 2]));$
    if $\mathsf{ded}(X[b - 2k + \delta_+ \,..\, c + 2k + \delta_-)) \leq v$ **then**
      $L'_v[\delta] := \max(L'_v[\delta], c + 2k + \delta_-);$
    if $L'_v[\delta] \neq -\infty$ **then**
      $L_v[\delta] := L'_v[\delta] + \mathsf{LCP}(T(X[a \,..\, b + c + \delta - L'_v[\delta])), X[L'_v[\delta] \,..\, d));$

structure of Lemma 4.6.4. To process the clusters, we note that there are $\mathcal{O}(k^2)$ indices in clusters, so each index of the cluster may be solved with a standard cubic time Dyck edit distance algorithm in $\mathcal{O}(k^{2+3})$ total time (see Fried et al. [Fried et al., 2024] for details). By Lemmas 4.7.4 and 4.7.5, we can compute the set of trapezoids, clusters, and decomposition tree in time $\mathcal{O}(k^2)$. So in total, we have $\mathcal{O}(k^5)$ update time. □

4.8 Missing Proofs of Section 4.3

4.8.1 Lazy Propagation

For our sub-polynomial dynamic Dyck edit distance approximation algorithm, we often wish to find parentheses with specific height conditions, such as twins of a given parenthesis or the index of the parenthesis corresponding to the parent node of a given child node. In general, we wish to quickly answer what we call *minimum height range queries*, queries $\mathsf{range}(i, h)$ that return the maximum range starting at i such that all parentheses in this range have heights greater than h.

In this section, we describe how we can use segment trees to pre-process a parenthesis string X to answer such range queries in logarithmic time, and then show how to use a well-known technique *lazy propagation* to correctly maintain this data structure when X undergoes dynamic

updates.

The segment tree is a binary tree. Each leaf node of the segment tree $\mathcal{T}$ for X will correspond to an index $0 \leq i < |X|$. Leaf nodes will be ordered such that the leftmost leaf node corresponds to index 0 and the rightmost leaf node corresponds to index $|X| - 1$. Each internal node will be the parent of two consecutive nodes from the following level and will correspond to the range of indices in the concatenation of the range of indices of its children. Additionally, each node v with corresponding range $[i, j]$ stores the minimum height of an index in this range, denoted $minh(v)$. For a leaf node v, $minh(v) = h(i)$ where i is the index corresponding to v.

Algorithm 20: Answering minimum height range queries with segment trees

```
1  Combine ([ℓ1 .. r1], [ℓ2 .. r2]):
2      if r1 == ℓ2 − 1 then
3          Return [ℓ1 .. r2];
4      else
5          Return [ℓ1 .. r1];
6  GetMinRange (v, i, h):
7      if i > v.r then
8          Return ∅;
9      if i ≤ v.ℓ and minh(v) > h then
10         Return [v.ℓ .. v.r];
11     if i ≤ v.ℓ and minh(v.left) ≤ h then
12         Return GetMinRange (v.left, i, h)
13     else if i ≤ ℓ and minh(v.left) > h then
14         Return Combine([v.left.ℓ, v.left.r], GetMinRange(v.right, i, h))
15     else
16         Return
             Combine(GetMinRange(v.left, i, h), GetMinRange(v.right, i, h));
```

Claim 4.8.1. *For a parenthesis string X, Algorithm 20 answers a minimum height range query (i, h) in $\mathcal{O}(\log^2 n)$ time.*

Proof. We first build the segment tree as described above for X. We can then pass the root of the tree to the `GetMinRange` function along with $\ell = 0, r = |X| - 1$ and the query values i

and h. Whenever the range of v, the node we are considering in the `GetMinRange` function lies completely to the left of the index i, we do not add it to our range or consider it any further. Conversely, if the range of v lies fully to the right of i, if it has minimum height above h we return the full range to be passed to the `Combine` function later. If the range of v lies fully to the right i but the minimum height in the left half of the range is not above h, then we only recurse on the left child, which corresponds to the left half of the range. If instead, the left child range has minimum height fully above h, then we recurse on the right child and will return the full left child range concatenated with any parts of the right child found to be in the solution as well. Finally, if the range of v is not fully to the right of i we just recurse on both children.

Note that we want the algorithm to return a contiguous range starting from i. Therefore, when we have potential pieces of the solution from both a left and right child, we use the `Combine` function to take the union of the left and right children's partial solutions if the resulting range would be contiguous, or otherwise only take the left child's returned range.

We call a node *active* if is passed to `GetMinRange` at some timestep when Algorithm 20 is run. We show by induction that there are only j active nodes from level j of the segment tree. As a base case, we note that there is 1 node, the root, at level 1 in the tree and it is passed directly to the algorithm at the start. We now assume that there are $j-1$ active nodes at level $j-1$. We note that at most 1 active node on each level may contain j in its range. This node may recurse on both its left and right child in the following level. For any node whose range falls completely to the left of i, we do not continue recursing on this node's subtree any further. This is true for any node whose range falls completely to the right of i with minimum height greater than h as well. Finally, for any remaining node, which must have a range falling completely to the right of i, the algorithm only recurses on either the left or right child. Therefore, all nodes recurse on at most 1 child except for

the node whose range contains i, which recurses on 2, i.e., there are at most $j - 1 + 1 = j$ active nodes in the next level of the tree.

Each active node only requires constant time ignoring their recursive calls. Since the height of a segment tree is $\mathcal{O}(\log n)$, there will be $1 + 2 + \ldots + \mathcal{O}(\log n)$ active nodes, and in total, the run-time of Algorithm 20 will be $\mathcal{O}(\log^2 n)$. □

While we only care about answering such queries in poly-logarithmic time, we note that improving the run-time for each query to $\mathcal{O}(\log n)$ is possible. For any pair of active nodes v_1, v_2 on the same level of the segment tree with ranges $[\ell_1 \,..\, r_1], [\ell_2 \,..\, r_2]$, respectively, such that $i < \ell_1 < \ell_2$, we note that only one of these active nodes needs to recurse on any children. If $minh(v_1) > h$, then the algorithm returns $[\ell_1 \,..\, r_1]$ as normal without recursing. If instead $minh(v_1) \leq h$, this means that it is not possible for $[\ell_2 \,..\, r_2]$ to be in the solution to the range query since there is already some parenthesis between ℓ_2 and i with height smaller than h. We may put this idea into practice by maintaining a queue of active nodes at each level in sorted order according to their ranges. Before processing an active node v_2, we just check the range $[\ell_1 \,..\, r_1]$ and minimum height of the node v_1 in the queue to the left of v_2. If $\ell_1 > i$ and $minh(v_1) \leq h$, we do not have to continue recursing on v_2 and can return an empty range. Now, by induction again we may show that there are at most 3 active nodes per level of the segment tree. In the top level of the tree, trivially there is 1 active node. Now we look at a lower level j and assume there are 3 active nodes. 1 such node v may correspond to a range containing i and so may need to recurse on both children. Any active node to the left of v in the queue will not recurse on any children since their corresponding range falls completely to the left of i. If there are two active nodes to the right of v, one of them will not recurse by our above argument. The remaining active node to the right of v will only recurse

on a single child as in the proof of Claim 4.8.1. Therefore, there are 3 active nodes in level $i+1$, completing the inductive argument. In total, we will visit less than $3 \log n$ active nodes, and so the runtime will be $\mathcal{O}(\log n)$.

We now discuss how to quickly update the segment tree to handle dynamic insertions, deletions, and substitutions in the parenthesis string. First, we want to make sure the height of the segment tree is always $\mathcal{O}(\log n)$. Initially, this will be true due to our construction where each leaf corresponds to a single index and every parent node concatenates the two ranges of its children. We may continue to guarantee $\mathcal{O}(\log n)$ height by using a self-balancing tree such as an AVL tree to handle when changes happen to X. Every deletion or insertion corresponds to an insertion or deletion of a leaf. We may additionally need to recompute $minh$ values for all nodes in the path to any inserted, deleted or rotated nodes after a rebalance, which may be done in $\mathcal{O}(\log n)$ time as $\mathcal{O}(\log n)$ nodes may be affected by a single update to X. Furthermore, an update of X at index i changes the height of all parentheses with indices greater than i. If the update is an inserted opening parenthesis or a deleted closing parenthesis, we must increment the height of all following parentheses by 1. If the update is a deleted opening parenthesis or inserted closing parenthesis, heights decrease by 1. If the update is a substitution, this is equivalent to a deletion and then insertion of the matching parenthesis, so the height changes by 2 for all subsequent parentheses in the relevant direction.

To handle efficient increments and decrements to the height of large sets of nodes in the segment tree, we use a well-known technique called *lazy propagation*. At each node, we will add an additional variable to store any increment to its range. When we access the min height of a node's range, we will add the new variable's value to the min height to include any updates to the parenthesis string. We can propagate any increments in a parent node to its children precisely

when we are traversing through the segment tree during a range query and reset the parent node's increment variable to 0. This guarantees that the children have enough information to correctly compute their minimum height without needing to update every node, instead updating only when those nodes are needed. Algorithm 21 provides pseudo-code of the update function which takes a range of parentheses and the value of the change in height for all parentheses in this range.

Algorithm 21: Update the heights in the segment tree when a dynamic edit occurs.

```
1 Update (v, ℓ, r, incr):
2     if ℓ > r then
3         Return;
4     if v.ℓ == ℓ and v.r == r then
5         v.incr ← v.incr + incr;
6     else
7         Update(v.left, ℓ, min(v.left.r, r));
8         Update(v.right, max(v.right.ℓ, ℓ), r);
```

Claim 4.8.2. *Algorithm 21 takes $\mathcal{O}(\log n)$ time.*

Proof. The proof is by induction. An active node is any node passed to `Update` at some time step of the algorithm. We argue inductively that there are at most 4 active nodes per level of the segment tree. At the root level, this is trivially true. We assume this is true up to some level j. The leftmost and rightmost nodes may recurse on both children. However, the inner two nodes will not recurse since they are fully in the range of parentheses that need to be incremented. Therefore, we just set these two nodes' $incr$ variables accordingly and do not recurse. Thus, there will only be 4 nodes in level $j + 1$.

Since there are 4 active nodes per level and the height of the segment tree is $\mathcal{O}(\log n)$, we have $\mathcal{O}(\log n)$ total run time. □

Part III

MPC String Problems

Chapter 5: Upper and Lower Bounds for Edit Distance in Space-Efficient MPC

5.1 Introduction

The Massively Parallel Computations (MPC) model has emerged as a theoretical model of interest in the last decade to reason about popular distributed processing frameworks such as MapReduce, Spark, and Hadoop, which are widely used for large-scale data analysis. The MPC model consists of multiple processors each with local memory sublinear in the size of input. The input is distributed across processors and computation happens in synchronous rounds. Between consecutive rounds, the partial results through local computations are communicated across processors subject to their local memory constraints. This inter-round communication poses a major bottleneck in the MPC setting. More formally, in the MPC model, each machine has $n^{1-\epsilon+o(1)}$ memory for $\epsilon > 0$ where n is the input size, and the number of machines is $\Omega(n^{\epsilon})$ which is sufficient to hold all the data across the machines. An MPC algorithm is called *space-efficient* if the total memory across all the processors is $n^{1+o(1)}$. The primary goal here is to design space-efficient MPC algorithms minimizing the number of rounds, aka round complexity, which is directly proportional to inter-round communications.

In recent years, the design of MPC algorithms has garnered significant attention in both theory and practice domains [Karloff et al., 2010, Goodrich et al., 2011, Beame et al., 2017, Im et al., 2023]. Despite this focus, the relationships among various problems and lower bounds in

the MPC model remain poorly understood. A recent work by Ghaffari, Kuhn, and Uitto [Ghaffari et al., 2019] has made strides in establishing lower bounds for classic problems, such as maximum matching and vertex cover. These bounds are conditioned on one of the main conjectures in MPC, specifically addressing the hardness of graph connectivity — determining if a graph has one or two connected components (known as the 1-cycle, 2-cycle conjecture) in MPC. Despite this progress, there is limited headway in proving lower bounds for MPC models beyond these specific cases.

Conversely, fine-grained complexity has emerged as a distinct sub-discipline, aiming to comprehend the complexities and relationships of low-polynomial time problems in non-distributed settings. Among some of the greatest contributions of fine-grained complexity are the Strong Exponential Time Hypothesis (SETH) [Impagliazzo and Paturi, 2001, Impagliazzo et al., 2001], and the Orthogonal Vector Conjecture (O.V.) [Williams, 2005]. The O.V. problem consists of two sets of binary vectors A and B, $|A| = |B| = n$, in d dimensional space, $d = \omega(\log n)$, and asks if there is $a \in A$, and $b \in B$ such that $\langle a, b \rangle = 0$. The *Orthogonal Vector conjecture* states that no truly subquadratic time algorithm exists for the O.V. problem unless SETH is false. Given its increasing relevance, a compelling question to strive for is whether we can employ fine-grained complexity arguments to establish connections among various problems and hardness results in the MPC model. Furthermore, how does approximation help overcome these fine-grained complexity barriers in the MPC model?

One of the challenging and extensively studied problems in the domain of fine-grained complexity is the string edit distance problem, which also serves as a primary focus in this paper. The edit distance between two strings is the minimum number of insertions, deletions, and substitutions needed to transform one string into the other. Edit distance is one of the fundamental measures of string similarity, serving as the foundational principle that propels applications from small-scale

applications such as `diff` and `awk` to large-scale problems including database queries, genomics, search engines, and even social networks. Understanding the complexity of edit distance computation has been the subject to numerous studies ranging from exact algorithm design [Sellers, 1974, Wagner and Fischer, 1974, Landau and Vishkin, 1988] to fast approximations [Andoni et al., 2010, Boroujeni et al., 2018, Chakraborty et al., 2020], sublinear-time estimation [Koucký and Saks, 2020, Bringmann et al., 2022a, Goldenberg et al., 2022], streaming and parallel computation [Chakraborty et al., 2016, Cheng et al., 2021, Hajiaghayi et al., 2019]. Consequently, it has been shown that approximating edit distance within a constant factor is possible in $\tilde{O}(n^{1+\epsilon})$ time for any constant $\epsilon > 0$ [Andoni and Nosatzki, 2020].

Computing edit distance in the MPC model has also received significant attention [Boroujeni et al., 2018, Hajiaghayi et al., 2019, Boroujeni and Seddighin, 2019]. Hajiahgayi, Seddighin, and Sun [Hajiaghayi et al., 2019] and Boroujeni, Ehsani, Ghodsi, Hajiaghayi, and Seddighin [Boroujeni et al., 2018] give MPC algorithms for edit distance in constant and logarithmic rounds, respectively, achieving $(1 + \epsilon)$, and $(3 + \epsilon)$ approximations. However, their algorithms are not space-efficient due to the utilization of superlinear total memory. Later works [Hajiaghayi et al., 2019, Boroujeni and Seddighin, 2019] improved the total memory requirement but still failed to achieve space-efficiency. Presently, there does not exist any algorithm to compute edit distance exactly or approximately in the MPC model with total space $\mathcal{O}(n^{1+o(1)})$ and sub-polynomial rounds. The existing MPC lower bounds [Ghaffari et al., 2019, Nanongkai and Scquizzato, 2020] do not shed any light on this stark absence, and prior PRAM results [Apostolico et al., 1990] cannot be simulated in MPC as the number of MPC machines is limited to sub-linear.

5.1.1 Our Contribution and Techniques

In this paper we investigates the edit distance problem in the MPC model while presenting new space-efficient algorithms and lower bound results.

Exact Algorithm for Edit Distance in the MPC Model. We begin by introducing a space-efficient MPC algorithm that computes the exact edit distance between a pair of strings, each of length n, in $\mathcal{O}(n^\epsilon)$ rounds.

Theorem 5.1.1. *For every $\epsilon \in (0, 1)$, there is an MPC algorithm that, given strings X and Y, computes $\mathsf{ed}(X, Y)$ using $\mathcal{O}(n^\epsilon)$ communication rounds with $\mathcal{O}(n^\epsilon)$ processors and $\mathcal{O}(n)$ total memory.*

In this algorithm, we transform the textbook dynamic programming solution for edit distance into an MPC-compatible form, resulting in an $\mathcal{O}(n^\epsilon)$-round algorithm with tight $\mathcal{O}(n)$ total memory. We start by briefly discussing the classic solution before discussing the challenges and techniques for adapting it to the MPC model. Given two strings X, Y of length n, we define a dynamic program table $\mathcal{D}$ as $\mathcal{D}[i, j] := \mathsf{ed}(X[0\,..\,i), Y[0\,..\,j))$, representing the edit distance between prefixes of X and Y of lengths i and j, respectively. To compute the entries of $\mathcal{D}$ and determine $\mathcal{D}[|X|, |Y|] = \mathsf{ed}(X, Y)$, we use the following recursive formula:

$$\mathcal{D}[i, j] = \min(\mathcal{D}[i-1, j] + 1, \mathcal{D}[i, j-1] + 1, \mathcal{D}[i-1, j-1] + \delta(X[i], Y[j])) \tag{5.1}$$

where $\delta(X[i], Y[j]) = 0$ if $X[i] = Y[j]$, otherwise 1. We also have base cases

$$\mathcal{D}[i, 0] = \mathsf{ed}(X[0\,..\,i), \varepsilon) = \mathsf{ed}(\varepsilon, Y[0\,..\,i)) = \mathcal{D}[0, i] = i$$

for any $i \in [0\,..\,n]$ where ε is the empty string. See the original works [Vintsyuk, 1968, Needleman and Wunsch, 1970, Sellers, 1974, Wagner and Fischer, 1974] for further discussion of this dynamic programming solution in the sequential setting.

In the MPC setting, the primary challenge in computing this table arises from the limitation that a single processor, with local memory of size $\mathcal{O}(S)$, (where $S = n^{1-\epsilon}$), can only store substrings of X and Y with a maximum length of $\mathcal{O}(S)$. Consequently, each processor can only compute an $\mathcal{O}(S) \times \mathcal{O}(S)$ block of $\mathcal{D}$ in a single communication round, as the recursive formula for $\mathcal{D}[i, j]$ necessitates access to X and Y to compute $\delta(X[i], Y[j])$. Thus if the table is computed row-by-row without any sophistication, it would require $\mathcal{O}(n/S)$ communication rounds per row and up to $\mathcal{O}(n^2/S^2)$ communication rounds to compute the entire $\mathcal{D}$. Instead, we compute $\mathcal{D}$ in a diagonal fashion. We initiate the process by making a processor locally compute entries $\mathcal{D}[0\,..\,S, 0\,..\,S]$ and pass the boundaries of the computed entries to two processors during the first communication round. Subsequently, one processor receives $\mathcal{D}[S, 0\,..\,S]$ and computes entries $\mathcal{D}[S\,..\,2S, 0\,..\,S]$, while the other processor receives $\mathcal{D}[0\,..\,S, S]$ and computes entries $\mathcal{D}[0\,..\,S, S\,..\,2S]$. These two processors then pass their boundary entries to three processors in total, continuing the process. By computing S diagonals of $\mathcal{D}$ during each communication round, we reduce the number of rounds needed to compute all of $\mathcal{D}$ from $\mathcal{O}(n^2/S^2)$ to $\mathcal{O}(n/S) = \mathcal{O}(n^\epsilon)$ rounds.

To ensure our algorithm operates within $\mathcal{O}(n)$ total memory, several essential considerations must be taken into account. Firstly, substrings of X and Y must be exchanged between proces-

sors during each communication round. This guarantees that processors responsible for computing each diagonal acquire the necessary substrings of X and Y to calculate their assigned block of $\mathcal{D}$. Secondly, we must ensure that a processor does not exceed their $\mathcal{O}(S)$ local memory while computing boundary values. Although each processor only needs to transmit $\mathcal{O}(S)$ boundary values to neighboring processors, they are in charge of computing $\mathcal{O}(S^2)$ entries in total. To address this, we ensure that a processor only stores $\mathcal{O}(S)$ computed entries at any given time, avoiding unnecessary communication rounds. Similar to our global strategy of computing diagonals of $\mathcal{D}$, we can compute diagonals of a block locally and only store a single diagonal at a time to avoid storing too many intermediate entries of $\mathcal{D}$ when computing the boundary values. These additional details and techniques collectively allow our algorithm to run within $\mathcal{O}(n)$ total memory. Further elaboration on this algorithm can be found in Section 5.3.

Approximate MPC Edit Distance. Next, we give the first space-efficient MPC algorithm to approximate the edit distance in sub-polynomial communication rounds. We say an algorithm $\mathcal{A}$ *α-approximates* a problem X if, for all instances v of problem X, it holds that $X(v)/\alpha \leq \mathcal{A}(v) \leq \alpha X(v)$. Formally we show the following.

Theorem 5.1.2. *Let X and Y be strings with $n = |X|$. For $\epsilon \in (0,1)$ and $\Theta(1) \leq r \leq \Theta(\frac{\log n}{\log\log n})$ where $n^{1-\epsilon} \geq n^{1/r}$, there is an MPC algorithm that $\mathcal{O}(r \cdot n^{1/r})$-approximates $\mathsf{ed}(X,Y)$ in $\mathcal{O}(r \cdot \epsilon^{-1})$ rounds of MPC using $\mathcal{O}(n^{\epsilon})$ processors and $\tilde{\mathcal{O}}(n \cdot (\log n)^{\mathcal{O}(r)})$ total memory.*

In the statement of Theorem 5.1.2, the parameter r governs a trade-off between the round complexity, the total space complexity, and the approximation ratio of our MPC algorithm. When $r = \Theta(1)$, the algorithm runs in constant communication rounds, uses near-linear space, and gives a polynomial approximation ratio $\mathcal{O}(n^{1/r})$. At the other extreme, when $r = \Theta(\frac{\log n}{\log\log n})$, the approx-

imation ratio decreases to poly-logarithmic, but the space complexity becomes $n^{1+\Theta(1)}$. As the focus of this work is on space-efficient MPC algorithms, we observe that the total memory becomes $n^{1+o(1)}$ as long as $r = o(\frac{\log n}{\log\log n})$. In particular, when $r = \Theta(\sqrt{\log n/\log\log n})$, the algorithm achieves a $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$ approximation ratio using sub-logarithmic communication rounds and $n \cdot 2^{\mathcal{O}(\sqrt{\log n \log\log n})}$ total memory. It remains an open problem to find a poly-logarithmic-factor approximation algorithm that still uses $n^{1+o(1)}$ total memory and $n^{o(1)}$ communication rounds.

Next, we provide a brief overview of the algorithm that is an MPC simulation of the approximate edit distance algorithm of Andoni, Krauthgamer, and Onak [Andoni et al., 2010] (or, more precisely, its recent simplified implementation by Bringmann, Cassis, Fischer, and Nakos [Bringmann et al., 2022a]). In this approach, we avoid computing the entire table $\mathcal{D}$ with n^2 entries. Instead, we approximate edit distances between substrings of X and Y using a tree with a small height and low degree. Each node in the tree corresponds to a substring of X, and each level represents a partition of X. At each node, we determine the minimum edit distance between its corresponding substring of X and various substrings of Y (the selection of the substrings of Y depends on the value of $\mathsf{ed}(X, Y)$). These fragment approximations are recursively combined, starting from the leaf nodes where at each leaf node we compute the exact edit distance between a single character of X and various characters of Y until we reach the root, which produces our approximation for $\mathsf{ed}(X, Y)$.

However, an optimal sequence of edits might align a substring of X starting at position i to a substring of Y starting at position j where i and j are far apart. Thus despite the recursive approach, we may need to compute the edit distance between up to $k \geq \mathsf{ed}(X, Y)$ substrings of Y for each substring of X. Therefore, one primary challenge in simulating this algorithm in MPC arises when k is large. In such cases, assigning a single processor with local memory

$\mathcal{O}(S)$ to each node of the tree proves inadequate, as an efficient method is required to compute k approximations per node. To address this, we allocate k/S processors for each node, with each processor computing a specific subset of the k different approximations. Additionally, when recursively combining the approximation results, a careful and synchronized approach is necessary to keep the number of communication rounds bounded. A pivotal step in this approach involves an elegant reduction to the *prefix sums* problem (with respect to various associative operators), that is solvable in constant rounds and with nearly linear total memory in MPC [Goodrich et al., 2011]. We show that leveraging the prefix sum problem enables the combination of k approximations across multiple processors in constant communication rounds and nearly linear total memory. For a more comprehensive understanding of the algorithm and this non-trivial reduction, please refer to Section 5.4.

Another big challenge in simulating the serial approximate edit distance algorithm arises when each of the $\mathcal{O}(n)$ leaf nodes (and subsequent internal nodes) demands k computations. This becomes particularly problematic when k is large, posing a risk of surpassing our total linear memory budget in MPC. To this end, the serial algorithm [Andoni et al., 2010, Bringmann et al., 2022a] employs a randomized pruning mechanism. While this slightly degrades the approximation guarantees, it effectively manages the challenge. The expected number of remaining nodes becomes $n^{1+o(1)}/k$, thereby requiring nearly linear total memory for the entire algorithm. The challenge lies in the fact that since the randomness is determined after the MPC algorithm starts, our algorithm needs to adaptively allocate processors to nodes without the ability to schedule assignments deterministically (the schedule remains unknown before the algorithm runs). To address this, we create a request set for processors to be assigned to node computations and utilize a reduction to the prefix sum problem to allocate processors and provide them with the necessary data for their

tasks.

MPC Orthogonal Vector Conjectures. We complement the above algorithms by introducing new lower-bound results for computing edit distance in the MPC model. To achieve this, we present the Strong O.V. conjecture and subsequently provide a reduction from O.V. to edit distance in the MPC model. Motivated by the fine-grained complexity and lack of lower bounds in the MPC model, our proposed Strong Orthogonal Vector Conjecture serves as the first super-logarithmic lower bound in MPC as well as the first lower bound conjecture for a non-graph problem in MPC.

Conjecture 5.1.3 (Strong (MPC) Orthogonal Vector Conjecture). For every constant $\epsilon \in (0, 1)$, there is no MPC algorithm that solves O.V. using $n^{\epsilon-\Omega(1)}$ communication rounds, $\Theta(n^{\varepsilon})$ processors, and $n^{1+o(1)}$ total memory.

Next, we provide some insights guiding our conjecture. A direct solution to O.V. involves computing the product of all $\mathcal{O}(n^2)$ pairs of vectors, which poses a considerable communication challenge in MPC. The computation of the dot product necessitates the vectors to be co-located on the same processor at a given time step. Consider a processor in our MPC model with local memory denoted as $S = n^{1-\epsilon+o(1)}$, where $\epsilon \in (0, 1)$. If the total memory across all processors is $n^{1+o(1)}$, implying $\mathcal{O}(n^{\epsilon})$ processors, then in a single communication round, at most $\mathcal{O}(n^{2-\epsilon+o(1)})$ pairs of vectors can be stored together on some processor. Consequently, any O.V. algorithm required to check all vector pairs will need $\Omega(\frac{n^2}{n^{2-\epsilon+o(1)}}) = n^{\epsilon-\Omega(1)}$ communication rounds in MPC. Despite a processor having the capability for unlimited local computations in a single round, the restricted local memory mandates that an O.V. algorithm employs multiple communication rounds to ensure the vectors are on the same processor at some point.

Even beyond straightforward solutions, the MPC model does not appear to offer improve-

ments in round complexity for an O.V. algorithm with $\mathcal{O}(n^{1+o(1)})$ total memory. For example, as an alternative strategy, rather than storing every pair of vectors on a processor during a given time step, we could adopt a distributed approach by only retaining pairs of coordinates from each vector pair. However, as we assume vectors can have small dimensions, such as $\log n$, this does not lead to any meaningful improvement in MPC complexity. To further improve on this, we may try to compute only certain bit products and show that the remaining ones can be inferred from these. Nevertheless, it is easy to construct examples where such inferences do not offer any improvement.[1] Thus with the above intuition, the O.V. algorithm is required to store the complete product (each coordinate) of every pair of vectors. However, without compressing this information, there is no way to improve the round complexity of the MPC algorithm.

In the above scenario, each machine has unrestricted processing time. However, if we restrict the MPC model where each processor has limited computational capacities between communication rounds, then we can propose the Weak O.V. conjecture that follows directly from the Strong Exponential Time Hypothesis (SETH). Such a model is well-motivated since real-world machines do have bounded computational capacity. We define a *quadratic-bounded computation MPC model*, where each processor with local memory $\mathcal{O}(S)$ can perform $\mathcal{O}(S^2)$ local computations per communication round.

Conjecture 5.1.4 (Weak (MPC) Orthogonal Vector Conjecture). For every constant $\epsilon \in (0, 1)$, there is no quadratic-bounded computation MPC algorithm that solves O.V. using $n^{\epsilon-\Omega(1)}$ communication rounds, $\Theta(n^{\varepsilon})$ processors, and $n^{1+o(1)}$ total memory.

[1]For an example of inferring bit products, consider bits a, b, c. First, let $a \cdot b = 0$ and $b \cdot c = 1$. An algorithm can infer that $a \cdot c = 0$ without multiplying the bits explicitly from these two products. However, even this simple inference falls short for some assignments of a, b, c. If we have $a \cdot b = 0, b \cdot c = 0$, then we cannot distinguish between the cases when $a = c = 1$ or at least one of a and c are 0. Thus, it is not clear whether an inference-based approach can lead to improved efficiency.

The Weak O.V. conjecture follows from SETH.

Observation 5.1.5. *The Weak O.V conjecture is true assuming SETH.*

Proof. Consider a quadratic-bounded computation MPC algorithm for the Orthogonal Vector problem. In a single round, each processor used in the algorithm can execute a maximum of $n^{2-2\epsilon+o(1)}$ computations. With $\mathcal{O}(n^{\epsilon})$ processors, the total computation performed in one round is $n^{2-\epsilon+o(1)}$. Therefore, by the Orthogonal Vector Conjecture, the algorithm must use at least

$$\frac{n^{2-o(1)}}{n^{2-\epsilon+o(1)}} = n^{\epsilon-o(1)}$$

communication rounds unless SETH is false. □

From MPC O.V. Conjectures to Edit Distance in the MPC model. Next, we introduce a reduction from O.V. to edit distance that, together with the Strong O.V. conjecture, establishes the first hardness for space-efficient edit distance computation in the MPC model.

Theorem 5.1.6. *For every constant $\epsilon \in (0, 1)$, if there is an MPC algorithm that solves exact string edit distance in MPC using R communication rounds, $\Theta(n^{\epsilon})$ processors, and $n^{1+o(1)}$ total memory, then there is an MPC algorithm that solves O.V. in $\mathcal{O}(R)$ rounds with the same space complexity.*

Our MPC reduction builds on the reduction of Backurs and Indyk [Backurs and Indyk, 2018] converting an instance of O.V. to an instance of edit distance. The key technique in their reduction involves vector gadgets, that converts each input vector into a string with size linear in the dimension d of the vectors. In this construction, orthogonal vectors have an edit distance of d, while

non-orthogonal vectors have an edit distance of $3m + (d - m)$, where m is the dot product of the vectors. These vector gadgets are concatenated with additional padding to ensure that any optimal edit sequence must align the gadgets of orthogonal vectors rather than non-orthogonal ones. Consequently, if the edit distance between the final pair of strings is below a certain threshold, the original sets of vectors must have included an orthogonal pair. In Section 5.5, we prove that this reduction can be performed in MPC with $\tilde{\mathcal{O}}(n)$ total memory and $\mathcal{O}(1)$ communication rounds and provide the reduction for completeness.

Using the above theorem, we can then give the first conditional hardness for edit distance in MPC. This justifies the significance of our approximation algorithm. It is noteworthy that our space-efficient exact algorithm for MPC edit distance aligns with the conjectured round complexity of $\mathcal{O}(n^{\epsilon})$, establishing its tightness.

Corollary 5.1.7. *If the Strong MPC Orthogonal Vector Conjecture is true, then any MPC exact string edit distance algorithm using $\mathcal{O}(n^{\epsilon})$ processors for $\epsilon \in (0, 1)$ and $n^{1+o(1)}$ total memory requires $n^{\epsilon-o(1)}$ communication rounds.*

Further Consequences of Strong MPC O.V. Conjecture. The Strong MPC Orthogonal Vector Conjecture has significant implications for proving hardness results in the MPC model. This is because, in fine-grained complexity, SETH-based lower bounds typically require reducing k-SAT to the O.V. problem. Typical properties seen in these reductions are (i) the gadgets employed in the reduction are of small size and (ii) they can be computed independently for each vector. This allows the reductions to be implemented in the MPC model using a constant number of rounds and $\mathcal{O}(n^{1+o(1)})$ total space. Following the ubiquity of O.V. based reductions, this gives new lower bound results for several other polynomial-time problems in the MPC model. Consequently, there

is a broad range of problems e.g., LCS [Abboud et al., 2015a], dynamic time warping [Bringmann and Künnemann, 2015], Fréchet distance [Bringmann, 2014], graph diameter [Roditty and Williams, 2013], single source max flow [Abboud et al., 2015b] etc., for which there does not exist any $n^{\epsilon-o(1)}$-round MPC algorithm using $\mathcal{O}(n^{\epsilon})$ processors and $n^{1+o(1)}$ total memory space assuming the Strong MPC O.V. conjecture. Formally one can show the following.

Theorem 5.1.8. *For any problem X if there exists a reduction f from O.V. (with binary vector sets $A, B \subseteq \{0,1\}^d$) to X such that (i) for each vector $v \in A \cup B$, size of the gadget $f(v)$ is polynomial in d, and (ii) the gadgets $f(v)$ can be computed locally, then assuming the Strong MPC O.V. conjecture, no MPC algorithm can solve X with $\mathcal{O}(n^{\epsilon})$ processors for $\epsilon \in (0,1)$, $n^{1+o(1)}$ total memory and $n^{\epsilon-\Omega(1)}$ communication rounds.*

As an example next we briefly discuss the graph diameter problem which computes the maximum distance between any pair of vertices. The diameter of a graph is the maximum distance between any pair of nodes in the graph. If one can compute breadth first search (BFS) in the MPC model following the ideas of [Behnezhad et al., 2019c, Alexandr et al., 2019, Andoni et al., 2020] using logarithmic rounds, then such an algorithm can be utilized to yield a logarithmic round 2-approximation algorithm for the graph diameter problem. However, leveraging the strong MPC O.V. conjecture, we can establish that computing an $(1.5-\epsilon)$ (for any constant $\epsilon > 0$) approximation for graph diameter requires polynomially many rounds for sparse graphs with a linear number of edges in the number of vertices.

Corollary 5.1.9. *Assuming strong MPC O.V. conjecture, given an unweighted undirected graph G on n nodes and $\mathcal{O}(n^{1+o(1)})$ edges, no MPC algorithm can decide whether diameter of G is ≤ 2 or ≥ 3 using $\mathcal{O}(n^{\epsilon})$ processors for $\epsilon \in (0,1)$, $n^{1+o(1)}$ total memory and $n^{\epsilon-\Omega(1)}$ communication*

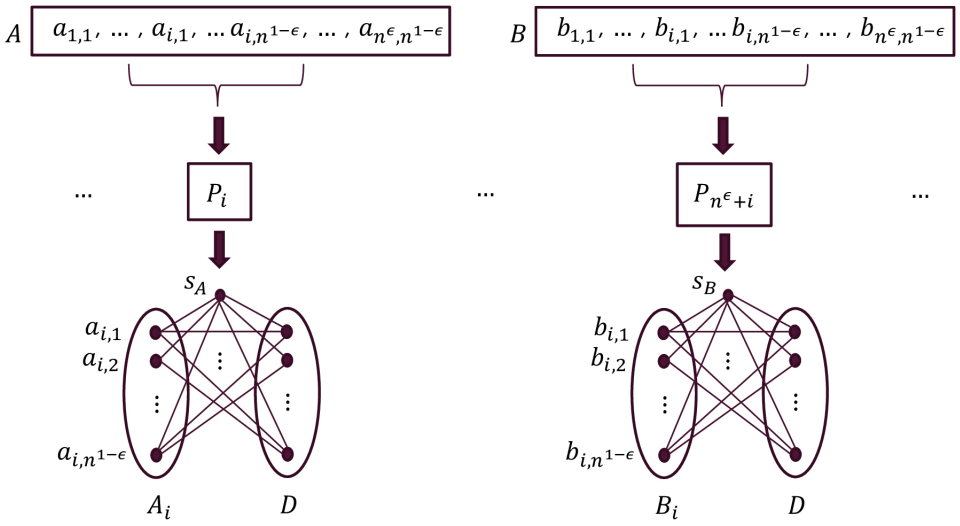


Figure 5.1: Example of reduction from O.V. to Graph Diameter problem in the MPC medel.

rounds.

Proof. Here we give a sketch of the proof. First, we discuss the reduction from O.V. to the graph diameter problem [Roditty and Williams, 2013]. Consider two sets of vectors $A, B \subseteq \{0,1\}^d$ that are involved in the O.V. problem. We generate a graph G as follows: G has vertex set A containing a vertex for each $a \in A$; vertex set B containing a vertex for each $b \in B$; vertex set D of size d; and two additional vertices, s_A and s_B. For each $a \in A$ and $d \in D$, we add an edge between a and d if $a[d] = 1$. We add similar edges between B and D. Finally, we connect all vertices in $A \cup D$ to s_A, all vertices in $B \cup D$ to s_B, and s_A to s_B. It can be easily verified that the diameter of G is 3 if and only if there exists a pair of vectors $a \in A$ and $b \in B$ that are orthogonal; otherwise, the diameter of G is 2.

To carry out this reduction in the MPC model, we first divide sets A and B, both of size n, into n^ϵ groups. Each group consists of $n^{1-\epsilon}$ vectors and is assigned to one processor. Using these vectors, each processor is able to compute locally the induced subgraph of G that corresponds to its group of vectors. Since the size of each locally induced subgraph is $\tilde{O}(n^{1-\epsilon+o(1)})$ the claim

follows. □

5.1.2 Other Related Works

Apart from the works [Boroujeni et al., 2018, Hajiaghayi et al., 2019, Boroujeni and Sedighin, 2019] mentioned earlier, there are additional studies focusing on parallel algorithms for edit distance and related problems. Krusche and Tiskin [Tiskin, 2010] work in the Bulk-Synchronous Parallel model of computation and use an algorithm for simple Monge matrix multiplication to solve the Longest Common Subsequence problem in logarithmic communication rounds. While the number of processors used is $\mathcal{O}(n^{\epsilon})$ for constant $0 < \epsilon < 1/3$, the local memory needed is $\mathcal{O}(n^{1-\epsilon/2})$, and so, an equivalent MPC algorithm would not be space-efficient. A recent work of [Chang and Zheng, 2024] provides the first weighted edit distance approximation algorithm in MPC with a $(1+\epsilon)$-approximation factor, $\mathcal{O}(S)$ local memory per processors, and $\mathcal{O}(n^2/S)$ total memory.

Before our conjecture, only a few studies had explored lower bounds for MPC in the context of other related problems. Roughgarden, Vassilvitskii, and Wang [Roughgarden et al., 2018] were able to prove that certain graph problems such as connectivity require $\Omega(\log_S n)$ communication rounds in MPC by utilizing a reduction of MPC algorithms to low-degree polynomials. Furthermore, borrowing hardness results from circuit complexity, they show that proving a problem requires $\omega(\log_S n)$ communication rounds in MPC would imply that $P \neq NC^1$, which would be a breakthrough result in circuit complexity. Charikar, Ma, and Tan [Charikar et al., 2020] extend these techniques to show similar results for the 1 vs. 2 cycle problem. In order to reason beyond the constant round lower bound barrier, Ghaffari, Kuhn, and Uitto [Ghaffari et al.,

2019] show conditional hardness results for MPC versions of maximum matching, vertex cover, maximal independent set, and coloring problems. All these are conditioned on the 1 vs. 2 cycle conjecture, claiming that connectivity cannot be done in better than $\mathcal{O}(\log n)$ rounds (aligning with the ideas from [Roughgarden et al., 2018] and matching an upper bound from [Rastogi et al., 2013]). Nanongkai and Michele Scquizzato [Nanongkai and Scquizzato, 2020] generalize these conditional hardness results and tie in complexity theory for sequential problems by showing that the 1 v. 2 cycle conjecture is equivalent to conjecturing that L, the set of problems solvable by deterministic Turing machine in logarithmic space, is not contained in the class of problems solvable in $o(\log n)$ rounds of MPC. Additionally, [Nanongkai and Scquizzato, 2020] shows that if NL, the set of problems solvable by nondeterministic Turing machine in logarithmic space, is not contained in the class of problems solvable in $o(\log n)$ rounds of MPC (a safer conjecture since $L \subseteq NL$), then several classic problems require at least $\Omega(\log n)$ MPC rounds such as perfect matching, single-source shortest paths, and diameter problem.

5.2 Preliminaries

5.2.1 String Notations

A *string* $Y \in \Sigma^n$ is a sequence of $|Y| := n$ characters from an *alphabet* Σ. For $i \in [0\,..\,n)$, we denote the ith character of Y with $Y[i]$. Formally, $Y[i\,..\,j)$ is a *fragment* of Y that can be represented using a reference to Y as well as its endpoints i, j. The fragment $Y[i\,..\,j)$ can be alternatively denoted as $Y[i\,..\,j-1]$, $Y(i-1\,..\,j-1]$, or $Y(i-1\,..\,j)$. A fragment of the form $Y[0\,..\,j)$ is a *prefix* of Y, whereas a fragment of the form $Y[i\,..\,n)$ is a *suffix* of Y. For strings $Y_1, Y_2, \ldots, Y_m \in \Sigma^n$, we let $\bigodot_{i=1}^{m} Y_i = Y_1 \bigodot Y_2 \bigodot \ldots \bigodot Y_m$ be the concatenation of these strings.

Additionally, $Y^m := \bigodot_{i=1}^{m} Y$ is the *mth power* of Y.

Problem 5.2.1 (String Edit Distance)**.** *Given two strings $X, Y \in \Sigma^n$, the* edit distance *of X and Y, $\mathsf{ed}(X, Y)$ is the minimum number of character insertions, deletions and substitutions needed to transform X into Y.*

5.2.2 Trees

A tree T is a connected, acyclic undirected graph. We consider rooted trees in this paper. A node v in a tree T is a *leaf* if it has no children and is an *internal node* otherwise. The *depth* of a node v is the number of edges from the root of the tree to v, and all nodes at depth i compose the ith *level* of the tree. The *height* of T is the maximum number of edges from the root node to a leaf node.

5.3 Exact MPC Edit Distance

In this section, we assume that input strings X, Y for the string edit distance problem have the same length. In the case that these strings have different lengths, we can pad the end of the shorter string by a special character that is not used in X or Y. We let $n = |X| = |Y|$, and we show that our MPC exact edit distance algorithm only needs tight $\mathcal{O}(n)$ total memory (unlike in the following section's approximation algorithm, which may utilize $n^{1+o(1)}$ total memory).

In order to provide an exact edit distance algorithm in MPC that matches the conditional hardness of Theorem 5.1.6, we adapt the classic dynamic programming solution we described in Subsection 5.1.1. Recall in this sequential solution, we compute table $\mathcal{D}$ for strings X, Y using Equation 5.1 in order to find $\mathsf{ed}(X, Y) = \mathcal{D}[|X|, |Y|]$. A tempting approach to compute all entries

of $\mathcal{D}$ would be to start at $\mathcal{D}[0,0]$ and compute all entries row-by-row or column-by-column. We note that an MPC processor can only compute as many entries of $\mathcal{D}$ as the size of its local memory $S = \mathcal{O}(n^{1-\epsilon})$. This local memory limitation combined with the observation that any processor computing an entry $\mathcal{D}[x,y]$ relies on the processor knowing the values of $\mathcal{D}[x-1,y]$ and $\mathcal{D}[x,y-1]$ means that even computing a single row or single column of $\mathcal{D}$ requires $\Omega(n/S) = \Omega(n^{\epsilon})$ communication rounds. Computing the full table of $\mathcal{D}$ in this way requires even more rounds and does not match the $\mathcal{O}(n^{\epsilon})$ communication round lower bound that we are aiming for.

To improve our computation of $\mathcal{D}$, we observe that instead of rows and columns, we may compute diagonals of $\mathcal{D}$. Using the observation that using Equation 5.1, computing each diagonal of $\mathcal{D}$ only relies on storing the values of the previous diagonal, we may compute a diagonal in MPC using a single round of communication since a diagonal of $\mathcal{D}$ has at most n entries and we have $\mathcal{O}(n)$ total memory. Since there are n diagonals of $\mathcal{D}$, we still need one further improvement to avoid using a communication round per diagonal. By assigning blocks of $\mathcal{D}$ with size S by S to each processor to compute, we can compute S diagonals of $\mathcal{D}$ using a single communication round. Therefore, in total we may use exactly $\mathcal{O}(n/S) = \mathcal{O}(n^{\epsilon})$ communication rounds to compute $\mathcal{D}$ in MPC.

First, we give pseudocode in Algorithm 22 to compute an S by S block of $\mathcal{D}$ locally on a single processor by computing diagonals of the block, and we analyze the complexity of this pseudocode in Lemma 5.3.1. One challenge of the local computation is that even an S by S block of $\mathcal{D}$ will exceed the local memory if the processor stores the entire block at once. To circumvent this issue, we use our previous observation that we only need to store the previous diagonal of $\mathcal{D}$ to compute a new diagonal, which we describe in the proof of Lemma 5.3.1.

Finally, we give pseudocode in Algorithm 23 to compute the exact edit distance of string X

and Y by computing diagonals of S by S blocks, and prove in Theorem 5.1.1 that this algorithm is space-efficient and runs in $\mathcal{O}(n^\epsilon)$ communication rounds of MPC, matching our conjectured lower bound.

Algorithm 22: Computes edit distance table $\mathcal{D}$ values for X and Y and returns border values

```
1  LocalDistance(L, T, X, Y):
2      for i from 0 to n do
3          D[i, 0] ← T[i];
4          for y from 1 to i − 1 do
5              x ← i − y;
6              D[x, y] =
                 min(D[x−1, y]+1, D[x, y−1]+1, D[x−1, y−1]+δ(X[x−1], Y[y−1]));
7          D[0, i] ← L[i];
8      for i from n + 1 to 2n do
9          for y from i − n to n do
10             x ← i − y;
11             D[x, y] =
                 min(D[x−1, y]+1, D[x, y−1]+1, D[x−1, y−1]+δ(X[x−1], Y[y−1]));
12     return D[n, 0 . . n] and D[0 . . n, n];
```

Lemma 5.3.1. *Given strings X, Y and arrays $\mathcal{L}, \mathcal{T}$ with $|X| = |Y| = n$ and $|\mathcal{L}| = |\mathcal{T}| = \mathcal{O}(n)$, Algorithm 22 can be completed in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space.*

Proof. We first consider the runtime of the algorithm. We note that any step within the loops requires constant time. As for the number of total iterations of each step, there are at most $n + 1$ iterations of both outer loops in Line 2 and Line 8 as i ranges from 0 to n and $n + 1$ to $2n$, respectively. The inner loop at Line 4 repeats less than n times since y ranges from 1 to $i - 1$, and i is at most n. The same is true for Line 9 since i begins at $n + 1$ in Line 8 and y begins at $i - n \geq 1$ and loops until n. Therefore, both inner and outer loops iterate at most $\mathcal{O}(n)$ times, and so, the total runtime of the algorithm is $\mathcal{O}(n^2)$.

Now, we argue that the algorithm can run in $\mathcal{O}(n)$ in constant space. First, we note that if

the algorithm simultaneously stores all values of $\mathcal{D}$ that are computed, it would already be over the space limits since the algorithm computes all $\mathcal{O}(n^2)$ entries of $\mathcal{D}$. Therefore, we describe a modification to the algorithm that clears some memory proactively throughout the algorithm. While we do not give the pseudo-code for this, the modification to the given pseudo-code is straightforward and described below.

We do not remove any entries of $\mathcal{D}$ that have a 0 or n in either index. Since indices range up to n, there are $4n$ such indices, which does not violate our memory constraints. As for all other entries of $\mathcal{D}$, during iterations of Line 8, we only store entries of $\mathcal{D}$ computed in the previous two iterations of the loop, and we clear all prior computed entries (except those with a 0 or n index). The same is true for iterations of Line 8, except in the first two iterations. In the first iteration of Line 8 we instead clear all entries of $\mathcal{D}$ computed prior except the last two iterations of Line 2, and in the second iteration of Line 8, we clear all entries of $\mathcal{D}$ computed prior except the last iteration of Line 2 and the previous iteration of the current loop. As argued above, since the inner loops at Line 4 and Line 9 run at most $\mathcal{O}(n)$ times, the algorithm computes only $\mathcal{O}(n)$ entries per iteration of the outer loops and therefore can store all computed entries of $\mathcal{D}$ from the previous two iterations and current iteration in $\mathcal{O}(n)$ memory.

Finally, we argue that the algorithm is still well-defined when only storing entries of $\mathcal{D}$ computed in the previous two iterations (and entries with a 0 or n index) using induction on i. To compute $\mathcal{D}[x, y]$, we need the values of $\mathcal{D}[x-1, y], \mathcal{D}[y, x-1],$, and $\mathcal{D}[x-1, y-1]$. When $i < 2$, Line 6 does not run. For $i = 2$, only $\mathcal{D}[1, 1]$ is computed, requiring $\mathcal{D}[0, 1], \mathcal{D}[1, 0], \mathcal{D}[1, 1]$, which are always kept in memory. Similarly for $i = 3$, only $\mathcal{D}[1, 2]$ and $\mathcal{D}[2, 1]$ are computed, which require $\mathcal{D}[1, 1]$ from the previous iteration and entries of $\mathcal{D}$ with a 0 in one of the indices. We now take an arbitrary $i > 3$. Notice that we compute $\mathcal{D}[x, y]$ in Line 6 and Line 11 for all $x, y \in [1, n]$

such that $i = x + y$. To compute $\mathcal{D}[x, y]$, we need the values of $\mathcal{D}[x-1, y], \mathcal{D}[y, x-1]$,, and $\mathcal{D}[x-1, y-1]$. Since $x+y = i$, we have that $x-1+y = x+y-1 = i-1$ and $x-1+y-1 = i-2$. All of the needed entries of $\mathcal{D}$ to compute any $D[x, y]$ with $x + y = i$ are from iteration $i - 1$ and $i - 2$, which by induction are stored in memory. □

Algorithm 23: Computes edit distance between X and Y in MPC

```
ExactMPCEditDistance(X, Y):
    n ← |X|;
    S ← n^{1−ε};
    P ← ∅;
    Distribute X_i = X[iS .. (i+1)S) to processor μ_i^X for i ∈ [0 .. n^ε);
    Distribute Y_i = Y[iS .. (i+1)S) to processor μ_i^Y for i ∈ [0 .. n^ε);
    Distribute T_{0,i} = {iS, iS+1, iS+2, ..., (i+1)S−1} to processor μ_i^Y for
      i ∈ [0 .. n^ε);
    Distribute L_{i,0} = {iS, iS+1, iS+2, ..., (i+1)S−1} to processor μ_i^X for
      i ∈ [0 .. n^ε);
    for i from 0 to n^ε − 1 do
        Send X_i and L_{i,0} from μ_i^X to μ_0^Y;
        Add μ_i^Y to P;
        for μ_j^Y in P in parallel do
            T_{i+j+1,j}, L_{i+j+1,j+1} ← LocalDistance(L_{i+j,j}, T_{i+j,j}, X_{i+j}, Y_j);
            Send L_{i+j+1,j+1} and X_{i+j} to μ_{j+1}^Y (except for j = n^ε − 1);
    for i from n^ε to 2n^ε − 1 do
        Remove μ_{i−n^ε}^Y from P;
        for μ_j^Y in P in parallel do
            T_{i+j+1,j}, L_{i+j+1,j+1} ← LocalDistance(L_{i+j,j}, T_{i+j,j}, X_{i+j}, Y_j);
            Send L_{i+j+1,j+1} and X_{i+j} to μ_{j+1}^Y;
    return T_{n^ε, n^ε−1}[S];
```

Theorem 5.1.1. *For every $\epsilon \in (0, 1)$, there is an MPC algorithm that, given strings X and Y, computes $\mathsf{ed}(X, Y)$ using $\mathcal{O}(n^\epsilon)$ communication rounds with $\mathcal{O}(n^\epsilon)$ processors and $\mathcal{O}(n)$ total memory.*

Proof. We prove that Algorithm 23 satisfies the theorem statement when each processor has local memory of size $\mathcal{O}(S) = \mathcal{O}(n^{1-\epsilon})$. First, we discuss the communication round complexity and

number of processors used by the algorithm. Distributing substrings of X and Y of length S as well as the additional arrays of length S in Line 7 and Line 8 to $\mathcal{O}(n^\epsilon)$ processors can be performed in a single communication round using $\mathcal{O}(n^\epsilon)$ processors with $\mathcal{O}(S)$ local memory. We note that after these beginning steps, the only processors used are μ_i^X and μ_i^Y for $i \in [0\,..\,n^\epsilon)$, so there are indeed only $\mathcal{O}(n^\epsilon)$ processors needed across the entire algorithm. Each iteration for $i \in [0\,..\,n^{\epsilon-1}]$ of the outer loop at Line 9 requires a single communication round to send X_i and $\mathcal{L}_{i,0}$ from μ_i^X to μ_0^Y since each only has size $\mathcal{O}(S)$. An additional communication round for each $i \in [0\,..\,2n^\epsilon - 1]$ is needed for all processors μ_j^Y to send $\mathcal{L}_{i+j+1,j+1}$ and X_{i+j} of size $\mathcal{O}(S)$ to processor μ_{j+1}^Y for $j \in [0\,..\,n^\epsilon - 1]$ since these communications can occur in parallel for all j during a single iteration i. Thus, there is an aggregate constant number of communication rounds per iteration of the outer loop at Line 9. A similar argument shows that a constant number of communication rounds is needed per iteration of the second outer loop at Line 15. Since there are $2n^\epsilon - 1$ iterations of outer loops at Line 9 and Line 15, Algorithm 23 uses $\mathcal{O}(n^\epsilon)$ communications rounds in total.

We now show that the algorithm can be completed with local memory of size $\mathcal{O}(S) = \mathcal{O}(n^{1-\epsilon})$ per processor. Each processor μ_i^X (μ_i^Y) initially stores substrings of X (Y) and array $\mathcal{L}_{i,0}$ ($\mathcal{T}_{0,i}$) of length $\mathcal{O}(S)$. We note that processors μ_i^X never store anything further. As for all other processors μ_i^Y, it is easy to see that the size of local memory is determined by the required space to compute the `LocalDistance` subroutine on Line 13 and Line 18. For a single call to the subroutine, the parameters $\mathcal{L}_{i+j,j}, \mathcal{T}_{i+j,j}, X_{i+j}, Y_j$ all have size $\mathcal{O}(S)$ and therefore fit into memory of size $\mathcal{O}(S)$. Additionally, by Lemma 5.3.1 and Algorithm 22, the subroutine itself requires memory of size $|X_{i+j}| = \mathcal{O}(S)$, and the return values $\mathcal{T}_{i+j+1}, \mathcal{L}_{i+j+1,j+1}$ of the subroutine are of size $\mathcal{O}(S)$ as well. We observe that the parameters and return values of each call to the subroutine are indexed by i, which increases each iteration (and therefore increases between each call to the

subroutine). Thus, a processor μ_i^Y does not need to store any parameters or return values with index $i-1$ during the ith iteration, and so, local memory of size $\mathcal{O}(S)$ fits all necessary data for Algorithm 23

Since there are $\mathcal{O}(n^\epsilon)$ processors each with local memory of size $\mathcal{O}(S) = \mathcal{O}(n^{1-\epsilon})$, Algorithm 23 requires $\mathcal{O}(n)$ total memory. As for the correctness of the algorithm, we note that our subroutine computes entries of $\mathcal{D}$ exactly according to Equation 5.1, the dynamic programming solution outlined in Subsection 5.1.1. Algorithm 23 returns $\mathcal{T}_{n^\epsilon, n^{\epsilon-1}} = \mathcal{D}[n, n]$, which is equivalent to the edit distance of X and Y. We leave further proof of this fact to previous works on the subject [Vintsyuk, 1968, Needleman and Wunsch, 1970, Sellers, 1974, Wagner and Fischer, 1974]. □

5.4 Approximate MPC Edit Distance

In this section, we provide the first MPC algorithm that approximates string edit distance in $\tilde{\mathcal{O}}(n^{1+o(1)})$ total memory and sub-polynomial communication rounds by adapting the serial algorithm of [Andoni et al., 2010] (as interpreted in [Bringmann et al., 2022a]). Previous MPC edit distance algorithms utilized super-linear total memory or a large number of communications and so, at the cost of an $n^{o(1)}$ approximation factor, we are the first to give a space-efficient MPC algorithm with sub-polynomial communication rounds.

To explain the challenge of MPC string edit distance in almost-linear memory and few communication rounds, we view computing the edit distance between strings X and Y from the equivalent perspective of trying to match the characters of X to characters of Y. Any number of characters in each string may be inserted, substituted, or deleted for a constant cost each, and we want

to find some perfect, non-crossing matching for the remaining characters of both strings such that matched characters have the same type. The minimum-cost sequence of edits that leads to a valid perfect matching defines our edit distance between X and Y, and it is easy to see that this is equivalent to the definition from Problem 5.2.1. One observation we can make is that if $\mathsf{ed}(X, Y) \leq k$ for some constant $k \in \mathbf{Z}$, a character at position i in X will be matched to a character at position j in Y such that $|i - j| \leq k$, otherwise $\mathsf{ed}(X, Y)$ would not be bounded by k. Therefore, if k is very small, smaller than the local memory of a processor in our MPC model, edit distance could be computed by partitioning X and Y into blocks that fit into local memory and calculating local edit distances between these blocks. Unfortunately, when k is much larger than the size of local memory, it is possible that characters at the start of X are matched to characters at the end of Y. In this case, an algorithm would have to compute local edit distances for all pairs of blocks of X and Y to find an optimal edit sequence, which would require up to $\mathcal{O}(n^\epsilon)$ communication rounds (as seen in our exact lower and upper bounds) or a super-linear total memory in MPC.

In order to mitigate the issue of having to check each pair of blocks, we leverage our approximation factor to limit the number of pairs of blocks our MPC algorithm has to compute the local edit distance for in expectation. First, in order to partition X and Y into such blocks, we define a useful structure and distance that allows us to approximate edit distance with a divide-and-conquer approach.

Definition 5.4.1 (Partition Tree [Andoni et al., 2010, Bringmann et al., 2022a])**.** A degree-B *partition tree* T for interval $I = [a \,..\, b]$ is a tree such that each internal node has degree B and each node v of the tree is labeled by a non-empty interval such that

1. Root node r has interval $I_r = I$.

2. Internal node v with children $v_0, \ldots, v_{B-1}$ has interval $I_v = I_{v_0} \cup \ldots \cup I_{v_{B-1}}$ where $|I_{v_i}| = |I_{v_j}|$ for all $i, j \in [0\,..\,B)$.
3. For the ith leaf node ℓ_i, the interval $I_{\ell_i} = \{i + a\}$ is just the ith element of I.

In our approximation algorithm, we will be constructing a partition tree for $I = [0\,..\,|X|)$ so that each level of the tree corresponds to a partition of X into blocks. At the leaf level, we always consider any node to represent a single character of X. Therefore, we assume that $|X|$ is a power of B, and in order to guarantee this we may pad the end of X with unused symbols at the cost of an additive B factor in the total memory of our algorithm where $B = n^{1/r}$ for some $\Theta(1) \leq r \leq \Theta(\frac{\log n}{\log \log n})$ (which does not change the overall total memory size of $\mathcal{O}(n)$). We assume that $|X| = |Y|$ for convenience, although when the input strings have unequal lengths, we may pad the shorter string with an unused symbol to increase its length. Since $\mathsf{ed}(X, Y) \geq ||X| - |Y||$, this padding only comes at the cost of an additional factor of 2 in the worst-case approximation ratio.

We now give the following recursive formula that we use to approximate the edit distance between strings using a partition tree.

Definition 5.4.2 (Partition Tree Distance [Andoni et al., 2010, Bringmann et al., 2022a])**.** Given strings X, Y, a degree-B partition tree T for interval $[0\,..\,|X|)$, and shift $s \in \mathbf{Z}$. For node v in T, we define the *partition tree distance* $\mathsf{td}(X, Y)$ as:

1. If v is a leaf, with $I_v = \{i\}$, $\mathsf{td}_{v,s}(X, Y) = \mathsf{ed}(X[i], Y[i + s])$. When $i + s \notin [0\,..\,|Y|)$, we assume $\mathsf{ed}(X[i], Y[i + s] = 1$.
2. If v is an internal node with children $v_0, .., v_{B-1}$, then

$$\mathsf{td}_{v,s}(X, Y) := \sum_{i=0}^{B-1} \min_{s' \in \mathbf{Z}} (\mathsf{td}_{v_i,s'}(X, Y) + 2|s - s'|).$$

Furthermore, we define $\mathsf{td}(X, Y) := \mathsf{td}_{r,0}(X, Y)$, where r is the root of T.

The following lemma from [Bringmann et al., 2022a] shows the relationship between partition tree distance and edit distance for two strings X and Y. We encourage readers to look to [Bringmann et al., 2022a] for proof of this and later cited statements as we want to keep the focus of this section on the general ideas and technical contributions we make when overcoming the difficulties of the MPC model.

Lemma 5.4.3 ([Bringmann et al., 2022a, Lemma 6]). *Let X, Y be strings, T be a degree-B partition tree for interval $[0\,..\,|X|)$, and of height $h = \log_b |X|$. Then,*

$$\mathsf{ed}(X, Y) \leq \mathsf{td}(X, Y) \leq 2Bh \cdot \mathsf{ed}(X, Y)$$

As seen from this lemma, by appropriately choosing our degree B, we can provide a good approximation of the edit distance.

Even though we have now defined a modular approach to computing an edit distance approximation, we have not reached an MPC algorithm that has the complexity we are targeting. At each node, according to the definition, we must check every integer shift of a block in Y, which may mean $\mathcal{O}(n)$ shifts per node. Since there are $\mathcal{O}(n)$ nodes in the partition, computing an edit distance for each shift will clearly violate either our target constant round or total memory complexity. We take two important steps to handle the leaf level, which we also utilize for all higher levels of the tree as well. First, we note that at any node, as observed previously if we know that $\mathsf{ed}(X, Y) \leq k$, we do not have to check any shift of a block of Y larger than k. By only checking shifts within the set $[-k\,..\,k]$, we bring down the number of computations at the leaf level to $\mathcal{O}(nk)$. As an aside, in the rest of the section, we assume we know the value of k such that $k = \mathsf{ed}(X, Y)$. In reality, as

we do not know k, we may run $\Theta(\log(n))$ copies of our algorithms given below in parallel with all powers of 2 from 2 to n in place of k.

Definition 5.4.4 (Capped Partition Tree Distance, [Bringmann et al., 2022a])**.** Given strings X and Y with partition tree T and $k \in \mathbf{Z}_+$, define the *capped partition tree distance* as follows:

$$\mathsf{td}^{\leq k}_{v,s}(X,Y) := \min\left(k, \sum_{i=0}^{B-1} \min_{s' \in [-k..k]} (\mathsf{td}^{\leq k}_{v_i,s'}(X,Y) + 2|s-s'|)\right).$$

As shown in [Bringmann et al., 2022a], we have $\mathsf{td}^{\leq k}_{r,0}(X,Y) = \min(k, \mathsf{td}(X,Y))$ where r is the root node of T.

For a node v, since $\mathsf{td}^{\leq k}_{v,s}(X,Y)$ is only an approximation of $\mathsf{td}_{v,s}(X,Y)$, we must be careful not to incur too much error when building up our approximation up the partition tree. We assign a *precision* β_v to each node v of the partition tree that informs our algorithm of the maximum additive error we are willing to incur from approximating the partition tree distance for this node. However, if we naively compute $\mathsf{td}^{\leq K}_{v,s}(X,Y)$ for some $s \in [-k,\ldots,k]$ from the approximations for v's children $v_0,\ldots,v_{B-1}$, each with their own approximation precision $\beta_{v_0},\ldots,\beta_{v_{B-1}}$, respectively, then we may incur error up to $\beta_{v_0} + \beta_{v_1} + \ldots + \beta_{v_{B-1}}$. If this sum is larger than β_v, then we risk blowing up the additive error factor of our approximation.

In order to mitigate the growth of additive error, we utilize an intricate technique introduced by [Andoni et al., 2010] called *precision sampling*. Using carefully chosen precisions according to an exponential distribution, we can use this technique to recover an approximation of $\mathsf{td}^{\leq k}_{v,s}$ for every $s \in [-k\,..\,k]$ from the approximations of its B children nodes.

Lemma 5.4.5 (Precision Sampling Lemma, [Andoni et al., 2010], [Bringmann et al., 2022a], [Bringmann et al., 2022b])**.** *Fix parameters $\gamma > 0$, $\alpha \geq 1$, $\beta \geq 0$, and let $A_1,\ldots,A_B \geq 0$ be real*

numbers. Independently sample $u_1, \ldots, u_B$ *from the exponential distribution* $Exp(\mathcal{O}(\log(\gamma^{-1})))$ *(for a sufficiently large hidden constant). There is an* $\mathcal{O}(B \log(\gamma^{-1}))$*-time algorithm* recover *satisfying for all* $\tilde{A}_1, \ldots \tilde{A}_B$ *with success probability at least* $1 - \gamma$*:*

1. *If* $\tilde{A}_i \geq \frac{1}{\alpha} \cdot A_i - \beta u_i$ *for all* i*, then* $\mathsf{recover}(\tilde{A}_1, \ldots, \tilde{A}_B, u_1, \ldots, u_B) \geq \frac{1}{2\alpha}\Sigma_i A_i - \beta$.
2. *If* $\tilde{A}_i \leq \alpha \cdot A_i + \beta u_i$ *for all* i*, then* $\mathsf{recover}(\tilde{A}_1, \ldots, \tilde{A}_B, u_1, \ldots, u_B) \leq 2\alpha\Sigma_i A_i + \beta$.

We have explained how we work with the capped partition tree distance; we still need to deal with the obvious issue that for large k, $\mathcal{O}(nk)$ computations per level of the partition tree will still greatly exceed our memory limitations (or cause us to use too many communication rounds). However, the magic of precision sampling (and its recovery algorithm) is that for a node v with children $v_0, \ldots, v_{B-1}$, we can choose β_{v_i} using a randomized process such that $\mathbb{E}[\beta_{v_i}^{-1}] = \mathcal{O}(\text{polylog}(n))\mathbb{E}[\beta_v^{-1}]$ and guarantee with high probability that for any node v with depth d_v in the partition tree, $\mathbb{E}[\beta_v^{-1}] = 1/k \cdot (\log n)^{\mathcal{O}(d_v)}$. Since $0 \leq \mathsf{td}_{v,s}^{\leq k}(X, Y) \leq |I_v|$, if $|I_v| \leq \beta_v$, then we can simply use 0 as our approximation for $\mathsf{td}_{v,s}^{\leq k}$ without violating our allowed additive approximation error. Furthermore, we can then bound the number of total nodes in the partition tree that we actually need to process, and it turns out that the remaining nodes become manageable in nearly linear total memory.

Definition 5.4.6 (Active Nodes)**.** Given strings X, Y with partition tree T and precisions β_v for each v in T, we call v an *active* node if $|I_v| > 1/\beta_v$. For all *inactive* nodes v in T with $|I_v| \leq 1/\beta_v$ and $s \in [-k \,..\, k]$, we substitute 0 for $\mathsf{td}_{v,s}^{\leq k}(X, Y)$.

Lemma 5.4.7 (Active Nodes Lemma, Lemma 31 [Bringmann et al., 2022a])**.** *Given a partition tree* T *with degree* B *such that for any node* v *in* T *with depth* d_v*,* $\mathbb{E}[\beta_v^{-1}] = 1/k \cdot (\log n)^{\mathcal{O}(d_v)}$*, the total number of active nodes is bounded by* $\frac{n}{k} \cdot \log(n)^{\mathcal{O}(\log_B(n))}$ *with constant probability.*

We have now described the general framework and organization of the approximation algorithm. There are a few details left to discuss to implement the algorithm in MPC that we describe next. In the case that k is larger than the local memory S of a single processor, we note that we must assign multiple processors per active node in order to compute all $\mathcal{O}(k)$ shifts for that node. By assigning $\mathcal{O}(k/S)$ processors per node, we can make sure that no processor has to exceed its local memory. According to Definition 5.4.4, we must compute a *range min query* for an active node v to approximate the partition tree distance for that node's interval. Let $\Delta_{v_i,s'}$ be the output of the recovery algorithm for children nodes $v_0, v_1, \ldots, v_{B-1}$ in T, $s \in [-k \,..\, k]$ that approximates $\mathsf{td}^{\leq k}_{v_i,s'}(X, Y)$, we want to calculate

$$\tilde{A}_{v_i,s} = \min_{s' \in [-k..k]} (\Delta_{v_i,s'} + 2|s - s'|).$$

This computation becomes more difficult when we must synchronize the computation across multiple copies of the node. We first partition the set of $\{-k, -k+1, \ldots, k\}$ across the processors designated to node v. Then, we argue that the range min query can be answered by reducing to a problem that has already been solved in MPC, prefix sum. We borrow the following result from [Goodrich et al., 2011] stating that prefix sum can be computed in MPC with constant communication rounds and $\mathcal{O}(n)$ total memory.

Theorem 5.4.8 ([Goodrich et al., 2011]; see also [Hu et al., 2019])**.** *Consider a binary associative operator $\oplus$ (that is, a semigroup). For every $\epsilon \in (0, 1)$, there is an MPC algorithm that, given a sequence $(a_i)_{i=1}^n$, computes the sequence of* prefix sums *$(s_i)_{i=1}^n$ with $s_i = \bigoplus_{j=1}^i a_i$, using $\mathcal{O}(\epsilon^{-1})$ communication rounds and $\mathcal{O}(n^\epsilon)$ processors with $\mathcal{O}(n^{1-\epsilon})$ local memory each.*

We now show an equivalence between range min queries and the *prefix min sum* problem,

a prefix sum problem with the minimum function standing in as the binary operator. Using this theorem and this equivalence, we can guarantee the same complexity for the two problems.

Lemma 5.4.9. *For every $\epsilon > 0$, there is an MPC algorithm that, given values $D_{-k}, D_{-k+1}, \ldots, D_k$, computes $\min_{s' \in [-k..k]} (D_{s'} + 2|s - s'|)$ for all $s \in [-k..k]$ using $\mathcal{O}(\epsilon^{-1})$ communication rounds and $\mathcal{O}(k)$ total memory with $\mathcal{O}(n^{1-\epsilon})$ local memory per processor.*

Proof. We show that the range min query described in the lemma statement is equivalent to the prefix minima. First, we observe that we can split this problem into two sub-problems as follows

$$\min_{s' \in [-k..k]} (D_{s'} + 2|s - s'|) = \min \left(\min_{s' \in [-k..s]} (D_{s'} + 2(s - s')), \min_{s' \in [s..k]} (D_{s'} + 2(s' - s)) \right).$$

First, we consider the first sub-problem for $s' < s$,

$$\begin{aligned} \min_{s' \in [-k..s]} (D_{s'} + 2(s - s')) &= 2s + \min_{s' \in [-k..s]} (D_{s'} - 2s') \\ &= 2s + \min_{s' \in [-k..s]} \tilde{D}_{s'} \end{aligned}$$

where we define $\tilde{D}_{s'} = D_{s'} - 2s'$. The value $\min_{s' \in [-k..s]} \tilde{D}_{s'}$ is a prefix minimum of the sequence $\tilde{D}_{-K}, \ldots, \tilde{D}_k$ and so, by Theorem 5.4.8, we can compute these values in the target space and communication round complexities. By symmetry, we can also solve the other sub-problems in the same space and rounds, and computing the final minimum on two values can be done locally on any processor after receiving the answers to the relevant two sub-problems. □

We note that in addition to range min queries, we show that the recovery algorithm of the Precision Sampling Lemma, Lemma 5.4.5 can also be done in constant rounds of MPC with no

additional memory requirements.

Algorithm 24: Recover edit distance approximation using precision sampling.

```
Recover (Ã_1, Ã_2, ... Ã_B, α, γ):
    λ ← O(α^{-2} log(γ^{-1}));
    Sample u_{i,j} ~ Exp(1) for i ∈ [B], j ∈ [λ] independently;
    u_i ← min_j u_{i,j} for each i ∈ [B];
    M̃_j = max_{i∈[B]} Ã_i / u_{i,j} for all j ∈ [λ];
    return ln 2 · median_{j∈[λ]} M̃_j;
```

Observation 5.4.10. *For every $\epsilon > 0$, there is an MPC algorithm that given $\tilde{A}_1, \tilde{A}_2, \ldots \tilde{A}_B$, outputs $A_1, A_2, \ldots, A_B$ satisfying Lemma 5.4.5 using $\mathcal{O}(\epsilon^{-1})$ communication rounds and $\mathcal{O}(B)$ total memory and $\mathcal{O}(n^{1-\epsilon})$ local memory per processor.*

Proof. We show the recovery algorithm for Lemma 5.4.5 from [Bringmann et al., 2022a] in Algorithm 24 for completeness. We note that the necessary min and median functions can be computed locally on each processor as λ is a constant. Lastly, the max functions can be computed in $\mathcal{O}(B)$ total memory according to Theorem 5.4.8. □

The final detail for our approximation algorithm we briefly mention before giving the full implementation is how we distribute the workload dynamically across the processors. Initially, we must construct the partition tree and assign precisions to each node in the tree. Since there are $\mathcal{O}(n)$ total nodes, we assign sets of S nodes to $\mathcal{O}(n/S)$ *leader* processors, where S is the size of the local memory per processor, and the leader processors compute the intervals and precision sampling parameters for each node from top to bottom of the tree. We intend to approximate $\mathsf{td}^{\leq K}v, s(X, Y)$ only for active nodes, implicitly taking 0 as approximation for inactive nodes. Once the leader processors compute the precision sampling parameters and identify all active nodes, the leader processor of each active node creates a request for a set of k/S *worker* processors to be assigned

the task of performing the computations related to that node. It is the worker processors that then compute the approximate partition tree distances in parallel and send these approximations to the workers handling the parent node. Further details on how we assign worker processors and distribute information between them are in the proof of Theorem 5.1.2 below.

Before describing the complete MPC implementation of the algorithm, we also mention how the computations within leaf nodes are organized. Unlike internal nodes, we compute the actual edit distance between characters of X and Y for leaf nodes, and so, we must distribute characters of X and Y efficiently in near-linear total memory. At a leaf node, we need one character of X and $\mathcal{O}(k)$ characters of Y according to Definition 5.4.2. However, again when $k \geq S$, this will require some synchronization between the processors storing Y and the workers responsible for any given leaf node. It turns out that broadcasting characters of Y to a set of relevant processors can be done with the prefix sum problem, similarly to our range min query solution, in constant rounds and $\mathcal{O}(mk)$ total memory, where m is the number of active leaf nodes. We leave the proof of the following lemma to the end of the section as the algorithm is long, though not conceptually complex.

Lemma 5.4.11. *For every $\epsilon > 0$, there is an MPC algorithm that, given a partition tree T with active leaf nodes $\ell_1, \ell_2, \ldots, \ell_m$ and an integer $k \in \mathbf{Z}$, computes $\mathsf{td}_{\ell_i,s}(X, Y)$ for all $i \in [1\,..\,m]$ and $s \in [-k\,..\,k]$ in $\mathcal{O}(\epsilon^{-1})$ communication rounds of MPC using $\mathcal{O}(n^\epsilon)$ processors and $\tilde{\mathcal{O}}(mk)$ total memory.*

In the proof of the following theorem, we give the full algorithm encapsulating the general framework and details described previously. Algorithm 25 gives the pseudo-code for computing $\mathsf{td}(X, Y)$, utilizing the subroutines for the leaf nodes, range min queries, and precision sampling

recovery as described above. We now argue about the complexity of simulating this algorithm in MPC.

Algorithm 25: Computes $\mathsf{td}(X,Y) \leq k$ for strings X, Y and constants $0 < \delta < \epsilon$ where $\mathsf{ed}(X,Y)$

```
ApproximateEditDistance (X, Y, ε, r, k):
    B ← n^{1/r}; S ← n^{1−ε};
    for ℓ from 0 to r do
        for i ∈ [B^ℓ] do
            Sample u_i^ℓ from Exp(λ);
            β_i^ℓ ← β_{⌊i/S⌋}^{ℓ−1} / u_{v_i};
            // Only active nodes do the following
            if n/B^ℓ > 1/β_i^ℓ then
                Create ⌈k/S⌉ copies v_{i,1}^ℓ, v_{i,2}^ℓ, ..., v_{i,⌈k/S⌉}^ℓ of ith node on level i;
                for j ∈ [⌈k/S⌉] do
                    S(v_{i,j}^ℓ) ← {−k + jS, −k + jS + 1, ... − k + (j + 1)S − 1};
                    x(v_{i,j}^ℓ) ← [i(n/B^ℓ), (i + 1)(n/B^ℓ) − 1];
    Δ_{v_i^r, s} ← LeafRequests({v_j^r : j ∈ [n] × [⌈k/S⌉]}) for
      i ∈ [n] × [⌈k/S⌉], s ∈ [−k .. k];
    for ℓ from r − 1 to 0 do
        for i ∈ [B^ℓ] do
            // Assigning 0 approximations for inactive nodes
            if n/B^ℓ ≤ 1/β_i^ℓ then
                Δ_{v_i^{ℓ+1}, s} ← 0 for all s ∈ {−k, ..., k};
            // Range min query, Lemma 5.4.9
            Ã_{i,s}^ℓ ← min_{s′∈{−k,...,k}} Δ_{v_i^{ℓ+1}, s′} + 2 · |s − s′|;
            Δ_{v_i^ℓ, s} ←
              Recover(Ã^ℓ_{⌊i/B^{ℓ−1}⌋, s}, Ã^ℓ_{⌊i/B^{ℓ−1}⌋+1, s} + 1, ..., Ã^ℓ_{⌊i/B^{ℓ−1}⌋+B−1, s} + B − 1) for
              s ∈ [−k .. k];
    Return min(min_{−k≤s≤k} Δ_{v_0^0, s}, k);
```

Theorem 5.1.2. *Let X and Y be strings with $n = |X|$. For $\epsilon \in (0,1)$ and $\Theta(1) \leq r \leq \Theta(\frac{\log n}{\log\log n})$ where $n^{1-\epsilon} \geq n^{1/r}$, there is an MPC algorithm that $\mathcal{O}(r \cdot n^{1/r})$-approximates $\mathsf{ed}(X,Y)$ in $\mathcal{O}(r \cdot \epsilon^{-1})$ rounds of MPC using $\mathcal{O}(n^{\epsilon})$ processors and $\tilde{\mathcal{O}}(n \cdot (\log n)^{\mathcal{O}(r)})$ total memory.*

Proof. Let $n = |X|$, and let $S = \tilde{\mathcal{O}}(n^{1-\epsilon})$ be the size of local memory per processor for $\epsilon > 0, \Theta(1) \leq r \leq \Theta(\frac{\log(n)}{\log\log n})$. Define $B = n^{1/r}$. We assume $S \geq B$ as per the theorem statement.

As mentioned previously, since we do not know k, we run $\Theta(\log(n))$ copies of the algorithm in parallel, only adding a logarithmic factor into the total memory and local memory of each processor. For the ith copy of the algorithm, we let $k = 2^i$, and our final solution is the output of the copy with the smallest k whose approximation is greater than its k, which will not change the asymptotic approximation guarantee of the algorithm.

We now analyze the steps and complexity of Algorithm 25. First, we discuss the partition tree construction within the loop beginning at Line 3. We note that a degree-B partition tree of height r has $m = 1 + B + B^2 + \ldots + n = \mathcal{O}(n)$ nodes, and we can assign an index ℓ to each node v_ℓ so that the ith child of the node v_j has label $\ell = i + jB$. To now construct the tree, we can use n^ϵ leader processors μ_i^L for $0 \leq i \leq n^\epsilon$. Leader i will be in charge of nodes with labels in $[iS \,..\, (i+1)S)$. Each leader can then define the interval I_v for each node v according to Definition 5.4.1 without any communication rounds. In addition to the interval, we need to determine the precision β_i for each node v_i in the partition tree. As per Line 6, the precisions depend on u_i, which can be sampled locally on each node's processor, and the precision β_j for the parent node v_j of v_i. Note that, due to our node labeling and deterministic distribution of nodes, all leader processors know the leader processor containing the children of their nodes. Since each leader processor contains S nodes and each processor has local memory of size S, we only need a single communication round to send all precisions of parent nodes at level d to children nodes at level $d + 1$ via their relevant leader processors. Therefore, in total, we need $\mathcal{O}(r)$ communication rounds, proportional to the height of the partition tree, to compute all precisions. Once we determine precisions and intervals for each node, each leader can determine if a node is active or inactive.

One important facet of the algorithm is making sure that the number of active nodes is at most $\mathcal{O}(\frac{n}{k} \cdot \log(n)^{\mathcal{O}(\log_B(n))})$. According to Lemma 5.4.7, this bound is true in expectation. To

guarantee this bound with high probability, we begin by running $\Theta(\log n)$ copies of the partition tree initialization described in the previous paragraph in parallel. This only adds a $\Theta(\log(n))$ factor to the total memory size of our algorithm. Each leader processor then computes the number of active nodes they contain, and the sum of all active nodes is computed in $\mathcal{O}(\epsilon^{-1})$ communication rounds, which can be done by Theorem 5.4.8 (though it is a much simpler problem than prefix sums). We then use the leaders from the partition tree copy with the smallest number of active nodes to guarantee the upper bound on the number of active nodes with high probability. Since each active node requires computing $\mathcal{O}(k)$ tree distance approximations, we therefore only need $\tilde{\mathcal{O}}(n \cdot \log(n)^{\mathcal{O}(\log_B(n))})$ total memory to store all such approximations. We assume this bound moving forward.

We next discuss how to adaptively assign *worker* processors for each active node. For each active node, we need k/S worker processors to compute the range min queries according to Lemma 5.4.9 (with a single worker responsible for multiple active nodes if $k < S$). However, since active nodes are not known until after we randomly assign their precisions, we must coordinate the assignment of worker processors. For each node, the leader processor determines whether the node is active and, if so, posts a request for workers handling the computations assigned to that node. We can number the requests with consecutive integer identifiers using the prefix-sums approach (see [Hu et al., 2019] for further details) in $\mathcal{O}(\epsilon^{-1})$ communication rounds. Then, we may assign worker processors $[ik/S \,..\, (i+1)k/S)$ to request i (with fractional assignments if $k < S$). The algorithm guarantees that the leader processor knows the identifiers of its requests in order to communicate with the assigned worker processors. The leader processor for active node v will need to send u_v, β_v, and I_v, as well as the (range of) identifiers of the worker processors for the parent node of v to each of its worker processors. This can be implemented in $\mathcal{O}(\epsilon^{-1})$ communi-

cation rounds using the standard broadcast subroutine (with each group of workers responsible for a given active node).

The algorithm partitions all shifts in $[-k \,..\, k]$ into sets of size S, and each worker processor is in charge of a single set in this partition, defined in Line 10. We handle the leaf nodes in Line 12 according to Algorithm 26 with $\mathcal{O}(\epsilon^{-1})$ communication rounds and $\mathcal{O}(n)$ total memory by Lemma 5.4.11. We then compute our approximations one level at a time, starting with the leaf level in the loop at Line 13. Worker processors for node v_i compute approximations $\tilde{A}_{i,s}$ of the partition tree distance according to Lemma 5.4.9 in Line 17 for all $s \in [-k \,..\, k]$ and the worker processors corresponding to the parent node of v_i receive all the approximations from its children nodes after precision sampling according to Observation 5.4.10 (including those of v_i). Specifically, if the jth worker processor is assigned the jth set of shifts from the partition of $[-k \,..\, k]$ discussed above, then that worker processor receives only the approximations for the shifts in the jth set. No worker processor receives more than S approximations, which fits in the local memory, and therefore we only need $\mathcal{O}(1/\epsilon)$ communication rounds per level of the partition tree. Any approximations not computed are evaluated as 0, according to Line 16. Once all B children approximations for a given shift are computed, the worker processors for the children run the precision sampling recovery algorithm according to Observation 5.4.10 and send the result to the worker processor of the parent node. This process of computing range min queries, sending the approximations to relevant parent worker processors, and then running the recovery algorithm is repeated one level of the tree at a time until we reach the root, which provides our final approximation.

For the entire algorithm, we have used $\tilde{\mathcal{O}}(n^{\epsilon})$ leader and worker processors and $\tilde{\mathcal{O}}(mk) = \mathcal{O}(n \cdot (\log n)^{\mathcal{O}(r)})$ total memory. Since we require a communication round per level of the tree to send the approximations between worker processors of children and parent nodes and the approxi-

mation subroutines for each level require $\mathcal{O}(\epsilon^{-1})$ communication rounds, we need $\mathcal{O}(r \cdot \epsilon^{-1})$ total communication rounds in total. Since we are simulating the algorithms of [Andoni et al., 2010], as implemented in [Bringmann et al., 2022a], the correctness of the output and approximation ratio follows from their results (see [Bringmann et al., 2022a, Theorem 1] for details). □

5.4.1 Proof of Lemma 5.4.11

Lemma 5.4.11. *For every $\epsilon > 0$, there is an MPC algorithm that, given a partition tree T with active leaf nodes $\ell_1, \ell_2, \ldots, \ell_m$ and an integer $k \in \mathbf{Z}$, computes $\mathsf{td}_{\ell_i,s}(X, Y)$ for all $i \in [1\,..\,m]$ and $s \in [-k\,..\,k]$ in $\mathcal{O}(\epsilon^{-1})$ communication rounds of MPC using $\mathcal{O}(n^{\epsilon})$ processors and $\tilde{\mathcal{O}}(mk)$ total memory.*

We discuss the steps and complexity of Algorithm 26. We assume that Algorithm 25 initializes the partition tree such that each leaf node ℓ stores its associated index $x(\ell)$ of string X and its set of associated shifts $S(\ell)$ (Lines 10 and 11 of Algorithm 25). We let $d = |S(\ell)|$ and assume that all (active) leaves have the same number of shifts.

First, we distribute all m leaves across $\lceil md/S \rceil$ processors and create a request for the characters of Y needed for each shift of an active node on these processors. This requires only a single communication round to distribute the leaves. We note that each leaf needs to compute k approximations and so, we need total space $\mathcal{O}(mk)$ to distribute and create all nodes and requests. The algorithm sends the entire request set to another set of $\lceil md/S \rceil$ request processors who sort the set of requests by the character index, which can be done in constant rounds with linear memory in the size of the request set according to [Goodrich et al., 2011].

Once we have sorted the requests, the algorithm identifies the first request α_i and the last

request β_i on request processor μ_i^R. The last request on processor μ_i^R is shared with the neighboring processor μ_{i+1}^R (except for the last processor in which $i+1$ would be out of bounds), which takes an additional communication round. For all unique character requests on request processors μ_i^R, the request is forwarded to the appropriate processors storing the requested character. In response, that processor provides the requested character. The only requests not answered will be requests shared by the previous processor as well according to β_{i-1}. Note that since requests are in sorted order, this guarantees that processors containing blocks of Y may respond to all requests made in this step in a single round of communication. Any character request that is needed more times than the local memory of a single processor will have requests on multiple subsequent request processors. Since we check β_{i-1} before requesting on each processor, only the leftmost request processor for a popular request will actually send out the request to the processors containing Y (and thus every processor storing S of the input characters will need to handle at most S requests, which is within its memory limits). We then distribute the response back to all subsequent request processors who need the character. The following lemma discusses how to do this using the prefix sums algorithm (Theorem 5.4.8).

Lemma 5.4.12. *Line 24 of Algorithm 26 can be completed in $\mathcal{O}(\epsilon^{-1})$ communication rounds of MPC using $\mathcal{O}(n^\epsilon)$ processors and $\mathcal{O}(mk)$ total memory.*

Proof. We formalize the problem as follows. We associate each request in the set of requests R with an index c in $[0\,..\,|Y|]$, the character $Y[c]$ (if already known) or ε (if the request has not been answered yet), and the destination processor μ_j^L, e.g., (c, ε, j). Note that the second position is changed to $Y[c]$ when a request is answered, and we consider ε to be lexicographically smaller than any character in our alphabet. Initially, when all requests are created, we sort the requests in

Algorithm 26: Computes tree distance requests for active leaf nodes.

```
1  LeafRequests(ℓ_1, ℓ_2, ..., ℓ_m):
2      S ← n^{1−ε}; P ← ⌈md/S⌉;
3      Distribute {ℓ_{iS/d}, ℓ_{iS/d+1}, ..., ℓ_{(i+1)S/d−1}} to processors μ_i^L for i ∈ [0, P);
4      for μ_i^L for i ∈ [0, P) in parallel do
5          for each leaf ℓ_j stored on μ_i^L do
6              Request X[x(ℓ_j)] from μ^X_{⌊j/S⌋};
7              R_i ← ∅;
8              for s ∈ S(ℓ_j) do
9                  R_i.append((x(ℓ_j) + s, i));
10         Send request set R_i to processor μ_i^R;
11     R ← ⋃_{i=0}^{n^ε−1} R_i;
12     Sort R across all processors μ_i^R by the first entry of each request in R;
13     for μ_i^R for i ∈ [0, P) in parallel do
14         α_i ← c where (c, j) is the first request in sorted order on μ_i^R;
15         β_i ← c where (c, j) is the last request in sorted order on μ_i^R;
16         Send β_i to μ_{i+1}^R;
17         p ← β_{i−1};
18         for each request (c, j) in sorted order do
19             if c ≠ p then
20                 Request Y[c] from μ^Y_{⌊c/S⌋};
21                 p ← c;
           // Lemma 5.4.12
22         Send Y[β_i] to all μ_j^R where α_j = β_i;
23         for each request (c, j) do
24             Send Y[c] to μ_j^L;
25     for μ_i^L for i ∈ [0, P) in parallel do
26         Return {ed(X[x(ℓ_j)], Y[x(ℓ_j) + s])} for all ℓ_j on μ_i^L, s ∈ S(ℓ_j);
```

order of their first position values in Line 14. For each set of subsequent requests for the same character at position c, the leftmost processor containing these requests receives $Y[c]$ in Line 20 and so, the leftmost request becomes $(c, Y[c], j)$, whereas all subsequent requests for position c stay as (c, ε, j'), where the values of j, j' are not important to our argument. We note that if we compare request triples by their first positions and use their second positions as a tie-breaker, the leftmost request for position c now satisfies the inequality $(c, Y[c], j) > (c, \varepsilon, j')$. We let $R_L(r)$ denote the set of requests that come before a given request $r \in R$ in the sorted set. For each

request for position c, answering the request is now equivalent to determining $\max\{r' \in R_L(r)\}$; this is precisely the prefix maxima problem; an instantiation of the prefix sums problem where the associative $\oplus$ operator is the maximum function. Using Theorem 5.4.8, this problem can be solved in the promised space and communication complexity. □

The final step is returning the requests (which have been answered by that point) to the original leaf processors, which can be done in one more round of communication.

5.5 Orthogonal Vector to Edit Distance in MPC

In this section, we give the reduction from [Backurs and Indyk, 2018] for orthogonal vector to edit distance and we prove that this reduction works in constant communication rounds of MPC with linear total memory. Similar to the graph diameter reduction as discussed in the further consequences of the string O.V. conjecture, this is another example of a reduction from O.V. where vector gadgets can be computed locally in MPC. For the sake of completeness, we give the reduction technique in more detail in this section. We name our two sets of d-dimensional vectors A and B where $n = |A| = |B|$ and for the reduction, we assume that $A, B \subseteq \{0,1\}^d$. With this assumption, any pair of non-orthogonal vectors must have a 1 in both vectors at the same index. At the heart of this reduction is a vector gadget that constructs a pair of strings such that the edit distance increases linearly in the number of indices that both vectors have a 1.

As per the notation of [Backurs and Indyk, 2018], the characters that compose our edit distance strings are integers in the set {1, 2, 3, 4, 5, 6}, but could instead be any set of symbols. We define $\ell_1 := 1000d$ and for an integer $x \in \{0,1\}$, the following bit gadget allows us to distinguish when two characters are both 1:

$$C_1(x) = \begin{cases} 2^{\ell_1}\ 0\ 1\ 1\ 1\ 2^{\ell_1} & \text{if } x = 0; \\ 2^{\ell_1}\ 0\ 0\ 0\ 1\ 2^{\ell_1} & \text{if } x = 1, \end{cases}$$

$$C_2(x) = \begin{cases} 2^{\ell_1}\ 0\ 0\ 1\ 1\ 2^{\ell_1} & \text{if } x = 0; \\ 2^{\ell_1}\ 1\ 1\ 1\ 1\ 2^{\ell_1} & \text{if } x = 1. \end{cases}$$

It is easy to verify that for $x, y \in \{0, 1\}$, $\mathsf{ed}(C_1(x), C_2(y)) = 3$ when $x = y = 1$, and otherwise, the edit distance is 1. This is the first building block that allows our edit distance to eventually signal that there exists a pair of orthogonal vectors. We next show two gadgets that we use to combine these bit gadgets. The following gadgets concatenate together the bit gadgets and put padding around both sides of the concatenated string.

We define a string $u = 1\ 0^{d-1}$ and integer $\ell_2 = 1000d^2$. For a vector $v \in \{0, 1\}^d$, we let v_i refer to the ith entry of v for $i \in [1, d]$ and define

$$G_1(v) := 4^{\ell_2} \left(\bigodot_{i=1}^{d} C_1(v_i) \right) 3^{\ell_1} \left(\bigodot_{i=1}^{d} C_1(u_i) \right) 4^{\ell_2}$$
$$G_2(v) := 3^{\ell_1} \left(\bigodot_{i=1}^{d} C_2(v_i) \right) 3^{\ell_1}.$$

We leave the formal proofs of the correctness of the gadgets of this section to [Backurs and Indyk, 2018], but for vectors $v_1, v_2 \in \{0, 1\}^d$, any optimal sequence of edits to transform $G_1(v_1)$ into $G_2(v_2)$ requires

$$2\ell_2 + \ell_1 + d + 2\min(v_1 \cdot v_2, u \cdot v_2)$$

edits. The main concept behind these gadgets is that due to the padding, there are only two edit sequences that can be optimal for any pair of vectors. The optimal edit sequence must either delete $\left(\bigodot_{i=1}^{d} C_1(v_i)\right)$ or $\left(\bigodot_{i=1}^{d} C_1(u_i)\right)$ depending on $\min(v_1 \cdot v_2, u \cdot v_2)$ since our bit gadgets have larger edit distance the more indices two vectors share that have a 1. Due to the length of the padding, which is quadratic in the dimension of the vectors, there is no other edit sequence that can require fewer edits than the two mentioned. To finish the reduction, we concatenate together these vector gadgets, with large sections of padding between each vector gadget, as seen in the following constructions for strings P_1 and P_2.

For any vector $v \in \{0,1\}^d$, let $t = |G_1(v)|$ and $T = 1000dt$. Let $f = 1^d$ and for sets of vectors from the orthogonal vector problem $A, B \subseteq \{0,1\}^d$, define

$$P_1 = \bigodot_{v \in V_1} (5^T G_1(v) 5^T)$$

$$P_2 = \left(\bigodot_{i=1}^{|V_1|} 5^T G_2(f) 5^T\right) \left(\bigodot_{v \in V_2} 5^T G_2(v) 5^T\right) \left(\bigodot_{i=1}^{|V_1|} 5^T G_2(f) 5^T\right)$$

$$P_1' = 6^{|P_2|} P_1 6^{|P_2|}.$$

Theorem 5.5.1 (Theorem 4 from [Backurs and Indyk, 2018])**.** *Given vector sets $V_1, V_2 \subseteq \{0,1\}^d$ where $n = |V_1| = |V_2|$, let $Y := 2|P_2| + n(2\ell_2 + \ell_1 + d + 2)$. If there is some $v_1 \in V_1, v_2 \in V_2$ such that $v_1 \cdot v_2 = 0$, then $\mathsf{ed}(P_1', P_2) \leq Y - 2$. Otherwise, $\mathsf{ed}(P_1', P_2) = Y$.*

Lemma 5.5.2. *Given an instance of the orthogonal vector problem with vector sets $A, B \subseteq \{0,1\}^d$ where $n = |A| = |B|$, there is an MPC algorithm that constructs P_1' and P_2 for A and B, respectively, with $\mathcal{O}(n^\epsilon)$ machines in $\mathcal{O}(1)$ communication rounds and $\tilde{O}(n)$ total memory.*

Proof. We go through the steps of the construction of P_1' and P_2. First, we note that since $\ell_1 =$

$\mathcal{O}(d)$, then $|C_1(x)|, |C_2(x)| = \mathcal{O}(d)$ for an integer $x \in \{0,1\}$. Next we consider $G_1(v)$ and $G_2(v)$ for a vector $v \in \{0,1\}^d$. Since $|C_1(x)| = \mathcal{O}(d)$ and $|C_2(x)| = \mathcal{O}(d)$, then $|\bigodot_{i=1}^{d} C_1(v_i)| = \mathcal{O}(d^2)$ and $|\bigodot_{i=1}^{d} C_2(v_i)| = \mathcal{O}(d^2)$. Similarly, since $u = 1\ 0^{d-1}$, we also have that $|\bigodot_{i=1}^{d} C_1(u_i)| = \mathcal{O}(d^2)$. Thus, $|G_1(v)| = \mathcal{O}(\ell_2 + d^2 + \ell_1) = \mathcal{O}(d^2)$ and $|G_2(v)| = \mathcal{O}(\ell_1 + d^2) = \mathcal{O}(d^2)$. For any processor storing vectors $v \in A$, the processor can locally construct $5^T G_1(v) 5^T$ for each vector in its memory since $|5^T G_1(v) 5^T| = \mathcal{O}(T + d^2) = \mathcal{O}(dt + d^2) = \mathcal{O}(d|G_1(v)| + d^2) = \mathcal{O}(d^3 + d^2)$ (we assume that dimension d is $polylog(n)$ and local memory of each processor is $\tilde{O}(n^{1-\epsilon})$. By a similar argument, any processor storing vectors of $v \in B$ can construct $5^T G_2(v) 5^T$. We assume that each processor knows $n = |A| = |B|$ and there are $\mathcal{O}(n^\epsilon)$ extra processors that can construct two copies of $\bigodot_{i=1}^{|V_1|} 5^T G_2(f) 5^T$ (one for the beginning of P_2 and one for the end of P_2. Finally, we assume that since each processor knows $n = |A| = |B|$ and d, we can use $\mathcal{O}(n^\epsilon)$ processors to construct two copies of $6^{|P_1|}$ (one copy for the beginning of P_1' and one copy for the end of P_1'). □

Theorem 5.1.6. *For every constant $\epsilon \in (0,1)$, if there is an MPC algorithm that solves exact string edit distance in MPC using R communication rounds, $\Theta(n^\epsilon)$ processors, and $n^{1+o(1)}$ total memory, then there is an MPC algorithm that solves O.V. in $\mathcal{O}(R)$ rounds with the same space complexity.*

Proof. The theorem statement follows directly from Theorem 5.5.1 and Lemma 5.5.2. □

Chapter 6: Massively Parallel Location-Sensitive String Problems

6.1 Introduction

One of the most well-studied topics in recent computer science history is the algorithmic aspects of string problems. The *longest palindrome substring* (LPS), the *longest common substring* (LCS), *string matching*, and *edit distance* are some of the more famous problems in this category. Some of these problems such as string matching, LCS, LPS, and *longest common prefix* (LCP) are *location-sensitive*, i.e. solutions to these problems require compiling information from continuous sequences of characters of the input string(s) and are dependent on the order of said characters and sequences. Efforts to solve these problems led to the discovery of several fundamental techniques such as *hashing algorithms* and *suffix trees* which are useful for solving location-sensitive problems. Both of these tools have numerous applications in several fields including DNA-sequencing, compiler design, anti-virus softwares, etc.

Suffix trees are tries made of the sorted suffixes of a given string, and computing a suffix tree is a classic problem in computer science literature. The concept was first introduced by Weiner [Weiner, 1973], and McCreight (JACM, 1976) [McCreight, 1976] showed that it is possible to create a *compressed* suffix tree of total size $O(n)$. Later, Ukkonen [Ukkonen, 1995] introduced an online $O(n)$-time algorithm that dynamically maintains an updated suffix tree as new characters are appended to the input string. There are numerous applications for the suffix tree data structure

such as *exact and approximate string matching*, *matching statistics*, *Lempel-Ziv decomposition*, and *longest repeated substring* [Gusfield, 1997]. Inspired by suffix trees, we discuss two particular applications of interest [Gall and Seddighin, 2020] at length in this paper:

1 Longest Palindrome Substring (LPS), and

2 Longest Common Substrings (LCS).

In addition to LPS and LCS, we consider the problem of Longest Common Prefix (LCP) queries, which is often a crucial step in suffix tree construction and can also be answered in constant time using an already-constructed suffix tree [Iliopoulos and Rytter, 2004].

Designing algorithms which are sublinear in space is crucial when the input data does not fit in the memory of available systems. In recent years, with the abundance of data and the increasing demand for large scale data processing, the size of datasets easily surpasses that of the typical machine's memory. Examples of the problem domains vary from social network graphs to DNA sequences, and efficiently analyzing such datasets has great economic and social impact. However, to address any problem subject to this restriction efficiently, we need alternative models of computation as the traditional RAM model is no longer feasible in practice. One approach to overcome this obstacle is to utilize a large pool of typical machines that have a substantially smaller amount of memory comparing to the input size. This simple idea is the key ingredient behind the modern distributed processing frameworks such as MapReduce, Hadoop, and Spark that are widely deployed across the industry at very large scales. The *Massively Parallel computation* (MPC) model is proposed by Karloff, Suri, and Vassilvitskii [Karloff et al., 2010], and enhanced in subsequent works [Andoni et al., 2014, Beame et al., 2017, Goodrich et al., 2011], to provide a theoretical foundation for algorithms implemented on such frameworks.

In the MPC model, problem data of size $O(n)$ is distributed among a set of machines each with sublinear memory, and algorithms proceed in several rounds. In each round, machines perform an arbitrary amount of computation on their local data independently. At the end of each round, machines can communicate with each other. Since the communication phase is often a major performance bottleneck in real world scenarios, the main goal in designing MPC algorithms is to optimize the round complexity while keeping total memory across all machines linear. Note that for some $0 < \epsilon < 1$, each machine has $O(n^{1-\epsilon})$ memory; during a single communication round, each machine may send and receive any number of messages such that the total size of the messages fits within its local memory constraints. In some of our algorithms below, we require that each machine may have a large enough local memory to store a message from every other machine in a single round of communication. To this end, we must make sure that the local memory of a machine $O(n^{1-\epsilon})$ is larger than the number of machines $O(n^{\epsilon})$, and so, we restrict ϵ to $0 < \epsilon \leq .5$. This restriction is well-motivated for the MPC model since with modern computing resources, the number of machines does not exceed the size of the local memory of each machine.

In this paper, we introduce a massively parallel framework for solving a wide range of *location-sensitive* string problems such as *string matching*, LCS, and LPS, which usually are easily solvable using either a *suffix tree* (or *hashing*) in the sequential setting. We simulate powerful tools such as the *suffix tree* of the given string and a novel MPC data structure for handling batches of LCP queries. We call this data structure an LCPQ oracle, and it is the first MPC data structure able to efficiently answer a large (even super-linear) number of LCP queries in a small number of communication rounds. In addition to demonstrating constant-round MPC reductions from the aforementioned problems of LPS and LCS to suffix trees, our framework provides the means to give algorithms using $O(1)$ communication rounds with tight $\widetilde{O}(n)$ total memory for LPS and

$\widetilde{O}(n^{1+\epsilon})$ memory for LCS by applying a technical analysis of string periodicity in each problem. Our novel LCPQ oracle is critical to the improved algorithms for LPS and LCS as well as the MPC suffix tree simulation.

Beyond discussing LCP, LPS and LCS, we also discuss improved algorithms for *suffix tree* construction. In the stronger *Adaptive* MPC (AMPC) model recently proposed in [Behnezhad et al., 2019d], each machine is subject to the same local memory and communication limitations as in MPC. However, the model is stronger because in each round, all communications can be stored in a large $O(n)$ shared memory that allows adaptive reads by all machines. We will utilize the shared memory to store *hashes* of suffixes, which is an interesting strategy that will allow us to avoid sending large messages between machines when comparing suffixes and instead just compare hash values of constant size in the shared memory. With this simpler communication, we show how to create an improved LCPQ oracle of a given string in AMPC and then subsequently the *suffix tree* in $\tilde{O}(n)$ memory and constant rounds [1].

6.1.1 Further Related Works

Among others, a specific family of graph problems known as *Locally Checkable Labeling* (LCL) problems – which includes vertex coloring, edge coloring, maximal independent set, and maximal matching to name a few – admit highly efficient MPC algorithms, and have been heavily studied during recent years [Behnezhad et al., 2019e, Behnezhad et al., 2019b, Assadi et al., 2019b, Assadi et al., 2019a, Behnezhad et al., 2019a, Ghaffari et al., 2020, Ghaffari and Uitto, 2019, Czumaj et al., 2018]. Another interesting family of problems studied in the MPC model are string problems.

[1] $\tilde{O}(n)$ is used to denote a linear size with a multiplicative polylogarithmic factor, i.e. $O(n\text{polylog}(n)) = \tilde{O}(n)$ and is used for convenience in the rest of the paper.

The *String Matching* problem, that is to find all occurrences of a pattern string in a text, has a wide range of applications on massive datasets from analyzing genomic data to Search Engine Indexing.

In a previous work, Hajiaghayi, Saleh, Seddighin, and Sun [Hajiaghayi et al., 2021] show that the *string matching* problem and also some of its variants can be solved efficiently in MPC. In particular, they show an $O(1)$ rounds algorithms for String Matching and Integer Convolution (using FFT) in MPC, and as a result they give efficient algorithms for String Matching with *wild-card* characters. Some building blocks in the string matching solution such as fast batch substring hash queries, which is a special case of block-based data structures, are also used in this paper (defined in later Section 6.4.1). Furthermore, our methods in this paper can be applied to the string matching problem (without wildcards) as an application of suffix trees [McCreight, 1976].

PRAM solutions to these problems also are heavily related to our work. Apostolico, Breslauer and Galil [Apostolico et al., 1994] give a PRAM solution for LPS with $O(n^2)$ total memory needed. While this solution requires more memory and rounds than we want if we were to convert it directly to MPC, our own optimal LPS algorithm utilizes their main idea that strings with multiple palindromes must be periodic. Iliopoulos and Rytter [Iliopoulos and Rytter, 2004] build a suffix tree from a simpler data structure, the suffix array, in PRAM in tight $\tilde{O}(n)$ total memory with $O(\log n)$ rounds. We follow this construction technique as well for our own suffix tree algorithm, with significant changes so that the transformation works in $O(1)$ rounds. Several other papers discuss related problems in PRAM such as [Lu and Lin, 1994] which solves the *longest common subsequence* problem and [Farach and Muthukrishnan, 1996] which solves suffix tree construction for binary strings, both in the PRAM model as well.

6.1.2 Problem Definition

A *string* $s \in \Sigma^n$ of length n is a sequence of n characters $s_0 s_1 \dots s_{n-1}$ from alphabet Σ. For any pair of integers $0 \leq l < r \leq n$, we use brackets and parentheses[2], i.e., $s[l : r)$, to denote a *substring* of string s which is a consecutive interval of characters $s_l s_{l+1} \dots s_{r-1}$ from index l to index $r-1$. The length of a substring $s[l : r)$ is equal to $r - l$. Every substring $s[l : n)$ which ends with s_{n-1} is called a *suffix* of s. Similarly, every substring $s[0 : r)$ which starts with s_0 is called a *prefix* of s.

One of the primitive tools in solving classical string problems is the **longest common prefix** (LCP) query. For two input strings of length n denoted by s and s', we define $\mathsf{LCP}_{s,s'}(i, j)$ as the length of the longest prefix shared between two suffixes $s[i : n]$ and $s'[j : n]$. For example, [Manber and Myers, 1993] uses an array containing $\mathsf{LCP}_{s,s}(\pi_i, \pi_{i+1})$ for every $0 \leq i < n-1$ as an auxiliary data structure alongside suffix arrays to mimic the capabilities of suffix trees. In the sequential setting, we can find the solution of LCP queries in $O(\log n)$ time by performing a *binary search* on the solution size and comparing the hashes of substrings of each suffix. Alternatively, if we already have a suffix tree built for string $s\#s'$, we may find the *lowest common ancestor* (LCA) in the suffix tree of the two suffixes $s[i : n]\#s'$ and $s'[j : n]$ (see following definition of suffix tree for more details).

Definition 6.1.1. Given strings s and s' each of length n, the LCP of indices i and j, denoted by $\mathsf{LCP}_{s,s'}(i, j)$, is equal to the length of the **longest common prefix** of suffixes $s[i : n)$ and $s'[j : n)$.

In the **longest common substring** problem (LCS), the input consists of two strings s and s' and our goal is to find the longest substring which is shared between the two strings. We assume

[2]This notation is often used for intervals $[l, r)$ that include l and exclude r.

that s and s' have the same length, which we denote by n. We use Σ to denote the alphabet of the strings.

In the **longest palindrome substring** problem (LPS), the goal is to find the longest substring of a given string s which reads the same both forward and backward. Let $\bar{s}$ denote the reverse of string s. Formally, a string s is a *palindrome* if $\bar{s} = s$. The length of s is also denoted by n and its alphabet is denoted by Σ.

The **suffix tree** of a given s is a rooted *trie*-like data structure in which every edge is labeled with a string, and it stores every *suffix* of s, i.e., $s[0:n), s[1:n), \ldots, s[n-1:n)$. Every node in the suffix tree represents a substring of s which is equal to the concatenation of the labels in the path from the root to this node. There is a leaf in the suffix tree for every *suffix* $s[i:n)$ $(0 \leq i < n)$, and the concatenation of the labels in the path from the root to this leaf results in suffix $s[i:n)$. See Figure 6.1 for an example for string $s =$ "bobocel". A *compressed* suffix tree of string s is a minimal suffix tree that does not have any vertex with degree 2, with an exception for the root, and thus the total number of vertices is bounded by $O(n)$. In addition, the total length of the labels on the edges of a compressed suffix tree is bounded by $O(n)$. A sequential algorithm can create a *compressed* suffix tree for any given string s with $O(n)$ running time [McCreight, 1976], and in $O(n)$ total time even if the characters are accessible one by one – and the suffix tree is maintained after every incoming character [Ukkonen, 1995]. Often we will need a suffix tree for two strings, and we follow the traditional approach of building a suffix tree on the concatenation of the two strings with a special character such as '#' between each string, e.g. $s\#s'$ represents the concatenation of strings s and s'.

Since the implementation of *compressed* suffix trees is complicated, Manber and Meyers [Manber and Myers, 1993] introduced a simpler and consequently less flexible data structure

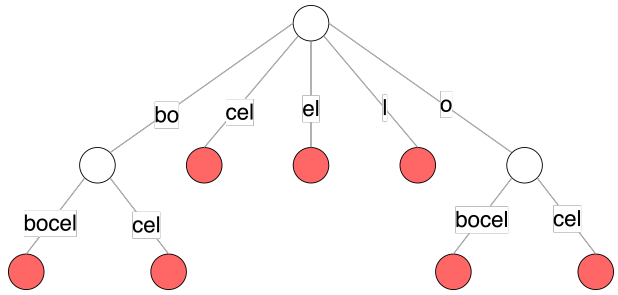


Figure 6.1: An example of *suffix tree* for example string "bobocel".

called **suffix array** as an alternative representation of suffix trees. Let $[n)$ denote the set of numbers $\{0, 1, 2, \ldots, n-1\}$. The suffix array of s is a permutation $\pi : [n) \to [n)$ which represents an order of the suffixes of s that is sorted in the lexicographical order, i.e., $s[\pi_i, n)$ is lexicographically smaller than $s[\pi_{i+1}, n)$ for every $i \in [n-1)$. When it is clear from context, we may simply refer to the ith suffix according to the order of the suffix array, i.e. $s[\pi_i, n)$, as suffix i. In the sequential setting, the suffix array of a given string s can be found in $O(n \log n)$ running time without using suffix tree by performing a *radix sort* for sorting the prefixes of length 2^i of every suffix at step i for a total of $O(\log n)$ steps.

6.2 KMR-Based Solutions

We start with a solution for location-sensitive problems using an existing, well-known KMR-based technique of [Karp et al., 1972]. While technically simple, we show these results for the sake of completeness, and the KMR algorithm gives us a point of comparison our own results. Our specialized algorithms for LPS and LCS we show in Sections 6.7 and 6.8 have a lower number of communication rounds and the LPS algorithm also improves the total memory size. Additionally the suffix tree construction we give utilizing our LCPQ ties the KMR approach and allows us to improve the communication rounds and total memory in AMPC (see Section 6.9.2)

In the KMR-based algorithm, we recursively define a hash function that we use in an $O(\frac{1}{\epsilon})$-round, $\widetilde{O}(n^{1+\epsilon})$-memory MPC algorithm for computing suffix array. First, we define this recursive hash function which maps substrings of an input string s to integers. When sorting strings in the construction of the KMR hash, we assume that we always sort in ascending order lexicographically. Additionally, if we refer to a substring $s[i, j]$ where $j \geq |s|$, we instead consider the substring $s[i, |s|)$.

Definition 6.2.1. Given a strings s and $\epsilon \in (0, 1)$, we build a KMR-based hash function $\mathcal{K}_{s,\epsilon}$. For all $1 \leq i \leq \frac{1}{\epsilon}$, $0 \leq j < n$, $\mathcal{K}_{s,\epsilon}(i, j)$ denotes a hash of $s[j, j + n^{i\epsilon}]$ defined as follows:

1 Sort all n^{ϵ}-length substrings of s and remove duplicates from the list. For all $0 \leq j < n$, let $\mathcal{K}_{s,\epsilon}(1, j)$ be the position of substring $s[j, j + n^{\epsilon})$ in this list.

2 Assume the hash function $\mathcal{K}_{s,\epsilon}(i-1, j)$ has been defined for all $0 \leq j < n$ where $1 \leq i \leq \frac{1}{\epsilon}$. For all $0 \leq j < n$, sort tuples $(\mathcal{K}_{s,\epsilon}(i-1, j), \mathcal{K}_{s,\epsilon}(i-1, j + n^{(i-1)\epsilon}), \mathcal{K}_{s,\epsilon}(i-1, j + 2n^{(i-1)\epsilon}), \ldots, \mathcal{K}_{s,\epsilon}(i-1, j + (n^{(i-1)\epsilon} - 1)n^{(i-1)\epsilon}))$ by comparing elements of the tuples from left to right and removing duplicate tuples from the list. For all $0 \leq j < n$, let $\mathcal{K}_{s,\epsilon}(i, j)$ be the position of substring tuple $(\mathcal{K}_{s,\epsilon}(i-1, j), \mathcal{K}_{s,\epsilon}(i-1, j + n^{(i-1)\epsilon}), \mathcal{K}_{s,\epsilon}(i-1, j + 2n^{(i-1)\epsilon}), \ldots, \mathcal{K}_{s,\epsilon}(i-1, j + (n^{(i-1)\epsilon} - 1)n^{(i-1)\epsilon}))$ in the sorted list.

Using sorting techniques in MPC, constructing the KMR-based hash function can be completed straightforwardly. Additionally, we prove the fact that by definition $\mathcal{K}_{s,\epsilon}(i, j)$ gives the position of the jth suffix of s in the suffix array when $i = 1/\epsilon$, so computing all such KMR-based hash values suffices for constructing the suffix array of s. Proof of this fact and the following theorems can be found in Appendix 6.11.

Theorem 6.2.2. *For $\epsilon \in (0, 1]$, there is an $O(\frac{1}{\epsilon})$-round* MPC *algorithm for a given string s using $\mathcal{K}_{s,\epsilon}$ for computing the* **suffix array** *of s in $\tilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor.*

With suffix array constructed, we may then build the suffix tree of any input string in constant rounds of MPC using the ideas of a PRAM algorithm from [Iliopoulos and Rytter, 2004] (proof in Appendix 6.11).

Theorem 6.2.3. *For $\epsilon \in (0, 1]$, there is an $O(\frac{1}{\epsilon})$-round* MPC *algorithm for suffix tree using $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

In the sequential setting, LPS and LCS each may be solved in linear time using suffix tree. For LPS, one can build a suffix tree on the input string $s = s_0 s_1 \dots s_{n-1}$ concatenated with the reverse string $\overline{s} = s_{n-1} \dots s_1 s_0$. The longest palindrome will then be the two suffixes in the tree with the lowest common ancestor where one suffix is from s and one suffix is from $\overline{s}$. Similarly for LCS, we build a suffix tree on the two input strings s and s' concatenated together and find the two suffixes with the lowest common ancestor in the suffix tree where one suffix is from s and one is from s'. These reductions are not completely trivial to implement in MPC, and we give a version of them formally in Appendix 6.11.2 and 6.11.3 after giving our own novel suffix tree construction using our Longest Common Prefix Query oracle.

Theorem 6.2.4. *For $\epsilon \in (0, 1]$, there are $O(\frac{1}{\epsilon})$-round* MPC *algorithms for* LCS *and* LPS *using $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

Now that we've shown a simple approach to some location-sensitive string problems and suffix tree, we next discuss our techniques involving the LCP problem which matches the state-of-the-art for suffix tree and improves the number of rounds and total space for LPS and LCS.

Additionally, these techniques will allow us to give a linear space algorithm for suffix tree construction in the AMPC model, which is not possible with the KMR approach.

6.3 Our Results and Techniques

We present a general framework for solving *location-sensitive* string problems in the MPC model. At the core of most of our algorithms, we rely on a novel data structure we call an LCPQ oracle which handles a large number of *Longest Common Prefix* (LCP) queries simultaneously in MPC. For example, when searching for the longest palindrome in a string it is necessary to do some sort of lexicographic comparisons to find the lengths of palindromes within the string. The LCPQ oracle allows us to efficiently evaluate such comparisons, and throughout this work, we will often reduce *location-sensitive* string problems to a polynomial number of LCP queries.

Problem	Model	Source	Rounds	Total Memory
LCPQ Oracle (with k queries)	MPC	Lemma 6.3.1 Theorem 6.3.2 Theorem 6.6.2	$O(1)$ $O(1)$ $O(\log\log n)$	$\widetilde{O}(n^{1+\epsilon}+k)$ $\widetilde{O}(n+k+\min(n,k)\cdot n^{\epsilon})$ $\widetilde{O}(n+k)$
LPS	MPC	Theorem 6.3.3	$O(1)$	$\widetilde{O}(n)$
LCS	MPC	Theorem 6.3.4	$O(1)$	$\widetilde{O}(n^{1+\epsilon})$
Suffix Array/Tree	MPC	Theorem 6.3.5,6.11.4	$O(1/\epsilon)$	$\tilde{O}(n^{1+\epsilon})$
Suffix Tree	AMPC	Theorem 6.3.6	$O(1)$	$\widetilde{O}(n)$

Table 6.1: The above table shows a summary of our results for LCPQ oracle, LPS, and LCS problems in MPC, as well as suffix tree constructions in MPC and AMPC using the LCPQ oracle. The local memory of each machine is $\widetilde{O}(n^{1-\epsilon})$ for a small constant ϵ.

Problem	Model	Source	Rounds	Notes
LPS	PRAM	[Apostolico et al., 1994]	$O(\log n)$	$O(n^2)$ memory/work
String Matching	MPC	[Hajiaghayi et al., 2021]	$O(1)$	$O(n)$ total memory
Suffix Array/Tree	PRAM	[Iliopoulos and Rytter, 2004]	$O(\log(n))$	$\tilde{O}(n)$ memory/work

Table 6.2: The existing results for LPS and suffix tree in the PRAM model, and string matching in MPC.

6.3.1 LCPQ Oracle

Given two strings s and s' of length n, an LCP query $q = (i, j)$ returns the length of the longest common prefix of the suffixes of s and s' which start at indices i and j respectively, i.e., the LCP of $s[i : n)$ and $s'[j : n)$. The main challenge of finding the longest common prefix between two suffixes is that a suffix may be very long, up to $O(n)$ if it includes most of the string. A single machine cannot compare two such suffixes of length $O(n)$ in its $O(n^{1-\epsilon})$-size local memory, and furthermore, a naive approach to computing the longest common prefix by comparing substrings of length $O(n^{1-\epsilon})$ sequentially could require $O(n^{\epsilon})$ rounds for long suffixes. By hashing every substring of size $n^{1-\epsilon}$, which we call *blocks*, and storing the hashes of subsequent blocks in a single machine, we can compare hashes of entire suffixes even of length $O(n)$ on a single machine. For example, for a suffix $s[i, n]$ we will store on a single machine the hashes of blocks $s[i, i + n^{\epsilon}]$, $s[i + n^{\epsilon}, i + 2n^{\epsilon}]$, etc. Notice that the starting index of each of the blocks considered in the previous example has a remainder of i modulo n^{ϵ}. Grouping blocks by their remainders after a modulo operation is a classic algorithmic technique, and we call this idea *Modular Partitioning*. Modular partitioning is the main way we can build and compare suffixes from their composite blocks efficiently on a single machine. Given query $q = (i, j)$, we then just have to make sure that the hashes of all blocks of suffixes $s[i : n)$ and $s'[j, n)$ are stored on the same machine together in

order to find a rough estimate of the longest common prefix between them.

Once we know the specific blocks where the prefixes no longer match, we can then easily compare these blocks of size $O(n^\epsilon)$ by sending a query to a machine that contains the (not hashed) substrings of each block using a single round of communication. Since each string can be broken into n^ϵ blocks of size $n^{1-\epsilon}$, in order to compare blocks directly we only need $O(n^{2\epsilon})$ machines which will each contain a unique pair of blocks from the input strings. This is not a novel technique, but we formalize this idea of *block-based data structures* in Section 6.4.1 since it is common to all of our MPC algorithms. The resulting data structure can answer a collection of $k = O(n^{1+\epsilon})$ arbitrary LCP queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ in $O(1)$ rounds.

Since our LCPQ oracle construction relies on hashing, our algorithms are necessarily randomized and succeed *with high probability*[3]. Additionally, our algorithms may need to try $O(\log n)$ different large prime numbers as the modulo base in our hash functions. To avoid $\log n$ communication rounds, we may try these prime numbers in parallel at the cost of an additional $O(\log n)$ factor in our space constraint. Furthermore, to perform modular partitioning, clearly local memory $O(n^{1-\epsilon})$ must be larger than $O(n^\epsilon)$ in order to find the hashes of blocks of length n^ϵ. Therefore, as discussed in the introduction, we bound ϵ by $0 < \epsilon \leq .5$ to make sure local memory $O(n^{1-\epsilon})$ is always large enough for our algorithms.

Lemma 6.3.1. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm, with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per machine, which initializes the* **Longest Common Prefix Query** (LCPQ) **Oracle** *in $O(1)$ rounds w.h.p., and then computes a collection of $k = O(n^{1+\epsilon})$ queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds.*

[3] w.h.p. (with high probability) should have an exponentially small probability of failure $\frac{1}{n^c}$ for some constant c we can choose.

With the previous techniques as a starting point, we give a more efficient LCPQ oracle, one of our main contributions, in Section 6.6. Specifically, we show that for any value of k which is either $O(n^{1-\epsilon})$ or $\Omega(n^{1+\epsilon})$, we can solve a batch of k queries in $O(1)$ rounds of MPC using tight total memory $O(n+k)$. For the values of $k = \omega(n^{1-\epsilon}) \cap o(n^{1+\epsilon})$, we present a compressed LCPQ oracle with $O(n + k + \min(n,k) \cdot n^{\epsilon})$ total memory. Intuitively, since the bottleneck is $\min(n,k) \cdot n^{\epsilon}$ for the values of k in this range, we end up having a sub-optimal total memory size if n and k are too close (within an $O(n^{\epsilon})$ multiplicative factor of each other); we introduce a different approach utilizing *modular partitioning* to show that solving the LCP queries with a total of $O(n+k)$ memory is achievable if we allow $O(\log \log n)$ rounds of communication in MPC (Lemma 6.6.2). In the uncompressed LCPQ oracle algorithm discussed before, recall that we hash blocks of size n^{ϵ} until we find the specific blocks where the prefix of two suffixes no longer match. We can simply repeat this process for blocks of size $n^{\epsilon/2}$, $n^{\epsilon/4}$, etc. to find smaller and smaller non-matching blocks until we find the exact longest common prefix length in at most $O(\log \log n)$ rounds. See Figure 6.2 or Table 6.3 to find more precise details about the performance of a *compressed* LCPQ oracle.

Theorem 6.3.2. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm with $\widetilde{O}(n^{1-\epsilon})$ memory per machine which initializes an* LCPQ **oracle** *in $O(1)$ rounds w.h.p., and then processes a collection of k queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds. The total memory used by this algorithm is $\widetilde{O}(n + k + \min(n,k) \cdot n^{\epsilon})$.*

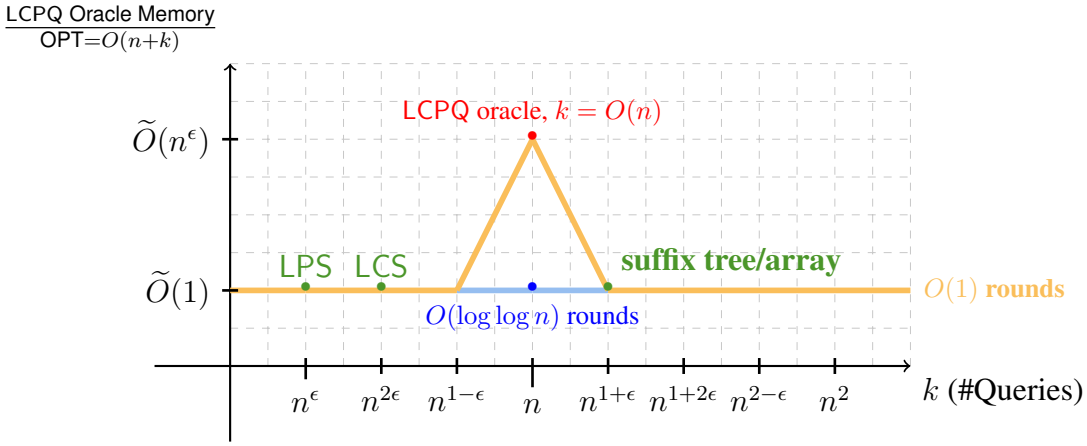


Figure 6.2: This plot illustrates the ratio of the memory blowup in a *compressed* LCPQ oracle for different ranges of k where k denotes the number of LCP queries in a batch. The vertical axis shows the ratio of memory blowup of the LCPQ oracle given query set $\mathcal{Q}$ of size k compared to the optimal memory $O(n + k)$. The horizontal axis shows the number of queries k. Both axes are logarithmic. The yellow path shows the memory performance of our best algorithm for a compressed LCPQ oracle in $O(1)$ rounds. The green points – corresponding to problems LPS (Section 6.7), LCS (Section 6.8), and suffix tree (Section 6.9.1) – show the problems in which we use memory-tight variants of the LCPQ oracle. The red point shows the problem of LCPQ oracle with $k = O(n)$ queries, in which we use sub-optimal memory for the LCPQ oracle, and the blue point demonstrate a memory-tight algorithm for the same problem if we allow $O(\log\log n)$ communication rounds in MPC.

6.3.2 LPS, LCS, and the Periodicity Properties

In Section 6.7, we present another main result, an algorithm for LPS using linear total memory and $O(1)$ communication rounds. A naive approach to solving LPS in the MPC model would be to distribute blocks of the input string to processors and have each processor find palindrome candidates. For any palindrome candidate, to find the actual full length of any palindrome would require comparing any two pairs of blocks since the palindrome may span the entire string, and to find the full length of many palindromes naively would require either super-linear memory or non-constant rounds. It might be tempting to suggest an MPC algorithm which queries the LCPQ oracle for all indices, and finds the longest palindrome centered at each location subsequently. However according to Figure 6.2, we end up with either a super-linear total memory or $O(\log \log n)$ rounds since we need to query every index and $k = O(n)$.

To avoid having to check each palindrome, we use an idea from [Apostolico et al., 1994] to analyze the periodicity of a string and limit the number of LCP queries we need to compute to be sublinear. Complemented by the novel compressed LCPQ oracle (see Section 6.6), our algorithm finds the maximum length palindrome per block using tight memory and constant rounds, as described in Theorem 6.3.3.

Theorem 6.3.3. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Palindrome Substring** (LPS) *with $\widetilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

Similarly, in Section 6.8 we present an algorithm for LCS that improves the communication rounds of prior techniques from $O(1/\epsilon)$ rounds to $O(1)$ rounds. The algorithm uses $\widetilde{O}(n^{1+\epsilon})$ total memory and is based on the periodicity properties of a pair of blocks with more than one LCS candidate pair. A *candidate pair* $p = (i, j)$ is a pair of suffixes i and j in two given strings s

and s' respectively, i.e. $s[i : n)$ and $s'[j : n)$, such that $\mathsf{LCP}(q) \geq 3n^{1-\epsilon}$ where $q = (i, j)$ is an LCP query for those pair of suffixes. As it is clear from the definition of the candidate pairs, they represent maximal common substrings with length at least $3n^{1-\epsilon}$. We use *pairwise block machines* to compute the longest common substring with length smaller than $3n^{1-\epsilon}$ by an exhaustive search. Therefore, it only remains to find the result of LCP queries for each candidate pair and take the maximum value.

The number of maximal candidate pairs excluding the pair of blocks with same period string is bounded by $O(n^{2\epsilon})$. However, if we include the candidate pairs for a common periodic range in both strings the number of LCP queries increases to $O(n)$. We can further reduce the total number of candidate pairs to $O(n^{2\epsilon})$ by finding the common periodic range imposed by a collection of more than one candidate pairs in a pair of blocks. The subsequent pair of indices in s and s' after the periodic range determine the candidate pair with maximum LCP in this pair of blocks. Thus, there are a collection of $O(n^{2\epsilon})$ candidate pairs, and the maximum LCP among these candidate pairs is equal to the LCS of two strings s and s'.

Theorem 6.3.4. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Common Substring** (LCS) *with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

6.3.3 Massively Parallel Suffix Tree using LCPQ

In our final section, we give our own LCPQ oracle-based algorithm for building the *suffix array* of a given string with $O(n^{1+\epsilon})$ total memory and $O(1/\epsilon)$ communication rounds. Using the suffix array to suffix tree conversion in Appendix 6.11.1, this allows us to build the suffix tree with our own LCPQ approach that ties the state-of-the-art KMR approach from Section 6.2. Moreover,

we show that this suffix array approach can be used in AMPC to improve suffix tree construction from $O(1/\epsilon)$ communication rounds and $\widetilde{O}(n^{1+\epsilon})$ total memory to $O(1)$ rounds and $\widetilde{O}(n)$ total memory.

For our LCPQ-based suffix array construction, we start by sorting the suffixes of a given input string by utilizing an LCPQ oracle which we initialize for string s and itself, to support a comparison-based parallel quick sort in total space $\tilde{O}(n^{1+\epsilon})$ similar to the one discussed in [Goodrich et al., 2011]. At each round, we choose n^{ϵ} random pivots for every remaining unsorted interval. In this process, the maximum interval size at round i is bounded by $\widetilde{O}(n^{1-\frac{i\epsilon}{2}})$ with high probability. Thus, after $O(1/\epsilon)$ rounds, we end up with $\widetilde{O}(n^{\frac{\epsilon}{2}})$-sized intervals which we sort locally using the LCPQ oracle. For comparing a pair of suffixes (i, j), we need to compare characters $s_{i'}$ and $s_{j'}$, the first unequal pair of characters we encounter if we start moving forward from indices i and j at the same rate. It is easy to observe that by definition $i' = i + \mathsf{LCP}_{s,s}(i, j)$ and $j' = j + \mathsf{LCP}_{s,s}(i, j)$, and so the LCPQ oracle provides all the information we need to find these indices.

Theorem 6.3.5. *For $\epsilon \in (0, .5]$, there is an $O(1/\epsilon)$-round* MPC *algorithm for computing the* **suffix array** *of a given string s with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor w.h.p..*

6.3.4 Suffix Tree in AMPC

Last but not least, we observe that in the *Adaptive Massively Parallel Computation* (AMPC) model, it is possible to build a fully-functional suffix tree[4] using $\widetilde{O}(n)$ total memory and in $O(1/(1-\epsilon))$ rounds. Note that LCP queries are handled by $O(\log n)$ sequential adaptive queries

[4]By a *fully-functional* suffix tree we mean one including backward and forward pointers as opposed to suffix tree we build in the MPC model which only captures the structure of the tree. A fully-functional suffix tree rarely relies on the suffix array and auxiliary block-based data structures for solving location-sensitive problems.

as we perform a binary search to find the length of an LCP query (i, j), i.e., the maximum l where $s[i : i + l)$ and $s'[j : j + l)$ have the same hash value. We retrieve the hash of a substring $s[l : r)$ using a *block-based* hashing data structure by spending only $O(1)$ adaptive queries. Since we use the $O(n)$ shared memory to store hashes, we avoid the requirement that $\epsilon \leq .5$, which stemmed from modular partitioning and the MPC versions of the block-based data structures.

Theorem 6.3.6. (Suffix Tree in AMPC) . *For $\epsilon \in (0, 1]$, there is an $O(1)$-round* AMPC *algorithm for computing the* **suffix tree** *of a given string s with $\tilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

We do not give the statements and their proofs, but utilizing the fully-functional suffix tree to solve LPS and LCS in $O(1)$ rounds and linear memory in AMPC easily follows using the reductions shown for MPC (Appendix 6.11.2 and 6.11.3).

6.4 MPC Implementation Details

In this section, we summarize a set of tools we frequently use in our MPC algorithms. First we introduce a technique called *Modular Partitioning* that is used to construct LCPQ oracles in Sections 6.5 and 6.6, which are an important building blocks of most of our algorithms. Next, we formalize two folklore MPC techniques called *Block-based Data Structures* and *Weighted Load Balancing*. We extensively use block-based data structures as auxiliary tools in the reduction of LPS and LCS to suffix tree in Appendix 6.11.2 and 6.11.3, as well as in the memory-optimal algorithms for LPS and LCS in Sections 6.7 and 6.8. Moreover, weighted load balancing is used in the construction of LCPQ oracles and suffix trees as well as in the reductions from LPS and LCS to suffix tree.

At the core, our algorithms use *hashing* and thus all of our results are randomized and succeed *with high probability*. There are additional sources of randomness in our algorithms such as the randomness induced by quick sort for finding the suffix array, but hashing is used in all algorithms described in this paper. We define the hash of a substring $s[l:r)$ as the following for a large enough prime number p.[5]

$$\mathsf{hash}\big(s[l:r)\big) = \Big(\sum_{i=l}^{r-1} s_i \cdot |\Sigma|^{r-1-i}\Big) \bmod p \tag{6.1}$$

6.4.1 Block-based Data Structures

The input string s is initially partitioned into n^ϵ consecutive *blocks* of size $O(n^{1-\epsilon})$, denoted by $b_0, b_1, \ldots, b_{n^\epsilon-1}$, so that each block fits into the memory of a single machine. We denote the machine which stores b_α by μ_α. Furthermore, we assume that μ_α has local access to the next block $b_{\alpha+1}$ and previous block $b_{\alpha-1}$ if either exists. Similarly, in problems that have two input strings s and s', we denote the blocks of s' by b'_β, and the dedicated machine that stores this block by μ'_β. In this paper, we use α and β variables when referring to the index of a block in s and s' respectively. We also always assume w.l.o.g. that both strings have length n. Otherwise, we pad the smaller one with enough number of '#' character, where '#' $\notin \Sigma$, to make their sizes equal.

Definition 6.4.1 (Blocks b_α and Block Machines μ_α)**.** There is a collection of n^ϵ machines $\{\mu_0, \mu_1, \ldots, \mu_{n^\epsilon-1}\}$ referred to as ***block machines*** for string s *(*and $\{\mu'_0, \mu'_1, \ldots, \mu'_{n^\epsilon-1}\}$ for string s'*)*, where μ_α *(*and μ'_β*)* contains the substring of the α-th ***block*** of s *(*and the β-th block of s'*)* denoted by $b_\alpha = s[\alpha n^{1-\epsilon} : (\alpha+1)n^{1-\epsilon})$.

[5] We can pick a set of $O(\log(n))$ random prime numbers, and process them in parallel, so that our algorithm succeeds *with high probability*.

We often use *block-based* data structures in our algorithms which is a standard technique in the MPC model. Roughly speaking, in a block-based data structure we gather a constant-sized information about every block b_α, say $f(b_\alpha)$[6], so that $\{f(b_0), f(b_1), \ldots, f(b_{n^\epsilon-1})\}$ can be copied and fit inside the memory of a single machine as long as $\epsilon \leq 0.5$. Furthermore, we use a *segment tree* inside each machine μ_α which computes $f(b_\alpha[l:r])$ for a total of $O(n^{1-\epsilon})$ substrings $b_\alpha[l:r]$ inside block b_α. Such data structures can answer a batch of $O(n)$ queries $f(s[l:r])$ about arbitrary substrings of s in 2 rounds of MPC. For example, we can define $f(s) = \mathsf{hash}(s)$, and thus query the hash of an arbitrary batch of $O(n)$ substrings efficiently in MPC. Function f needs to have the property that there exists a merge function g so that $f(s) = g\big(f(s[0:i]), f(s[i:n]), i, n\big)$ for every $0 < i < n$. For the example function hash as defined in (6.1), the corresponding function merge function g equals:

$$g\big(\mathsf{hash}(s[0:i]), \mathsf{hash}(s[i:n]), i, n\big) = \big(\mathsf{hash}(s[0:i]) \cdot |\Sigma|^{n-i} + \mathsf{hash}(s[i:n])\big) \bmod p$$

6.4.2 Modular Partitioning

The *Modular Partitioning* of an array of length n is to partition its elements based on their remainder when dividing by n^ϵ. Recall the definition of block machines μ_α from Definition 6.4.1. Within each block b_α, μ_α computes the hash of every substring of length n^ϵ starting in b_α, i.e., $\mathsf{hash}(s[i:i+n^\epsilon])$ for every $\alpha n^\epsilon \leq i < \min\{(\alpha+1)n^\epsilon, n\}$[7]. We allocate a machine λ_x for every $0 \leq x < n^\epsilon$ and gather into λ_x the hash values of every substring $s[i:i+n^\epsilon)$ such that $i \bmod n^\epsilon =$

[6]Function f can also be defined on any arbitrary array with length n instead of the string itself.
[7]Note that we have a local access to the next block, i.e., $b_{\alpha+1}$, inside μ_α.

x. In other words, we are assigning each index i to one of n^ϵ machines $\lambda_0, \lambda_1, \ldots, \lambda_{(n^\epsilon-1)}$ so that indices assigned to each machine have the same remainder in division by n^ϵ. This simple trick allows us to compute the hash of any substring whose length is a multiple of n^ϵ inside the local memory of some λ_x. A substring $s[i : i + un^\epsilon)$ is a concatenation of several substrings of length n^ϵ, specifically $s[i : i + n^\epsilon) + s[i + n^\epsilon : i + 2n^\epsilon) + \ldots + s[i + (u-1)n^\epsilon : i + un^\epsilon)$, all of which are assigned to the same machine $\lambda_{(i \bmod n^\epsilon)}$. For an example, see Figure 6.3 in which block machines μ_α and mod machines λ_x are specified for an example string s.

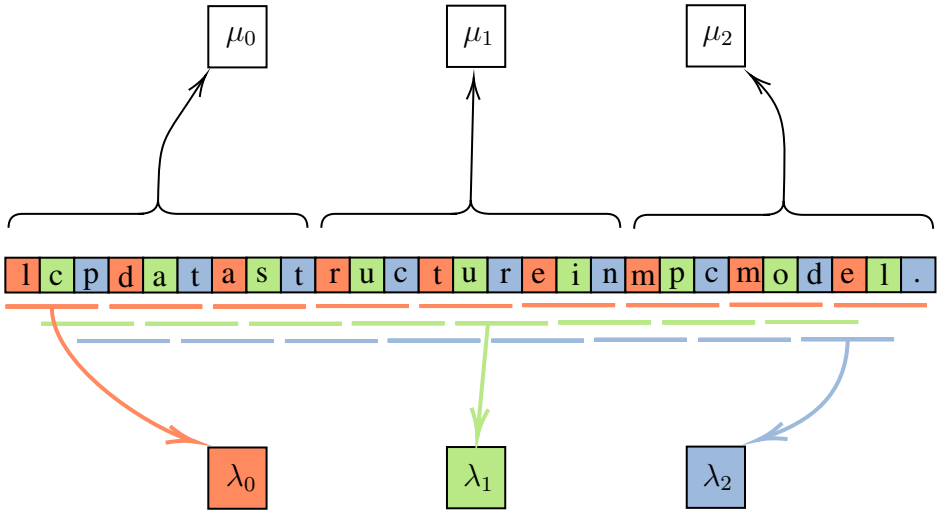


Figure 6.3: The modular partitioning is illustrated in this figure. In this example, $n = 27$ and $\epsilon = 1/3$, which means $n^\epsilon = 3$ and $n^{1-\epsilon} = 9$. There are 3 *block machines* μ_0, μ_1, and μ_2 which store the substrings of s corresponding to blocks b_0, b_1, and b_2 respectively. There are also 3 *mod machines* λ_0, λ_1, and λ_2. Each λ_x for $0 \leq x \leq 2$, stores the hash of every substring of length $n^\epsilon = 3$ which start at a location i so that $i \bmod 3 = x$.

Definition 6.4.2. There are a collection of n^ϵ machines $\{\lambda_0, \lambda_1, \ldots, \lambda_{n^\epsilon-1}\}$ for string s *(*and $\{\lambda'_0, \lambda'_1, \ldots, \lambda'_{n^\epsilon-1}\}$ for string s'*)*, where λ_x *(*and λ'_x*)* contains the hash of substrings of length $O(n^\epsilon)$ starting at indices i so that $i \bmod n^\epsilon = x$, and they are called ***mod machines***.

6.4.3 Weighted Load Balancing

Our machine and data structures are often required to answer large amounts of queries in parallel, i.e. $O(n)$ or $O(n^{1+\epsilon})$ queries. We first partition a query set into groups so that all queries that must be routed to the same machine to be answered are together. Unfortunately, more queries than the size of the local memory of a machine may be routed to a single machine. We use a 2-round algorithm *Weighted Load Balancing* to replicate the data stored in each machine proportionally to the number of queries assigned to that machine so that the local memory constraint is never violated.

We take our LCP query set $\mathcal{Q}$ as an example where each query $q \in \mathcal{Q}$ is a pair of indices (i, j) such that we want to find the LCP between the suffixes starting at index i in one string and j in another. We partition $\mathcal{Q}$ into $n^{2\epsilon}$ groups $Q_{x,y}$ based on the remainders of dividing both coordinates of each query by n^{ϵ}. Each group $Q_{x,y}$ contains all the queries $q = (i, j)$ such that $i \bmod n^{\epsilon} = x$ and $j \bmod n^{\epsilon} = y$. We may assign small groups, ones with size $O(n^{1-\epsilon})$, to one of $\lambda_{x,y}$ machines without violating the memory constraints. However, sometimes a large number of queries are concentrated in specific groups. The input queries might even consist of a single group of size $O(n^{1+\epsilon})$. To fix this issue, we replicate the data stored in each machine based on the number of queries assigned to it. See Algorithm 27 for more details. In Algorithm 27, the data with total size $O(n^{1+\epsilon})$ is partitioned into $n^{2\epsilon}$ groups $P_{x,y}$, and we aim to distribute the queries among a collection of $O(n^{2\epsilon})$ machines $\xi_{x,y}$ with possible repetitions[8].

[8]In the definition Algorithm 27, we label the servers and query groups with subscript indices in the range $[1, m]$. However, in our actual applications, servers may have any labelling such as a pair of subscripts, e.g. $\lambda_{x,y}$. The only requirement is that a query in P_i is expected to be sent to a copy of ξ_i for any ξ_i of the server set.

Algorithm 27: WeightedLoadBalancing($\mathcal{S} = \{\xi_1, \xi_2, ...\xi_m\}, \{P_1, P_2, ...P_m\}$)

for $\xi_x \in \mathcal{S}$ **do**
  $c_x \leftarrow \lceil \frac{|P_x|}{n^{1-\epsilon}} \rceil$;
  Replicate the data stored in machine ξ_x into c_x copies $\xi_x^{(1)}, \xi_x^{(2)}, \ldots, \xi_x^{(c_x)}$;
  Partition the elements of P_x into c_x groups $P_x^{(1)}, P_x^{(2)}, \ldots, P_x^{(c_x)}$ of almost equal size;
  Send $P_x^{(z)}$ to $\xi_x^{(z)}$ for every $1 \leq z \leq c_x$;
return $\{c_1, c_2, ..., c_m\}$;

6.5 Longest Common Prefix Query Oracle

A fundamental component of our later algorithms is a data structure for simultaneously processing a query batch consisting of $O(n^{1+\epsilon})$ arbitrary *Longest Common Prefix* (LCP) queries in $O(1)$ rounds of the MPC model. Recall the definition of LCP queries from Definition 6.1.1. An LCP query $q = (i, j)$ returns the length of the longest common prefix of the suffixes of two given strings s and s' starting at indices i and j respectively, denoted by $\mathsf{LCP}_{s,s'}(i, j)$. Along with the input strings s and s', we are also given a collection of LCP queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ where $k = O(n^{1+\epsilon})$.

Recall the definitions of *block machines* and *mod machines* from respectively Definitions 6.4.1 and 6.4.2. In order to use the data stored in a collection of mod machines $\lambda_0, \lambda_1, \ldots, \lambda_{n^\epsilon-1}, \lambda'_0, \lambda'_1, \ldots, \lambda'_{n^\epsilon-1}$ to estimate the length of the LCP queries, it is tempting to allocate a machine for each pair of remainders. In other words, if we allocate $O(n^{2\epsilon})$ machines $\lambda_{x,y}$ for every pair $0 \leq x, y < n^\epsilon$ so that it contains the data stored in λ_x as well as the data stored in λ'_y, we do not violate the memory constraints. However, we only need an $O(n^\epsilon)$-sized subset

$\Lambda \subseteq \{\lambda_{x,y}\}$ of mod machines. We can retrieve the required information from the following subset[9]:

$$\Lambda = \{\lambda_{0,0}, \lambda_{0,1}, \ldots, \lambda_{0,n^\epsilon-1}, \lambda_{1,0}, \ldots, \lambda_{n^\epsilon-1,0}\}.$$

The sketch of the algorithm is to only allocate $\lambda_{x,y}$ when there is a query $q = (i, j)$ where $i \bmod n^\epsilon = x$ and $j \bmod n^\epsilon = y$. In each λ_x we don't need to keep the hash of every substring with length n^ϵ and starting index $i \bmod n^\epsilon = x$. We can only keep the substrings with the length equal to the difference between two consecutive existing first coordinate of queries, i.e.,

$$D_{x,y} = \{d \mid \exists\ 1 \leq j \leq |Q_{x,y}| \quad \text{s.t.}\ \ d = i_{j+1} - i_j\} \tag{6.2}$$

where $i_1, i_2, \ldots, i_{|Q_{x,y}|}$ is the sorted sequence of the first coordinate of every (i, j) in $Q_{x,y}$. If we only keep the substrings with a length included in $D_{x,y}$, we have $O(n^{2\gamma})$ total memory, where the worst cost happens when all queries belong to $Q_{0,0}$ for example. We do another similar step for block machines $\mu_{\alpha,\beta}$ in the second stage of processing LCP queries.

The total memory used by all $\lambda_{x,y}$ machines is therefore bounded by $O(n^{1+\epsilon})$. For a given query $q = (i, j)$, so that $i \bmod n^\epsilon = x$ and $j \bmod n^\epsilon = y$, we use machine $\lambda_{x,y}$ to find the longest common prefix of $s[i : n]$ and $s'[j : n]$ whose length is a multiple of n^ϵ. Each machine $\lambda_{x,y}$ is able to find the hash of every substring with a length multiple of n^ϵ which start at indices with remainder x in s or with remainder y in s' when dividing by n^ϵ. Thus, we only need to redirect every query to a target machine $\lambda_{x,y}$ in order to finish the first phase. Similarly, we create $\mu_{\alpha,\beta}$ for every $0 \leq \alpha, \beta < n^\epsilon$ that contains the data stored in μ_α and μ'_β, see Figure 6.4 for an example.

[9] We will need this observation later in Section 6.6 to optimize the required memory for a *compressed* LCPQ oracle.

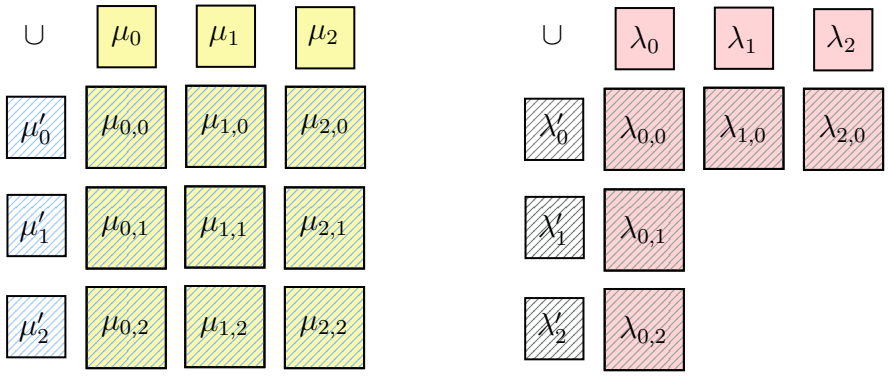


Figure 6.4: An illustration of *pairwise block machines* $\mu_{\alpha,\beta}$ and *pairwise mod machines* $\lambda_{x,y}$ for the same example. Pairwise block machines and pairwise mod machines have $O(n^{2\epsilon})$ and $O(n^{\epsilon})$ members respectively (9 and 5 members in this example), where in the former for every $0 \leq \alpha, \beta < n^{\epsilon}$, $\mu_{\alpha,\beta}$ stores the union of the data of μ_{α} and μ'_{β}, i.e., $\mu_{\alpha,\beta} = \mu_{\alpha} \cup \mu'_{\beta}$. Similarly, we have $\lambda_{x,y} = \lambda_x \cup \lambda'_y$ for every $0 \leq x, y < n^{\epsilon}$ such that $\min(x, y) = 0$.

Definition 6.5.1. There are two collections of $n^{2\epsilon}$ machines, $\{\mu_{\alpha,\beta} \mid 0 \leq \alpha, \beta < n^{\epsilon}\}$ and $\{\lambda'_{x,y} \mid 0 \leq x, y < n^{\epsilon}\}$, so that:

- $\mu_{\alpha,\beta}$ contains the substring of the α-th block of s and β-th block of s', respectively b_{α} and b'_{β}, and is called a ***pairwise block machine***.
- Similarly, $\lambda_{x,y}$ contains the hash of substrings of length $O(n^{\epsilon})$ starting at indices i in s so that $i \bmod n^{\epsilon} = x$, and indices j in s' so that $j \bmod n^{\epsilon} = y$, and is called a ***pairwise mod machine***.

We initialize a pairwise block-based data structure which fits into $O(n^{2\epsilon})$ machines each with $\widetilde{O}(n^{1-\epsilon})$ local memory by utilizing *Modular Partitioning* and *Weighted Load Balancing*. Next, we utilize the data stored in our data structure to find in parallel the length of every LCP query from $\mathcal{Q}$ using $O(1)$ rounds. Theorem 6.3.1 formally states our result about initializing and using the LCP data structure in the MPC model.

Lemma 6.3.1. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm, with $\widetilde{O}(n^{1+\epsilon})$ total memory*

and $\widetilde{O}(n^{1-\epsilon})$ *memory per machine, which initializes the* **Longest Common Prefix Query** (LCPQ) **Oracle** *in* $O(1)$ *rounds w.h.p., and then computes a collection of* $k = O(n^{1+\epsilon})$ *queries,* $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$*, in* $O(1)$ *rounds.*

The data structure processes the queries in two phases:

1 It first computes an estimated length of each query $q = (i, j)$. More precisely, it finds the longest common prefix starting at i and j whose length is a multiple of n^ϵ. In other words,

$$\max\{u \geq 0 \mid s[i : i + un^\epsilon] = s'[j : j + un^\epsilon]\}$$

2 Then, it finds the exact length of each query by gathering the queries that end in block b_α of s and block b'_β of s' into the same machine so that all such queries can be answered locally.

Algorithm 28: LCPInit(s, s')

for $0 \leq x < n^\epsilon$ **do**
  for $1 \leq u < n^{1-\epsilon}$ **do**
    Compute hash$(s[(u-1)n^\epsilon + x : un^\epsilon + x])$ in $\mu_{\lfloor u/n^{1-2\epsilon} \rfloor}$ and send it to λ_x;
    Compute hash$(s'[(u-1)n^\epsilon + x : un^\epsilon + x])$ in $\mu'_{\lfloor u/n^{1-2\epsilon} \rfloor}$ and send it to λ'_x;
  Replicate the data stored in mod machines λ_x and λ'_x into pairwise mod machines $\lambda_{x,y}$ and $\lambda_{y,x}$ respectively for all $0 \leq y < n^\epsilon$;
  Replicate the data stored in block machines μ_α and μ'_α into pairwise block machines $\mu_{\alpha,\beta}$ and $\mu_{\beta,\alpha}$ respectively for all $0 \leq \beta < n^\epsilon$;

Now that we have shown how to initialize the LCP data structure, which is formally summarized in Algorithm 28, we are ready to describe an $O(1)$-round MPC algorithm to answer a batch of queries $\mathcal{Q}$ of size $O(n^{1+\epsilon})$. In the rest of this section, we prove Theorem 6.3.1.

of Theorem 6.3.1. We first partition $\mathcal{Q}$ into $n^{2\epsilon}$ groups $Q_{x,y}$ based on the remainders of dividing both coordinate of each query by n^ϵ. Each group $Q_{x,y}$ contains all the queries $q = (i, j)$ such that

$i \bmod n^\epsilon = x$ and $j \bmod n^\epsilon = y$. We may assign small groups, ones with size $O(n^{1-\epsilon})$, to one of $\lambda_{x,y}$ machines without violating the memory constraints. However, sometimes a large number of queries are concentrated in specific groups. The input queries might even consist of a single group of size $O(n^{1+\epsilon})$. We utilize Algorithm 27 (WeightedLoadBalancing) to balance the number of replicas of each machine $\lambda_{x,y}$ based on the number of queries assigned to the same machine, i.e., $|Q_{x,y}|$.

We partition the queries and redirect them into the replicas of $\lambda_{x,y}$ created by Algorithm 27. Then, for a batch of queries $Q^{(z)}_{x,y}$ of size $O(n^{1-\epsilon})$, we use the local memory of a replica of $\lambda_{x,y}$ to find an estimate of the length of each query. See line (6) of Algorithm 29 for more details. We compute the longest common prefix of query $q = (i, j)$ whose length is a multiple of n^ϵ. In terms of a variable u^*, the longest common prefix for the query (i, j) found so far has length equal to $u^* n^\epsilon$. We also have $n^{2\epsilon}$ initially empty bags $F_{\alpha,\beta}$ for all $0 \leq \alpha, \beta < n^\epsilon$, one for each pair of blocks. We use the estimated length to find out the pair of blocks (b_α, b'_β) on which the LCP query $q = (i, j)$ ends, and we put it in the respective bag $F_{\alpha,\beta}$. We use Algorithm 27 again, to balance the number of replicas of each $\mu_{\alpha,\beta}$ based on the size of the bags $F_{\alpha,\beta}$, which represent the load, and to redirect the queries,.

Once every query is assigned to a replica $\mu^{(z)}_{\alpha,\beta}$, we start the second phase of processing queries. In this phase, our goal is to find the exact length of every query, and each query $q = (i, j)$ is already redirected to a machine $\mu^{(z)}_{\alpha,\beta}$ where b_α and b'_β are respectively blocks of s and s' on which the LCP of $q = (i, j)$ ends. Thus, the data stored in $\mu^{(z)}_{\alpha,\beta}$ is sufficient to find the exact LCP of $q = (i, j)$. Let $i' = i \bmod n^{1-\epsilon}$ and $j' = j \bmod n^{1-\epsilon}$ be indices of respectively b_α and b'_β with the same offset i and j have in their own blocks. Let d be the minimum of $\{i', j'\}$. We find the longest common prefix of $b_\alpha[i' - d : n^{1-\epsilon}]$ and $b'_\beta[j' - d : n^{1-\epsilon}]$, and we denote

by ℓ the length of this common prefix. We show that the length of query $q = (i, j)$ is equal to $\ell + \max\{\alpha n^{1-\epsilon} - i, \beta n^{1-\epsilon} - j\}$. See Algorithm 29 for more details. □

Algorithm 29: LCPBatchQueries$(s, s', \mathcal{Q})$

$F_{\alpha,\beta} \leftarrow \emptyset$ for all $0 \le \alpha, \beta < n^{\epsilon}$;
$Q_{x,y} \leftarrow \{q = (i, j) \in \mathcal{Q} \mid x = i \bmod n^{\epsilon} \wedge y = j \bmod n^{\epsilon}\} \quad \forall 0 \le x, y < n^{\epsilon}$;
$c_{x,y} \leftarrow$ WeightedLoadBalancing$(\{\lambda_{x,y} \mid 0 \le x, y < n^{\epsilon}\}, \{Q_{x,y} \mid 0 \le x, y < n^{\epsilon}\})$;
for $0 \le x, y < n^{\epsilon}, 1 \le z \le c_{x,y}$ **do**
    // in machine $\lambda^{(z)}_{x,y}$
    for $q = (i, j) \in Q^{(z)}_{x,y}$ **do**
        $u^* \leftarrow \max\{u \mid s[i : i + un^{\epsilon}] = s'[j : j + un^{\epsilon}]\}$;
        Add q to $F_{\lfloor \frac{i+u^*n^{\epsilon}}{n^{1-\epsilon}} \rfloor, \lfloor \frac{j+u^*n^{\epsilon}}{n^{1-\epsilon}} \rfloor}$;
$c'_{\alpha,\beta} \leftarrow$ WeightedLoadBalancing$(\{\mu_{\alpha,\beta} \mid 0 \le \alpha, \beta < n^{\epsilon}\}, \{F_{\alpha,\beta} \mid 0 \le \alpha, \beta < n^{\epsilon}\})$;
for $0 \le \alpha, \beta < n^{\epsilon}, 1 \le z \le c'_{\alpha,\beta}$ **do**
    // in machine $\mu^{(z)}_{\alpha,\beta}$
    for $q = (i, j) \in F^{(z)}_{\alpha,\beta}$ **do**
        $i' \leftarrow i \bmod n^{1-\epsilon}$;
        $j' \leftarrow j \bmod n^{1-\epsilon}$;
        $d \leftarrow \min\{i', j'\}$;
        $\ell \leftarrow \max\{\ell \mid b_{\alpha}[i' - d : i' - d + \ell] = b'_{\beta}[j' - d : j' - d + \ell]\}$;
        LCP$(i, j) \leftarrow \ell + \max\{\alpha n^{1-\epsilon} - i, \beta n^{1-\epsilon} - j\}$;

6.6 Compressed LCPQ Oracle

In this section, we study instances of the LCPQ oracle in which there are a smaller collection of queries $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, for example $k = O(n)$ as opposed to $O(n^{1+\epsilon})$ which we considered in the previous part. The ultimate question is whether it is possible to handle k queries for any value of k using $\widetilde{O}(n + k)$ total memory in a constant number of rounds since we need at least $\Omega(n + k)$ total memory to store the strings and queries. However, it is not possible to always get $\widetilde{O}(n+k)$ memory using our techniques. So the natural question is for which range of the values of k, we can can handle k queries in $O(1)$ rounds of MPC with $\widetilde{O}(n+k)$ total memory. In this section,

we study this problem thoroughly and optimize it for different ranges of k, which is summarized in Table 6.3 as well as in Figure 6.2.

In Lemma 6.6.1, we present an alternative approach for solving a batch of k LCP queries. The total memory used by this approach equals $\widetilde{O}(n + kn^{\epsilon})$ instead of $\widetilde{O}(n^{1+\epsilon} + k)$ total memory of Lemma 6.3.1. This implies a constant round MPC algorithm with total memory $\widetilde{O}(n + k + \min(n, k) \cdot n^{\epsilon})$ if we use Lemma 6.3.1 when $k = \Omega(n)$ and Lemma 6.6.1 when $k = O(n)$. Thus, the algorithm uses $\widetilde{O}(n + k + \min(n, k) \cdot n^{\epsilon})$ total memory since it needs to store the strings and queries.

Theorem 6.3.2. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm with $\widetilde{O}(n^{1-\epsilon})$ memory per machine which initializes an* LCPQ **oracle** *in $O(1)$ rounds w.h.p., and then processes a collection of k queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds. The total memory used by this algorithm is $\widetilde{O}(n + k + \min(n, k) \cdot n^{\epsilon})$.*

However, for the values of k in the range from $\omega(n^{1-\epsilon})$ to $o(n^{1+\epsilon})$, we still get a sub-optimal total memory of $O\big(n+k+\min(n, k)\cdot n^{\epsilon}\big) = \omega(n+k)$. In this case, we show an $O(\log \log n)$-round MPC algorithm with optimal total memory $O(n + k)$ which solves a batch of LCP queries of size $O(k)$ (See Lemma 6.6.2 for more details). Our memory-optimal algorithms for different ranges of k are demonstrated in Table 6.3. We start by showing the algorithm with $\widetilde{O}(n + k \cdot n^{1+\epsilon})$ memory and a constant number of rounds that handles a batch of k queries.

Lemma 6.6.1. *For $0 < \epsilon < 1/2$, there is an $O(1)$-round* MPC *algorithm, with $\widetilde{O}(n + k \cdot n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{\epsilon})$ memory per machine, which initiates the* LCPQ **Oracle** *in $O(1)$ rounds w.h.p., and then computes a collection of k queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds.*

Proof. Recall from our observation at the start of Section 6.5, we can assume the length of the

#Queries	Source	Rounds	Total Memory
$k = O(n^{1-\epsilon})$	Lemma 6.6.1	$O(1)$	$O(n + k \cdot n^{\epsilon}) = O(n)$
$k = \omega(n^{1-\epsilon})$, and $k = o(n^{1+\epsilon})$	Lemma 6.6.1 Theorem 6.6.2	$O(1)$ $O(\log \log n)$	$O(n + k + \min(n, k) \cdot n^{\epsilon})$ $O(n + k)$
$k = \Omega(n^{1+\epsilon})$	Lemma 6.3.1	$O(1)$	$O(n^{1+\epsilon} + k) = O(k)$

Table 6.3: The above table shows the different memory-optimal results we show in this section for different ranges of k, the number of LCP queries.

queries in $\mathcal{Q}$ is bounded by $O(n^{\epsilon})$. We need a total memory of $O(n + k \cdot n^{\epsilon})$ to store every substring of length $O(n^{\epsilon})$ starting at endpoints i and j (in s and s' respectively) for every query $q = (i, j) \in \mathcal{Q}$. Once we gathered the required information, we solve the LCP queries locally by comparing the characters of $s\big[i : \min(i + n^{\epsilon}, n)\big)$ and $s'\big[j : \min(j + n^{\epsilon}, n)\big)$ one by one. □

Theorem 6.6.2. *For $0 < \epsilon < 1/2$, there is an $O(\log \log n)$-round* MPC *algorithm, with $\widetilde{O}(n + k)$ total memory and $\widetilde{O}(n^{\epsilon})$ memory per machine, which uses the* LCPQ **Oracle** *to compute a collection of k queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, w.h.p.*

Proof. Again, we know from observation from Section 6.5 that the solution of every query $q = (i, j) \in \mathcal{Q}$ is bounded by $O(n^{\epsilon})$. Let $\mathcal{L}$ denote the guarantee on the maximum length of the current queries. We use an iterative approach that at each step reduces the current guarantee on the length of LCP queries from $O(\mathcal{L})$ to $O(\sqrt{\mathcal{L}})$. It is clear that after $O(\log \log n)$ iterations, the guarantee on the size of LCP queries collapses to $O(1)$. We use the ideas of Lemma 6.6.1 to solve the problem using $O(n + k)$ at that point to solve the problem locally.

Recall the *modular partitioning* technique from Section 6.4.2. To reduce the current guarantee from $\mathcal{L}$ to $O(\sqrt{\mathcal{L}})$ at each iteration, we perform a modular partitioning modulo $O(\sqrt{\mathcal{L}})$. Since the total number of characters on s and s' which are within substrings of length $O(\mathcal{L})$ starting from i or j for any query $q = (i, j)$ is bounded by $O(n)$, we need to query the hash of a total

of at most $O(n)$ substrings either to find out that $\mathsf{LCP}_{s,s'}(i,j) \leq \sqrt{\mathcal{L}}$ or to reduce the guarantee to $O(\sqrt{\mathcal{L}})$. We use a *block-based* data structure such that $f\big(s[l:r)\big) = \mathsf{hash}\big(s[l:r)\big)$ to handle the interval hash queries. Utilizing the computed modular partitioning, we estimate the length of each LCP with an additive error of $O(\sqrt{\mathcal{L}})$. This implies that we can reduce our guarantee on the maximum LCP among all queries in $\mathcal{Q}$ to $O(\sqrt{\mathcal{L}})$ by replacing each query $q = (i,j)$ by $q' = (i',j') = (i+\kappa, j+\kappa)$, where

$$\kappa = \left\lfloor \frac{\mathsf{LCP}_{s,s'}(i,j)}{\sqrt{\mathcal{L}}} \right\rfloor \cdot \sqrt{\mathcal{L}}.$$

□

6.7 Longest Palindrome in MPC

In this section, we give an improved solution to *Longest Palindrome Search* (LPS). Given a string $s = s_0 s_1 ... s_{n-1}$, let $\overline{s} = s_{n-1} ... s_1 s_0$ be the reverse of s. Note we also use $s[j:i)$ to represent the reverse of a substring $s[i+1:j+1)$ for $i < j$. A *palindrome* contained in a string s is a substring c of s such that $c = \overline{c}$. If c_0 is a palindrome in s such that for all other palindromes c_1 in s, $|c_0| > |c_1|$, then $\mathsf{LPS}(s) = c_0$. Our algorithm, Algorithm 33 below, will use $O(n^\epsilon)$ processors each with $\widetilde{O}(n^{1-\epsilon})$ memory for $\epsilon < .5$ and finish in $O(1)$ rounds. In LPS, finding short palindromes locally on an individual processor is very easy. However, like many of the other problems in the paper, finding long palindromes that have length larger than the memory of a processor becomes difficult since it requires coordination between processors to calculate the exact length. Moreover, it is possible that there are many such long palindromes, even as much as $\Omega(n)$ palindrome candidates. The existence of multiple palindromes that overlap in a string implies

that the section of the string containing these palindromes is periodic. For example, if there are two palindromes, a first palindrome to the left of a second palindrome, then the left side of the first palindrome must match its right side, which overlaps the left side of the second palindrome. Thus, each left side must match and therefore, this left side is exactly the repeated substring of this section of the string. We call this repeated substring the *period* of a substring if it spans the entire substring. By finding periods of each block of length $O(n^{1-\epsilon})$ of the original string s, we can keep track of palindrome candidates and find the largest palindrome per period without needing to check all $\Omega(n)$ candidates across the entire string, which would otherwise require either a non-constant number of rounds or super-linear memory.

Our algorithm for LPS, based on the properties of periodic substrings proven in [Apostolico et al., 1994], will be split into several steps. At the start of our main algorithm (Algorithm 33), we break the input string s into blocks b_α of size $n^{1-\epsilon}$ where $n = |s|, 0 \leq \alpha < n^\epsilon$. We then utilize the following algorithm, Algorithm 30, for finding palindrome candidates and a period string on each block. After finding each period w_α for block b_α, we find the exact range of the period and use the center of this range as our palindrome candidate for period w_α. Once we have our final set of palindrome candidates per period, we can use the LCPQ oracle to find the length of each palindrome by finding the longest common prefix of a palindrome center in string s with the corresponding center in $\overline{s}$ and return the longest palindrome.

The first algorithm we formally write, Algorithm 30, takes three adjacent blocks $b_{\alpha-1}, b_\alpha, b_{\alpha+1}$ as input and returns the period of block b_α if it is a periodic string as well as any potential palindrome candidates in case block b_α is not periodic. Algorithm 30 begins by storing the indices of palindrome centers from block b_α in an array C by checking if the block to the left of the center equals the block to the right. Without loss of generality we can assume all palindromes

have a single character as their center since we can simply add an additional character not found in the string between every two characters of the string to preserve all palindromes while converting any palindromes with a two character center to a one character center. Therefore, each index stored in C will be the center of a palindrome of length at least $2n^{1-\epsilon} - 1$ characters, i.e. the palindrome spans the entire block b_α. Then, we exploit the fact that any string will have periodic characters if there are at least two different palindromes that span the length of the string. If there are several palindrome centers within a block b_α with palindrome length $2^\epsilon - 1$, then to find the minimum period length we can take the *Greatest Common Divisor* (GCD) of the pairwise differences of the indices of these centers multiplied by 2 to account for the mirrored left and right side of each palindrome. We use this length to pull a period string w from the beginning of the block and return w alongside C.

Algorithm 30: LPSCandidate$(t = b_{\alpha-1}b^\alpha b^{\alpha+1})$

Initialize the list of LPS candidate indices to the empty set, $C \leftarrow \emptyset$;
`// Find palindrome center candidates in` $b\alpha$
for $n^\epsilon \leq j < 2n^\epsilon$ **do**
  if $t[j - n^\epsilon : j) = t[j + n^\epsilon : j)$ **then**
    $C \leftarrow C \cup \{j\}$;
if $|C| \geq 2$ **then**
  `// Calculate the minimum period length of` t
  For any $j, k \in C$ such that $j < k$, $\ell(j, k) := 2(k - j) + 1$;
  $\ell_0 \leftarrow gcd(\{l(j, k) - 1 | j, k \in C, j < k\}) + 1$;
else
  $l_0 \leftarrow 2n^\epsilon + 1$;
if $|C| \geq 1$ **then**
  `// Find the minimum period` w `of block` b_α
  Let the period string of b_α be $w := t[n^\epsilon, n^\epsilon + l_0]$;
else
  Find the maximum length palindrome centered at j in b_α using any sequential palindrome algorithm, store j in $C[0]$;
  $w \leftarrow -1$;
Return w, C;

In the following two lemmas, we show that Algorithm 30 correctly finds minimum length period strings in space $O(n^{1-\epsilon})$ given blocks of size $n^{1-\epsilon}$.

Lemma 6.7.1. *On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = |b_{\alpha+1}| = n^{1-\epsilon}$, Algorithm 30 will return w, C where C contains all indices of b_α that are centers of palindromes of length at least $2n^{1-\epsilon}+1$ and w is the minimum length period string if $|C| \geq 2$.*

Proof. On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = n^{1-\epsilon}$, any index j such that $n^\epsilon \leq j < 2n^\epsilon$ implies that $t[j : j+1) \in b_\alpha$. Thus, any palindrome candidate index added to C in step 5 of Algorithm 30 will be an index of b_α. Furthermore, index j is added to C if and only if $t[j-n^\epsilon : j) = t[j+n^\epsilon, j)$, i.e. j is a palindrome center with palindrome length $2n^\epsilon+1$.

From Lemma 3.3 of [Apostolico et al., 1994], $|C| \geq 2$ implies that block b_α is periodic, and for any $j, k \in C, j < k$, there is a period of length $l = 2(k-j)+1$. Thus, the minimum period length will be the GCD of all period lengths for any index pair of C, which we denote with l_0 in step 9 of Algorithm 30. Since each candidiate $j \in C$ is the center of a palindrome spanning the entire block b_α, the period string is repeated across the entire block and so we can take the first ℓ_0 characters as the period of the block. □

Lemma 6.7.2. *On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = |b_{\alpha+1}| = n^{1-\epsilon}$, Algorithm 30 uses $O(n^{1-\epsilon})$ memory.*

Proof. By Lemma 6.7.1, Algorithm 30 only considers candidates within block b_α of size $O(n^{1-\epsilon})$, so storing palindrome centers requires at most space $O(n^{1-\epsilon})$. Calculating the period and storing the period string of a single block of size $O(n^{1-\epsilon})$ similarly requires space $O(n^{1-\epsilon})$. □

After finding the period and palindrome centers of a block b_α, we then find the range of blocks surrounding b_α with matching period. We build a generic "period block" W, the concate-

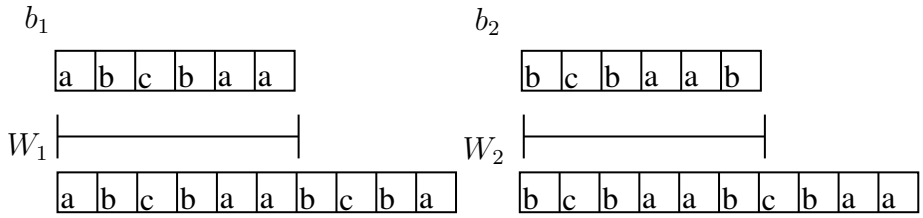


Figure 6.5: An example of two consecutive string blocks b_1 and b_2. Block b_1 has period $w_1 =$ "$abcba$", which is used to build W_1 as well as W_2, a rotation of $W1$, used to check that b_2 follows the same period.

nation of the period string with itself $\lceil \frac{n^{1-\epsilon}}{|w|} \rceil$ times. By carefully rotating W, we can find a range of block indices denoted $[\ell_1, r_1]$ such that the blocks $b_\alpha, j \in [\ell_1, r_1]$, including b_α, form a large periodic substring of the original string s. Figure 6.5 shows an example of a period block W_1 for a block b_1 and its subsequent rotation. After finding this block range, we then also find the exact indices in blocks b_{ℓ_1} and b_{r_1} where the period ends. The following subroutines, performed by each processor μ_α in parallel, lists the above steps, formalized.

Algorithm 31: LPSPeriodRange$(w_\alpha, H = [\mathsf{hash}(b_0), \mathsf{hash}(b_1), ..., \mathsf{hash}(b_{n^\epsilon - 1})])$

```
W_α ← ⌈n^{1−ε} / |w_α|⌉ · w_α;
ℓ, r ← −1;
for β = 0 to n^ε − 1 do
    if ℓ == −1 and α − β ≥ 0 then
        W_β ← rotate(W_α, −β · (|W_α| − n^{1−ε}));
        if H[α − β] ≠ W_β[0 : n^ε) then
            ℓ = α − β;
    if r == −1 and α + β < n^ε then
        W_β ← rotate(W_α, β · (|W_α| − n^{1−ε}));
        if H[α + β] ≠ W_β[0 : n^ε) then
            r = α + β;
    return ℓ, r;
```

Lemma 6.7.3. *Given the period string w_α of periodic block b_α and an array of block string hashes, Algorithm 31 finds the period range (ℓ, r) of consecutive blocks surrounding block b_α that fully contains a periodic substring with period w_α in space $\widetilde{O}(n^{1-\epsilon})$ w.h.p..*

Proof. The correctness of Algorithm 31 to find the period range (ℓ, r) relies on the claim that $W_\beta[0 : n^\epsilon)$ as constructed in steps 5 and 11 matches blocks $b_{\alpha-\beta}$ and $b_{\alpha+\beta}$, respectively, if these blocks are part of the period range with period w_α. We will prove this claim by induction on index j. For $j = 0$, this is trivial since we are given that block b_α is already periodic with period w_α.

Now let β be an index such that the claim holds true; we will prove that the claim holds for $\beta + 1$, first for the W_β constructed in step 11. Since the claim holds for β, we know that $W_\beta[0 : n^\epsilon) = b_{\alpha+\beta}$. Furthermore, W_β is just a rotation of W_α, a periodic string with period w_α as constructed in step 1. Therefore, string W_β contains a fragment of a period that must be the start of the next block $b_{\alpha+\beta+1}$ if the following block is in the period range. Thus W_β rotated to the right by $|W_\beta| - n^\epsilon$ characters will correctly match the remaining fragment to the next block as well as make sure the rest of the block is periodic since W_β is a rotation of period block string W_α. Therefore,

$$\begin{aligned} W_{\beta+1} &= rotate(W_\beta, (|W_\beta| - n^{1-\epsilon})) \\ &= rotate(W_\beta, (|W_\alpha| - n^{1-\epsilon})) \\ &= rotate(W_\alpha, \beta \cdot (|W_\alpha| - n^{1-\epsilon}) + |W_\alpha| - n^{1-\epsilon})) \\ &= rotate(W_\alpha, (\beta + 1) \cdot (|W_\alpha| - n^{1-\epsilon})) \end{aligned}$$

which is exactly the string we set $W_{\beta+1}$ to be in step 11. The same arguments hold for W_β constructed in step 5 but with left rotations since we are checking the reverse direction from β to $\beta - 1$. Thus, the claim holds.

From this claim, we can see that by checking that each block b_β surrounding b_α matches W_β, we will find some index β_1 such that block $b_{\alpha-\beta_1}$ does not match W_{β_1} in step 6 and some index β_2 such that block $b_{\alpha+\beta_2}$ that does not match W_{β_2} in step 12, which are the nearest left and right

blocks that fall outside of the period range. We then set our range indices accordingly in steps 7 and 12 so that $\ell = \alpha - \beta_1$ and $r = \alpha + \beta_2$.

Finally, Algorithm 31 requires $O(n^{1-\epsilon})$ space to store W_α and W_β. Furthermore, Algorithm 31 must also store its input array H, which contains the hash of each block. Since we have that $\epsilon < .5$ in our parallel setting, $|H| = n^\epsilon < n^{1-\epsilon}$, and so Algorithm 31 only requires $n^{1-\epsilon}$ space. □

Algorithm 32: LPSExactPeriodRange(α, w, ℓ, r)

1 $a_r \leftarrow |w| - (r-\alpha)(\lceil \frac{n^{1-\epsilon}}{|w_\alpha|} \rceil \cdot w_\alpha - n^{1-\epsilon}) \mod (|w|)$;
2 $a_\ell \leftarrow |w| - (\ell-\alpha+1)(\lceil \frac{n^{1-\epsilon}}{|w_i|} \rceil \cdot w_\alpha - n^{1-\epsilon}) \mod (|w|)$;
3 $h_w \leftarrow \mathsf{hash}(w)$;
4 Send $a_r, h_w, |w|$ to μ_r;
5 Send $a_\ell, h_w, |w|$ to μ_ℓ;
6 **for** $j = 0$ *to* $\lfloor \frac{n^{1-\epsilon}}{|w|} \rfloor$ *on processor* μ_r **do**
7 $t \leftarrow b_{r-1} b_r b_{r+1}$;
8 $k \leftarrow j \cdot |w| + (n^{1-\epsilon} - a_r)$;
9 **if** $h_w \neq \mathsf{hash}(t[k : k + |w|))$ **then**
10 $o_r \leftarrow k$;
11 **for** $j = 0$ *to* $\lfloor \frac{n^{1-\epsilon}}{|w|} \rfloor$ *on processor* μ_ℓ **do**
12 $t \leftarrow b_{\ell-1} b_\ell b_{\ell+1}$;
13 $k \leftarrow (2n^{1-\epsilon} + a_\ell) - j \cdot |w|$;
14 **if** $h_w \neq \mathsf{hash}(t[k - |w| : k))$ **then**
15 $o_\ell \leftarrow k - w$;
16 Return o_ℓ, o_r;

Lemma 6.7.4. *Given period w of block b_α and period range ℓ, r on processor μ_α, Algorithm 32 returns indices o_ℓ, o_r which are the indices of the leftmost and rightmost characters, respectively, denoting the last consecutive full period w in b_ℓ and b_r in space $\widetilde{O}(n^{1-\epsilon})$ and $O(1)$ rounds w.h.p.*

Proof. The main idea of Algorithm 32 is to iteratively check that the hash of the period w matches the hash of period-length chunks of the blocks b_ℓ and b_r. This idea itself seen in steps 6-12 and 13-19 is quite simple; the main difficulty is aligning the period with the blocks. Since the block

lengths are $n^{1-\epsilon}$, it is not guaranteed that the start of the period will align with the start of a block. Thus, in steps 1 and 2 we calculate the number of characters before the start of each block until the beginning of a full period.

When calculating value a_r in step 1, we follow a similar argument as in the proof of Lemma 6.7.3. The misalignment of block b_α and period block W_α built in Algorithm 31 consisting of repeated period w is initially just $\lceil \frac{n^{1-\epsilon}}{|w|} \rceil - n^{1-\epsilon}$. Therefore, each additional block extends the misalignment by another factor of $\lceil \frac{n^{1-\epsilon}}{|w|} \rceil - n^{1-\epsilon}$. Since the period range is the range of the period w repeated consecutively, misalignment can only be at most the size of the period before another starts; thus, we mod our misalignment by $|w|$. To calculate the length of the period fragment at the beginning of block b_r, we therefore have $(r - \alpha)(\lceil \frac{n^{1-\epsilon}}{|w|} \rceil - n^{1-\epsilon})$. Lastly, since we want to make sure we check for full periods and the beginning of block b_r is a period fragment, we subtract the length calculated up until now from $|w|$ to find out how many characters before b_r we must check to find a full period, and store this number in a_r. Step 8 on processor μ_r finds where a full period should start just before block b_r using a_r and steps 9 and 10 find the leftmost index that does not begin a full period $|w|$ and store it in o_r. By a symmetric argument, step 2 finds the number of characters after block b_ℓ until a full period and stores this in a_ℓ. Steps 15-17 then use a_ℓ to find the rightmost index in b_ℓ that does not begin a full period $|w|$ and stores this index in o_ℓ.

The number of rounds needed for the algorithm is only 2 since process μ_α must send and receive once each to processors μ_ℓ and μ_r to send integers $a_r, a_\ell, h_w, |w|$ and receive o_ℓ, o_r. Additionally, there is only $O(n^{1-\epsilon})$ space needed per processor since the size of the string blocks t will be $O(n^{1-\epsilon})$ and all auxiliary variables created to find o_ℓ and o_r trivially only require a constant amount of space. □

To wrap up the section, we present the full LPS algorithm for a string s on n^ϵ processors in the MPC model with $O(n)$ total memory and $O(1)$ rounds. First, each processor μ_α finds palindrome candidates and a period string w_α on block b_α using Algorithm 30. Each block then finds its surrounding period range using Algorithms 31 and 32. The center of the period range will be the longest palindrome center for all blocks in the period range and so we use the LCPQ oracle to check the length of the palindrome by finding how many characters to the left of the palindrome center match the characters to the right. After finding the longest palindrome on each block, we use a separate processor to compare each block's longest palindrome to find the maximum length palindrome across the entire string

Algorithm 33: LongestPalindromeSearch($s = \{s_0 s_\alpha ... s_{n-1}\}$)

Partition s into n^ϵ blocks of length $n^{1-\epsilon}$ where b_α denotes the αth block;
Distribute blocks $b_{\alpha-1}, b_\alpha, b_{\alpha+1}$ to processor μ_α for $0 \leq \alpha < n^\epsilon$;
Distribute $H = (\mathsf{hash}(b_1), \mathsf{hash}(b_2), ..., \mathsf{hash}(b_{n^\epsilon - 1}))$ to all processors;
$D \leftarrow \mathsf{LCPinit}(s, \overline{s})$;
for *processors* $\mu_\alpha, 0 \leq \alpha < n^\epsilon$ *in parallel* **do**
    $w_\alpha, C_\alpha \leftarrow \mathsf{LPSCandidate}(b^{\alpha-1} b^\alpha b^{\alpha+1})$;
    $\ell_\alpha, r_\alpha \leftarrow \mathsf{LPSPeriodRange}(w_\alpha, H)$;
    $o_{\ell_\alpha}, o_{r_\alpha} \leftarrow \mathsf{LPSExactPeriodRange}(\alpha, w_\alpha, \ell_\alpha, r_\alpha)$;
    $c_\alpha \leftarrow \frac{1}{2}\left((r_\alpha \cdot n^{1-\epsilon} + o_r) - ((\ell_\alpha + 1) \cdot n^{1-\epsilon} - o_{\ell_\alpha})\right) + ((\ell_\alpha + 1) \cdot n^{1-\epsilon} - o_{\ell_\alpha})$;
    if $s_{c_\alpha} \in b_\alpha$ **then**
        $d_\alpha \leftarrow 2 \cdot D.\mathsf{query}(s_{c_\alpha}, n - s_{c_\alpha}) + 1$;
$d_{max} = \max_{\alpha, 1 \leq \alpha \leq n^\epsilon}\{d_\alpha\}$;
Return (c_α, d_α) such that $d_\alpha = d_{max}$;

Theorem 6.3.3. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Palindrome Substring** (LPS) *with $\widetilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

Proof. For the correctness of Algorithm 33 we parse the algorithm one step at a time. First, each process μ_α receives its blocks $b_{\alpha-1}, b_\alpha, b_{\alpha+1}$ of length $n^{1-\epsilon}$ and an array of hashes of each block. Second, the LCPQ oracle is setup on s and its reverse string $\overline{s}$. In parallel, by Lemma 6.7.1, step 6

finds the largest palindrome candidate centers in block b_α and the period w_α of this block. In step 7, by Lemma 6.7.3, the range of block indices (ℓ_α, r_α) is found such that each block in this range matches a periodic string with period w_i. In step 8, by Lemma 6.7.4, the exact indices $o_{\ell_\alpha}, o_{r_\alpha}$ that denote the exact characters in b_{ℓ_α} and b_{r_α} where the period ends, specifically no more full periods continue to the left of o_{ℓ_α} nor to the right of o_{r_α}. At this point, we know that any maximum length palindrome must be at the center of this period range and spans at least the entire period range length. If there was another maximum length palindrome within this period range that was not the center, then it could not be longer than the period range since we know that one side would have a longer periodic section than the other. Therefore, we can consider only the very center of the period range for a maximum length palindrome. This center is therefore exactly halfway between the $(n^{1-\epsilon} - o_{\ell_\alpha})$-th character in block b_{ℓ_α} and the o_{r_α}-th character in block b_{r_α}. Since each block has length $n^{1-\epsilon}$, the first character of block b_{ℓ_α} has index $\ell_\alpha \cdot n^{1-\epsilon}$ in s and the first character of block b_{r_α} is at index $r_\alpha \cdot n^{1-\epsilon}$. Thus, the length of the period range will be

$$(r_\alpha \cdot n^{1-\epsilon} + o_r) - ((\ell_\alpha + 1) \cdot n^{1-\epsilon} - o_{\ell_\alpha})$$

and in step 9, we add half this length to the leftmost index of the period range, $((\ell_\alpha+1)\cdot n^{1-\epsilon}-o_{\ell_\alpha})$, to find the center of the period range c_α. In step 11, we use the LCPQ oracle to find the length of the palindrome by finding the length of the longest common prefix of the palindrome center in s with the palindrome center in $\overline{s}$, which is equivalent to finding the length of characters to the right of c_α in s that match the characters to the left of c_α, which is equivalent to the characters to the right of $n - c_\alpha$.

The number of rounds needed in steps 1-4 to distribute the blocks, hashes, and to set up the

LCPQ oracle is just $O(1)$ since each processor only receives $O(n^{1-\epsilon})$ information, which is the size of each processor's memory. Then, in steps 6 and 7, LPS_Candidate are performed locally on a single processor. By Lemma 6.7.4, Step 8 only requires $O(1)$ rounds and can be performed in parallel by each processor since each processor only sends a constant amount of information to two other processors. Step 11 requires $O(1)$ rounds as discussed in Section 6.5 and Section 6.6. Finally, finding the max palindrome across the entire string in step 14 can be done in 1 round since there is at most n^{ϵ} candidate indices, one per block, and there is $O(n^{1-\epsilon})$ memory per processor which is larger than n^{ϵ} since $\epsilon < .5$.

The memory per processor is trivially $O(n^{1-\epsilon})$ since each processor only gets 3 blocks of size $O(n^{1-\epsilon})$ and by Lemma's 6.7.2, 6.7.3, and 6.7.4. Since we use $O(n^{\epsilon})$ processors to store each block of the string, build the LCPQ oracle (see Corollary 6.3.2), and find the maximum length palindrome across the entire string, we therefore use $O(n^{\epsilon} \cdot n^{1-\epsilon}) = O(n)$ memory total.

□

6.8 Longest Common Substring in MPC

In this section, we show a constant-round MPC algorithm for finding the *Longest Common Substring* (LCS) of two strings s and s'. Without loss of generality, we assume both strings have length n. A substring $s[i : i+l)$ of string s is a consecutive interval of characters $s_i, s_{i+1}, \ldots, s_{i+l-1}$. Recall the definition of LCP queries from Section 6.5. $\mathsf{LCP}_{s,s'}(i, j)$ is the longest common prefix of two suffixes $s[i : n)$ and $s'[j : n)$. Thus,

$$\mathsf{LCS}_{s,s'} = \max_{i,j \in [n]} \mathsf{LCP}_{s,s'}(i,j) \tag{6.3}$$

Assume both strings are partitioned into n^ϵ consecutive blocks of size $O(n^{1-\epsilon})$, denoted by $b_1, b_2, \ldots, b_{n^\epsilon}$ for s and $b'_1, b'_2, \ldots, b'_{n^\epsilon}$ for s', so that each block fits into the memory of a single machine. We denote the machine which stored b_α and b'_β by μ_α and $\bar{\mu}_\beta$ respectively. Our high-level approach is to find the longest common substring whose starting points fall in a fixed pair of blocks, say b_α and b'_β. Let us denote this value by $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta)$, i.e, $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta) = \max_{i \in b_\alpha, j \in b'_\beta} \mathsf{LCP}_{s,s'}(i,j)$. Then, the final answer can be computed as the maximum of this partial solutions for each pair of blocks, i.e.,

$$\mathsf{LCS}_{s,s'} = \max_{\alpha,\beta \in [n^\epsilon]} \mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta) \tag{6.4}$$

Thus, our focus in this section is to efficiently compute $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta)$. We may omit subscripts in LCS and LCP functions whenever s and s' are implied from the context. We also need that this process requires only $O(1)$ machines, so that we can compute $O(n^{2\epsilon})$ different values in parallel and still have $O(n^{1+\epsilon})$ total memory.

Definition 6.8.1 (Candidate Pair)**.** For a pair of blocks b_α and b'_β, we call a pair of indices $p = (i,j)$, where $i \in b_\alpha$ and $j \in b'_\beta$, a *"candidate pair"* if $\mathsf{LCP}(i,j) \geq 3n^{1-\epsilon}$. The *"offset"* of a candidate pair $p = (i,j)$, denoted by $\delta(p)$, is defined as $i - j$.

We can find a set of candidate pairs by transferring $O(1)$ blocks into a single machine and

performing an exhaustive search. Let $P_{\alpha,\beta} = \{p_1, p_2, \ldots, p_k\}$ be the set of candidate pairs belonging to the pair of blocks b_α and b'_β. If there are multiple candidate pairs with the same offset, we only keep the one with the smallest i. We show that when there are multiple candidate pairs, $k > 1$, the subsequent blocks $b_{\alpha+1}$ and $b'_{\beta+1}$ are periodic. We will later use this periodicity property in our algorithm. Roughly speaking, assume we have two candidate pairs p_1 and p_2. This means that for every $i \in b_{\alpha+1}$,

$$s_i = s'_{i-\delta(p_1)} = s_{i-\delta(p_1)+\delta(p_2)}$$

This intuitively means that $b_{\alpha+1}$ is periodic with period length $\delta(p_1) - \delta(p_2)$. However, since $|\delta(p_1) - \delta(p_2)|$ might be as large as $2n^{1-\epsilon}$, we prove the periodicity properties for the concatenation of the next two blocks $b_{\alpha+1}b_{\alpha+2}$. In the following lemma, we formally show that the period lengths of both $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ are equal to $\gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}$.

Lemma 6.8.2. *The period length of both $b_{\alpha+1}b'_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ is equal to $\gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}$.*

Proof. Recall that $P_{\alpha,\beta}$ is the set of candidate pairs $(i, j) \in b_\alpha \times b'_\beta$ so that $\mathsf{LCP}(i, j) \geq 3n^{1-\epsilon}$. This implies that for any $p^\circ = (i^\circ, j^\circ) \in b_{\alpha+1}b_{\alpha+2} \times b'_{\beta+1}b'_{\beta+2}$, we have

$$s_{i^\circ} = s'_{j^\circ} \quad \text{if } \exists\, p = (i, j) \in P_{\alpha,\beta} \quad \text{s.t. } \delta(p) = \delta(p^\circ) \tag{6.5}$$

We can further expand the domain of variable p° in equation 6.5 by requiring either $i^\circ \in b_{\alpha+1}b_{\alpha+2}$ or $j^\circ \in b'_{\beta+1}b'_{\beta+2}$. As long as there is a candidate pair p with the same offset as p°, then we have either

$$\begin{cases} i^\circ \in b_{\alpha+1}b_{\alpha+2}\colon & i \leq (\alpha+1)n^{1-\epsilon} \leq i^\circ \Rightarrow j \leq j^\circ \\ j^\circ \in b'_{\beta+1}b'_{\beta+2}\colon & j \leq (\beta+1)n^{1-\epsilon} \leq j^\circ \Rightarrow i \leq i^\circ \end{cases}$$

Thus, if at least one of the indices in a pair p° is within the two subsequent blocks, i.e., $i^\circ \in b_{\alpha+1}b_{\alpha+2}$, and there is a candidate pair in $P_{\alpha,\beta}$ with the same offset, then $s_{i^\circ} = s'_{j^\circ}$. We exploit this fact to demonstrate that the length of the period string $w_{\alpha,\beta}$ of subsequent blocks, $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$, divides the offset difference of every pair of candidate pairs. More precisely, we show that the subsequent blocks are periodic with length $\delta(p_1) - \delta(p_2)$, which means that this length is a multiple of the minimal period length, i.e., $|w_{\alpha,\beta}|$.

$$(\delta(p_1) - \delta(p_2)) \bmod |w_{\alpha,\beta}| = 0 \quad \forall\, p_1, p_2 \in P_{\alpha,\beta} \tag{6.6}$$

To show that b_α and b'_β are periodic with length $\delta(p_1) - \delta(p_2)$, we use the same idea we intuitively mentioned before. Note that the period range $\delta(p_1) - \delta(p_2)$ is bounded by $2n^{1-\epsilon}$. Consider two indices $i, j \in b_{\alpha+1}b_{\alpha+2}$, so that $i - j = \delta(p_1) - \delta(p_2)$. Since i is within the two subsequent blocks, then $s_i = s'_{i-\delta(p_1)}$. In addition, since j is also within the subsequent block, then $s_j = s'_{j-\delta(p_2)}$. However,

$$i - \delta(p_1) = (i - j) - (\delta(p_1) - j) = \delta(p_1) - \delta(p_2) - \delta(p_1) + j = j - \delta(p_2)$$

Hence, $s_i = s_j$ since at least one index in each pair is within the periodic region. Now that we have shown that the period length of b_α divides the offset difference of every $p_1, p_2 \in P_{\alpha,\beta}$,

it is straight-forward to observe that the period length of b_α is equal to the GCD of all the offset differences in every pair of candidate pairs, i.e., the right hand side of Equation 6.7. We can prove the same thing for b'_β similarly.

$$|w_{\alpha,\beta}| = \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\} \tag{6.7}$$

□

As mentioned before, we aim to compute $\mathsf{LCS}(b_\alpha, b'_\beta)$ for every pair of blocks b_α and b'_β in parallel. We reduced this problem to a collection of $O(n^2)$ LCP queries according to Equation (6.3). Even by excluding every query which is not a candidate pair, we might have to make $\Omega(n^2)$ LCP queries, while an LCPQ oracle from Section 6.5 is capable of handling at most $O(n^{1+\epsilon})$ LCP queries in $O(1)$ rounds without blowing up the memory. To overcome this issue, we use the periodicity properties of the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ to limit the number of LCP queries we need to make for each pair of blocks to $O(1)$. At the beginning, we build three instances of the LCPQ oracle $\mathcal{D}_{s,s}$, $\mathcal{D}_{s,s'}$, and $\mathcal{D}_{s',s'}$. The algorithm consists of three stages:

1 Find the set of candidate pairs, $P_{\alpha,\beta}$, by an exhaustive search on the local data of $\mu_{\alpha,\beta}$. If $|P_{\alpha,\beta}| = O(1)$, we make a separate LCP query to $\mathcal{D}_{s,s'}$ for each candidate pair and we are done. Otherwise, the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ are periodic. Let $w_{\alpha,\beta}$ be the period string where $|w_{\alpha,\beta}| = \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}$ according to Lemma 6.8.2.

2 Next, we find the boundaries of the periodic pattern. The periodic pattern might repeat after the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ and span several blocks. Let $i^\circ \in \alpha^\circ$ and $j^\circ \in \beta^\circ$ be the first characters in s and s' which are inconsistent with the periodic pattern.

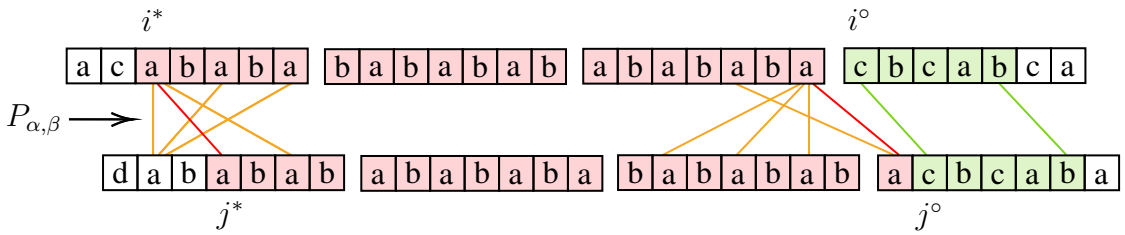


Figure 6.6: Candidate pairs, first inconsistent character, and the extension beyond periodic part

We call i° and j° the *first inconsistent characters*. To find the exact location of i° and j°, we utilize the LCPQ oracle we build for the strings and themselves, $\mathcal{D}_{s,s}$ and $\mathcal{D}_{s',s'}$. The first inconsistent characters in s and s' are respectively located at:

$$i^\circ = \alpha n^{1-\epsilon} + |w_{\alpha,\beta}| + \mathcal{D}_{s,s}.\mathsf{query}(\alpha n^{1-\epsilon}, \alpha n^{1-\epsilon} + |w_{\alpha,\beta}|)$$

$$j^\circ = \beta n^{1-\epsilon} + |w_{\alpha,\beta}| + \mathcal{D}_{s',s'}.\mathsf{query}(\beta n^{1-\epsilon}, \beta n^{1-\epsilon} + |w_{\alpha,\beta}|)$$

3 If there exists a candidate pair $p^* = (i^*, j^*) \in P_{\alpha,\beta}$ so that $\delta(p^*) = i^\circ - j^\circ$, then $\mathsf{LCP}(i^*, j^*)$ may extend beyond the boundaries of the periodic pattern. For this specific candidate pair, we make an LCP query to $\mathcal{D}_{s,s'}$. For all other candidate pairs in $P_{\alpha,\beta}$, we claim that knowing the location of i° and j° is sufficient for computing $\mathsf{LCP}(i, j)$.

Algorithm 34 demonstrates the algorithm in more details. In what follows, we prove that the described algorithm correctly finds the *Longest Common Substring* of two strings s and s', $\mathsf{LCS}_{s,s'}$, and the MPC memory constraints are not violated during $O(1)$ rounds.

Theorem 6.3.4. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Common Substring** (LCS) *with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

Proof. The algorithm starts by building the set of candidate pairs using an exhaustive search on each $\mu_{\alpha,\beta}$ in parallel. There are a collection of $O(n^{2\epsilon})$ machines $\{\mu_{\alpha,\beta} \mid 0 \leq \alpha, \beta < n^\epsilon\}$ each

Algorithm 34: $\mathsf{LCS}(s = \{s_0 s_1 ... s_{n-1}\}, s' = \{s'_0 s'_1 ... s'_{n-1}\},)$

1 $\mathcal{D}_{s,s}, \mathcal{D}_{s,s'}, \mathcal{D}_{s',s'} \leftarrow \mathsf{LCPInit}(s,s), \mathsf{LCPInit}(s,s'), \mathsf{LCPInit}(s',s')$;
2 **for** *all processors* $\mu_{\alpha,\beta}$, $0 \le \alpha, \beta < n^{\epsilon}$, *in parallel* **do**
3 $\quad P_{\alpha,\beta} \leftarrow \Big\{(i,j) \in b_\alpha \times b'_\beta \mid \begin{cases} s[i : i+3n^{1-\epsilon}) = s'[j : j+3n^{1-\epsilon}) \\ s[i-1 : i-1+3n^{1-\epsilon}) \neq s'[j-1 : j-1+3n^{1-\epsilon}) \end{cases}\Big\}$;
4 $\quad w_{\alpha,\beta} \leftarrow b_{\alpha+1} b_{\alpha+2}[1 : \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}]$;
$\quad$ `// Find the boundaries of the periodic region`
5 $\quad i^\circ \leftarrow (\alpha+1)n^{1-\epsilon} + \mathcal{D}_{s,s}.\mathsf{query}((\alpha+1)n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon} + |w_{\alpha,\beta}|)$;
6 $\quad j^\circ \leftarrow (\beta+1)n^{1-\epsilon} + \mathcal{D}_{s',s'}.\mathsf{query}((\beta+1)n^{1-\epsilon}, (\beta+1)n^{1-\epsilon} + |w_{\alpha,\beta}|)$;
7 $\quad p^\circ \leftarrow (i^\circ, j^\circ)$;
8 $\quad S_{\alpha,\beta} \leftarrow \max_{(i,j) \in P_{\alpha,\beta}} \min\{i^\circ - i, j^\circ - j\}$;
$\quad$ `// Find the extension beyond the periodic region`
9 $\quad$ **if** *there is any* $p^* = (i^*, j^*) \in P_{\alpha,\beta}$ *such that* $\delta(p^*) = \delta(p^\circ)$ **then**
10 $\quad\quad S_{\alpha,\beta} \leftarrow \max\{S_{\alpha,\beta}, \mathcal{D}_{s,s'}.\mathsf{query}(i^*, j^*)\}$;

with memory $O(n^{1-\epsilon})$, which are initialized with the appropriate data in the beginning. Recall the construction of $\mu_{\alpha,\beta}$ machine from Section 6.5. We allow each machine to have access to 3 subsequent blocks in each string additionally, i.e., $\{b_{\alpha+1}, b_{\alpha+2}, b_{\alpha+3}, b'_{\beta+1}, b'_{\beta+2}, b'_{\beta+3}\}$, so that we can locally compute the set of candidate pairs $P_{\alpha,\beta}$ in each machine $\mu_{\alpha,\beta}$ (Line 3 of Algorithm 34 which follows the definition of *candidate pairs* from Definition 6.8.1). Note that if there are several candidate pairs with the same offset, we keep the one with smallest i. Therefore, the total size of $P_{\alpha,\beta}$ is bounded by $O(n^{1-\epsilon})$.

The next step is to reduce the number of potential candidates in each machine to $O(1)$, which may count up to $O(n^{1-\epsilon})$ and in that case we need to make $O(n^{1-\epsilon)}$ queries per machine if we naively query each candidate pair.[10] If the number of candidate pairs is already $O(1)$, we can query each candidate pair separately using an LCPQ oracle. As mentioned before we initiate 3

[10]The LCPQ oracle actually supports $O(n^{1-\epsilon})$ queries per machine, however we minimize the number of required queries anyways for more flexibility.

instances of LCPQ oracle as the following which fit the memory limits.

$$\begin{cases} \mathcal{D}_{s,s} \leftarrow \mathsf{LCPInit}(s, s) \\ \mathcal{D}_{s,s'} \leftarrow \mathsf{LCPInit}(s, s') \\ \mathcal{D}_{s',s'} \leftarrow \mathsf{LCPInit}(s', s') \end{cases}$$

We use $\mathcal{D}_{s,s}$ and $\mathcal{D}_{s',s'}$ for finding the borders of the periodic pattern in s and s' respectively, and $\mathcal{D}_{s,s'}$ allows us to query the LCP of an individual candidate pair. According to Lemma 6.8.2, the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ are periodic with period length

$$|w_{\alpha,\beta}| = \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}.$$

Once we know the period length $|w_{\alpha,\beta}|$, we query $\mathcal{D}_{s,s}$ to find where the periodic pattern ends in s, aka *the first inconsistent character* or i°. We can pick any two pair of indices with distance $|w_{\alpha,\beta}|$ which both belong to $b_{\alpha+1}b_{\alpha+2}$, for example, $(\alpha+1)n^{1-\epsilon}$ and $(\alpha+1)n^{1-\epsilon}+|w_{\alpha,\beta}|$. Similarly, we find j° by querying $\mathcal{D}_{s',s'}$, and let $p^\circ = (i^\circ, j^\circ)$. The LCP of every candidate pair except one needs to end at either i° or j°.

Note that a character within the periodic range never matches with a first inconsistent character. Therefore, no candidate pair with an offset not equal to $\delta(p^\circ)$ can extend beyond p°. On the other hand, we know that every candidate pair $p = (i, j)$ has an LCP which is at least $\min\{i^\circ - i, j^\circ - j\}$ since the periodic pattern continues up to p°. Thus, we don't even need to query $\mathcal{D}_{s,s'}$ for almost all of the candidate pairs. However, if there is a candidate pair $p^* = (i^*, j^*)$ so that $\delta(p^*) = \delta(p^\circ)$, it may go beyond p° as long as the two strings have the same inconsistent

characters, i.e, $s_{i^\circ+1} = s'_{j^\circ+1}, s_{i^\circ+2} = s'_{j^\circ+2}, \ldots$. Only for this candidate pair, which is unique since we keep only one candidate if there were several with the same offset initially, we make a query to $\mathcal{D}_{s,s'}$. Since we already have computed the LCP of every pair $i, j \in b_\alpha \times b'_\beta$, we return the maximum LCP as the partial solution of this pair of blocks $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta)$.

The total number of queries we make to LCPQ oracles is bounded by $O(n^{2\epsilon})$, since there are only $O(1)$ LCP queries for each pair of blocks. Therefore, the total memory we need for LCP data structures is bounded by $O(n^{1+\epsilon})$. □

6.9 Massively Parallel Suffix Tree

6.9.1 Suffix Array with the LCPQ Oracle

In this section, we give our LCPQ oracle-based approach to suffix tree construction. The following algorithm ties that of the KMR suffix array construction in Section 6.2 in both communication and total rounds. Additionally, this construction allows for later improvement in the AMPC model, which the KMR approach does not.

We wish to find the **suffix array** of a given string s with length n in the MPC model with $O(1/\epsilon)$ rounds and $\widetilde{O}(n^{1-\epsilon})$ memory per machine for some $0 < \epsilon \leq 0.5$. In addition, the total memory needs to be $\widetilde{O}(n^{1+\epsilon})$. The *suffix array* of a given string s is just a permutation $\pi : [n] \to [n]$ over the suffixes of s so that they are sorted lexicographically based on π. We employ a quick-sort-like approach, similar to [Goodrich et al., 2011], which sorts an array in $O(1/\epsilon)$ rounds in MPC. Note that we need to make comparison queries carefully since we can't answer each of them in $O(1)$ time as opposed to the sorting of an array of integers.

Theorem 6.3.5. *For $\epsilon \in (0, .5]$, there is an $O(1/\epsilon)$-round* MPC *algorithm for computing the* **suffix**

Algorithm 35: SuffixArray(s)

Let $\ell \leftarrow \lceil \frac{2}{\epsilon} \rceil - 1$;
Let $L_0, L_1, L_2, \ldots, L_\ell$ be a collection of lexicographically ordered bags of sets;
Add $\{1, 2, \ldots, n\}$ to L_0;
$D_{s,s} \leftarrow$ LCPInit(s,s);
$\pi \leftarrow \{(i, i) \mid 0 \leq i < n\}$;
for $1 \leq z \leq \ell$ **do**
  for $1 \leq t \leq |L_z|$ **do**
    Let $U_t = \{u_{t,1}, u_{t,2}, \ldots, u_{t,k}\}$ be the t-th set in L_z ordered by the lexicographical order;
    if $z = \ell$ **then**
      // We have $k = O(n^{\epsilon/2})$.
      SmallChunkSort(t, U_t, π)
    else
      PivotFiltering$\big(t, U_t, O(n^{\epsilon/2})\big)$
Relabel π so that the relative order of π_i's is preserved, and the range of π is $[n]$;

array *of a given string s with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor w.h.p..*

Proof. In the first step $z = 1$, we uniformly randomly select $n^{\epsilon/2}$ pivots for each remaining bag in L_z, and therefore, we need to perform a total of $O(n^{1+\epsilon/2})$ lexicographical comparison of two suffixes of s in the first step. There are a total of $\ell = (2/\epsilon) - 1$ steps to finish the sorting, and at the end of the z-th step ($1 \leq z \leq \ell$), we have $O(n^{z\epsilon/2})$ disjoint subsequences with expected size $O(n^{1-z\epsilon/2})$, which is concentrated, and then each of them needs to be sorted recursively. See Algorithm 36 for more details. PivotFiltering(t, U_t, V) assumes U_t is the t-th unsorted chunk, in the relative ordering of the chunks, and decomposes U_t into V smaller chunks based on $V - 1$ pivots so that the smaller chunks are relatively sorted, and will be processed in the next step.

In each step $1 \leq z < 2/\epsilon$, we again choose $n^{\epsilon/2}$ random pivot for each of those subsequences, but still the total number of comparisons is bounded by $O(n^{\epsilon/2} \cdot n^{z\epsilon/2} \cdot n^{1-z\epsilon}/2) = O(n^{1+\epsilon/2})$. We first sort the randomly selected pivots using Algorithm 37 since $V = O(n^{\epsilon/2})$, which we explain shortly. We group the remaining elements of a bag using the result of their comparison with the

$O(n^{\epsilon/2})$ sorted pivots of the same bag.

Algorithm 36: PivotFiltering$(t, U_t = \{u_{t,1}, u_{t,2}, \ldots, u_{t,k}\}, V)$

Let p_0 and p_V be dummy elements which are by definition lexicographically smaller and lexicographically greater respectively than any other suffix;
Let $p'_1, p'_2 \leq \ldots \leq p'_{V-1}$ be $V-1$ uniformly random indices from $[k]$;
// Sorting the pivots
Let $\pi' \leftarrow \{(i,i) \mid 1 \leq i < V\}$;
SmallChunkSort$(0, \{p'_1, p'_2, \ldots, p'_{V-1}\}, \pi')$;
Let $p_{\pi'_i} \leftarrow p'_i$ for all $1 \leq i < V$;
// Decompose U_t based on $O(V)$ pivots into $v_0, v_1, \ldots, v_{V-1}$.
$\kappa_{i,j} \leftarrow D_{s,s}$.query$(p_i, u_{t,j})$ for every $0 \leq i \leq V$ and $1 \leq j \leq k$;
Let $v_i \leftarrow \left\{ u_{t,j} \in U_t \mid \begin{cases} s[p_i + \kappa_{i,j}] \leq s[u_{t,j} + \kappa_{i,j}], \text{and} \\ s[p_{i+1} + \kappa_{i+1,j}] > s[u_{t,j} + \kappa_{i+1,j}] \end{cases} \right\}$ for every $0 \leq i < V$;
Add $v_0, v_1, \ldots, v_{V-1}$ to L_{z+1} so that v_i has index $(t-1)n^{\epsilon/2} + i$ in L_{z+1};

Finally, in the last step $z = \ell$, we have at most $O(n^{\epsilon/2})$ suffixes in each bag, and we can query every pair of suffixes in every bag to create a collection Q of $O(n^{1+\epsilon/2} \cdot n^{\epsilon/2}) = O(n^{1+\epsilon})$ queries. We locally sort the small number, $O(n^{\epsilon/2})$ of suffixes by comparing each pair of them together and use it as an oracle for sorting. See Algorithm 37 for more details. According to Theorem 6.3.1, we can handle all these queries in $O(1)$ round and $O(n^{1+\epsilon})$ total memory using our LCPQ oracle. Knowing the result of the collection of queries $\mathcal{Q}$, we can compare the respective characters and compare two suffixes lexicographically. Ultimately, we sort each bag of size $O(n^{\epsilon/2})$ using SmallChunkSort subroutine, and the resulting function π is sorted lexicographically but exceeds the range limits. The last step is to relabel π so that the range of the function is indeed $[0, n)$ and also π is a valid permutation.

□

Algorithm 37: SmallChunkSort$(t, U_t = \{u_{t,1}, u_{t,2}, \ldots, u_{t,k}\}, \pi)$ s.t $k = O(n^{\epsilon/2})$

$\kappa_{i,j} \leftarrow D_{s,s}.\mathsf{query}(u_{t,i}, u_{t,j}) \quad \forall (i,j) \leq [k] \times [k]$;

$$\eta_{i,j} \leftarrow \begin{cases} 1 & \text{if } s[u_{t,i} + \kappa_{i,j}] \leq s[u_{t,j} + \kappa_{i,j}] \\ 0 & \text{if } s[u_{t,i} + \kappa_{i,j}] > s[u_{t,j} + \kappa_{i,j}] \end{cases};$$

Sort U_t locally using $\eta_{i,j}$'s as a comparison oracle to get $W_t = \{w_{t,1}, w_{t,2}, \ldots, w_{t,k}\}$;

Let $\pi_{w_{t,i}} \leftarrow tn + i$ for every $1 \leq i \leq k$;

6.9.2 Suffix Tree in AMPC

We will now show that in the AMPC model, we are able to improve our LCPQ oracle-based suffix array and suffix tree construction to $\tilde{O}(n)$ total memory and $O(1)$ communication rounds. The main flexibility that AMPC allows us from MPC is providing an $O(n)$ shared memory in which we can preprocess the input string s and answer $O(n)$ LCP queries within $\tilde{O}(n)$ space. In the preprocessing step, for all indices $i, 0 \leq i < n$, we store $\mathsf{hash}(s[i:n))$. Utilizing these partial hashes, we can then find the hash of any interval in string s, and by performing a binary search on interval length, we can find exactly the interval where two prefixes match in $\log(n)$ memory. The following algorithm performs such a recursive search.

Algorithm 38: LCP2Queries(i, j, o, k)

```
1 if hash(s[i : i + o)) = hash(s[j : j + o)) then
2     if k = 1 or hash(s[i : i + o + 1)) ≠ hash(s[j : j + o + 1)) then
3         return o;
4     o ← o + ⌈(n−j)/2^k⌉;
5     LCP2Queries(i, j, o, k + 1);
6 else
7     o ← o − ⌈(n−j)/2^k⌉;
8     LCP2Queries(i, j, o, k + 1);
```

Lemma 6.9.1. *For a preprocessed string s, given any indices $i < j$ such that $0 \leq i < j < |s|$ and $s[i] = s[j]$, $k = 1$, and $o = n - j$ as input, Algorithm 38 computes the LCP of two indices in s in $O(1)$ rounds and $O(\log^2 n)$ total space with high probability.*

Proof. Algorithm 38 is essentially performing a binary search on potential LCP lengths until it finds the exact length of the longest common palindrome between two prefixes of s starting at i and j. Therefore, we will prove the above claim by induction.

The algorithm ends in line 3 if the hash of the prefixes of s starting at i and starting at j are equal and $k = 1$, which means the entire suffix starting at i is equal to the suffix at j. If $k = 1$, then Algorithm 38 has just began and therefore $o = n - j$. Thus, if the hash of the prefixes of s starting at i and j are equal up to o characters, then the prefixes are completely equal up until the end of the string s and o is returned as the LCP length. Alternatively, if the hash of the prefixes are equal up to o characters but not $o + 1$, we also stop since we've found the LCP. If LCP$(i, j) = 1$, we will eventually stop due to this second condition since $s_i = s_j$ but $s_{i+1} \neq s_{j+1}$ when $k = \log n$.

We will assume without loss of generality that $\mathsf{LCP}(i,j) > 0$, which is easily verifiable without the need for hashing.

Assume that for the kth recursive call to Algorithm 38, the algorithm is correct. Consider the $(k+1)$th call. If $\mathsf{LCP}(i,j) \geq o$, then we find in step 1 that $\mathsf{hash}(s[i:i+o])$=$\mathsf{hash}(s[j:j+o])$ and call the algorithm again with $o + \lceil \frac{n-j}{2^{k+1}} \rceil < o + \lceil \frac{n-j}{2^k} \rceil$. Similarly, if $\mathsf{LCP}(i,j) < o$, we call the algorithm again with $o - \lceil \frac{n-j}{2^{k+1}} \rceil > o - \lceil \frac{n-j}{2^k} \rceil$. Since the algorithm was correct up to the kth recursive call, and we continue the search for the next smallest interval in both cases, the algorithm continues correctly. In the case that $\mathsf{LCP}(i,j) = o$, then $\mathsf{hash}(s[i:i+o+1)) \neq \mathsf{hash}(s[j:j+o+1))$ and we still stop at step 3 correctly and output $\mathsf{LCP}(i,j) = 0$.

This algorithm has $\log(n)$ recursive calls since k increases from 1 to at most $\log(n)$ before terminating. Since this algorithm runs locally on a single processor and for sufficiently large n, $n^\epsilon > \log(n)$, then Algorithm 38 uses only $O(1)$ rounds and requires $O(\log n)$ space per LCP query. We gain an additional $O(\log n)$ space factor to guarantee a good hash function with high probability. □

With our updated LCP query algorithm and Lemma 6.9.1, we have reduced the memory requirements of the LCPQ oracle construction from $O(n^{1+\epsilon})$ in MPC to $\tilde{O}(n)$ in AMPC with $O(1)$ rounds. We are also able to update the $\mathsf{PivotFiltering}$ parameters from Section 6.9.1 in AMPC to select $n^{1-\epsilon}$ pivots at once rather than n^ϵ. With high probability, we will then only need two rounds of pivot filtering to sort the suffixes. This change is seen in the third parameter at step 12.

With the updated LCPQ oracle we therefore can use Algorithm 39 to construct the suffix array π for a string s in $O(1)$ rounds and $\tilde{O}(n)$ memory since constructing the suffix array requires $O(n)$ queries per pivot sort and the LCPQ oracle requires $O(\log(n))$ memory per query. Further-

Algorithm 39: SuffixArray(s)

Let $\ell \leftarrow \lceil \frac{2}{\epsilon} \rceil - 1$;
Let $L_0, L_1, L_2, \ldots, L_\ell$ be a collection of lexicographically ordered bags of sets;
Add $\{1, 2, \ldots, n\}$ to L_0;
$D_{s,s} \leftarrow$ LCPInit(s,s);
$\pi \leftarrow \{(i, i) \mid 0 \leq i < n\}$;
for $1 \leq z \leq \ell$ **do**
    for $1 \leq t \leq |L_z|$ **do**
        Let $U_t = \{u_{t,1}, u_{t,2}, \ldots, u_{t,k}\}$ be the t-th set in L_z ordered by the lexicographical order;
        if $z = \ell$ **then**
            // We have $k = O(n^{\epsilon/2})$.
            SmallChunkSort(t, U_t, π)
        else
            PivotFiltering$\big(t, U_t, O(n^{(1-\epsilon)/2})\big)$
Relabel π so that the relative order of π_i's is preserved, and the range of π is $[n]$;

more, by Corollary 6.11.5, we can construct the suffix tree in the same space and rounds. Thus, we have our main AMPC result:

Theorem 6.3.6. (Suffix Tree in AMPC) . *For $\epsilon \in (0, 1]$, there is an $O(1)$-round* AMPC *algorithm for computing the* **suffix tree** *of a given string s with $\tilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

6.10 Conclusion

We have shown how to build a suffix tree in MPC in $O(n^{1+\epsilon})$ memory and constant rounds and in AMPC in $\tilde{O}(n)$ memory and constant rounds. For further improvement in MPC, one may look to improve the LCPQ oracle. Currently that is a main bottleneck that requires $O(n^{1+\epsilon})$ memory in MPC. Specifically, for an LCP query set $\mathcal{Q}$ of size k where $n^{1-\epsilon} < k < n^{1+\epsilon}$, a new solution to answer all queries in $O(n)$ total memory may be possible. Currently, we do not have this solution, and future work may be able to find such an LCPQ oracle. The following

problem encapsulates this idea:

Problem 6.10.1. *Given a string s of length n, initialize an* LCPQ **oracle** *with linear memory in $O(1)$ rounds of* MPC *that is capable of handling $O(n)$ arbitrary* LCP *queries in $O(1)$ rounds of* MPC *with $\widetilde{O}(n)$ total memory.*

Unfortunately, our current algorithm for suffix array construction (and therefore, suffix tree construction) requires $O(n^{1+\epsilon})$ comparisons when performing the pivot filtering. Therefore, even if the LCPQ structure is improved, our current suffix array algorithm would still require $O(n^{1+\epsilon})$ memory. If we can find a way to reduce the number of LCP queries needed, we may be able to have constant rounds and $\tilde{O}(n)$ total memory in MPC for suffix tree construction.

Problem 6.10.2. *Given a string s of length n, find the* **suffix tree** *in $O(1/\epsilon)$ rounds of* MPC *with $\widetilde{O}(n)$ total memory.*

Finally, the memory for LCS may still be improved. Specifically, when the LCS of two strings is small, our current solution simply checks every pair of blocks for the small solution, of which there are $n^{2\epsilon}$ pairs. If all pairs of blocks would not be needed, and instead only $O(n^{\epsilon})$ pairs, it would be possible to improve the LCS algorithm in MPC further as large solutions to LCS that span multiple blocks should be easier to detect than small solutions.

Problem 6.10.3. *Given two strings s and s' of length n, find* $\mathsf{LCS}_{s,s'}$ *in $O(1)$ rounds of* MPC *with $\widetilde{O}(n)$ total memory. It is guaranteed that* $\mathsf{LCS}_{s,s'} = o(n^{2\epsilon})$.

6.11 KMR Suffix Array and Tree Proofs

We give a proof that the constructed hashes correspond to the sorted order of substrings of s with length $n^{i\epsilon}$ for $1 \leq i \leq \frac{1}{\epsilon}$. With this lemma, it is easy to see that setting $i = \frac{1}{\epsilon}$ gives us the

sorted order of all suffixes of s, i.e. the suffix array (see Definition 6.2.1).

Lemma 6.11.1. *Given a string s, for all $1 \leq i \leq \frac{1}{\epsilon}, 0 \leq j, j' < n$, then $\mathcal{K}_{s,\epsilon}(i,j) < \mathcal{K}_{s,\epsilon}(i,j')$ if and only if $s[j, j+n^{i\epsilon}) < s[j', j'+n^{i\epsilon})$.*

Proof. We prove the above lemma inductively on i. First we set $i = 1$. For any $0 \leq j, j' < n$, if $s[j, j+n^{\epsilon}] < s[j', j'+n^{\epsilon}]$, then the position of $s[j, j+n^{\epsilon}]$ comes before $s[j', j'+n^{\epsilon}]$ in a sorted list of all substrings of length n^{ϵ} of s. Therefore, we have $\mathcal{K}_{s,\epsilon}(1,j) < \mathcal{K}_{s,\epsilon}(1,j')$. Similarly, if $\mathcal{K}_{s,\epsilon}(1,j) < \mathcal{K}_{s,\epsilon}(1,j')$ then it must be that the position of $s[j, j+n^{\epsilon}]$ comes before $s[j', j'+n^{\epsilon}]$ in a sorted list of all substrings of length n^{ϵ} of s. Therefore, $s[j, j+n^{\epsilon}] < s[j', j'+n^{\epsilon}]$.

Now, for the inductive hypothesis we assume that the lemma statement is true for some $i-1 \geq 1$. If $s[j, j+n^{i\epsilon}) < s[j', j'+n^{i\epsilon})$, then we know that for some $0 \leq k < n^{\epsilon}$, $s[j + kn^{(i-1)\epsilon}, j + (k+1)n^{(i-1)\epsilon}) < s[j' + kn^{(i-1)\epsilon}, j' + (k+1)n^{(i-1)\epsilon})$ and for all $1 \leq k' < k$, $s[j+k'n^{(i-1)\epsilon}, j+(k'+1)n^{(i-1)\epsilon}) = s[j'+k'n^{(i-1)\epsilon}, j'+(k'+1)n^{(i-1)\epsilon})$. By the inductive hypothesis, we then know that $\mathcal{K}_{s,\epsilon}(i-1, j+kn^{(i-1)\epsilon}) < \mathcal{K}_{s,\epsilon}(i-1, j'+kn^{(i-1)\epsilon})$ and $\mathcal{K}_{s,\epsilon}(i-1, j+k'n^{(i-1)\epsilon}) < \mathcal{K}_{s,\epsilon}(i-1, j'+k'n^{(i-1)\epsilon}$. Therefore,

$$
\begin{aligned}
&(\mathcal{K}_{s,\epsilon}(i-1,j), \mathcal{K}_{s,\epsilon}(i-1, j+n^{(i-1)\epsilon}), \mathcal{K}_{s,\epsilon}(i-1, j+2n^{(i-1)\epsilon}), \ldots, \mathcal{K}_{s,\epsilon}(i-1, j+(n^{(i-1)\epsilon}-1)n^{(i-1)\epsilon})) \\
&< (\mathcal{K}_{s,\epsilon}(i-1,j'), \mathcal{K}_{s,\epsilon}(i-1, j'+n^{(i-1)\epsilon}), \ldots, \mathcal{K}_{s,\epsilon}(i-1, j'+(n^{(i-1)\epsilon}-1)n^{(i-1)\epsilon})) \\
&\implies \mathcal{K}_{s,\epsilon}(i,j) < \mathcal{K}_{s,\epsilon}(i,j').
\end{aligned}
$$

By following the same steps in the reverse direction, we case also see that if $\mathcal{K}_{s,\epsilon}(i,j) < \mathcal{K}_{s,\epsilon}(i,j')$ then it must be the case that $s[j, j+n^{i\epsilon}) < s[j', j'+n^{i\epsilon})$ according to the induction

hypothesis. Thus, the lemma is proven by induction. □

Lemma 6.11.2. *For $\epsilon \in (0, 1]$, there is an $O(\frac{1}{\epsilon})$-round* MPC *algorithm for computing $\mathcal{K}_{s,\epsilon}$ of a given string s with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor.*

Combining these two lemmas gives us our suffix array construction in $O(1/\epsilon)$ communication rounds and $\widetilde{O}(n^{1+\epsilon})$ total memory:

Proof. We note that sorting $O(n)$ strings of length n^ϵ in MPC can be done in $O(1)$ rounds, $\widetilde{O}(n^{1+\epsilon})$ total space with $\widetilde{O}(n^{1-\epsilon})$ memory per processor (for example by using a modified pivot filtering algorithm from Section 6.9.1). Additionally there are only $O(\frac{1}{\epsilon})$ sorting rounds according to Definition 6.2.1, so in total we use $O(\frac{1}{\epsilon})$ rounds to construct $\mathcal{K}_{s,\epsilon}$. □

Proof of Theorem 6.2.2. Note that $\mathcal{K}_{s,\epsilon}(\frac{1}{\epsilon}, j)$ is the hash of the entire suffix of s starting at index j for any $0 \leq j < n$. Therefore, by Lemma 6.11.1 computing $\mathcal{K}_{s,\epsilon}(\frac{1}{\epsilon}, j)$ for all j suffices for computing the suffix array of s. By Lemma 6.11.2, we have the desired number of communication rounds and required space. □

6.11.1 Converting Suffix Array to Suffix Tree

Now we are ready to use the constructed suffix array to build a suffix tree. Let $\pi : [n] \to [n]$ be the permutation returned by the suffix array algorithm. Then $s_{\pi_i} s_{\pi_i+1} \dots s_{n-1}$ is the i-th substring of s lexicographically. We start by finding the LCP of each pair of adjacent suffixes in the suffix array, which we store in an array denoted LCP^π, e.g. $\mathsf{LCP}^\pi[i] = \mathsf{LCP}_{s,s}(\pi_i, \pi_{i+1})$. We populate the n indices of LCP^π using our LCPQ oracle from Section 6.5. Once the LCP^π array is made, we utilize the ideas from Iliopoulos and Rytter [Iliopoulos and Rytter, 2004] complemented

by a block-based data structure on the minimum values of the LCP^{π} array across each block to build the internal nodes of the suffix tree in constant rounds and $\tilde{O}(n^{1+\epsilon})$ total memory.

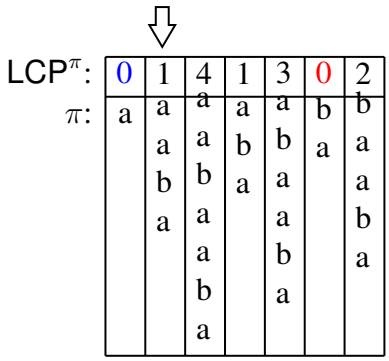


Figure 6.7: The above figure is an example of the suffix array π and LCP^{π} array for string "aabaaba". Furthermore, we look at π_1 as an example, the second suffix in the suffix array denoted by the arrow. The nearest left suffix and nearest right suffixes with a smaller LCP^{π} value are the suffixes "a" and "ba", as denoted by the blue and red colorings. The indices of these suffixes will therefore be $\mathcal{L}[\pi_1]$ and $\mathcal{R}[\pi_1]$, respectively.

To describe the details of our suffix tree algorithm, we review the solution of [Iliopoulos and Rytter, 2004] to address the difficulties we need to confront when implementing their approach in MPC. First, they observe that each non-leaf node in a suffix tree is the longest common ancestor (LCA) of two consecutive suffixes in the suffix array. They therefore start by finding the LCP of each pair of consecutive suffixes in the suffix array, which in turn gives the depth of the LCA of the same pair of vertices in the suffix tree. As we mentioned before we can find the LCP of pairs of suffixes using our LCPQ oracle. Following the methods of [Iliopoulos and Rytter, 2004], we need to compute two more arrays. The first array is a nearest smallest neighbor array SN, which is defined as:

$$\begin{cases} \mathsf{SN}[i] = & \begin{cases} \mathcal{L}[i], & \text{if } \mathsf{LCP}^{\pi}[\mathcal{L}[i]] \geq \mathsf{LCP}^{\pi}[\mathcal{R}[i]] \\ \mathcal{R}[i] & \text{o.w.} \end{cases} \\ \mathcal{L}[i] = & \max_{j<i} \mathsf{LCP}^{\pi}[j] < \mathsf{LCP}^{\pi}[i], \\ \mathcal{R}[i] = & \min_{j>i} \mathsf{LCP}^{\pi}[j] < \mathsf{LCP}^{\pi}[i]. \end{cases}$$

Intuitively, $\mathsf{SN}[i]$ will correspond to the parent of suffixes i and $i-1$ since it is the nearest pair of suffixes that have a shorter LCP^{π} value than $\mathsf{LCP}^{\pi}[i]$. See Figure 6.7 for an example of the LCP^{π} array and finding $\mathcal{L}$ and $\mathcal{R}$ values for an index. To break the ties between multiple pairs of suffixes that have the same LCP, we will also compute an array named $\mathsf{Leftmost}[i]$ which is the leftmost sibling of the LCA of suffixes i and $i-1$, defined formally as:

$$\mathsf{Leftmost}[i] = \begin{cases} 0 & \text{if } \mathsf{LCP}^{\pi}[i] = 0 \\ \min_{j \in [1,i]} \{\mathsf{LCP}^{\pi}[j] = \mathsf{LCP}^{\pi}[i], j \geq \mathsf{SN}[i]\} & \text{o.w.} \end{cases}$$

By doing this, [Iliopoulos and Rytter, 2004] observed that the parent of each non-leaf vertex i may be correctly and uniquely identified by $j = \mathsf{Leftmost}[\mathsf{SN}[i]]$ with label $(j, \mathsf{LCP}^{\pi}[j])$. It is not hard to find the father of leaves as well using the same two arrays $\mathsf{Leftmost}$ and SN. The following lemma from [Iliopoulos and Rytter, 2004] summarizes the relationships between these arrays and the suffix tree:

Lemma 6.11.3 (Lemma 1, [Iliopoulos and Rytter, 2004]). *Given internal node $(k_1, \mathsf{LCP}^{\pi}[k_1])$ and leaf node (k_2) where k_1, k_2 are the indices of leaf nodes corresponding to the suffixes $s[k_1 : n), s[k_2 : n)$ respectively:*

1 *The parent of* $(k_1, \mathsf{LCP}^\pi[k_1])$ *is* $(j, \mathsf{LCP}^\pi[j])$ *for* $j = \mathsf{Leftmost}[\mathsf{SN}[k_1]]$.

2 *If* $\mathsf{LCP}^\pi[k_2] \leq \mathsf{LCP}^\pi[k_2+1]$ *then the parent of* (k_2) *is* $(j, \mathsf{LCP}^\pi[j])$ *for* $j = \mathsf{Leftmost}[k_2]$. *Else the parent of* (k_2) *is* $(k_2+1, \mathsf{LCP}^\pi[k_2+1])$.

Now, to compute the SN and Leftmost arrays in MPC, we will first find and store LCP^π using the compressed LCPQ oracle. Specifically, on all block processors μ_α, $0 \leq \alpha < n^\epsilon$, the processor with block $b_\alpha = s[in^{1-\epsilon} : (i+1)n^{1-\epsilon}]$ as defined in 6.4.1, we will store $\mathsf{LCP}^\pi[j]$ for any index j such that $s_j \in b_\alpha$. Additionally, on every processor μ_α, we find the minimum over all stored LCP^π values of the previous step, which we denote as

$$m_i := \min_{j \in [in^{1-\epsilon}, (i+1)n^{1-\epsilon})} (\mathsf{LCP}^\pi[j]).$$

We will distribute these minimum to all processors so that any processor μ_α has a block-based data structure M such that $M[i] = m_i$. We then easily find $\mathcal{L}[k]$ and $\mathcal{R}[k]$ by finding the nearest left and right blocks b_ℓ and b_r such that $m_\ell, m_r < \mathsf{LCP}[k]$ and then send queries to find the exact value of $\mathcal{L}[k], \mathcal{R}[k]$ to processors μ_ℓ, μ_r, respectively, of all indices k where b_ℓ, b_r are the nearest blocks with a smaller longest common prefix value than $\mathsf{LCP}[k]$. If $\Omega(n^{1-\epsilon})$ queries are sent to a single processor, we will duplicate the processor and distribute the queries evenly using Algorithm 27. Since there is only one query per index to find SN, there is $O(n)$ total queries, which means we only need constant rounds of communication for this step. A similar set of steps is used to then build the Leftmost array for block b^i on processor μ_i. If $\mathsf{LCP}^\pi[k] = 0$ for an index k, then $\mathsf{Leftmost}[k]$ is trivially 0. Otherwise, we check the minimum LCP^π values of each neighboring blocks to the left of b^i to see if any have an equal value to $\mathsf{LCP}^\pi[k]$. If there is an equal minimum, then we query the processor with that block to find the exact value of $\mathsf{Leftmost}[k]$; if there is not an

equal minimum, we query μ_ℓ again since it is the first block with a smaller minimum since either that block may have an index j such that $\mathsf{LCP}^\pi[j] = \mathsf{LCP}^\pi[k]$ or there is no such index with an equal longest common prefix for a $j \geq \mathsf{SN}[i]$ as required by the definition of $\mathsf{Leftmost}$. In this case, $\mathsf{Leftmost}[k] = k$. Again, since we have n indices that require a single processor query, we can use Algorithm 27 to build the $\mathsf{Leftmost}$ array in constant rounds. We formally give the suffix tree construction by Algorithm 12 below.

Algorithm 40: $\mathsf{SuffixTree}(\pi, s)$

1 Distribute blocks of the suffix array from π such that processor μ_α has $\pi_{\alpha n^\epsilon}, \pi_{\alpha n^\epsilon+1}, ..., \pi_{(\alpha+1)n^\epsilon-1}$ for all $0 \leq \alpha < n^\epsilon$;
2 $D_{s,s} \leftarrow \mathsf{LCPInit}(s, s)$;
3 $\mathsf{LCP}^\pi[j] \leftarrow D_{s,s}.\mathsf{query}(\pi_j, \pi_{j-1})$ for all $0 \leq j < n$;
4 $m_\alpha \leftarrow \min_{j \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon}]}(\mathsf{LCP}^\pi[j])$ for all $0 \leq \alpha < n^\epsilon$;
5 $\mathcal{L}, \mathcal{R} \leftarrow \mathsf{Build}\mathcal{LR}(\mathsf{LCP}^\pi, \{m_\alpha | 0 \leq \alpha < n^\epsilon\})$;
6 $P \leftarrow \{(k, \mathcal{L}[k], \mathcal{R}[k], \mathsf{LCP}^\pi[\mathcal{L}[k]], \mathsf{LCP}^\pi[\mathcal{R}[k]]) | 0 \leq k < n\}$;
7 $\mathsf{SN} \leftarrow \mathsf{BuildSN}(P)$;
8 $\mathsf{Leftmost} \leftarrow \mathsf{BuildLeftmost}(\mathsf{LCP}^\pi, \mathsf{SN})$;
9 **for** $0 \leq k < n$ **do**
10 Add leaf node (k) to suffix tree;
11 **if** $k == \mathit{Leftmost}[k]$ **then**
12 Add internal node $(k, \mathsf{LCP}^\pi[k])$ to suffix tree;

Using Algorithm 12 and all its subroutines, we then can prove our main result.

Theorem 6.11.4. *For $\epsilon \in (0, .5]$, there is an $O(1/\epsilon)$-round* MPC *algorithm for computing the* **suffix tree** *of a given string s with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

Proof. We initialize the LCPQ oracle for s with itself in step 2 and then perform $O(n)$ queries to find the LCP between all pairs of consecutive suffixes in the suffix array in step 3. By Theorem 6.3.1, the LCP initialization and $O(n)$ LCP queries can be performed in $O(1)$ rounds and $\tilde{O}(n^{1+\epsilon})$ total memory. Each processor μ_α will be in charge of storing the $O(n^{1-\epsilon})$ values of $\mathsf{LCP}^\pi[j]$

for all indices j in its block of the suffix array. In step 4, we use these LCP^π values to find the minimum LCP of two consecutive suffixes per suffix array block, which can be done locally on each block machine and then sent to all other machines in a single round since each of the $n^\epsilon < n^{1-\epsilon}$ machines only sends a single value to all other machines. Next, we call a subroutine described in the following paragraph to fill in the $\mathcal{L}$ and $\mathcal{R}$ as defined previously.

The below algorithm, Algorithm 21, finds $\mathcal{L}$ and $\mathcal{R}$ in $O(n)$ total space and $O(1)$ rounds. First, we will discuss building the $\mathcal{L}$ array in steps 3-8 and 20-22, where $\mathcal{L}[k]$ contains the index of the nearest pair of consecutive suffixes in the suffix array to the left of k that has a smaller LCP than that $\mathsf{LCP}^\pi[k]$. The $\mathcal{R}$ array has the same steps but mirrored to find the nearest pair of suffixes with a smaller LCP to the right of k and can be seen in steps 9-14 and 23-24. First, for each index k, we check all LCP^π values for an LCP smaller than $\mathsf{LCP}^\pi[k]$ within the block containing k only for indices to the left of k. This is performed locally on a single processor. If $\mathcal{L}[k]$ is not found in the same block as k, we then check the minimum LCP^π values of neighboring blocks to the left of the block containing k using our block-based data structure until we find the nearest left block with a minimum LCP^π smaller than $\mathsf{LCP}^\pi[k]$. Since the minimum LCP^π value of this block is smaller than $\mathsf{LCP}^\pi[k]$, we are guaranteed to find $\mathcal{L}[k]$ at this block. We add requests to a request set so that we can query these left blocks for $\mathcal{L}[k]$ for all indices k in parallel. Before actually sending the queries, we use $\mathsf{WeightedLoadBalancing}$ to make sure each processor can handle its requests and copy any processor when needed. Since each index k for $0 \leq k < n$ only has a single query to find $\mathcal{L}$, then there are only $O(n)$ queries total to find the $\mathcal{L}$ and $\mathcal{R}$ arrays. Therefore, steps 17-25 only requires $O(1)$ rounds and $O(\tilde{n^{1-\epsilon}})$ space per processor after $\mathsf{WeightedLoadBalancing}$. To handle each $\mathcal{L}$ query in steps 20-22, a processor is given the index k and its $\mathsf{LCP}^\pi[k]$ and can compare the $\mathsf{LCP}^\pi[k]$ locally to its own values of LCP until it finds the maximum index in its suffix array block

with a smaller LCP than $\mathsf{LCP}^\pi[k]$. The maximum index with a smaller LCP is also the rightmost index, i.e. we find $\mathcal{L}[k]$ which is the closest index to k with a smaller LCP than $\mathsf{LCP}^\pi[k]$. As mentioned previously, it is easy to check that the steps to find $\mathcal{R}[k]$ mirror those to find $\mathcal{L}[k]$ and require the same space and number of rounds.

Algorithm 41: Build$\mathcal{LR}(\mathsf{LCP}^\pi, \{m_\alpha | 0 \leq \alpha < n^\epsilon\})$

1 $F_\alpha \leftarrow \emptyset$ for $0 \leq \alpha < n^\epsilon$;
2 **for** $0 \leq \alpha < n^\epsilon$ **do**
  // in machine μ_α
3 **for** $k \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon}]$ **do**
4 **if** $\exists k' \in [\alpha n^{1-\epsilon}, k)$ *s.t.* $\mathit{LCP}^\pi[k'] < \mathit{LCP}^\pi[k]$ **then**
5 $\mathcal{L}[k] \leftarrow \max_{k' \in [\alpha n^{1-\epsilon}, k)} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
6 **else**
7 $\ell \leftarrow \max_{\beta < \alpha} \{\beta : m_\beta < m_\alpha\}$;
8 Add $(k, \mathsf{LCP}^\pi[k])$ to F_ℓ;
9 **if** $\exists k' \in (k, (\alpha+1)n^{1-\epsilon})$ *s.t.* $\mathit{LCP}^\pi[k'] < \mathit{LCP}^\pi[k]$ **then**
10 $\mathcal{R}[k] \leftarrow \min_{k' \in (k, (\alpha+1)n^{1-\epsilon})} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
11 **else**
12 $r \leftarrow \min_{\beta > \alpha} \{\beta : m_\beta < m_\alpha\}$;
13 Add $(k, \mathsf{LCP}^\pi[k])$ to F_r;
14 $\{c_\alpha \mid 0 \leq \alpha < n^\epsilon\} \leftarrow$ WeightedLoadBalancing$(\{\mu_\alpha | 0 \leq \alpha < n^\epsilon\}, \{F_\alpha | 0 \leq \alpha < n^\epsilon\})$;
15 **for** $0 \leq \alpha < n^\epsilon, 1 \leq z \leq c_\alpha$ **do**
  // in machine $\mu_\alpha^{(z)}$
16 **for** $(k, \mathit{LCP}^\pi[k]) \in F_\alpha^{(z)}$ **do**
17 **if** $k > (\alpha+1)n^{1-\epsilon}$ **then**
18 $\mathcal{L}[k] \leftarrow \max_{k' \in [\alpha n^{1-\epsilon}, k)} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
19 **else**
20 $\mathcal{R}[k] \leftarrow \min_{k' \in (k, (\alpha+1)n^{1-\epsilon})} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
21 Return $\mathcal{L}, \mathcal{R}$;

Using the $\mathcal{L}$, $\mathcal{R}$, and LCP^π arrays, we can build the SN array according to its definition. We call another subroutine Algorithm 6 to build the SN array with these inputs in steps 7-8 of the suffix tree construction. Each processor μ_α can find and store the block of SN for the suffixes in its block of the suffix array. Therefore, this step trivially runs in constant rounds, $O(n^{1-\epsilon})$ space per machine, and $O(n)$ total memory.

Algorithm 42: BuildSN($P = \{(k, \mathcal{L}[k], \mathcal{R}[k], \mathsf{LCP}^\pi[\mathcal{L}[k]], \mathsf{LCP}^\pi[\mathcal{R}[k]]) | 0 \leq k < n\}$)

```
1 for (k, L[k], R[k], LCP^π[L[k]], LCP^π[R[k]]) ∈ P do
2 |   if LCP^π[L[k]] ≥ LCP^π[R[k]] then
3 |   |   SN[k] ← L[k];
4 |   else
5 |   |   SN[k] ← R[k];
6 return SN;
```

After building the SN array, we can construct the final array before making the suffix tree, the Leftmost array. In step 9, we call the BuildLeftmost subroutine, described below, which builds Leftmost according to its definition using similar steps to the Build$\mathcal{LR}$ subroutine.

Algorithm 15 begins in steps 3 and 4 for a suffix index k by setting any index with $\mathsf{LCP}^\pi[k] = 0$ to 0, which is the correct value by the definition of the Leftmost array. If $\mathsf{LCP}^\pi[k] \neq 0$, let b_i be the block containing k of size $n^{1-\epsilon}$, i.e. $i = \lceil \frac{k}{n^{1-\epsilon}} \rceil$. Then, in steps 5 and 6, the algorithm searches for the farthest block b_j from b_i of length $n^{1-\epsilon}$ with a minimum LCP^π value equal to $\mathsf{LCP}^\pi[k]$ between b_k and the block containing $\mathsf{SN}[k]$. If $\mathsf{SN}[k]$ is to the right of k, no such block exists and we know that $\mathsf{Leftmost}[k] = k$ by definition which is fulfilled in step 9. If $\mathsf{SN}[k]$ is to the left of $\mathsf{LCP}^\pi[k]$, we know that the block containing $\mathsf{SN}[k]$ is the farthest block to the left of b_i with a smaller minimum LCP^π value than $\mathsf{LCP}[k]$. Thus, if we do not find such a block b_j with a minimum LCP^π equal to $\mathsf{LCP}[k]$, we know no such block exists between b_i and the block containing $\mathsf{SN}[k]$ and again in step 9 we set $\mathsf{Leftmost}[k]$ to k. If we find such a block b_j, then in steps 6 and 7 we add a query to be handled later to find Leftmost on block b_j in processor μ_j^π. Step 10 calls AdjustRepetitions2 to make sure each processor only needs to handle $O(n^{1-\epsilon})$ queries out of the total $O(n)$ queries, and then steps 13 and 14 find $\mathsf{Leftmost}[k]$ according to its definition by finding the leftmost index on block b_j for index k with an LCP^π value equal to $\mathsf{LCP}^\pi[k]$.

Since each index k only requires a single query, AdjustRepetitions2 will successfully allow all $O(n)$ potential queries to be handled in $O(1)$ rounds, total space $O(n)$, and $O(n^{1-\epsilon})$ memory per processor. Thus, BuildLeftmost will build the leftmost array in the desired space and rounds.

Algorithm 43: BuildLeftmost($\mathsf{LCP}^\pi, \mathsf{SN}$)

$F_\alpha \leftarrow \emptyset$ for $0 \leq \alpha < n^\epsilon$;
for $0 \leq \alpha < n^\epsilon$ **do**
  // in machine μ_α
  for $k \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon})$ **do**
    if $\mathit{LCP}^\pi[k] == 0$ **then**
      $\mathsf{Leftmost}[k] \leftarrow 0$;
    else if $\exists \beta, \lceil \frac{\mathsf{SN}[k]}{n^{1-\epsilon}} \rceil \leq \beta < \lceil \frac{k}{n^{1-\epsilon}} \rceil$ *s.t.* $m_\beta = \mathit{LCP}^\pi[k]$ **then**
      $\beta' \leftarrow \min_{\beta \in [\lceil \frac{\mathsf{SN}[k]}{n^{1-\epsilon}} \rceil, \lceil \frac{k}{n^{1-\epsilon}} \rceil)}$;
      Add $(k, \mathsf{LCP}^\pi[k])$ to $F_{\beta'}$;
    else
      $\mathsf{Leftmost}[k] \leftarrow k$;
$\{c_\alpha \mid 0 \leq \alpha < n^\epsilon\} \leftarrow$ WeightedLoadBalancing($\{\mu_\alpha | 0 \leq \alpha < n^\epsilon\}, \{F_\alpha | 0 \leq \alpha < n^\epsilon\}$);
for $0 \leq \alpha < n^\epsilon, 1 \leq z \leq c_\alpha$ **do**
  // in machine $\mu_\alpha^{(z)}$
  for $(k, \mathit{LCP}^\pi[k]) \in F_\alpha^{(z)}$ **do**
    $\mathsf{Leftmost}[k] \leftarrow \min_{j \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon}]} \{\mathsf{LCP}^\pi[j] = \mathsf{LCP}^\pi[k]\}$;
return Leftmost;

With the addition of the SN and Leftmost arrays, we may build the suffix tree according to [Iliopoulos and Rytter, 2004]. We simply add internal nodes for every unique Leftmost suffix. Edges are detailed in Lemma 6.11.3 and are implicitly already provided to us using the LCP^π, SN, and Leftmost arrays.

□

Corollary 6.11.5. *Given a string s, its suffix array π, and its LCP^π array, we can construct the suffix tree of s in MPC with $O(1)$ rounds and $O(n)$ total memory.*

Proof. This corollary is a natural result of the previous proof. Note that to build a suffix tree from

π and LCP^{π} in MPC, we only need to run the $\mathsf{Build}\mathcal{LR}$, $\mathsf{BuildSN}$, and $\mathsf{BuildLeftmost}$ subroutines. All three of these algorithms use $\mathsf{WeightedLoadBalancing}$ to run in $O(1)$ rounds and $O(n)$ total memory. □

6.11.2 LCS with Suffix Tree in MPC

We will now use our suffix tree structure augmented with the $\mathsf{LCP}^{\pi}, \mathcal{L}, \mathcal{R}, \mathsf{SN}, \mathsf{Leftmost}$, and suffix arrays to solve the LCS and LPS applications in $O(1/\epsilon)$ rounds and $\widetilde{O}(n^{1+\epsilon})$ total memory. Our algorithms for these problems utilizing the suffix tree in MPC are different than prior sequential solutions due to an inability to perform tree searches in constant rounds. To overcome this issue, we look at each internal node of the suffix tree and find its descendants using the ordering of the suffix array and block-based data structures. We will formally state this idea for the LCS application first in this section.

Given two input strings s and s' for the LCS problem, we first follow the steps of a typical suffix tree solution and concatenate the strings with a special character such as \$ to separate the strings. For example, given input strings $s =$ "apple" and $s' =$ "banana", we build the suffix tree on concatenated string "apple\$banana". Once we have our suffix tree, we then search the internal nodes of the tree in parallel for the LCS; note there may be at most $O(n)$ internal nodes, one per suffix. Any internal node represents the longest common ancestor between at least two suffixes in the suffix tree. If any internal node is the ancestor of suffixes from both strings s and s', then it corresponds to a common substring between s and s', and we therefore want to find the internal node corresponding to the longest common substring. However, to check if an internal node is an ancestor of a suffix from each input string typically requires a search of the tree, e.g. a depth-first

search. Unfortunately, since the suffix tree can have a depth of $O(n)$, a depth-first search could require up to $O(n^\epsilon)$ rounds of communication, one per processor. We avoid a depth-first search entirely by using the underlying suffix array and the additional suffix information we computed when converting suffix array to suffix tree. Given an internal node $(k, \mathsf{LCP}^\pi[k])$ of the suffix tree, we observe that the range of indices in the suffix array that are descendants of this internal node is $(\mathcal{L}[k], \mathcal{R}[k])$.

Observation 6.11.6. *Given internal node $v = (k, \mathsf{LCP}^\pi[k])$, a leaf node $u = (j)$ is a descendant of v if and only if $\mathcal{L}[k] \leq j < \mathcal{R}[k]$.*

The previous statement follows since any suffix with index k' within this range has $\mathsf{LCP}^\pi[k'] > \mathsf{LCP}^\pi[k]$, which implies that the k'th suffix indeed contains the prefix of length $\mathsf{LCP}^\pi[k]$ corresponding to the internal node $(k, \mathsf{LCP}^\pi[k])$. Note that only one side is inclusive because the $\mathsf{LCP}^\pi[k]$ is defined as the longest common prefix between π_k and π_{k-1}.

By keeping track of whether or not each block of the suffix array has a suffix from each string, we can easily find the lowest internal node on each processor μ_α that has a leaf descendant from each input string and then take the lowest internal node across all processors to find the longest common substring. To this end, we will simply construct an additional array $\mathcal{B}$ that has a value of 1 if an internal node of the suffix tree has a child from both strings to complete the algorithm.

With the above algorithm that relies on our suffix tree construction, we have the following corollary:

Corollary 6.11.7. (LCS using Suffix Tree). *Let k be the number of communication rounds needed to construct suffix tree w.h.p. in* MPC *with $O(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor for $\epsilon \in (0, .5]$. Then, there is an $O(k)$-round* MPC *solution for* LCS *of two input strings s, s'*

Algorithm 44: LCS$(s = \{s_0 s_1 ... s_{n-1}\}, s' = \{s'_0 s'_1 ... s'_{n-1}\})$

Let $t = \{s_0 s_1 ... s_{n-1} \$ s'_0 s'_1 ... s'_{n-1}\}$;
$\pi \leftarrow$ SuffixArray(t);
SuffixTree(π, t);
Build$\mathcal{B}()$;
$mx_\alpha, mxk_\alpha \leftarrow 0$ for $0 \leq \alpha < n^\epsilon$;
for *internal nodes* $(k, LCP^\pi[k])$ **do**
    // On processor $\mu_{\lfloor \frac{k}{n^{1-\epsilon}} \rfloor}$
    $\alpha \leftarrow \lfloor \frac{k}{n^{1-\epsilon}} \rfloor$;
    if $\mathcal{B}[k] == 1$ *AND* $LCP^\pi[k] > mx_\alpha$ **then**
        $mx_\alpha \leftarrow \mathsf{LCP}^\pi[k]$;
        $mxk_\alpha \leftarrow k$;
Return mx_α, mxk_α such that $mx_\alpha = \max_{0 \leq beta < n^\epsilon}\{mx_\beta\}$;

with $\widetilde{O}(n^{1+\epsilon})$ *total memory and* $\widetilde{O}(n^{1-\epsilon})$ *memory per processor w.h.p.*

Note that the construction of $\mathcal{B}$ is written below in Algorithm 45. The construction of $\mathcal{B}$ is long but straightforward. First, a block-based data structure is created and stored on all block machines that stores whether or not a block has a suffix from s and $\overline{s}$. Then, for each internal node $(k, \mathsf{LCP}^\pi[k])$, we simply check if any blocks in the range of blocks between $\mathcal{L}[k], \mathcal{R}[k]$ have a suffix from s and $\overline{s}$ using the block-based data structure. If not, we check the specific blocks containing $\mathcal{L}[k]$ and $\mathcal{R}[k]$ as a last chance and set $\mathcal{B}[k]$ accordingly.

6.11.3 LPS with Suffix Tree in MPC

For our second suffix tree application, we solve the LPS problem in $O(1/\epsilon)$ rounds and $\widetilde{O}(n^{1+\epsilon})$ total memory. Once again, we use the suffix tree structure augmented with the same arrays created during the suffix tree construction. Instead of building a suffix tree on two concatenated input strings as in LCS, we build the suffix tree on the single input string s concatenated with the reverse of the input string $\overline{s}$. Any internal node that is an ancestor of both a suffix from s starting at

Algorithm 45: Build$\mathcal{B}()$

for $0 \leq \alpha < n^{\epsilon}$ **do**
  // On processor μ_{α}
  $h_{\alpha}, \overline{h_{\alpha}} \leftarrow 0$;
  if $\exists k \in (\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon})$ *such that* $\$ \in s[\pi_k : n)$ **then**
    $h_{\alpha} \leftarrow 1$;
  if $\exists k \in (\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon})$ *such that* $\$ \notin s[\pi_k : n)$ **then**
    $\overline{h_{\alpha}} \leftarrow 1$;
  Send $h_{\alpha}, \overline{h_{\alpha}}$ to all processors μ_{β} for $0 \leq \beta < n^{\epsilon}$;
$F_{\alpha} \leftarrow \emptyset$ for $0 \leq \alpha < n^{\epsilon}$;
for *internal nodes* $(k, \mathit{LCP}^{\pi}[k])$ **do**
  // On processor $\mu_{\lfloor \frac{k}{n^{1-\epsilon}} \rfloor}$
  $\alpha \leftarrow \lfloor \frac{k}{n^{1-\epsilon}} \rfloor$;
  $f_k, \overline{f_k} \leftarrow 0$;
  if $\exists \beta, \lfloor \frac{\mathcal{L}[k]}{n^{1-\epsilon}} \rfloor < \beta < \lfloor \frac{\mathcal{R}[k]}{n^{1-\epsilon}} \rfloor$ *s.t.* $h_{\beta} = 1$ **then**
    $f_k \leftarrow 1$;
  if $\exists \beta, \lfloor \frac{\mathcal{L}[k]}{n^{1-\epsilon}} \rfloor < \beta < \lfloor \frac{\mathcal{R}[k]}{n^{1-\epsilon}} \rfloor$ *s.t.* $\overline{h_{\beta}} = 1$ **then**
    $\overline{f_k} \leftarrow 1$;
  Add $(k, \mathcal{L}[k], f_k, \overline{f_k})$ to $F_{\lfloor \frac{\mathcal{L}[k]}{n^{1-\epsilon}} \rfloor}$;
  Add $(k, \mathcal{R}[k], f_k, \overline{f_k})$ to $F_{\lfloor \frac{\mathcal{R}[k]}{n^{1-\epsilon}} \rfloor}$;
$\{c_{\alpha} \mid 0 \leq \alpha < n^{\epsilon}\} \leftarrow$ WeightedLoadBalancing$(\{\mu_{\alpha} | 0 \leq \alpha < n^{\epsilon}\}, \{F_{\alpha} | 0 \leq \alpha < n^{\epsilon}\})$;
for $0 \leq \alpha < n^{\epsilon}, 1 \leq z \leq c_{\alpha}$ **do**
  // in machine $\mu_{\alpha}^{(z)}$
  for $(k, o, f_k, f'_k) \in F_{\alpha}^{(z)}$ **do**
    if $\exists j$ *s.t.* $\$ \in s[\pi_j : n)$ *for* $\alpha n^{1-\epsilon} \leq j < o$ *OR* $o \leq j < (\alpha+1)n^{1-\epsilon}$ **then**
      $f_k \leftarrow 1$;
    if $\exists j$ *s.t.* $\$ \notin s[\pi_j : n)$ *for* $\alpha n^{1-\epsilon} \leq j < o$ *OR* $o \leq j < (\alpha+1)n^{1-\epsilon}$ **then**
      $\overline{f_k} \leftarrow 1$;
    Send $(f_k, \overline{f_k})$ to $\mu_{\lfloor \frac{k}{n^{1-\epsilon}} \rfloor}$;
for *internal nodes* $(k, \mathit{LCP}^{\pi}[k])$ **do**
  // On processor $\mu^{\pi}_{\lfloor \frac{k}{n^{1-\epsilon}} \rfloor}$
  $\mathcal{B}[k] = 0$;
  if $f_k == 1$ *AND* $\overline{f_k} == 1$ **then**
    $\mathcal{B}[k] = 1$;

index i and a suffix from $\overline{s}$ starting at $n-i$ corresponds to a palindrome since it is a common prefix of the substring to the left and right of index i in s. Unfortunately, as in the LCS suffix tree solution, we cannot perform efficient tree searches in constant rounds and small memory to find the lowest internal node in a sequential fashion. Instead, we will once again search all internal nodes of the suffix tree in parallel with an additional array $\mathcal{P}$ that we must construct that will answer queries for whether or not an internal node corresponds to a palindrome of s or not, i.e. is the ancestor of a suffix and its correlated reverse suffix.

Algorithm 46: LPS$(s = \{s_1 s_2 ... s_n\}$

```
Let t = {s_1 s_2 ... s_n $ s_n s_{n-1} ... s_1};
π ← SuffixArray(t);
SuffixTree(π, t);
BuildP();
mx_i, mxk_i ← 0 for 0 ≤ i < n^ε;
for internal nodes (k, LCP^π[k]) do
    // On processor μ^π_{⌊k/n^{1-ε}⌋}
    i ← ⌊k/n^{1-ε}⌋;
    if P[k] == 1 AND LCP^π[k] > mx_i then
        mx_i ← LCP^π[k];
        mxk_i ← k;
Return mx_i, mxk_i such that mx_i = max_{0≤j<n^ε}{mx_j};
```

With Algorithm 46, which relies on our suffix tree construction and subroutine Algorithm 47, we have the following corollary:

Corollary 6.11.8. (LPS using Suffix Tree). *For $\epsilon \in (0, .5]$, there is an $O(1/\epsilon)$-round* MPC *algorithm for computing the* LPS *of an input strings s with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

Note that to build $\mathcal{P}$ we follow a similar set of steps in Algorithm 47 to the construction of $\mathcal{B}$ for LCS. For an internal node $(k, \mathsf{LCP}^{\pi}[k])$, instead of checking for a suffix descendant of both

strings between $\mathcal{L}[k]$ and $\mathcal{R}[k]$, we simply check for a suffix and its reverse suffix between $\mathcal{L}[k]$ and $\mathcal{R}[k]$.

Algorithm 47: Build$\mathcal{P}()$

$\mathcal{P}[k] \leftarrow 0$ for $0 \leq k < n$;
for $0 \leq \alpha < n^\epsilon$ **do**
    // On processor μ_α
    $\ell_\alpha \leftarrow \max\{j | \exists k \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon})$ s.t. $j < k$ and $\pi_j = n - \pi_k\}$;
    $r_\alpha \leftarrow \min\{j | \exists k \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon})$ s.t. $j > k$ and $\pi_j = n - \pi_k\}$;
    Send ℓ_α and r_α to all processors;
$F_j \leftarrow \emptyset$ for $0 \leq j < n^\epsilon$;
for *internal nodes* $(k, \mathsf{LCP}^\pi[k])$ **do**
    // On processor $\mu_{\lfloor \frac{k}{n^{1-\epsilon}} \rfloor}$
    $\alpha \leftarrow \lfloor \frac{k}{n^{1-\epsilon}} \rfloor$;
    $\alpha_\ell \leftarrow \lfloor \frac{\mathcal{L}[k]}{n^{1-\epsilon}} \rfloor$;
    $\alpha_r \leftarrow \lfloor \frac{\mathcal{R}[k]}{n^{1-\epsilon}} \rfloor$;
    if $\exists \beta \in (\alpha_\ell, \alpha_r)$ *s.t.* $\mathcal{L}[k] \leq \ell_\beta \leq \mathcal{R}[k]$ *or* $\mathcal{L}[k] \leq r_\beta \leq \mathcal{R}[k]$ **then**
        $\mathcal{P}[k] \leftarrow 1$;
    Add $(k, \mathcal{L}[k], \mathcal{R}[k])$ to $F_{\alpha_\ell}, F_{\alpha_r}$;
$\{c_\alpha \mid 0 \leq \alpha < n^\epsilon\} \leftarrow$ WeightedLoadBalancing$(\{\mu_\alpha^\pi | 0 \leq \alpha < n^\epsilon\}, \{F_\alpha | 0 \leq \alpha < n^\epsilon\})$;
for $0 \leq \alpha < n^\epsilon, 1 \leq z \leq \alpha$ **do**
    // in machine $\mu_\alpha^{(z)}$
    for $(k, \mathcal{L}[k], \mathcal{R}[k]) \in F_\alpha^{(z)}$ **do**
        if $\exists j \in [\alpha n^{1-\epsilon}, (\alpha+1)n^{1-\epsilon}]$ *s.t.* $\pi_j = n - \pi_k$ *and* $\mathcal{L}[k] < j, k < \mathcal{R}[k]$ **then**
            $\mathcal{P}[k] \leftarrow 1$;

Bibliography

[Abboud, 2014] Abboud, A. (2014). Hardness for easy problems. Presented at Satellite Workshop of ICALP (YR-ICALP).

[Abboud et al., 2019] Abboud, A., Addanki, R., Grandoni, F., Panigrahi, D., and Saha, B. (2019). Dynamic set cover: improved algorithms and lower bounds. In Charikar, M. and Cohen, E., editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 114–125. ACM.

[Abboud et al., 2015a] Abboud, A., Backurs, A., and Williams, V. V. (2015a). Tight hardness results for LCS and other sequence similarity measures. In Guruswami, V., editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society.

[Abboud et al., 2018] Abboud, A., Backurs, A., and Williams, V. V. (2018). If the current clique algorithms are optimal, so is Valiant's parser. *SIAM J. Comput.*, 47(6):2527–2555.

[Abboud et al., 2015b] Abboud, A., Williams, V. V., and Yu, H. (2015b). Matching triangles and basing hardness on an extremely popular conjecture. In Servedio, R. A. and Rubinfeld, R., editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 41–50. ACM.

[Aho and Peterson, 1972] Aho, A. V. and Peterson, T. G. (1972). A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312.

[Akmal and Jin, 2021] Akmal, S. and Jin, C. (2021). Faster algorithms for bounded tree edit distance. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 12:1–12:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Akutsu et al., 2010a] Akutsu, T., Fukagawa, D., and Takasu, A. (2010a). Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348.

[Akutsu et al., 2010b] Akutsu, T., Fukagawa, D., and Takasu, A. (2010b). Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348.

[Alexandr et al., 2019] Alexandr, Stein, C., and Zhong, P. (2019). Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In Baier, C., Chatzigiannakis, I., Flocchini, P., and

Leonardi, S., editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Alman et al., 2024] Alman, J., Duan, R., Vassilevska Williams, V., Xu, Y., Xu, Z., and Zhou, R. (2024). More asymmetry yields faster matrix multiplication. *CoRR*, abs/2404.16349.

[Alstrup et al., 2000] Alstrup, S., Brodal, G. S., and Rauhe, T. (2000). Pattern matching in dynamic texts. In Shmoys, D. B., editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, pages 819–828. ACM/SIAM.

[Amir et al., 2019] Amir, A., Boneh, I., Charalampopoulos, P., and Kondratovsky, E. (2019). Repetition Detection in a Dynamic String. In Bender, M. A., Svensson, O., and Herman, G., editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Andersson and Nilsson, 1994] Andersson, A. and Nilsson, S. (1994). A new efficient radix sort. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 714–721. IEEE Computer Society.

[Andoni et al., 2010] Andoni, A., Krauthgamer, R., and Onak, K. (2010). Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 377–386. IEEE Computer Society.

[Andoni et al., 2014] Andoni, A., Nikolov, A., Onak, K., and Yaroslavtsev, G. (2014). Parallel algorithms for geometric graph problems. In *STOC*, pages 574–583.

[Andoni and Nosatzki, 2020] Andoni, A. and Nosatzki, N. S. (2020). Edit distance in near-linear time: it's a constant factor. In *61st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2020*, pages 990–1001. IEEE.

[Andoni and Onak, 2012] Andoni, A. and Onak, K. (2012). Approximating edit distance in near-linear time. *SIAM J. Comput.*, 41(6):1635–1648.

[Andoni et al., 2020] Andoni, A., Stein, C., and Zhong, P. (2020). Parallel approximate undirected shortest paths via low hop emulators. In Makarychev, K., Makarychev, Y., Tulsiani, M., Kamath, G., and Chuzhoy, J., editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 322–335. ACM.

[Apostolico et al., 1990] Apostolico, A., Atallah, M. J., Larmore, L. L., and McFaddin, S. (1990). Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988.

[Apostolico et al., 1994] Apostolico, A., Breslauer, D., and Galil, Z. (1994). Parallel detection of all palindromes in a string. In *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*, pages 497–506. Springer.

[Assadi et al., 2019a] Assadi, S., Bateni, M., Bernstein, A., Mirrokni, V. S., and Stein, C. (2019a). Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *SODA*, pages 1616–1635.

[Assadi et al., 2019b] Assadi, S., Chen, Y., and Khanna, S. (2019b). Sublinear algorithms for $\delta+1$ vertex coloring. In *SODA*, pages 767–786.

[Assadi et al., 2018] Assadi, S., Onak, K., Schieber, B., and Solomon, S. (2018). Fully dynamic maximal independent set with sublinear update time. In Diakonikolas, I., Kempe, D., and Henzinger, M., editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 815–826. ACM.

[Backurs and Indyk, 2018] Backurs, A. and Indyk, P. (2018). Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097.

[Backurs and Onak, 2016] Backurs, A. and Onak, K. (2016). Fast algorithms for parsing sequences of parentheses with few errors. In Milo, T. and Tan, W., editors, *35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016*, pages 477–488. ACM.

[Bannai et al., 2017] Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., and Tsuruta, K. (2017). The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514.

[Bar-Yossef et al., 2004] Bar-Yossef, Z., Jayram, T. S., Krauthgamer, R., and Kumar, R. (2004). Approximating edit distance efficiently. In *45th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2004*, page 550–559. IEEE.

[Baswana et al., 2011] Baswana, S., Gupta, M., and Sen, S. (2011). Fully dynamic maximal matching in o(log n) update time. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 383–392.

[Bateni et al., 2023] Bateni, M., Esfandiari, H., Fichtenberger, H., Henzinger, M., Jayaram, R., Mirrokni, V., and Wiese, A. (2023). Optimal fully dynamic *k*-center clustering for adaptive and oblivious adversaries. In Bansal, N. and Nagarajan, V., editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 2677–2727. SIAM.

[Batu et al., 2006] Batu, T., Ergun, F., and Sahinalp, C. (2006). Oblivious string embeddings and edit distance approximations. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, page 792–801. SIAM.

[Beame et al., 2017] Beame, P., Koutris, P., and Suciu, D. (2017). Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58.

[Behnezhad et al., 2019a] Behnezhad, S., Brandt, S., Derakhshan, M., Fischer, M., Hajiaghayi, M., Karp, R. M., and Uitto, J. (2019a). Massively parallel computation of matching and mis in sparse graphs. In *ACM SIGACT*, pages 481–490.

[Behnezhad et al., 2019b] Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Knittel, M., and Saleh, H. (2019b). Streaming and massively parallel algorithms for edge coloring. In *ESA*, pages 15:1–15:14.

[Behnezhad et al., 2019c] Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., and Mirrokni, V. S. (2019c). Near-optimal massively parallel graph connectivity. In Zuckerman, D., editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1615–1636. IEEE Computer Society.

[Behnezhad et al., 2019d] Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., Mirrokni, V. S., and Schudy, W. (2019d). Massively parallel computation via remote memory access. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 59–68. ACM.

[Behnezhad et al., 2019e] Behnezhad, S., Hajiaghayi, M., and Harris, D. G. (2019e). Exponentially faster massively parallel maximal matching. In *FOCS*, pages 1637–1649.

[Bender et al., 2005] Bender, M. A., Farach-Colton, M., Pemmasani, G., Skiena, S., and Sumazin, P. (2005). Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94.

[Bernstein, 2016] Bernstein, A. (2016). Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574.

[Bernstein et al., 2020a] Bernstein, A., Gutenberg, M. P., and Saranurak, T. (2020a). Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1123–1134. IEEE.

[Bernstein et al., 2020b] Bernstein, A., Gutenberg, M. P., and Wulff-Nilsen, C. (2020b). Near-optimal decremental SSSP in dense weighted digraphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1112–1122. IEEE.

[Bhattacharya et al., 2019] Bhattacharya, S., Henzinger, M., and Nanongkai, D. (2019). A new deterministic algorithm for dynamic set cover. In Zuckerman, D., editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 406–423. IEEE Computer Society.

[Bhattacharya et al., 2023] Bhattacharya, S., Kiss, P., and Saranurak, T. (2023). Dynamic $(1+\epsilon)$-approximate matching size in truly sublinear update time. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1563–1588, Los Alamitos, CA, USA. IEEE Computer Society.

[Bhattacharya and Kulkarni, 2019] Bhattacharya, S. and Kulkarni, J. (2019). Deterministically maintaining a (2 + ϵ)-approximate minimum vertex cover in O($1/\epsilon^2$) amortized update time. In Chan, T. M., editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1872–1885. SIAM.

[Bhattacharya and Kulkarni, 2020] Bhattacharya, S. and Kulkarni, J. (2020). An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In Chawla, S., editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2509–2521. SIAM.

[Bille, 2005] Bille, P. (2005). A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1–3):217–239.

[Bojańczyk and Walukiewicz, 2008] Bojańczyk, M. and Walukiewicz, I. (2008). Forest algebras. In Flum, J., Grädel, E., and Wilke, T., editors, *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*, volume 2 of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press.

[Boroujeni et al., 2021] Boroujeni, M., Ehsani, S., Ghodsi, M., Hajiaghayi, M., and Seddighin, S. (2021). Approximating edit distance in truly subquadratic time: Quantum and mapreduce. *J. ACM*, 68(3):19:1–19:41.

[Boroujeni et al., 2018] Boroujeni, M., Ehsani, S., Ghodsi, M., Hajiaghayi, M. T., and Seddighin, S. (2018). Approximating edit distance in truly subquadratic time: Quantum and mapreduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1170–1189. SIAM.

[Boroujeni et al., 2019] Boroujeni, M., Ghodsi, M., Hajiaghayi, M., and Seddighin, S. (2019). $(1 + \epsilon)$-approximation of tree edit distance in quadratic time. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 709–720. ACM.

[Boroujeni and Seddighin, 2019] Boroujeni, M. and Seddighin, S. (2019). Improved mpc algorithms for edit distance and ulam distance. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 31–40.

[Brakensiek and Rubinstein, 2020] Brakensiek, J. and Rubinstein, A. (2020). Constant-factor approximation of near-linear edit distance in near-linear time. In *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 685–698. ACM.

[Bringmann, 2014] Bringmann, K. (2014). Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 661–670. IEEE Computer Society.

[Bringmann et al., 2022a] Bringmann, K., Cassis, A., Fischer, N., and Nakos, V. (2022a). Almost-optimal sublinear-time edit distance in the low distance regime. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1102–1115.

[Bringmann et al., 2022b] Bringmann, K., Cassis, A., Fischer, N., and Nakos, V. (2022b). Improved sublinear-time edit distance for preprocessed strings. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, pages 32:1–32:20.

[Bringmann et al., 2024] Bringmann, K., Fischer, N., van der Hoog, I., Kipouridis, E., Kociumaka, T., and Rotenberg, E. (2024). Dynamic dynamic time warping. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 208–242. SIAM.

[Bringmann et al., 2020] Bringmann, K., Gawrychowski, P., Mozes, S., and Weimann, O. (2020). Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). *ACM Trans. Algorithms*, 16(4).

[Bringmann et al., 2019] Bringmann, K., Grandoni, F., Saha, B., and Williams, V. V. (2019). Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. *SIAM J. Comput.*, 48(2):481–512.

[Bringmann and Künnemann, 2015] Bringmann, K. and Künnemann, M. (2015). Quadratic conditional lower bounds for string problems and dynamic time warping. In Guruswami, V., editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society.

[Buneman et al., 2003] Buneman, P., Grohe, M., and Koch, C. (2003). Path queries on compressed XML. In *29th International Conference on Very Large Data Bases, VLDB 2003*, page 141–152. Morgan Kaufmann.

[Bunke and Shearer, 1998] Bunke, H. and Shearer, K. (1998). A graph distance metric based on the maximal common subgraph. *Pattern Recogn. Lett.*, 19(3–4):255–259.

[Cassis et al., 2023] Cassis, A., Kociumaka, T., and Wellnitz, P. (2023). Optimal algorithms for bounded weighted edit distance. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2177–2187, Los Alamitos, CA, USA. IEEE Computer Society.

[Chakraborty et al., 2020] Chakraborty, D., Das, D., Goldenberg, E., Koucký, M., and Saks, M. E. (2020). Approximating edit distance within constant factor in truly sub-quadratic time. *Journal of the ACM*, 67(6):1–22.

[Chakraborty et al., 2016] Chakraborty, D., Goldenberg, E., and Koucký, M. (2016). Streaming algorithms for embedding and computing edit distance in the low distance regime. In Wichs, D. and Mansour, Y., editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 712–725. ACM.

[Chang and Zheng, 2024] Chang, Y.-J. and Zheng, D. W. (2024). Fully scalable massively parallel algorithms for embedded planar graphs. In *Proceedings of the Thirty-Five Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*. SIAM.

[Charalampopoulos et al., 2020a] Charalampopoulos, P., Gawrychowski, P., and Pokorski, K. (2020a). Dynamic longest common substring in polylogarithmic time. In Czumaj, A., Dawar, A., and Merelli, E., editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Charalampopoulos et al., 2020b] Charalampopoulos, P., Kociumaka, T., and Mozes, S. (2020b). Dynamic string alignment. In Gørtz, I. L. and Weimann, O., editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 9:1–9:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Charalampopoulos et al., 2020c] Charalampopoulos, P., Kociumaka, T., and Wellnitz, P. (2020c). Faster approximate pattern matching: A unified approach. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 978–989.

[Charikar et al., 2004] Charikar, M., Chekuri, C., Feder, T., and Motwani, R. (2004). Incremental clustering and dynamic information retrieval. *SIAM J. Comput.*, 33(6):1417–1440.

[Charikar et al., 2020] Charikar, M., Ma, W., and Tan, L.-Y. (2020). New lower bounds for massively parallel computation from query complexity. *ArXiv*, abs/2001.01146.

[Chawathe, 1999] Chawathe, S. S. (1999). Comparing hierarchical data in external memory. In *25th International Conference on Very Large Data Bases, VLDB 1999*, pages 90–101. Morgan Kaufmann.

[Chen et al., 2013a] Chen, A., Chu, T., and Pinsker, N. (2013a). Computing the longest increasing subsequence of a sequence subject to dynamic insertion. *CoRR*, abs/1309.7724.

[Chen et al., 2013b] Chen, A., Chu, T., and Pinsker, N. (2013b). The dynamic longest increasing subsequence problem. *arXiv preprint arXiv:1309.7724*.

[Chen et al., 2020] Chen, L., Goranci, G., Henzinger, M., Peng, R., and Saranurak, T. (2020). Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In Irani, S., editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1135–1146. IEEE.

[Cheng et al., 2021] Cheng, K., Farhadi, A., Hajiaghayi, M., Jin, Z., Li, X., Rubinstein, A., Seddighin, S., and Zheng, Y. (2021). Streaming and small space approximation algorithms for edit distance and longest common subsequence. In Bansal, N., Merelli, E., and Worrell, J., editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 54:1–54:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Chi et al., 2022a] Chi, S., Duan, R., and Xie, T. (2022a). Faster algorithms for bounded-difference min-plus product. In *33rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 1435–1447. SIAM.

[Chi et al., 2022b] Chi, S., Duan, R., Xie, T., and Zhang, T. (2022b). Faster min-plus product for monotone instances. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1529–1542.

[Chomsky and Schützenberger, 1963] Chomsky, N. and Schützenberger, M.-P. (1963). The algebraic theory of context-free languages. In Braffort, P. and Hirschberg, D., editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. Elsevier.

[Chumachenko et al., 2022] Chumachenko, K., Iosifidis, A., and Gabbouj, M. (2022). Weighted edit distance for country code recognition in license plates. In *30th European Signal Processing Conference, EUSIPCO 2022*, pages 1111–1115. IEEE.

[Chuzhoy et al., 2020] Chuzhoy, J., Gao, Y., Li, J., Nanongkai, D., Peng, R., and Saranurak, T. (2020). A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In Irani, S., editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1158–1167. IEEE.

[Clifford et al., 2022] Clifford, R., Gawrychowski, P., Kociumaka, T., Martin, D. P., and Uznanski, P. (2022). The dynamic k-mismatch problem. In Bannai, H. and Holub, J., editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Cole and Hariharan, 2002] Cole, R. and Hariharan, R. (2002). Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782.

[Czumaj et al., 2018] Czumaj, A., Lacki, J., Madry, A., Mitrovic, S., Onak, K., and Sankowski, P. (2018). Round compression for parallel matching algorithms. In *ACM SIGACT*, pages 471–484.

[Das et al., 2023] Das, D., Gilbert, J., Hajiaghayi, M., Kociumaka, T., and Saha, B. (2023). Weighted edit distance computation: Strings, trees, and dyck. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, page 377–390, New York, NY, USA. Association for Computing Machinery.

[Das et al., 2022a] Das, D., Gilbert, J., Hajiaghayi, M., Kociumaka, T., Saha, B., and Saleh, H. (2022a). $\tilde{O}$(n+poly(k))-time algorithm for bounded tree edit distance. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 686–697. IEEE.

[Das et al., 2022b] Das, D., Kociumaka, T., and Saha, B. (2022b). Improved approximation algorithms for Dyck edit distance and RNA folding. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022*, volume 229 of *LIPIcs*, pages 49:1–49:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Demaine et al., 2010] Demaine, E. D., Mozes, S., Rossman, B., and Weimann, O. (2010). An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1).

[Demetrescu and Italiano, 2004] Demetrescu, C. and Italiano, G. F. (2004). A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992.

[Dürr, 2023] Dürr, A. (2023). Improved bounds for rectangular monotone min-plus product and applications. *Inf. Process. Lett.*, 181(C).

[Farach and Muthukrishnan, 1996] Farach, M. and Muthukrishnan, S. (1996). Optimal logarithmic time randomized suffix tree construction. In auf der Heide, F. M. and Monien, B., editors, *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings*, volume 1099 of *Lecture Notes in Computer Science*, pages 550–561. Springer.

[Farach-Colton et al., 2000] Farach-Colton, M., Ferragina, P., and Muthukrishnan, S. (2000). On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011.

[Ferragina et al., 2009] Ferragina, P., Luccio, F., Manzini, G., and Muthukrishnan, S. (2009). Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1).

[Fine and Wilf, 1965] Fine, N. J. and Wilf, H. S. (1965). Uniqueness theorems for periodic functions. *Proc. Am. Math. Soc.*, 16(1):109–114.

[Fontan et al., 2016] Fontan, L., Ferrané, I., Farinas, J., Pinquier, J., and Aumont, X. (2016). Using phonologically weighted levenshtein distances for the prediction of microscopic intelligibility. In *17th Annual Conference of the International Speech Communication Association, Interspeech 2016*, pages 650–654. ISCA.

[Fried et al., 2022] Fried, D., Golan, S., Kociumaka, T., Kopelowitz, T., Porat, E., and Starikovskaya, T. (2022). An improved algorithm for the k-Dyck edit distance problem. In Naor, J. S. and Buchbinder, N., editors, *33rd Annnual ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 3650–3669. SIAM.

[Fried et al., 2024] Fried, D., Golan, S., Kociumaka, T., Kopelowitz, T., Porat, E., and Starikovskaya, T. (2024). An improved algorithm for the k-dyck edit distance problem. *ACM Transactions on Algorithms*, 20(3):26:1–25.

[Gall and Seddighin, 2020] Gall, F. L. and Seddighin, S. (2020). Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. *CoRR*, abs/2010.12122.

[Gao et al., 2020] Gao, Y., He, M., and Nekrich, Y. (2020). Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In *28th Annual European Symposium on Algorithms, ESA 2020*, volume 173 of *LIPIcs*, pages 54:1–54:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Gawrychowski, 2011] Gawrychowski, P. (2011). Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In *19th Annual European Symposium on Algorithms, ESA 2011*, volume 6942 of *LNCS*, pages 421–432. Springer.

[Gawrychowski and Janczewski, 2021] Gawrychowski, P. and Janczewski, W. (2021). Fully dynamic approximation of LIS in polylogarithmic time. In Khuller, S. and Williams, V. V., editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 654–667. ACM.

[Gawrychowski et al., 2018] Gawrychowski, P., Karczmarz, A., Kociumaka, T., Lacki, J., and Sankowski, P. (2018). Optimal dynamic strings. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1509–1528.

[Gerlach et al., 2021] Gerlach, A., Wiemerslage, A., and Kann, K. (2021). Paradigm clustering with weighted edit distance. In *18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 107–114. Association for Computational Linguistics.

[Ghaffari et al., 2020] Ghaffari, M., Grunau, C., and Jin, C. (2020). Improved mpc algorithms for mis, matching, and coloring on trees and beyond. *arXiv preprint arXiv:2002.09610*.

[Ghaffari et al., 2019] Ghaffari, M., Kuhn, F., and Uitto, J. (2019). Conditional hardness results for massively parallel computation from distributed lower bounds. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663.

[Ghaffari and Uitto, 2019] Ghaffari, M. and Uitto, J. (2019). Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *SODA*, pages 1636–1653. SIAM.

[Goldenberg et al., 2022] Goldenberg, E., Kociumaka, T., Krauthgamer, R., and Saha, B. (2022). Gap edit distance via non-adaptive queries: Simple and optimal. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 674–685. IEEE.

[Goldenberg et al., 2019] Goldenberg, E., Krauthgamer, R., and Saha, B. (2019). Sublinear algorithms for gap edit distance. In Zuckerman, D., editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1101–1120. IEEE Computer Society.

[Goldenberg et al., 2020] Goldenberg, E., Rubinstein, A., and Saha, B. (2020). Does preprocessing help in fast sequence comparisons? In *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 657–670.

[Goodrich et al., 2011] Goodrich, M. T., Sitchinava, N., and Zhang, Q. (2011). Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 374–383.

[Gorbachev and Kociumaka, 2024] Gorbachev, E. and Kociumaka, T. (2024). Bounded edit distance: Optimal static and dynamic algorithms for small integer weights. *CoRR*, abs/2404.06401.

[Gupta and Peng, 2013] Gupta, M. and Peng, R. (2013). Fully dynamic (1+ e)-approximate matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 548–557. IEEE Computer Society.

[Gusfield, 1997] Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, USA.

[Gutenberg et al., 2020] Gutenberg, M. P., Williams, V. V., and Wein, N. (2020). New algorithms and hardness for incremental single-source shortest paths in directed graphs. In Makarychev, K., Makarychev, Y., Tulsiani, M., Kamath, G., and Chuzhoy, J., editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 153–166. ACM.

[Haeupler et al., 2019] Haeupler, B., Rubinstein, A., and Shahrasbi, A. (2019). Near-linear time insertion-deletion codes and $(1+\epsilon)$-approximating edit distance via indexing. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, page 697–708. ACM.

[Hajiaghayi et al., 2021] Hajiaghayi, M., Saleh, H., Seddighin, S., and Sun, X. (2021). String matching with wildcards in the massively parallel computation model. In *SPAA*, pages 275–284.

[Hajiaghayi et al., 2019] Hajiaghayi, M., Seddighin, S., and Sun, X. (2019). Massively parallel approximation algorithms for edit distance and longest common subsequence. In Chan, T. M., editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1654–1672. SIAM.

[Hanauer et al., 2022] Hanauer, K., Henzinger, M., and Schulz, C. (2022). Recent advances in fully dynamic graph algorithms - A quick reference guide. *ACM J. Exp. Algorithmics*, 27:1.11:1–1.11:45.

[Harel and Tarjan, 1984] Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355.

[Harrison, 1978] Harrison, M. A. (1978). *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[Henzinger and Kale, 2020] Henzinger, M. and Kale, S. (2020). Fully-Dynamic Coresets. In Grandoni, F., Herman, G., and Sanders, P., editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 57:1–57:21, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Henzinger et al., 2016] Henzinger, M., Krinninger, S., and Nanongkai, D. (2016). Dynamic approximate all-pairs shortest paths: Breaking the $o(mn)$ barrier and derandomization. *SIAM Journal on Computing*, 45(3):947–1006.

[Hu et al., 2019] Hu, X., Yi, K., and Tao, Y. (2019). Output-optimal massively parallel algorithms for similarity joins. *ACM Trans. Database Syst.*, 44(2):6:1–6:36.

[Iliopoulos and Rytter, 2004] Iliopoulos, C. and Rytter, W. (2004). On parallel transformations of suffix arrays into suffix trees. In *Australasian Workshop on Combinatorial Algorithms (AWOCA)*, volume 11, pages 11–4.

[Im et al., 2023] Im, S., Kumar, R., Lattanzi, S., Moseley, B., and Vassilvitskii, S. (2023). Massively parallel computation: Algorithms and applications. *Found. Trends Optim.*, 5(4):340–417.

[Impagliazzo and Paturi, 2001] Impagliazzo, R. and Paturi, R. (2001). On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375.

[Impagliazzo et al., 2001] Impagliazzo, R., Paturi, R., and Zane, F. (2001). Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530.

[Indyk, 2001] Indyk, P. (2001). Algorithmic applications of low-distortion geometric embeddings. In *42nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2001*, page 10. IEEE.

[Ishida et al., 2005] Ishida, Y., Inenaga, S., Shinohara, A., and Takeda, M. (2005). Fully incremental lcs computation. In Liśkiewicz, M. and Reischuk, R., editors, *Fundamentals of Computation Theory*, pages 563–574, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Jurafsky and Martin, 2009] Jurafsky, D. and Martin, J. H. (2009). *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International.

[Karczmarz et al., 2022] Karczmarz, A., Mukherjee, A., and Sankowski, P. (2022). Subquadratic dynamic path reporting in directed graphs against an adaptive adversary. In Leonardi, S. and Gupta, A., editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1643–1656. ACM.

[Karloff et al., 2010] Karloff, H. J., Suri, S., and Vassilvitskii, S. (2010). A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948.

[Karp et al., 1972] Karp, R. M., Miller, R. E., and Rosenberg, A. L. (1972). Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 125–136.

[Karp and Rabin, 1987] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260.

[Kempa and Kociumaka, 2022] Kempa, D. and Kociumaka, T. (2022). Dynamic suffix array with polylogarithmic queries and updates. In Leonardi, S. and Gupta, A., editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1657–1670. ACM.

[Kim and Park, 2004] Kim, S.-R. and Park, K. (2004). A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312. Combinatiorial Pattern Matching.

[Klein, 1998] Klein, P. N. (1998). Computing the edit-distance between unrooted ordered trees. In *6th Annual European Symposium on Algorithms, ESA 1998*, volume 1461 of *LNCS*, pages 91–102. Springer.

[Kociumaka, 2018] Kociumaka, T. (2018). *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw.

[Kociumaka et al., 2023] Kociumaka, T., Mukherjee, A., and Saha, B. (2023). Approximating edit distance in the fully dynamic model. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1628–1638.

[Kociumaka et al., 2021] Kociumaka, T., Porat, E., and Starikovskaya, T. (2021). Small space and streaming pattern matching with k edits. In *62nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2021*, pages 885–896. IEEE.

[Kociumaka et al., 2015] Kociumaka, T., Radoszewski, J., Rytter, W., and Waleń, T. (2015). Internal pattern matching queries in a text and applications. In Indyk, P., editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM.

[Kociumaka and Saha, 2020] Kociumaka, T. and Saha, B. (2020). Sublinear-time algorithms for computing & embedding gap edit distance. In Irani, S., editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1168–1179. IEEE.

[Kociumaka and Seddighin, 2021] Kociumaka, T. and Seddighin, S. (2021). Improved dynamic algorithms for longest increasing subsequence. In Khuller, S. and Williams, V. V., editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 640–653. ACM.

[Koide et al., 2020] Koide, S., Xiao, C., and Ishikawa, Y. (2020). Fast subtrajectory similarity search in road networks under weighted edit distance constraints. *Proceedings of the VLDB Endowment*, 13(12):2188–2201.

[Koucký and Saks, 2020] Koucký, M. and Saks, M. E. (2020). Constant factor approximations to edit distance on far input pairs in nearly linear time. In *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 699–712. ACM.

[Koucký and Saks, 2023] Koucký, M. and Saks, M. E. (2023). Simple, deterministic, fast (but weak) approximations to edit distance and dyck edit distance. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 5203–5219.

[Kozen, 1997] Kozen, D. C. (1997). *Automata and Computability*. Springer New York.

[Kurtz, 1996] Kurtz, S. (1996). Approximate string searching under weighted edit distance. In *3rd South American Workshop on String Processing, WSP 1996*, pages 156–170. Carleton University Press.

[Kuszmaul, 2019] Kuszmaul, W. (2019). Dynamic time warping in strongly subquadratic time: Algorithms for the low-distance regime and approximate evaluation. In Baier, C., Chatzigiannakis, I., Flocchini, P., and Leonardi, S., editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019*, volume 132 of *LIPIcs*, pages 80:1–80:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Landau et al., 1998] Landau, G. M., Myers, E. W., and Schmidt, J. P. (1998). Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582.

[Landau and Vishkin, 1988] Landau, G. M. and Vishkin, U. (1988). Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78.

[Landau and Vishkin, 1989] Landau, G. M. and Vishkin, U. (1989). Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169.

[Levenshtein, 1965] Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710.

[Lu and Lin, 1994] Lu, M. and Lin, H. (1994). Parallel algorithms for the longest common subsequence problem. *IEEE Trans. Parallel Distributed Syst.*, 5(8):835–848.

[Manber and Myers, 1993] Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948.

[Mao, 2021] Mao, X. (2021). Breaking the cubic barrier for (unweighted) tree edit distance. In *62nd Annual IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 792–803. IEEE.

[McCreight, 1976] McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272.

[Mehlhorn et al., 1997] Mehlhorn, K., Sundar, R., and Uhrig, C. (1997). Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17:183–198.

[Mitzenmacher and Seddighin, 2020] Mitzenmacher, M. and Seddighin, S. (2020). Dynamic algorithms for LIS and distance to monotonicity. In Makarychev, K., Makarychev, Y., Tulsiani, M., Kamath, G., and Chuzhoy, J., editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 671–684. ACM.

[MongoDB, 2022] MongoDB (2022). MongoDB JSON schema examples tutorial. `https://www.mongodb.com/basics/json-schema-examples`. Accessed: 2022-03-03.

[Mortensen, 2006] Mortensen, C. W. (2006). Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525.

[Myers, 1986] Myers, E. W. (1986). An $o(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266.

[Myers, 1995] Myers, G. (1995). Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92.

[Nanongkai and Saranurak, 2017] Nanongkai, D. and Saranurak, T. (2017). Dynamic spanning forest with worst-case update time: adaptive, las vegas, and O($n^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129. ACM.

[Nanongkai et al., 2017] Nanongkai, D., Saranurak, T., and Wulff-Nilsen, C. (2017). Dynamic minimum spanning forest with subpolynomial worst-case update time. In Umans, C., editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 950–961. IEEE Computer Society.

[Nanongkai and Scquizzato, 2020] Nanongkai, D. and Scquizzato, M. (2020). Equivalence classes and conditional hardness in massively parallel computations. *CoRR*, abs/2001.02191.

[Navarro and Sadakane, 2014] Navarro, G. and Sadakane, K. (2014). Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3).

[Needleman and Wunsch, 1970] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453.

[Nishimoto et al., 2020] Nishimoto, T., I, T., Inenaga, S., Bannai, H., and Takeda, M. (2020). Dynamic index and lz factorization in compressed space. *Discrete Applied Mathematics*, 274:116–129. Stringology Algorithms.

[Nussinov and Jacobson, 1980] Nussinov, R. and Jacobson, A. B. (1980). Fast algorithm for predicting the secondary structure of single-stranded rna. *Proceedings of the National Academy of Sciences*, 77(11):6309–6313.

[Paige and Tarjan, 1987] Paige, R. and Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989.

[Peris and Marzal, 2002] Peris, G. and Marzal, A. (2002). Fast cyclic edit distance computation with weighted edit costs in classification. In *16th International Conference on Pattern Recognition, ICPR 2002*, pages 184–187. IEEE, IEEE Computer Society.

[Pettie and Ramachandran, 2005] Pettie, S. and Ramachandran, V. (2005). A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing*, 34(6):1398–1431.

[Rastogi et al., 2013] Rastogi, V., Machanavajjhala, A., Chitnis, L., and Das Sarma, A. (2013). Finding connected components in map-reduce in logarithmic rounds. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 50–61.

[Roditty and Williams, 2013] Roditty, L. and Williams, V. V. (2013). Fast approximation algorithms for the diameter and radius of sparse graphs. In Boneh, D., Roughgarden, T., and Feigenbaum, J., editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 515–524. ACM.

[Roditty and Zwick, 2016] Roditty, L. and Zwick, U. (2016). A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733.

[Roughgarden et al., 2018] Roughgarden, T., Vassilvitskii, S., and Wang, J. R. (2018). Shuffles and circuits (on lower bounds for modern parallel computation). *J. ACM*, 65(6).

[Saha, 2014] Saha, B. (2014). The Dyck language edit distance problem in near-linear time. In *55th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2014*, pages 611–620. IEEE Computer Society.

[Saha, 2015] Saha, B. (2015). Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 118–135.

[Saha, 2017] Saha, B. (2017). Fast & space-efficient approximations of language edit distance and rna folding: An amnesic dynamic programming approach. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 295–306. IEEE.

[Sankowski, 2004] Sankowski, P. (2004). Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 509–517. IEEE Computer Society.

[Sankowski, 2007] Sankowski, P. (2007). Faster dynamic matchings and vertex connectivity. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 118–126. SIAM.

[Seddighin and Seddighin, 2022] Seddighin, M. and Seddighin, S. (2022). $3 + \epsilon$ approximation of tree edit distance in truly subquadratic time. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022*, volume 215 of *LIPIcs*, pages 115:1–115:22. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Selkow, 1977] Selkow, S. M. (1977). The tree-to-tree editing problem. *Inform. Process. Lett.*, 6(6):184–186.

[Sellers, 1974] Sellers, P. H. (1974). On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793.

[Shapiro and Zhang, 1990] Shapiro, B. A. and Zhang, K. (1990). Comparing multiple RNA secondary structures using tree comparisons. *Comput. Appl. Biosci.*, 6(4):309–318.

[Skiena, 2020] Skiena, S. (2020). *The Algorithm Design Manual, Third Edition*. Texts in Computer Science. Springer.

[Solomon, 2016] Solomon, S. (2016). Fully dynamic maximal matching in constant update time. In Dinur, I., editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 325–334. IEEE Computer Society.

[Tai, 1979] Tai, K.-C. (1979). The tree-to-tree correction problem. *J. ACM*, 26(3):422–433.

[Tiskin, 2008] Tiskin, A. (2008). Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603.

[Tiskin, 2010] Tiskin, A. (2010). Fast distance multiplication of unit-monge matrices. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, page 1287–1296, USA.

[Touzet, 2005] Touzet, H. (2005). A linear tree edit distance algorithm for similar ordered trees. In *16th Annual Conference on Combinatorial Pattern Matching, CPM 2005*, page 334–345. Springer.

[Ukkonen, 1985] Ukkonen, E. (1985). Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118.

[Ukkonen, 1995] Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

[van den Brand et al., 2023] van den Brand, J., Liu, Y. P., and Sidford, A. (2023). Dynamic maxflow via dynamic interior point methods. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, page 1215–1228, New York, NY, USA. Association for Computing Machinery.

[van den Brand and Nanongkai, 2019] van den Brand, J. and Nanongkai, D. (2019). Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 436–455. IEEE Computer Society.

[van den Brand et al., 2019] van den Brand, J., Nanongkai, D., and Saranurak, T. (2019). Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In Zuckerman, D., editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480. IEEE Computer Society.

[Vintsyuk, 1968] Vintsyuk, T. K. (1968). Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57.

[Wagner and Fischer, 1974] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

[Weiner, 1973] Weiner, P. (1973). Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE.

[Williams, 2005] Williams, R. (2005). A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357–365. Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004).

[Williams and Xu, 2020] Williams, V. V. and Xu, Y. (2020). Truly subcubic min-plus product for less structured matrices, with applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–29. SIAM.

[Zhang and Shasha, 1989] Zhang, K. and Shasha, D. E. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262.