Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Range Counting

(Technical Report CS-TR-4533 and UMIACS-TR-2003-101)

Qingmin Shi* Joseph F. JaJa† Christian W. Mortensen‡

Abstract

We present linear-space sublogarithmic algorithms for handling the *three-dimensional dominance reporting problem* and the *two-dimensional range counting problem*. Under the RAM model as described in [M. L. Fredman and D. E. Willard. "Surpassing the information theoretic bound with fusion trees", *Journal of Computer and System Sciences*, 47:424–436, 1993], our algorithms achieve $O(\log n / \log\log n + f)$ query time for 3-D dominance reporting, where f is the number of points reported, and $O(\log n / \log\log n)$ query time for 2-D range counting case. We extend these results to any constant dimension d achieving $O(n(\log n / \log\log n)^{d-3})$-space and $O((\log n / \log\log)^{d-2} + f)$-query time for the reporting case and $O(n(\log n / \log\log n)^{d-2})$-space and $O((\log n / \log\log n)^{d-1})$ query time for the counting case.

1 Introduction

Given a set S of d-dimensional points (d is assumed to be a constant), we wish to store these points in a data structure so that, given a query point q, the points in S that *dominate* q, which we will refer to as *proper* points, can be reported or counted quickly. A point $p = (p_1, p_2, \ldots, p_d)$ is S dominates $q = (q_1, q_2, \ldots, q_d)$ if and only if $p_i \geq q_i$ for all $i = 1, \ldots, d$. Without loss of generality, we assume that no two points in S have the same x-, y-, or z-coordinates. A number of geometric retrieval problems involving iso-oriented objects can be reduced to this problem (see for example [6]). Solutions to this problem have also been used recently in dealing with the so-called *temporal range queries* on time-series data [13].

Throughout this paper, we will use n to represent the input size, f to represent the output size (for the reporting case), and ϵ to represent an arbitrarily small positive constant. We define c as $\log^\epsilon n$[1]. Given a set S of d-dimensional points $(x_1, x_2, \ldots, x_d)$, a point in S with the largest x_i-coordinate smaller than or equal to a number α is called the *x_i-predecessor* of α and the one with the smallest x_i-coordinate larger than or equal to α is called the *x_i-successor* of α. The *x_i-rank* of a number α (with respect to S) is defined as the number of points in S whose x_i-coordinates are smaller than or equal to α. The *x_i-rank* of a d-dimensional point is defined as the x_i-rank of its x_i-coordinate. Let $i \leq j$ be two integers, we will use $[i..j]$ to denote the set of integers $\{i, i+1, \ldots, j\}$

*Institute of Advanced Computer Studies, University of Maryland, College Park, MD 20742. Email: qshi@umiacs.umd.edu

†Institute of Advanced Computer Studies, University of Maryland, College Park, MD 20742. Email: joseph@umiacs.umd.edu

‡IT University of Copenhagen, Email: cworm@itu.dk

[1] In this paper, we always assume that the logarithmic operations are to the base two.

and $[i..j]^k$ to denote $\underbrace{[i..j] \times \cdots \times [i..j]}_{k}$. Given a tree, we will assign a *label* to each node v, which is equal to the number of siblings of v to its left.

Our model of computation is the RAM model as described in [8], in which it is assumed that each word consists of w bits and the size of the data set never exceeds 2^w, i.e. $w \geq \log n$. In addition, arithmetic and bitwise logical operations take constant time. We assume each coordinate of a point is given as an integer in a single word.

In [4], Chazelle and Edelsbrunner proposed two linear-space algorithms to handle the 3-D dominance reporting problem. The first achieves $O(\log n + f \log n)$ query time and the second achieves $O(\log^2 n + f)$ query time. These two algorithms were later improved by Makris and Tsakalidis [9] to yield $O((\log\log n)^2 \log\log\log n + f \log\log n)$ and $O(\log n + f)$ query time respectively. The previous best linear-space algorithm for the 2-D dominance counting problem is due to Chazelle [3], which achieves $O(\log n)$ query time.

In [12], Shi and JaJa improved the query performance of the $O(\log n + f)$ time algorithm of Makris and Tsakalidis for the reporting case to $O(\log n/\log\log n + f)$ query time and that of the $O(\log n)$ time algorithm of Chazelle for the counting case to $O(\log n/\log\log n)$ query time, but at the expense of increasing the storage costs in both cases by a factor of $\log^\epsilon n$.

In this paper, we show how to reduce the storage cost required for handling the 3-D reporting and the 2-D counting problems while maintaining the same query performance as in [12], thus obtaining the fastest query time algorithms using linear space. In fact, our 3-D reporting algorithm solves a more general "layered" 3-D reporting in linear space and $O(\log n/\log\log n + f)$ query time. Since answering a general range counting query is equivalent to answering a constant number of dominance counting queries, our results for the dominance counting problems is valid for the general range counting problems as well. We also extend these results to the high dimensional case. The results of this paper are summarized as the following two theorems.

Theorem 1.1. *For any constant $d \geq 3$ there exist data structures such that any d-dimensional dominance reporting query can be handled in $O((\log n/\log\log n)^{d-2}+f)$ time using $O(n(\log n/\log\log n)^{d-3})$ space.*

Theorem 1.2. *For any constant $d \geq 2$ there exist data structures such that any d-dimensional range counting query can be handled in $O((\log n/\log\log n)^{d-1})$ time using $O(n(\log n/\log\log n)^{d-2})$ space.*

In Section 2, we summarize some previously known results that will be heavily used in this paper. We briefly review the non-linear space algorithms (that appeared in [12]) in Section 3 and describe the new linear-space algorithm for the 3-D reporting case in Section 4. In Section 5, we give a linear-space solution to the more general layered 3-D dominance reporting problem and we extend it to higher dimensions in Section 6. The 2-D counting case and its extensions to higher dimensions are discussed in Sections 7 and 8 respectively.

2 Preliminaries

2.1 Q-heaps and Fusion Trees

Q-heap and *fusion trees* achieve sublogarithmic search time on one-dimensional data. The following two lemmas are shown in [7] and [8] respectively.

Lemma 2.1. *Assume that in a database of n elements, we have available the use of precomputed tables of size $o(n)$. Then it is possible to construct a fusion tree data structure of size $O(n)$ space,* *which has a worst-case $O(\log n/\log\log n)$ time for performing member, predecessor and rank operations.*

Lemma 2.2. *Suppose Q is a subset with cardinality $m < \log^{1/5} n$ lying in a larger database S consisting of n elements. Then three exists a Q-heap data structure of size $O(m)$ that enables insertion, deletion, member, and predecessor queries on Q to run in constant worst-case time, provided access is available to a precomputed table of size $o(n)$.*

Note that in Lemma 2.2, the look-up table of size $o(n)$ is shared among all the Q-heaps built on subsets of S.

2.2 Fast Fractional Cascading

Let T be a tree rooted at w and having a maximum degree of c at each node. A node v in T contains a sorted list $L(v)$ of elements. The total number of elements in all the lists is n. Such a tree is called a *catalog tree* [2]. An *iterative search* on T is defined as follows.

> Given a query item x, and an embedded tree F of T, which is rooted at w and has p nodes, find the predecessor of x in each of the lists associated with the nodes of F.

The fractional cascading technique [5] can be used to organize T and its associated lists so that an iterative search can be performed in $O(t(n) + p\log c)$ time, in which $t(n)$ is the time it takes to identify the predecessor of x in $L(w)$ and $\log c$ is the cost of finding the predecessor of x in each of the remaining $p-1$ lists. This technique uses $O(n)$ space. Note that when c is not a constant, the time spent at each node is not a constant either.

In [12], we combined the Q-heap technique and the fractional cascading and obtained the *fast fractional cascading* structure, which achieves constant search time at each node when the maximum degree of T is polylogarithmic in n. The storage cost, however, is increased to $n\log^{\epsilon} n$. This result was improved in [14] by reducing the storage cost to linear while maintaining the same query performance. This result is summarized in the following lemma.

Lemma 2.3. *Let T be a catalog tree rooted at w with a total number n of items in its associated lists, and let $c = \log^{\epsilon} n$ be the maximum degree of a node in T. There exists a $O(n)$-space data structure derived from T such that an iterative search operation specified by a query item x and an embedded tree F with p nodes can be performed in $O(t(n) + p)$ time, where $t(n)$ is the time it takes to identify the predecessor of x in $L(w)$.*

2.3 Handling 3-Sided 2-D Reporting Queries Using Cartesian Trees

A Cartesian tree [15] C is a binary tree defined on a finite set of 2-D points, say $p_1, p_2, \ldots, p_n$, sorted by their x-coordinates. The root of this tree is associated with the point p_i with the largest y-coordinate. Its left child is the root of the Cartesian tree built on $p_1, \ldots, p_{i-1}$, and its right child is the root of the Cartesian tree built on $p_{i+1}, \ldots, p_n$.

In [12], we explained how the Cartesian trees can be modified to efficiently handle the 3-sided 2-D reporting queries, i.e. to identify the points (x, y) that satisfy $a \le x \le b$ and $y \ge c$, where a, b, and c are three numbers provided by the query, and showed the following Lemma.

Lemma 2.4. *Let C be the modified Cartesian tree that corresponds to a set of n 2-D points. A 3-sided 2-D range query given as (a, b, d) can be handled in $O(t(n) + f)$ time using $\mathcal{D}(C)$ of size $O(n)$, where $t(n)$ is the time to identify the nodes corresponding to the successor of a and the predecessor of b, and $\mathcal{D}(C)$ is a transformation of C to support the nearest common ancestor search in constant time.*

In the rest of this paper, whenever we refer to a Cartesian tree, we mean its transformation that is suitable for 3-sided 2-D range reporting queries. We can also use the Cartesian tree to index a set of d-dimensional points $(x_1, x_2, \ldots, x_d)$ based on any two of their dimensions. If the points are first sorted by the x_i-dimension and are recursively picked during the construction of the tree according to their x_j-coordinates, then the resulting Cartesian tree is called an (x_i, x_j)-Cartesian tree.

3 Previous $O(\log / \log\log n + f)$-Query Time Algorithms for 3-D Dominance Reporting and 2-D Dominance Counting

In this section, we briefly review the data structures proposed in [12], upon which our new algorithms are based.

3.1 An $O(n \log^{\epsilon} n)$-Space and $O(\log / \log\log n + f)$-Query Time Algorithm for 3-D Dominance Reporting

The skeleton of the data structure is a balanced search tree of degree $c = \log^{\epsilon} n$ (thus of height $O(\log n / \log\log n)$) built on the points in S sorted by *decreasing* z-coordinates. A Q-heap $K(v)$ is used to index the keys stored at each internal node v. Let $M(v)$ be the *maximal set* of the points stored in the subtree rooted at v, excluding the points that are already associated with the ancestors of v (the maximal set of a point set R is the set of points $p \in R$ such that for all $p' \in R$ where $p \neq p'$ the projection of p onto the x-y plane is not dominated by the projection of p' onto the x-y plane). In addition to the Q-heap, each node v is associated with several Cartesian trees: an (x,z)-Cartesian tree $D(v)$ and c (x,y)-Cartesian trees $D_1(v), D_2(v), \ldots, D_c(v)$. The (x,z)-Cartesian tree $D(v)$ stores the *maximal set* $M(v)$ of v; and $D_i(v)$ stores the union of the maximal sets associated with the leftmost i children of v. It is easy to see that the storage cost of this data structure is $O(n \log^{\epsilon} n)$, since the tree T and the associated Q-heaps requires $O(n)$ space, and each point is stored in at most one (x, z)-Cartesian tree and c (x, y)-Cartesian trees.

To answer a 3-D dominance query specified by the point (q_1, q_2, q_3), we first identify, in $O(\log n / \log\log n)$ time using the Q-heaps, the path Π from the root to the leaf that corresponds to the z-successor of q_3. We then search the tree recursively, starting from the root. Note that we do not visit any node that is in a subtree rooted at the right sibling of a node on Π. For each node v visited, finding the $f(v)$ proper points in $M(v)$ is equivalent to a 3-sided 2-D range query due to the properties of a maximal set (see [9] for more details) and thus can be handled in $O(f(v))$ time using $D(v)$ (assuming that we already know the leftmost and rightmost leaf nodes of $D(v)$ that are in the query range). Suppose the kth child of v from the left is on Π ($k = c + 1$ if v is not on Π). Note that we cannot afford to visit the each of the leftmost $k - 1$ children of v. Instead, we visit such a *proper* child u only if there is at least one proper point in $M(u)$. We do so by searching $D_{k-1}(v)$ for proper points and mark the children of v they come from. Since each point reported from $D_{k-1}(v)$ will also be reported at a child w of v, each point may be reported twice.

We build a fusion tree on the x-coordinates to index the points stored in each of the $c + 1$ Cartesian trees associated with the root w. Furthermore, we connect all the Cartesian trees using a modified fractional cascading structure of size $O(n \log^{\epsilon} n)$ (see [12] for more details). These additional data structures do not asymptotically increase the storage cost and allow each Cartesian tree associated with a non-root node to be searched in constant time, plus the time it takes to retrieve proper points (Lemma 2.4).

3.2 An $O(n \log^{\epsilon} n)$-Space and $O(\log n/\log\log n)$-Query Time Algorithm for 2-D Dominance Counting

As in the 3-D dominance reporting case, we use a balanced tree T of degree c (thus of height $h = O(\log n/\log\log n)$) as the skeleton of our data structure for handling the 2-D dominance counting query. The tree T is built on the x-ranks of the points in S sorted in decreasing order. Each internal node v of T is associated with two-secondary structures: a *router* $r(v)$ and a *counter* $c(v)$. The router $r(v)$ stores, for each point p in the subtree rooted at v, the label of the child of v to whose subtree p belongs. These labels are sorted in order of the *decreasing* y-coordinates of the corresponding points. Let $n(v)$ be the number of points stored in the subtree rooted at v. The counter $c(v)$ is a two-dimensional array of size $(m(v)-1)\times c$, where $m(v) = \lceil n(v)\log c/\log n\rceil$. The item $c(v)[i][j]$ stores the number of labels among those of the first $i\log n/\log c$ points in $r(v)$ which are smaller than or equal to j. In addition to the tree T, we have two fusion trees built respectively on the increasing x- and y-coordinates of the points in S.

Since each entry of a router is a number in $[1..c]$, it can be encoded using $\log c$ bits. Thus the space required for maintaining all the routers is $O(n)$ words. Each item in a counter occupies a single word, and therefore the space used to stored all the counters is $O(cn)$ words.

To answer a query q given as (q_1, q_2), we first replace q_1 and q_2 with their respective x- and y-ranks r_1 and r_2 with respect to the points in S using the two fusion trees. We then in $O(\log n/\log\log n)$ time identify the path Π from the root of T to the leaf node that corresponds to the y-successor of r_2. For each node v on Π, suppose the label of its child u which is also on the path is j. We can compute in constant time the number of points stored in the subtrees rooted at the leftmost $j-1$ subtrees by looking up an appropriate entry in the counter $c(v)$ and a global table of size $O(n)$ that is shared by the search processes at all the nodes on Π. Details can be found in [12].

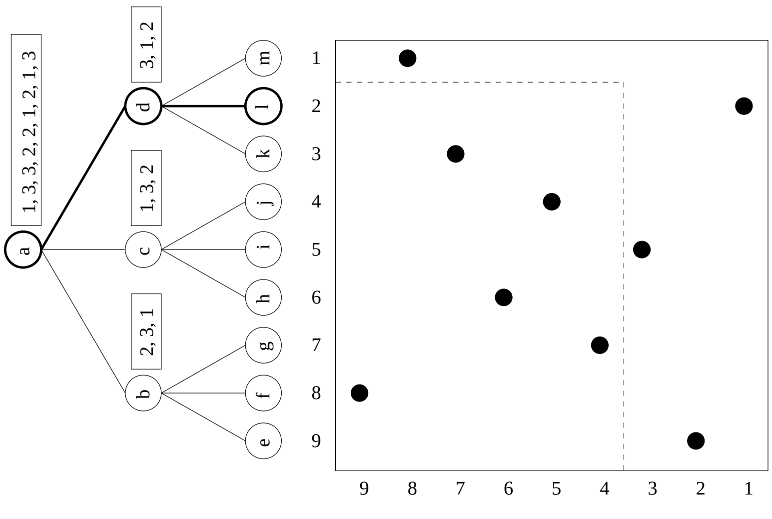

Figure 1: An example for the 2-D dominance counting query where $d = 2$, $n = 9$, $h = 2$, and $c = 3$.

Figure 1 gives an example on how such a 2-D dominance counting is handled. Consider the query $(2, 4)$. We calculate the answer to this query along the path from the the root to the leaf node l, which corresponds to the x-coordinate 2. At root a, since its child d is also on the path, we compute the number of points contributed by the subtree rooted at b and c by counting among the first 6 entries of $r(a)$ the number of entries whose value is in $[1..2]$; and we get 4. Since only two of these 6 entries have value 3, we continue to visit node d, which is also on the path, and count among the first 2 entries of $r(d)$ the number of entries whose value is in [1..1], which is 1. We continue to search node l and count zero point there. Hence the answer to the query is $4+1+0=5$.

4 A Linear-Space Algorithm with $O(\log n/\log\log n+f)$ Query Time for 3-D Dominance Reporting

In this section, we improve upon the result in Section 3.1 to achieve $O(\log n/\log\log n+f)$ query time and linear space. Two factors contributed to the non-linear space requirement of the data structure described there. First, the fractional cascading technique uses non-linear space. Second, with each node v, c (x,y)-Cartesian trees were needed to ensure that we can in constant time find the proper children. The first difficulty no longer exists since we now have at hand the fast fractional cascading structure [14]. The rest of this section is devoted to overcoming the second difficulty.

Let $S(v)$ be the union of the maximal sets associated with the children of v. We associate with each point $p=(p_1,p_2,p_3)$ in $S(v)$ a *layer* $p.l$, which is the label of the child of v from whose maximal set that point comes. To achieve linear space for the 3-D dominance reporting problem, it is sufficient to solve the following problem.

Problem 4.1. *Build a data structure for $S(v)$ such that, given a query (x_1,x_2,h), where $h\in[1..c]$, the $k(v)$ layers, from each of which there is at least one proper point p, which satisfies $p_1\geq x_1, p_2\geq x_2$, and $p.l\leq h$, can be identified in $O(k(v))$ time.*

We now discuss how to handle this problem using only $O(|S(v)|)$ space. Let $u_1,u_2,\ldots,u_c$ be the children of v in T and let $M(u_i)=\{(x_{i,1},y_{i,1},z_{i,1}),(x_{i,2},y_{i,2},z_{i,2}),\ldots,(x_{i,n_i},y_{i,n_i},z_{i,n_i})\}$ for $i=1,2,\ldots,c$. Since $M(u_i)$ is maximal, we can assume without loss of generality that $x_{i,1}<x_{i,2}<\cdots<x_{i,n_i}$ and $y_{i,1}>y_{i,2}>\cdots>y_{i,n_i}>y_{i,n_i+1}=-\infty$. Now consider the projections of these points to the x-y plane. We define a set $G(v)$ of vertical segments in the x-y plane as follows: $G_i(v)=\{(x_{i,j};y_{i,j+1},y_{i,j})|j=1,\ldots,n_i\}$ and $G(v)=\bigcup_{i=1,\ldots,c}G_i(v)$. We say the segments in $G_i(v)$ are from layer i. Figure 4 gives an example of such a set of segments, with segments from different children depicted using lines of different thicknesses. It is obvious that $|S(v)|=|G(v)|$. We denote $|G(v)|$ as $N(v)$.

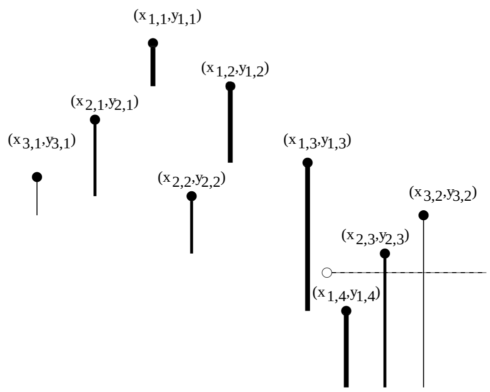

Figure 2: The dominance query and orthogonal segment intersection.

Lemma 4.1. *Let $s=(x_0,+\infty;y_0)$ be a semi-infinite horizontal line. For each $i\in\{1,\ldots,c\}$, u_i contains at least one point whose projection to the x-y plane dominates (x_0,y_0) if and only if there* *exists a vertical segment* $(x_{i,j}; y_{i,j+1}, y_{i,j}) \in G(v)$ *that intersects* s *and furthermore, there is at most one such vertical segment.*

Proof. If a segment $(x_{i,j}; y_{i,j+1}, y_{i,j}) \in G(v)$ intersects s, then by definition, $(x_{i,j}, y_{i,j})$ dominates (x_0, y_0). On the other hand, suppose $(x_{i,j}, y_{i,j})$ dominates (x_0, y_0). Then either the segment $(x_{i,j}; y_{i,j+1}, y_{i,j})$ intersects s, or $y_0 < y_{j+1} \neq -\infty$, which means $j < n_i$. Therefore, the point $(x_{i,j+1}, y_{j+1})$ also dominates (x_0, y_0). Repeating this process ensures that we can find one vertical segment corresponding to u_i which intersects s. Finally, since the projections of the vertical segments corresponding to u_i do not overlap, s can only intersect one of these segments. $\square$

As a result of Lemma 4.1, identifying the proper children of v in $O(k(v))$ time can be achieved if we design an indexing scheme on $G(v)$, such that given s and an integer h, the segments from $G_i(v)$, with $i = 1, 2, \ldots, h$, which intersect s can be reported in $O(k(v))$ time.

Note that this is not simply a segment intersection reporting problem, as we cannot afford to report every segment in $G(v)$ that intersects s. This is the case because some of them may come from other layers than those in $[1..h]$. Nevertheless, we can solve this problem by performing a segment intersection *counting* query followed by a table look-up operation. This segment intersection counting query is defined as computing the number of segments in $G(v)$ that intersect s (these segments are not necessarily from the layers in $[1..h]$).

We first discuss the table look-up operation. The list of proper children of v with respect to a 3-D dominance reporting query can be represented as a vector $r = (k(v), I_1, I_2, \ldots, I_{k(v)})$, where I_i, with $i = 1, \ldots, k(v)$, is the index of a proper child. Obviously, the bit-cost of this vector is $O(c \log c)$. Once we obtain such a vector, we can retrieve from it the index of the proper children one by one in $O(k(v))$ time.

Lemma 4.2. *The vector* r *is uniquely defined by the y-rank* g *of* y_0 *in the set of endpoints of* $G(v)$*, the value of* h*, and the number* k *of segments in* $G(v)$ *which intersect* s*.*

Proof. Let $y_1, y_2, \ldots, y_{N(v)}$ be the list of y-coordinates of the points in $G(v)$ in sorted order. Consider two consecutive such y-coordinates y_j and y_{j+1}. It is easy to see that the list of segments in $G(v)$ sorted from left to right which intersect the query segment $s = (x_0, +\infty; y_0)$, with y_0 varying in the range $[y_j, y_{j+1})$, remains the same. Among these segments, the rightmost k intersect s. And, knowing h, we can uniquely remove those coming from the rightmost $c - h$ children of v. $\square$

Since only one segment from $G_i(v)$ could possibly intersect s, the value of k is bounded by c. The value of h is also bounded by c and the y-rank of y_0 is bounded by $N(v)$. Therefore, we can create a look-up table containing $N(v)$ words, each corresponding to a possible y-rank of y_0. The $\log n$ bits of each such word is sufficient to record for each possible combination of k and h, the vector r that has been uniquely determined ($c^3 \log c < \log n$ for large enough n).

Among the three indices g, h, and k, g can be computed in constant time by applying the fast fractional cascading technique on the y-coordinates of the points in $\bigcup_{i=1,\ldots,c} M(u_i)$, and h is known using the Q-heap associated with v. Thus we only need to show that the value of k can be computed in constant time. We first give the following lemma.

Lemma 4.3. *A 3-D dominance counting query on a set* R *of* $m < \log^{\epsilon} n$ *points can be handled in constant time using* $O(m)$ *space.*

Proof. An answer to such a query is uniquely decided by the ranks of the query point in R with respect to the x-, y-, and z-coordinates, which can be computed by applying the Q-heap techniques in constant time and $O(m)$ space. These three ranks are used to index a $m \times m \times m$ look-up table to obtain the correct answer. Since $m < \log^{1/5} n$, any possible answer can be represented using only O(log log n) bits. Therefore all m^3 (not necessarily distinct) possible answers can be compacted into a single word ($m^3 \log\log n < \log n$ for large enough n). □

We now explain how to compute the value of k in constant time. We partition the endpoints of the segments in $G(v)$ into $N(v)/c$ horizontal stripes $P_1, \ldots, P_{N(v)/c}$, each containing c endpoints. Let $B_1, \ldots, B_{N(v)/c-1}$ be the boundaries such that B_i separates P_i and P_{i+1}. We associate with each boundary the maximal subset S_i of $G(v)$ such that every segment in S_i intersects B_i, and with each stripe P_i the maximal subset T_i of $G(v)$ such that every segment in T_i crosses the entire stripe P_i. We also denote the subset of segments in $G(v)$ that are completely inside P_i as R_i. Note that a segment can belong to up to $N(v)/c - 1$ subsets associated with the boundaries and up to $N(v)/c - 2$ subsets associated with the stripes. However, the size of each S_i or T_i is bounded by c. The total size of all the subsets associated with the boundaries is equal to the number of intersections between the segments in $G(v)$ and the $N(v)/c-1$ boundaries. Notice that each $G_i(v)$, with $i = 1, \ldots, c$, contributes at most $N(v)/c - 1$ such intersections. Thus the total size of all the subsets associated with the boundaries is $O(N(v))$. Similarly, the total size of all the subsets associated with the stripes is also $O(N(v))$. And finally, $\sum_{i=1,\ldots,N(v)/c} |R_i| = O(N(v))$.

Given a query segment $s = (x_0, +\infty; y_0)$, we can determine, using the fast fractional cascading structure, the stripe P_{j+1} within which it falls. Without loss of generality, suppose this is not the first nor the last stripe, and hence the two boundaries B_j and B_{j+1} exist. The number of segments in $G(v)$ that intersect s can be computed as $A + B - C + D$, where A and B are respectively the numbers of segments in S_j and S_{j+1} that intersect s, C is the number of segments in T_j that intersect s, and D is the number of segments in R_i that intersect s. Computing A and B is equivalent to a 2-D dominance counting query on the lower endpoints of the segments in S_j and the upper endpoints of the segments in S_{j+1} respectively; computing C is equivalent to a 1-D dominance counting query on the x-coordinates of the segments in T_{j+1}; and computing D is equivalent to a 3-D dominance counting query on the segments $(x; y_1, y_2)$ in R_{j+1} in the form $(x \geq x_0, y_1 \leq y_0, y_2 \geq y_0)$. Since the size of each of the sets involved is bounded by $O(\log^\epsilon n)$, by Lemma 4.3, these computation can be performed in $O(1)$ time and in linear space.

Theorem 4.1. *There exists data structures such that any three-dimensional dominance reporting query can be handled in $O(\log n/\log\log n + f)$ time using $O(n)$ space.*

5 Layered 3-D Dominance Reporting

In this section, we present an algorithm for a more general *layered* 3-D dominance reporting problem, whose solution enables an efficient extension of the results to higher dimensions. Within this setting, each of the n 3-D point p in S is assigned a *layer* $p.l$, in addition to its coordinates. Let $\mathcal{P}([1..c])$ denote the power set of $[1..c]$. Given a query as (q_1, q_2, q_3, L), where $L \in \mathcal{P}([1..c])$, a point p is *proper* if and only if p dominates q and, in addition, $p.l \in L$. By assigning all the points in S to the same layer and letting L contain only that layer, we obtain the standard 3-D dominance reporting problem. In sections 5.1 and 5.2, we discuss the algorithms for the *layered* and *double-layered* 3-sided 2-D reporting problems, which will form the building blocks for our algorithm for the layered 3-D dominance reporting problem (the concept of *layers* was also used in [11] and [10]). These algorithms build on known data structures for the non-layered case (see e.g. [1] for a survey). Our overall strategy is the same as in Section 3.1 except that we replace the (x, y)-Cartesian trees with the new data structures.

5.1 Layered 3-Sided 2-D Reporting

Suppose F is a set of $m \leq n$ two-dimensional points with x-coordinate $p.x \in [1..m]$, y-coordinate $p.y \in [1..m]$ and layer $p.l \in [1..c]$. The layered 3-sided 2-D reporting problem is defined as follows.

Problem 5.1. *Develop a data structure for F such that, given a query (a, b, d, L) with $a, b, d \in [1..m]$ and $L \in \mathcal{P}([1..c])$, we can quickly report the points p that satisfy $a \leq p.x \leq b$, $p.y \geq d$, and $p.l \in L$.*

We will store L as a bit vector in a single word. Further, we will assume that we are allowed to use a constant number of global look-up tables of size $O(n)$ words. Intuitively, the reader should view F as the the points stored in the (x, y)-Cartesian trees of a node as described in Section 3.1. The above problem can be reduced to the following *layered* range-maximum problem.

Problem 5.2. *Develop a data structure for F such that, given a query (a, b, L) with $a, b \in [1..m]$ and $L \in \mathcal{P}([1..c])$, we can quickly report the point p such that $p.y$ is maximized under the conditions $a \leq p.x \leq b$ and $p.l \in L$.*

We will later show how to handle Problem 5.2 in constant time and linear space. With such an algorithm, we can handle the layered 3-sided 2-D range query in $O(f)$ time using linear space, f being the output size, as follows. We first find the point p with x-coordinate between a and b such that $p.y$ is maximized under the condition that $p.l \in L$. If $p.y < d$, then no point in $S(v)$ is proper. Otherwise, we report the point p and recursively apply two layered range-maximum queries $(a, p.x - 1, L)$ and $(p.x + 1, b, L)$.

What remains to be shown is that a layered range-maximum query on F can be handled in constant time. We first give a non-linear space solution, and then discuss how to reduce the space to linear.

5.1.1 A Non-Linear Space Solution

We build a binary tree T on the increasing x-coordinates of the m points in F. We associate with each leaf node v of T the two-dimensional arrays $v.\text{left}[0..\log m][1..c]$ and $v.\text{right}[0..\log m][1..c]$. Let u be the ancestor of v whose distance to v is l (the distance between v and itself is 0). Then $v.\text{left}[l][j]$ stores the maximum y-coordinate of the points with layer j corresponding to the leaf nodes between the leftmost leaf node of the subtree rooted at u and and the leaf node v. Similarly, $v.\text{right}[l][j]$ stores the maximum y-coordinate of the points with layer j corresponding to the leaf nodes between v and the rightmost leaf node of the subtree rooted at u.

Now suppose we are given a query (a, b, L) with $a < b$. Let v_a and v_b be two leaf nodes corresponding to the two points whose x-coordinates are a and b respectively. We can in constant time locate the nearest common ancestor u of v_a and v_b. Let h be the height of u (the height of a leaf node is 0). Then the point that satisfies the query is either the point in $v_a.\text{right}[h-1][l]$ with maximum y-coordinate where $l \in L$ or the point in $v_b.\text{left}[h-1][l]$ with maximum y-coordinate where $l \in L$.

It follows that, to show that a layered range-maximum query can be handled in constant time, it is sufficient to show that, given an array $X[1..c]$ of elements in $[1..m]$, we can preprocess X into a data structure so that, given a subset L of $[1..c]$, the value $\max\{X[l] | l \in L\}$ can be computed in constant time.

To do so, we replace each entry in X with its rank among the elements in X. Each possible array thus can be represented using $O(\log\log n \log^{\epsilon} n)$ bits. Since each possible set L of layers specified by a query can be represented using $O(\log^{\epsilon} n)$ bits, we can construct a global look-up table of

$O(n)$ entries, each occupying a word and corresponding to one of the $2^{O(\log\log n \log^{\epsilon} n + \log^{\epsilon} n)} = O(n)$ possible instances of the query, i.e. a possible combination of an array X and a set L, and storing the answer to that query.

The overall space required (beside the global look-up table) for this solution is $O(m \log m \log^{\epsilon} n)$ because each of the m leaf nodes is associated with an array of size $O(\log m \log^{\epsilon} n)$.

5.1.2 Reducing the Space to Linear

We now describe how to reduce the space usage to $O(m)$. We sort the points in F by their x-coordinates and group them into blocks each with $O(\log m \log^{2\epsilon} n)$ points. For each block and for each layer, we take the point in the block with maximum y-coordinate. We then build a structure T' just described over these points. Since the number of such points is $O(m/(\log m \log^{\epsilon} n))$, the size of T' is $O(m)$. Given a query (a, b, L), the interval $[a, b]$ can be partitioned into three parts $[a..(l-1)]$, $[l..r]$, and $[(r+1)..b]$, in which $[l..r]$, if it exists, corresponds to the maximum sequence of consecutive blocks that are fully contained in $[a..b]$. Therefore, the output of the query (a, b, L) is one of the three points reported by the queries $(a, l-1, L)$, (l, r, L), and $(r+1, b, L)$, with the largest y-coordinate. The (l, r, L) query can be handled in constant time using T'.

What remains is to describe how we query each block B of points in constant time. This is a special layered range-maximum problem in which the problem size $|B|$ is at most $O(\log m \log^{2\epsilon} n)$. First, we replace the y-coordinates of the points with their ranks among the y-coordinates. Next, we build a tree $T(B)$ with degree $O(\log^{\delta} n)$ (for a sufficiently small constant $\delta > 0$) and thus constant height on the x-coordinates of the points in B. Since the tree is of constant depth, we can solve the query on B in constant time as long as the following problem can be solved in constant time: given a node v, and two integers $k_1, k_2 \in [1..\log^{\delta} n]$ with $k_1 \leq k_2$ and $L \in \mathcal{P}([1..c])$, how to decide in constant time the point with the maximum y-coordinate which is stored in the children of v with labels between k_1 and k_2, and is from a layer in L. Again, the number of possible query instances is bounded by $2^{O((\log\log n)^2 \log^{\delta} n + 2\log\log n + \log^{\epsilon} n)} = O(n)$ and thus the answer to each such instance can be obtained in constant time using a global look-up table of size $O(n)$. Therefore, any layered range-maximum query on any block can be handled in linear space and constant time and Lemma 5.1 follows.

Lemma 5.1. *There exists a data structure such that any layered three-sided two-dimensional reporting query as defined in Problem 5.1 can be handled in $O(f)$ time using linear space.*

5.2 Double-layered 3-Sided 2-D Reporting

We now discuss the *double-layered* 3-sided 2-D reporting problem, which is more general than the one addressed in Section 5.1. For this new problem, each of the m point $p = (p.x, p.y)$ in F is assigned two layers $p.l_1$ and $p.l_2$. A double-layered 3-sided 2-D reporting problem is defined as:

Problem 5.3. *Develop a data structure for F such that, given a query (a, b, d, L_1, L_2), where $L_1, L_2 \in \mathcal{P}([1..c])$, we can quickly report the proper points p, which satisfy $a \leq p.x \leq b$, $p.y \geq d$, $p.l_1 \in L_1$, and $p.l_2 \in L_2$.*

Following the same argument as in Section 5.1, we can reduce the above problem to the following *double-layered* range maximum problem.

Problem 5.4. *Develop a data structure for F such that, given a query (a, b, L_1, L_2) with $a, b \in [1..m]$ and $L_1, L_2 \in \mathcal{P}([1..c])$, we can quickly report the points p such that $p.y$ is maximized under the conditions $a \leq p.x \leq b$, $p.l_1 \in L_1$, and $p.l_2 \in L_2$.*

Our algorithm for Problem 5.4 is almost identical to the one for Problem 5.2 and runs in constant time. So we will only comment on the necessary changes to that algorithm. We start from the non-linear space solution. For each leaf node v, we replace v.left$[0..\log m][1..c]$ and v.right$[0..\log m][1..c]$ with two three-dimensional arrays v.left$[0..\log m][1..c][1..c]$ and v.right$[0..\log m][1..c][1..c]$, each entry storing a maximum value corresponding to a distinct pair of layers. To show that a double-layered range-maximum query can be handled in constant time, it is sufficient to show that, given an array $X[1..c][1..c]$ of elements in $[1..m]$, we can preprocess X into a data structure so that, given $L_1, L_2 \in \mathcal{P}([1..c])$, the value $\max\{X[l_1][l_2] | l_1 \in L_1, l_2 \in L_2\}$ can be identified in constant time. As we did in Section 5.1, we solve this problem by using a global-lookup table. This is possible because the number of possible query instances is $2^{O(\log\log n \log^{2\epsilon} n + 2\log^{\epsilon} n)} = O(n)$. The space used, aside from this global-lookup table is $O(m \log m \log^{2\epsilon} n)$.

To reduce the space to linear, we group the points into blocks, each with $O(\log m \log^{4\epsilon} n)$ points. For each block, we pick for each possible pair of layers, the point with maximum y-coordinate. The data structure described in the previous paragraph is then used to index the points thus picked and the space used is $O(m)$. Following similar argument as in Section 5.1, the double-layered range maximum problem on each block can be solved in constant time using linear space provided that we have available a global look-up table of size $2^{O((\log\log n)^3 \log^{\delta} n + 2\log\log n + 2\log^{\epsilon} n)} = O(n)$.

Lemma 5.2. *There exists a data structure such that any double-layered three-sided two-dimensional reporting query as defined in Problem 5.3 can be handled in $O(f)$ time using linear space.*

5.3 Layered 3-D Dominance Reporting

Note that by replacing the c (x, y)-Cartesian trees associated with each of the internal node v in the data structure described in Section 3.1 with the data structure for solving the layered 3-sided 2-D reporting problem (m is equal to the number of points in the union of the maximal sets of v's children), we immediately obtain an alternative $O(n)$-space and $O(\log n/\log\log n + f)$-query time solution for the 3-D dominance reporting problem. However, in order to make the solution extendable to higher dimensions, we need a stronger result than in Theorem 4.1. We now describe how to use Lemmas 5.1 and 5.2 to solve the layered 3-D dominance reporting problem.

The skeleton of our structure is the same tree T described in Section 4. At each node v of T, we store two structures: $D(v)$, and $R(v)$. $D(v)$ is the structure of Lemma 5.1 built on the maximal set $M(v)$, and $R(v)$ is the structure of Lemma 5.2 built on the union $S(v)$ of the maximal sets of the children of v. For each point p in $S(v)$, if p comes from the maximal set of the child of v with label l, then, in addition to the layer $p.l_1$ initially assigned to p, we assign a second layer $p.l_2 = l + 1$ to p.

The search process for a query (q_1, q_2, q_3, L) is almost the same as described in Section 4. The difference is that, at each node v visited, we use the query (q_1, q_2', q_3, L) on $D(v)$ to report the proper points in $M(v)$, where q_2' is the x-coordinate of the y-successor of q_2 in $M(v)$, which can be computed in constant time using fast fractional cascading. Suppose the jth child of v is also on the path Π. To decide which child of v should be visited, we use the double-layered 3-sided 2-D reporting query $(q_1, q_2, L, [1..(j-1)])$ on $R(v)$.

Notice that Lemmas 5.1 and 5.2 are valid only when the points are from a rank space. This is not the case here since we only have at each internal node of the primary tree a subset of the original data set. However, this problem can be easily solved by using the fast fractional cascading structure to replace the coordinates of each point with its x- and y-ranks in that subset.

Theorem 5.1. *There exists a data structure such that any layered three-dimensional dominance reporting query can be handled in $O(\log n/\log\log n + f)$ time using $O(n)$ space.*

6 Handling d-Dimensional Dominance Reporting Using Dimension Increasing Lemma

In this section, we extend the results in Section 5 to handle d-dimensional dominance reporting queries on a set of n d-dimensional points, with $d \geq 3$. We do so by introducing the *dimension increasing lemma* (a similar lemma for a dynamic set of points was proved in [10]), which basically states that, for any $k < d$, if we can handle a layered range reporting (resp. counting) query in k-dimensional space, then we can handle a layered range reporting (resp. counting) query in $(k+1)$-dimensional space by increasing both the space and query time by a factor of $O(\log n / \log\log n)$. In this section, we define $\mathcal{L}^\epsilon = [1..\log^\epsilon n]$ for any constant $0 < \epsilon < 1/5$, and assume that we have available a global look-up table of size $O(n)$ as required by the algorithms in Section 5.

Let $(S, +)$ be a semigroup. We now define what a $(\mathcal{Q}, \epsilon)$ data structure is. Each element $e \in (\mathcal{Q}, \epsilon)$ has a point $e.\mathrm{p}$, a layer $e.\mathrm{layer} \in \mathcal{L}^\epsilon$ and a semigroup element $e.\mathrm{s} \in S$. The constant ϵ satisfies $0 < \epsilon < 1/5$ and $\mathcal{Q}$ is a set of predicates on points. $(\mathcal{Q}, \epsilon)$ is assumed to support a query $(q, L) \in \mathcal{Q} \times \mathcal{P}(\mathcal{L}^\epsilon)$ whose answer is $\sum_{e \in (\mathcal{Q},\epsilon) \,|\, q(e.\mathrm{p}) \wedge e.\mathrm{layer} \in L} e.\mathrm{s}$.

Lemma 6.1. *Suppose we have a $(\mathcal{Q}, \epsilon)$ structure. Suppose further f is an injective function from points to integers in single words which can be evaluated in constant time. Then there exists a $(\mathcal{Q}', \epsilon/2)$ structure where $\mathcal{Q}'$ is the set of predicates q' that can be written as $q'(p) = i \leq f(p) \leq j \wedge q(p)$ for $q \in \mathcal{Q}$ and integers i and j. Further, if $m = |(\mathcal{Q}', \epsilon/2)| \leq n$ is the number of elements in $(\mathcal{Q}', \epsilon/2)$ then:*

1. *Each element in $(\mathcal{Q}', \epsilon/2)$ is stored in $O(\log m / \log\log n)$ $(\mathcal{Q}, \epsilon)$ structures.*
2. *Each $(\mathcal{Q}, \epsilon)$ structure contains at most m elements.*
3. *Given a query in $(\mathcal{Q}', \epsilon/2)$, we can answer it by performing $O(\log m / \log\log n)$ queries in $(\mathcal{Q}, \epsilon)$ structures and then return the semigroup sum of the answers as result.*

Further the space usage besides the space usage in item 1 is $O(m)$ and the queries to perform in item 3 can be determined in constant time per query.

Proof. We make a search tree T containing the elements of $(\mathcal{Q}', \epsilon/2)$ in the leaves, such that T has degree $d = \lfloor \log^{\epsilon/2} n \rfloor$ and hence height $O(\log m / \log\log n)$. For $e \in (\mathcal{Q}', \epsilon/2)$ we use $f(e.\mathrm{p})$ as key in T. Let $v \in T$ be an internal node. We keep in v a q-heap containing the keys stored to guide the search through v. Further, we keep in v a secondary $(\mathcal{Q}, \epsilon)$ structure. Let $e \in (\mathcal{Q}', \epsilon/2)$ be an element stored in a leaf descendant to v which is also descendant to the child of v with label l. We then store an element e' in the secondary structure in v with $e'.\mathrm{p} = e.\mathrm{p}$, $e'.\mathrm{s} = e.\mathrm{s}$ and $e'.\mathrm{layer} = l + d \cdot e.\mathrm{layer}$. Note that $e'.\mathrm{layer} \in \mathcal{L}^\epsilon$.

Now suppose we are given a query $(q', L') \in \mathcal{Q}' \times \mathcal{P}(\mathcal{L}^{\epsilon/2})$ where $q'(p) = i \leq f(p) \leq j \wedge q(p)$ and assume without loss of generality that $i \neq j$. The interval $[i \ldots j]$ identifies a set $M \subseteq T \times \mathcal{P}(\mathcal{L}^{\epsilon/2})$ with $|M| = O(\log m / \log\log n)$ as follows. Let Π_{left} and Π_{right} be the two paths from the root of T to the two leaves that respectively correspond to the successor of i and the predecessor of j. Further, let w be the lowest internal node on both Π_{left} and Π_{right}. Suppose the lth child of w is on Π_{left} and the rth child is on Π_{right}. Then M contains the element $(w, [(l+1)..(r-1)])$. Suppose next that v is an internal node on Π_{left} (resp. Π_{right}) which is not on Π_{right} (resp. Π_{left}). Suppose also that the lth (resp. rth) child of v is on Π_{left} (resp. Π_{right}). Then M contains the element $(v, [1..l-1])$ (resp. $(v, [r+1..d])$). For each $(v, L) \in M$ we perform the query $(q, \{l_1 + d \cdot l_2 \in \mathcal{L}^\epsilon \mid l_1 \in L \wedge l_2 \in L'\})$ in the secondary structure of v and then we return the semigroup sum in S of the answers to these queries as answer. Using the q-heaps stored in the nodes of T we can find M in time $O(\log m / \log\log n)$ and the lemma follows. $\square$

By using the layered 3-D dominance reporting structure of Theorem 5.1 (for the dominance reporting problem where we have m points with layers from $[1..\log^{\epsilon} n]$ and a global look-up table of size $O(n)$) as a basis and applying Lemma 6.1 $d-3$ times, we immediately obtain Theorem 1.1.

7 A Linear-Space Algorithms with $O(\log n/\log\log n)$ query time for 2-D Dominance Counting

In this section, we show how to modify the data structure mentioned in Section 3.2 so that the space usage is reduced to linear and the query performance remains the same. We consider the case where the coordinates of the n points are integers in $[1..n]$. The general case can be converted to this one by replacing q_i of a query $(q_1, \ldots, q_d)$ with its x_i-rank r_i with respect to S. This conversion can be performed in $O(\log n/\log\log n)$ time using the fusion tree technique with an additional space usage of $O(n)$.

Recall that in Section 3.2, we associate with each node v two structures: a router $r(v)$ and a counter $c(v)$. The overall size of the routers is $O(n)$. Therefore the bottleneck in terms of space usage is the set of counters.

We can view a router with m points as a set F of two-dimensional points in $[1..m] \times [1..c]$, and hence the corresponding counter can be viewed as a counting structure that allows a 2-D dominance counting query on F to be handled in constant time. Let $h = \log n/\log\log n$. The solution in Section 3.2 uses $O(m/h \cdot \log^{\epsilon} n)$ space. In this section, we give a more space-efficient solution that uses only $O(m/h)$ space. The following theorem shows a more general result which will also be used for d-dimensional dominance counting.

Lemma 7.1. *Assume we are given $m \leq n$ points F in $[1..m] \times [1..c]^{d-1}$. Then there exists a dominance counting structure for F using $O(m/h)$ words supporting queries in $O(1)$ time. This structure makes use of a precomputed global look-up table of size $O(n)$.*

Proof. We create a tree of height 3 on the x-coordinates of the points in F, sorted by decreasing x-coordinates, in the following way (see Figure 3). At level 0 we have m leafs. The nodes at level 1 have degree h and the nodes at level 2 have degree c^{d-1}. It follows that the root node at level 3 has degree $m/(hc^{d-1})$.

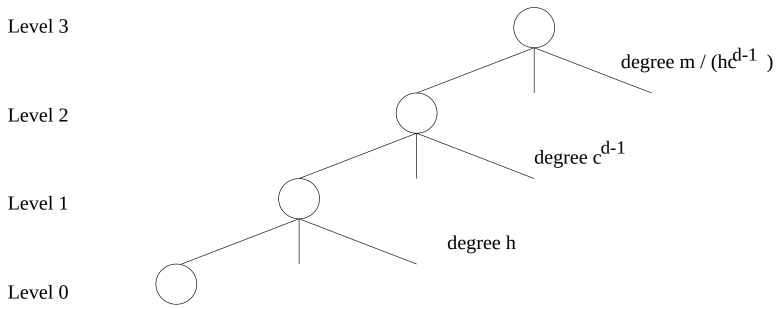

Figure 3: The tree structure for the proof of Lemma 7.1

For each node v at levels 1 and 2, we store a table $L(v)$ indexed by $[1..c]^{d-1}$. At entry $(u_2, \ldots, u_d)$ of $L(v)$ we store the number of points $p = (p_1, p_2, \ldots, p_d)$ which satisfy the following two conditions: (i) $p_2 \geq u_2, \ldots, p_d \geq u_d$; (ii) p_1 corresponds to a leaf node in one of the subtrees rooted at the siblings of v which are to the left of v.

For each node v at level 2, each entry of $L(v)$ uses $O(\log n)$ bits. Since there are $O(m/(hc^{d-1}))$ nodes at level 2, the total number of words needed for all the tables of level 2 nodes is $O(m/h)$.

For each node v at level 1, $O(\log(hc^{d-1})) = O(\log\log n)$ bits are sufficient to represent each entry of $L(v)$. Since there are $O(n/h)$ such nodes. The overall bit-size of the tables at level 1 is $O(c^{d-1}\log\log n \cdot m/h)$. For a small enough constant $\epsilon > 0$, $O(c^{d-1}\log\log n) = O(\log n)$ and hence the word-cost of these tables is $O(m/h)$.

We pack the nodes at level 0 into m/h chunks, each containing h nodes that have the same parent. Since each such node describes a point in $[1..c]^{d-1}$, the number of bits required to represent a chunk is $O(h(d-1)\log c) = O(\log n)$. Hence the word-cost for representing all the m/h chunks is $O(m/h)$.

Now suppose we are given a query $(r_1, \ldots, r_d)$. The answer to this query is computed by aggregating the numbers we obtain along the path from the root to the leaf node that corresponds to the number r_1. For the two nodes u_1 and u_2 on this path at levels 1 and 2 respectively, the desired numbers can be obtained from entries $(r_1, \ldots, r_d)$ in $L(u_1)$ and $L(u_2)$. At level 0, since the number of possible chunks is $O(n)$, the number of points in a chunk that dominates $(r_1, \ldots, r_d)$ can be found by looking up in a global table of size $O(n)$. □

8 d-Dimensional Dominance Counting

The techniques described in Section 7 can be extended to higher dimensions. As we did in that section, we build a tree with degree d on the first dimension. At each node v, we have a router that record for each point in the subtree rooted at v which subtree of v's child this point comes from. Given a query $q = (r_1, \ldots, r_n)$, what we need to compute at each of the nodes visited (one at each level) is the number of points $p = (p_1, \ldots, p_d)$ coming from the subtrees rooted at the left most k children of that node which satisfy $p_2 \geq r_2, \ldots, p_d \geq r_d$.

Following the same approach as in Section 7, a router can be viewed as a set F of d-dimensional points in $[1..m]^{d-1} \times [1..c]$. By setting $e = d-1$, the following lemma immediately gives Theorem 1.2.

Lemma 8.1. *For any $e \in [1..d]$, there exists a dominance counting structure for $m \leq n$ points in $[1..m]^e \times [1..c]^{d-e}$ using $O(mh^{e-2})$ words of memory and $O(h^{e-1})$ query time. The structure needs a precomputed table with $O(n)$ words.*

Proof. We prove the lemma by induction on e. For $e = 1$, this is just Lemma 7.1. For the inductive step, suppose Lemma 8.1 holds for $e = k - 1 \geq 1$. We show that the lemma holds for $e = k$. We create a tree R with m leaves and degree c on the decreasing x_k-coordinates of the m points in F.

For each internal node u of R, we associate a dominance counting structure for $e = k-1$ on the points in the subtree rooted at u. By the induction hypothesis, the overall size of such structures for all the nodes in R is bounded by $O(\sum_{j=0}^{h} c^j (m/c^j) h^{k-3}) = O(mh^{k-2})$.

Now suppose we are given a query $(r_1, \ldots r_d)$, we compute the answer to the query by aggregating the count along the path from the root of R to the leaf node that corresponds the point whose x_k-coordinate is r_e. Consider an internal node u on this path. Suppose its jth child is also on the path. Then the count contributed by u is the output of the query $(r_1, \ldots, r_{k-1}, j, r_{k+1}, \ldots, r_d)$ on the dominance counting structure of u for $e = k - 1$; and this count can be computed in $O(h^{k-2})$ time.

Since the height of R is $O(\log m / \log c) = O(h)$, the time it takes to search R is $O(h^{k-2}h) = O(h^{k-1})$. This completes the inductive step. □

Comment: Note that Lemma 6.1 applies to counting queries as well. It would also be possible to obtain Theorem 1.2 by modifying the 2-D counting algorithm in Section 7 to support layers.

References

[1] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of the 14th ACM Symp. on Parallel Algorithms and Architecture (SPAA)*, pages 258–264, Aug. 2002.

[2] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, Aug. 1986.

[3] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–463, June 1988.

[4] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete Comput. Geom.*, 3:113–126, 1987.

[5] B. Chazelle and L. J. Guibas. Fractional Cascading: I. A data structure technique. *Algorithmica*, 1(2):133–162, 1986.

[6] H. Edelsbrunner and M. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14:124–127, 1982.

[7] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[8] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[9] C. Makris and A. K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, 1998.

[10] C. W. Mortensen. Fully-dynamic orthogonal range reporting on RAM (Preliminary version). Technical Report TR-2003-22, The IT University of Copenhagen, 2003.

[11] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 618–627, 2003.

[12] Q. Shi and J. JaJa. Fast algorithms for 3-d dominance reporting and counting. Technical Report CS-TR-4437, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[13] Q. Shi and J. JaJa. Fast algorithms for a class of temporal range queries. In *Workshop on Algorithms and Data Structures (WADS'03)*, pages 91–102, Ottawa, Canada, 2003.

[14] Q. Shi and J. JaJa. Fast fractional cascading and its applications. Technical Report CS-TR-4502, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[15] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.