ABSTRACT

Title of Thesis: DEPENDENT ROUNDING SPEEDUPS AND APPLICATIONS

Thomas Wrona
Master of Science, 2024

Thesis Directed by: Professor Aravind Srinivasan
Department of Computer Science

Dependent rounding is a rounding method on bipartite graphs with several good theoretical properties, including negative correlation and total degree preservation. However, a naive implementation of the dependent rounding algorithm runs in at least cubic time with respect to the number of vertices present, which is untenable for larger graphs. To alleviate this problem, we introduce several speedup methods, including reformulations and parallelism. These methods can be used to reduce the runtime of dependent rounding by several orders of magnitude, making it feasible to use repeatedly on large graphs. We also present potential applications of dependent rounding enabled by these speedups. These applications are largely modifications to machine learning algorithms, such as SignSGD, to include dependent rounding in order to improve a rounding method or make a sampling method more consistent. We hope that dependent rounding can serve as a new method to improve machine learning techniques both theoretically and empirically.

DEPENDENT ROUNDING SPEEDUPS AND APPLICATIONS

by

Thomas Wrona

Thesis 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
Master of Science
2024

Advisory Committee:
Professor Aravind Srinivasan, Chair/Advisor
Professor Laxman Dhulipala
Professor Furong Huang

Acknowledgments

I am grateful to Professor Aravind Srinivasan for giving me this opportunity, Professors Laxman Dhulipala and Furong Huang for being on the committee as well as providing additional input, Jiayi Wu for his assistance, and my friends and family for their support.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1: Introduction

Graphs, especially bipartite graphs, have become fundamental for various computing problems. Bipartite graphs may be used to represent set membership (by representing sets and their members as two disjoint partitions that together form the vertices of a graph) or any two-dimensional matrix (the indices of the first dimension in one partition and the second dimension in the second partition) and so on [1]. The most popular application of bipartite graphs as of late is machine learning, where a bipartite graph represents each layer of the machine learning architecture, with input and output features as the two partitions and weights as weighted edges between them. Machine learning (ML) noticeably suffers from growing graph sizes as the number of features used in typical architectures increases dramatically. As bipartite graph use cases have grown in recent years, approximation or sampling methods may be needed to reduce the complexity of these problems to more realistic scales.

Dependent rounding, introduced in [2], is an improved method of rounding the edges of bipartite graphs with some convenient properties. In particular, applying dependent rounding to a graph results in almost-preserved degrees, where, for each vertex, the sum of edge weight values incident to the vertex will differ by less than one from the original sum. Additionally, the edges incident to each vertex have negatively correlated rounded weights. These will be discussed in further detail later. These properties are beneficial in many theoretical formulations, including k-medians [3] and network design [2, 4], among other graph problems. However, dependent rounding rarely sees direct use in practical applications.

Despite all the potential of dependent rounding, it is not a fast process when compared to standard independent rounding methods (e.g. round each value in succession), which generally are constant time, linear work (with respect to the number of elements rounded) algorithms. While most rounding and sampling methods can easily use parallelism, dependent rounding is a superlinear and highly sequential process. The original dependent rounding algorithm cannot benefit significantly from parallelism, as every step depends on all previous steps. So, as graph problems apply to increasing graph sizes, dependent rounding, as originally conceived, becomes infeasible to apply. In this paper, one of our goals is to propose a new dependent rounding algorithm feasible for use on large graphs.

Moving forward, the most promising application of dependent rounding is machine learning. Machine learning weights are often represented as a bipartite graph connecting features in one layer to the next. Therefore, dependent rounding can be converted into a sampling method by applying it to sampling probabilities and using the resulting 0's and 1's to decide whether to sample an edge. Neural network pruning is a field with many recent breakthroughs, probabilistic sampling being one of many possible pruning methods [5]. However, any current probabilistic pruning will prune weights using independent sampling. Dependent rounding has several advantages over any independent sampling method, notably degree preservation and negative dependence. Therefore, we believe that it is not unreasonable to take probabilistic pruning methods and perform dependent rounding on the probabilities instead of standard independent sampling. Given the properties of dependent rounding, we will present several possible avenues to improve machine learning techniques. We hope that our new methods will increase the accuracy of these techniques to be worth the increased time and memory costs of dependent rounding.

1.1 Outline of Thesis

In Chapter 2, we discuss some related work. In particular, we briefly overview some work on dependent rounding, parallelism, and machine learning.

In Chapter 3, we analyze the original dependent rounding algorithm, then introduce our new algorithm to speed up the process. We then analyze its runtime both theoretically and experimentally.

In Chapter 4, we propose some possible applications of this new dependent rounding algorithm, primarily in the area of machine learning. We discuss some preliminary experiments and some additional areas of interest where dependent rounding may offer improvement.

Chapter 5 concludes the thesis.

Chapter 2: Related Work

2.1 Dependent Rounding

We will start by describing independent rounding. Independent rounding is any method where each member of a set of values is rounded independently of all others. In general, we will denote x as an original value and X as the rounded value of x. X is often, but not necessarily, a random variable. For example, standard rounding takes values and independently rounds each to the closest integer. We can write this as:

$$X = \begin{cases} \lfloor x \rfloor + 1, & x \geq \lfloor x \rfloor + \frac{1}{2} \\ \lfloor x \rfloor, & \text{otherwise} \end{cases} \tag{2.1}$$

Another independent rounding method is stochastic rounding, where a value $x \in [0, 1]$ is rounded up with probability proportional to its distance to $\lfloor x \rfloor$. This means values are more likely to be rounded to the closest integer. More precisely, this is:

$$X = \begin{cases} \lfloor x \rfloor + 1, & \text{with probability } x - \lfloor x \rfloor \\ \lfloor x \rfloor, & \text{otherwise} \end{cases} \tag{2.2}$$

This method is unbiased, as:

$$E[X] = (\lfloor x \rfloor + 1) \times (x - \lfloor x \rfloor) + \lfloor x \rfloor \times (1 - x + \lfloor x \rfloor) = x \tag{2.3}$$

Dependent rounding is a more elaborate version of rounding a set of values. The chance of rounding a value up or down is dependent on some other values in the set that is being rounded. Note that dependent rounding can refer to any scheme that does this, but we will use the term to refer to the scheme introduced in [2]. The scheme begins with a bipartite graph $G = (A, B, E, w)$ where:

- A and B are disjoint partitions of the vertices of G.
- E is the set of edges between A and B.
- w is the set of weights. Here, we assume without loss of generality that w_{ij} is between 0 and 1. Otherwise, we can shift w_{ij} to $w_{ij} - \lfloor w_{ij} \rfloor$ for each value before rounding and then shift back afterward. Also, note that the algorithm can be modified to round to any precision p other than 1 by multiplying each value by $\frac{1}{p}$ before first shift and by p after the return shift.

Dependent rounding returns a modified bipartite graph $G' = (A, B, E', w')$, where $w'_{ij} \in \{0, 1\}$ and E' no longer contains the edges (i, j) where $w'_{ij} = 0$, i.e. the edges with weights rounded to 0 are removed. There are three relevant properties to dependent rounding:

1. Marginal distribution - $\forall i, j$,

$$P(w'_{ij} = 1) = w_{ij} \tag{2.4}$$

. This is a guarantee that all resulting edge weights are unbiased, a convenient property shared with stochastic rounding.

2. Degree preservation - $\forall i, j$,

$$\sum_j w'_{ij} \in \{\lfloor \sum_j w_{ij} \rfloor, \lceil \sum_j w_{ij} \rceil\} \tag{2.5}$$

$$\sum_i w'_{ij} \in \{\lfloor \sum_i w_{ij} \rfloor, \lceil \sum_i w_{ij} \rceil\} \tag{2.6}$$

. In words, this indicates that the sum of edge weights incident to every vertex remains within 1 after rounding, minimizing variance.

3. Negative correlation - $\forall b \in \{0, 1\}, S \subseteq [|B|], i$,

$$P[\wedge_{j \in S} w'_{ij} = b] \leq \prod_{i \in S} P[w'_{ij} = b] \tag{2.7}$$

This property also holds for all j and $S \subseteq [|A|]$. Here $[n]$ denotes the set $\{1, 2, \dots, n\}$. As a result, neighboring edges have lower probability of being rounded to the same value.

We can see that dependent rounding has many strengths but a crucial weakness appears: while independent rounding schemes are always $O(1)$ time and $O(|A||B|)$ work operations (i.e. constant time and work proportional to the number of edges), dependent rounding has a work and time complexity of $O(|A||B|(|A| + |B|))$, which is superlinear with respect to the number of edges. The difference in time complexity is significant enough that dependent rounding on even moderately sized graphs (on the order of 100,000 edges) can be unrealistic.

Dependent rounding historically has been relegated to theoretical algorithms, but we can look at them for inspiration on potential realized applications for dependent rounding. The original paper [2] includes algorithms related to scheduling and routing. For example, we can frame broadcast scheduling as a problem where A is the set of time slots and B is the set of topics and apply dependent rounding to assign topics to time slots. We can also schedule jobs on multiple machines by creating a vertex in A for each job and a vertex in B for each machine. Dependent rounding can also round relaxed linear programming solutions to an integer solution [3], partially by transforming the relaxed solutions into a partition each. This method can be used to solve the k-means problem, also called the connected facility location problem [4].

2.2 Parallelism

As far as we know, due to the theoretical nature of dependent rounding, there is no work on improving its runtime, whether in serial or parallel. Therefore, we have instead based our work on speedup methods for other graph problems, namely butterfly enumeration and depth-first search (DFS).

In a bipartite graph $G = (A, B, E)$, we define a butterfly to be any set $\{a_0, a_1, b_0, b_1\}$ such that $a_0, a_1 \in A, b_0, b_1 \in B, (a_0, b_0), (a_0, b_1), (a_1, b_0), (a_1, b_1) \in E$. In other words, it is a pair of vertices in A and a pair of vertices in B where all four possible edges among them exist (or have nonzero weight in the weighted case). Notice that a butterfly is equivalent to a four-cycle in a graph. We also define a wedge from A to B to be any set $\{a_0, a_1, b\}$ such that $a_0, a_1 \in A, b \in B, (a_0, b), (a_1, b) \in E$. This is a pair of vertices in A and a vertex in B such that both possible edges among them exist (or have nonzero weight). It follows that every butterfly can be split into two wedges from A to B or two wedges from B to A.

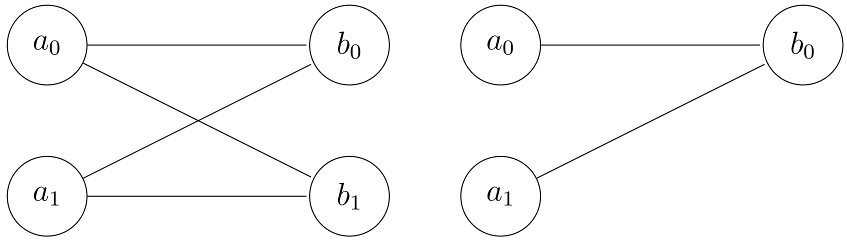

Figure 2.1: Example of a butterfly (left) and a wedge (right).

Previous work [6] indicates that butterfly counting is a process amenable to significant parallelism, implying low time complexity. Most of the methods of butterfly counting perform the following:

1. In parallel, for each pair a_1, a_2 out of $\binom{|A|}{2}$ possible pairs, count the number of wedges from that pair to vertices in B and call the result $W_{a_1 a_2}$. The total number of butterflies from a_1, a_2 to B will be $\binom{W_{a_1 a_2}}{2}$. This is $O(\log |B|)$ time and $O(|A|^2|B|)$ work, as we must do a prefix sum operation over $|B|$ elements ($O(\log |B|)$ time, $O(|B|)$ work) $\binom{|A|}{2}$ times in parallel.

2. Use the parallel prefix sum algorithm to sum $\binom{W_{a_1 a_2}}{2}$ across all $a_1, a_2 \in A$. This is $O(\log |A|)$ time and $O(|A|^2)$ work.

We can see from this that the total number of possible butterflies in a bipartite graph is:

$$|\{a_0, a_1, b_0, b_1 \mid a_0, a_1 \in A, b_0, b_1 \in B, (a_0, b_0), (a_0, b_1), (a_1, b_0), (a_1, b_1) \in E\}| \leq \binom{|A|}{2}\binom{|B|}{2}$$

The size of this number is a strong indicator that many butterflies may exist even in graphs with a reduced number of edges. Assuming that this is true, using butterflies in dependent rounding can shorten both the length of cycles used in the rounding process and the runtime of a single rounding step. We are more interested in butterfly enumeration, which does not benefit from the speedup of using prefix sum. However, the base idea of splitting A into pairs still serves as inspiration for our algorithm.

Dependent rounding, as originally proposed, is dependent on DFS, which is not typically amenable to parallelism. In fact, work-efficient parallel DFS can only decrease runtime from $O(n+m)$ to $O(n)$, which is only nontrivial for dense graphs. Recent work has more significantly increased the speed (down to $O(\sqrt{n})$ time complexity) while being nearly work efficient [7]. Despite this speedup, running DFS only once and maintaining information without repeated calls is generally more effective to reduce runtime when possible. Also, using butterfly-based methods cuts much of the need for DFS. These ideas lead us to believe that the optimal DFS methods are not worth the increased work, as they would only decrease the complexity of a small subset of the problem. Nevertheless, we remain interested in further study of parallel DFS techniques to serve as an alternative to improving dependent rounding.

2.3 Machine Learning

Machine learning could be a primary driver for future dependent rounding applications. Any machine learning technique involving data simplification or approximation is a strong candidate for using dependent rounding due to its consistency of information retention from improved concentration bounds and degree preservation. SignSGD [8] is an optimization technique that involves using the sign of the gradient (as opposed to full gradient) to perform gradient descent. In other words, SignSGD takes the gradient tensor for each layer and individually replaces each element with its sign. This has the advantage of significant compression while also achieving convergence rates near or at state-of-the art (ADAM [9], SGD). However, rounding small gradients to their sign can result in over-corrections in early gradient descent, and it is our belief that more delicate rounding methods can remedy this problem. Applying different rounding methods to SGD also may provide further insight into why SignSGD is so effective.

Neural network pruning is another machine learning technique involving data simplification that uses data removal. A general conclusion of neural network pruning work [5] is that randomization techniques do not perform as well as simpler methods. The most basic weight magnitude pruning, which removes the smallest magnitude weights, significantly outperforms simple random pruning, which also has a large variance in effectiveness. The most effective randomized methods involve pruning probabilities generated by examining the possible effect of each weight on the result of the neural network [10–12]. However these methods require many samples to be effective, limiting the amount of effective pruning. There is some indication that some randomized methods can perform better at low levels of pruning [13], but in all other cases, magnitude weight pruning remains state-of-the-art. This conclusion is not surprising given the extra introduced noise in the methods. We believe that the problems with introducing stochasticity in machine learning are due to the inherent inconsistency of rounding and sampling techniques, preventing the benefits of unbiased pruning from being realized. As dependent rounding partially mitigates these issues, we hope it shows promise in increasing the effectiveness of randomized machine learning methods, either by increasing accuracy or decreasing required sampling.

The lottery ticket hypothesis [14] augments neural network pruning by proposing that the original random values used to initialize a neural network retain their importance throughout training. The authors suggest that magnitude weight pruning should be followed by resetting all non-pruned values to their original values and further training, and that repeating this several times can lead to fast training with high levels of pruning. This is an open area of research. It may be possible to replace magnitude weight pruning with rounding-based pruning methods to see if magnitude weight pruning is innate to the lottery ticket hypothesis. It is also unknown how much precision in the original random values is necessary, or if an approximation of the values is also effective (e.g. their sign [15]). According to the authors of the lottery ticket hypothesis, using just the sign breaks down when training large networks, but we believe that this is an underexplored idea.

Chapter 3: Dependent Rounding Speedup

3.1 Original Algorithm

The dependent rounding algorithm described in [2] is not sufficiently fast to be used realistically. We will now describe the original process used for dependent rounding (note that we are using the case where the edges all have values between 0 and 1, as the algorithm can be shifted and scaled to handle any range of values). First, the algorithm finds the set of what we will call floating edges, which are edges that have a non-integer value. A graph representation of floating edges is created. Then, the algorithm will repeatedly find cycles or maximal paths to work with using Depth-First Search (DFS). Any time that a cycle or maximal path P is found, the algorithm will split P into M_1 and M_2, where M_1 is the set of odd-numbered edges and M_2 is the set of even-numbered edges along P. The numbering is arbitrary and is a consequence of the path selected, not innate to the edges. All that matters is that M_1 contains no adjacent edges and the same for M_2. This is done so that we can properly maintain the sum of edge weights incident to each vertex, as each vertex has at most two incident edges, one odd-numbered and one even-numbered: whatever we add to one edge we must subtract from the other. There are now two possibilities for how we modify the edges along the path:

1. Raise the first edge - α - If we raise the first edge's value, need to lower the second edge's

value to maintain the sum incident to the shared vertex. This can be extrapolated along the path to get that all edges in M_1 will be raised and all edges in M_2 will be lowered. In order to get the maximum amount that edges can be changed in this manner (without pushing values above 1 or below 0), α, we get

$$\alpha = \min\{\{1 - x_{i,j} \mid \forall (i,j) \in M_1\}, \{x_{i,j} \mid \forall (i,j) \in M_2\}\} \tag{3.1}$$

2. Lower the first edge - β - In this case, we are lowering all edges in M_1 and raising all edges in M_2. This results in

$$\beta = \min\{\{x_{i,j} \mid \forall (i,j) \in M_1\}, \{1 - x_{i,j} \mid \forall (i,j) \in M_2\}\} \tag{3.2}$$

Given these two possible modifications, we would like to now generate a probability that either of them happens while remaining unbiased. In order to maintain the expected value of each edge, we must have that, given probability p that M_1 is raised and probability $1-p$ that M_1 is lowered, $p\alpha - (1-p)\beta = 0$, resulting in $p = \frac{\beta}{\alpha+\beta}$. After this is done, we know that at least one of the edges will be set to 0 or 1, and so ceases to be floating. This step can be repeated until no floating edges remain and all of the weights are rounded.

Note that the size of P is at most $|A| + |B|$ and therefore the time to perform a rounding step is $O(|A|+|B|)$. There are $|E| \leq |A||B|$ total rounding steps, bringing the total running time to $O(|A||B|(|A|+|B|))$.

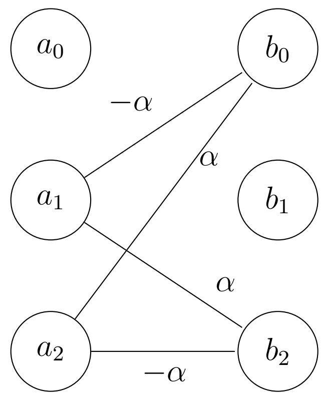

Figure 3.1: Example of rounding butterfly step where the depicted edges show the modifications to each edge value. Note that the sum of the modifications incident to each vertex is 0.

Algorithm 1 Original Dependent Rounding Algorithm

Require: Bipartite graph (A, B, E, w)
Ensure: Rounded bipartite graph (A, B, E, w)
  Floating edges $F \subseteq E$
  while $F \neq \emptyset$ **do**
    $P \leftarrow$ cycle or maximal path in (A, B, F)
    $M_1 \leftarrow$ odd numbered edges in P
    $M_2 \leftarrow$ even numbered edges in P
    $\alpha \leftarrow$ minimum of distances to 1 of all edges in M_1 and distances to 0 of all edges in M_2
    $\beta \leftarrow$ minimum of distances to 0 of all edges in M_1 and distances to 1 of all edges in M_2
    $r \leftarrow$ random number uniformly between 0 and 1
    if $r < \frac{\beta}{\alpha+\beta}$ **then**
      $w_{i,j} \leftarrow w_{i,j} + \alpha$ for $(i, j) \in M_1$
      $w_{i,j} \leftarrow w_{i,j} - \alpha$ for $(i, j) \in M_2$
    else
      $w_{i,j} \leftarrow w_{i,j} - \beta$ for $(i, j) \in M_1$
      $w_{i,j} \leftarrow w_{i,j} + \beta$ for $(i, j) \in M_2$
    end if
    The edge that has been rounded to 0 or 1 is removed from F and E (if 0)
  end while

3.2 New Algorithm

3.2.1 Overview

We have developed a few possible improvements to the dependent rounding scheme:

1. Length of DFS step - Each rounding step can take as much as $O(|A| + |B|)$ time, but cycles can be (and often are) as small as size 4. Therefore, we can speed up dependent rounding by focusing on small cycles. Our new algorithm utilizes this idea by searching for every possible 4-cycle, called butterflies, to round first. Originally inspired by very fast parallel algorithms for counting butterflies [6], this idea can also be parallelized easily so that multiple rounding steps can be performed at the same time.

2. Waste of work - Each rounding step does a partial DFS of the floating edges in the graph, but then completely ignores this gathered data when moving into the next rounding step. We can speed up dependent rounding by doing a full DFS and then reusing as much information as possible between rounding steps. This should be effective as only one edge is removed in each step, and restoring a DFS tree only involves finding one edge to reconnect the split DFS tree from the last iteration. Our new algorithm utilizes this idea by creating an Euler tour and then performing minor changes to it each rounding step. This is especially important when considering that, even though there has been some recent progress on nearly work-optimal DFS [7], actual work-optimal DFS is known to not be amenable to parallelism.

3. Forests are easy to round - We can round away all remaining cycles, leaving only a forest.

Rounding a forest is rather easy, as it consists entirely of maximal paths between leafs, which are easy to find in an Euler Tour. After a single DFS iteration, we can round by searching for a path between any two leafs, performing a rounding step, then splitting the Euler Tour along the edge that is no longer floating.

So, our new algorithm will start by rounding butterflies of floating edges, then will round remaining cycles, then will end by rounding away the remaining forest.

Algorithm 2 New Dependent Rounding Algorithm

Require: Bipartite graph (A, B, E, w)
Ensure: Rounded bipartite graph (A, B, E''', w''')
(A, B, E', w') = Round butterflies (A, B, E, w)
$(A, B, E'', w''), C$ = Round cycles (A, B, E', w')
(A, B, E''', w''') = Round forest $(A, B, E'', w''), C$

3.2.2 Rounding Butterflies

Recall that any 4-length cycle in a bipartite graph is a butterfly. Each butterfly consists of wedges, which are any 2 edges that share a vertex. A lazy way to iterate through every single butterfly is to iterate through all possibilities of each of the 4 vertices. To save work, we can instead start by picking a pair of vertices in A, then iterate through B to find wedges from the pair in A to elements of B (2 of which form a butterfly). This takes us from $O(|A|^2|B|^2)$ checks to $O(|A|^2|B|)$ checks. An additional benefit of using this method is that the sets of edges coming out of two disjoint pairs in A are also disjoint. This lends itself to parallelism, as all we require to round two different cycles at the same time is for the cycles to be edge-disjoint.

Therefore, in order to round all butterflies, we can start by splitting A into disjoint pairs using a method similar to round-robin tournaments, then in parallel (for each pair) iterate through

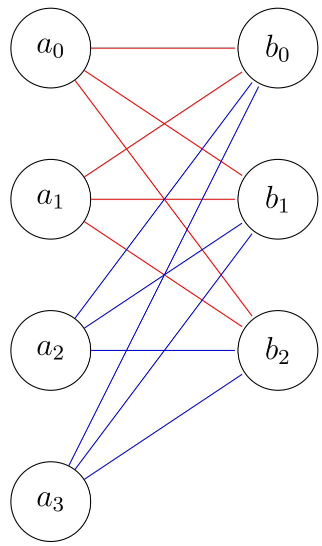

Figure 3.2: Example of how pairing off vertices in A can be used to split edges into disjoint sets. We can then search for butterflies in each set in parallel.

B and search for two wedges. These form a butterfly, which we round, destroying exactly one of the wedges. Then the algorithm will continue to look for another wedge and round until B is exhausted and all possible butterflies with the two chosen vertices in A no longer exist. There will be a total of $|A| - 1$ rounds, each with $|A|/2$ pairs of vertices, resulting in all $\binom{|A|}{2}$ possible pairs being explored. Therefore, we know that all possible butterflies will be explored and removed if they exist.

We also know of an improved method to round all butterflies (specifically those that include two specified vertices from A) that is faster theoretically but not empirically. This consists of enumerating in parallel all wedges to B that exist, then rounding away in parallel pairs of them until only one exists. For a given pair from A, this will take $O(\log |B|)$ time and $O(|B|)$ work. However, this method was empirically slower and likely would require an immense number of processors to be worth considering.

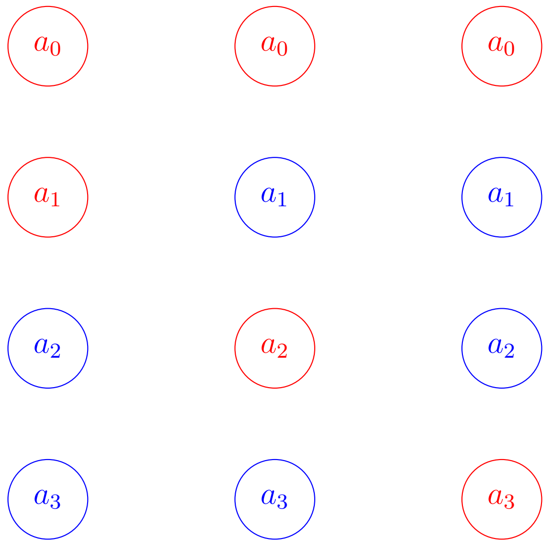

Figure 3.3: Example of how vertices in A will be paired off using a round robin tournament method. Each column is a round.

Algorithm 3 Round Butterflies

Require: Bipartite graph (A, B, E, w)
Ensure: Bipartite graph with no butterflies (A, B, E', w')
  for all $round \in \{1, \ldots, |A| - 1\}$ **do**
    $us \leftarrow$ round robin pairs corresponding to $round$
    for all $pair \in \{1, \ldots, |A|/2\}$ in parallel **do**
      $u1, u2 \leftarrow us[pair]$
      Iterate through $\{1, \ldots, |B|\}$ and continually find first two vertices $v1, v2$ that both have wedges from $u1, u2$
      Do a dependent rounding step on cycle $P = (u1, v1, u2, v2, u1)$
      The edge that has been rounded to 0 or 1 is removed from F
    end for
  end for

3.2.3 Rounding Cycles

Now the floating edges will no longer form any butterflies, but can still form cycles. It is possible to continue by using the original dependent rounding algorithm on the remaining edges. However, in order to save work, we can use an Euler tour to only do DFS once. Euler tours, introduced in [16], are Eulerian cycles (i.e. they see each edge exactly once) of directed edges which are formed by ”splitting” the edges of an otherwise undirected tree. These are created by using any spanning tree algorithm on the graph of floating edges, then finding each directed tree edge’s ”next” edge, and finally going through this list to enumerate the cycle.

Once we have an Euler tour, we can iterate through every non-tree edge. When given one of those edges (i, j), we use the Euler tour to find a path from j to i, trim this path to its shortest length, then perform a dependent rounding step on $P = [j, \ldots, i, j]$. This will remove one edge along the path from the set of floating edges. We must trim the path to the shortest length in order to avoid the case where both of the directed edges corresponding to any undirected edge are in the path (e.g. the path contains both a, b and b, a). If this occurs, it is possible for us to continually both add and subtract the same value to this edge, leading to no actual progress. In order to keep the Euler tour up to date, we must modify it by adding the nontree edge and removing the rounded edge, both constant time and linear work operations. Obviously, if those edges are the same, we do not need to modify the Euler tour at all.

3.2.4 Rounding a Forest

The remaining floating edges will form a forest. Forests are easy to round, using a method that parallelism can easily be applied to. If we take the Euler tour for any tree, we can find a max-

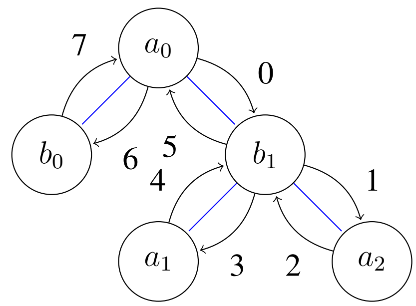

Figure 3.4: Example of an Euler tour (labeled order of black edges) for this tree (blue edges).

Algorithm 4 Round Cycles

Require: Bipartite graph (A, B, E, w)
Ensure: Bipartite graph with no cycles (A, B, E', w'), remaining Euler tour C
  $T \leftarrow$ a spanning tree of the floating edges $F \subseteq E$
  $C \leftarrow$ Euler tour of T
  for all $e = (e_1, e_2) \in F - T$ **do**
    $P \leftarrow$ path in Euler tour from e_1 to e_2
    Trim P to remove detours
    Perform a dependent rounding step on P
    The edge that has been rounded to 0 or 1 is removed from F
    The rounded edge also replaces (e_1, e_2) in C
  end for

imal path by taking the Euler tour between 2 neighboring leaves, and then perform a dependent rounding step on this path, then split the Euler tour along the rounded edge. We will now have 2 Euler tours to which we can repeat this process in parallel until no floating edges remain.

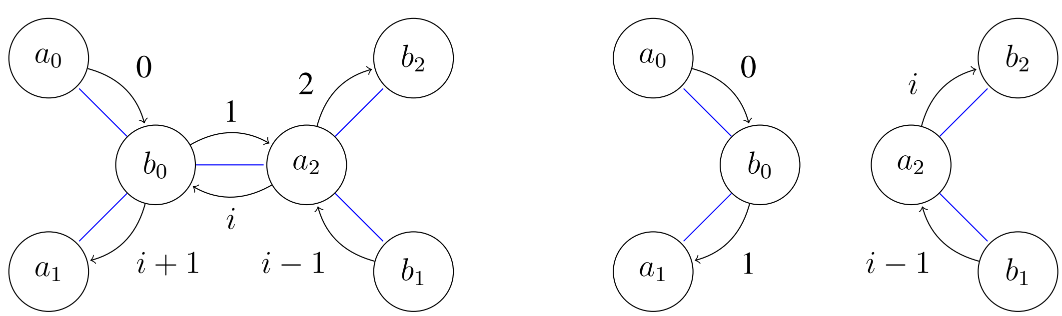

Figure 3.5: Example of an edge deletion in an Euler tour from left to right. The remainder of the Euler tour (2 to $i-2$ and $i+2$ to end) is not shown. Edge addition is the same in reverse.

Algorithm 5 Round Forest

Require: Bipartite graph with no cycles (A, B, E, w), Euler tour C
Ensure: Rounded bipartite graph (A, B, E', w')
  $P \leftarrow$ path between any two leafs in C
  Perform a dependent rounding step on P
  Split C along the edge that has been rounded to 0 or 1 into C_1, C_2
  In parallel: Round Forest $(A, B, E, w), C_1$, and Round Forest $(A, B, E, w), C_2$

3.3 Runtime Analysis

We have seen that the original dependent rounding algorithm runs in time $O(|A||B|(|A| + |B|))$. We can analyze the new algorithm in parts:

1. Butterfly rounding - Each round robin pair runs in $O(|B|)$ time. There are $O(|A|)$ rounds of $O(|A|)$ of these done in parallel, meaning the work is $O(|A|^2|B|)$ and the time is $O(|A||B|)$. The time can be reduced to $O(|A| \log |B|)$ by picking out pairs of wedges to B in parallel, but this is not effective empirically.

2. Cycle rounding - There are at most $O(|A|\sqrt{|B|})$ floating edges remaining after butterfly removal [17]. Finding a spanning tree takes at most $O(|A| + |B|)$ time and $O(|A|\sqrt{|B|})$ work. Euler tour generation is constant time and $O(|A|\sqrt{|B|})$ work. For each edge, it is $O(|A| + |B|)$ time and work (dominated by trimming the path). This is a total of $O(|A||B|\sqrt{|B|})$ time and work.

3. Forest rounding - There are at most $|A| + |B| - 1$ remaining floating edges. Each rounding step takes $O(\log(|A|+|B|))$ time and $O(|A|+|B|)$ work, then splits the remaining problem in two. Therefore, the total is $O(\log^2(|A| + |B|))$ time and $O((|A| + |B|) \log(|A| + |B|))$ work.

The overall time complexity is $O(|A||B|\sqrt{|B|})$ time and $O(|A|^2|B|)$ work, assuming that $\sqrt{|B|} \leq |A| \leq |B|$. This means that we have $\frac{|A|}{|B|}$ times less work and $\frac{|A|}{\sqrt{|B|}}$ times less time when compared to the original algorithm, which can be quite significant for imbalanced graphs. The current pain point is trimming the path in cycle rounding, which, if improved, could result in the time being reduced to $O(|A|\sqrt{|B|}\log|B|)$ at best.

Table 3.1: Theoretical Runtime Analysis. Assumes $\sqrt{|B|} \leq |A| \leq |B|$. Note how time and work complexity are dominated by cycle rounding.

Step	Time Complexity	Work Complexity
Original Algorithm	$O(\|B\|^2\|A\|)$	$O(\|B\|^2\|A\|)$
Rounding Butterflies	$O(\|A\|\log\|B\|)$	$O(\|A\|^2\|B\|)$
Rounding Cycles	$O(\|A\|\|B\|\sqrt{\|B\|})$	$O(\|A\|\|B\|\sqrt{\|B\|})$
Rounding Forest	$O(\log^2(\|A\|+\|B\|))$	$O((\|A\|+\|B\|)\log(\|A\|+\|B\|))$
Total	$O(\|A\|\|B\|\sqrt{\|B\|})$	$O(\|A\|\|B\|\sqrt{\|B\|})$

3.4 Experimental Results

We tested three different dependent rounding schemes: an original naive Python implementation, a second Python implementation based on the new algorithm, and a C++ implementation that uses parallelism and the new algorithm [1]. The parallelism is performed using the C++ library ParlayLib [18]. Each algorithm was run with random 2-D matrices (size $|A| \times |B|$) as input, with $|A|$ and $|B|$ being powers of 2 starting from 8 and going to 16384 unless time/space constraints are hit. Without loss of generality, we only consider cases where $|A| \leq |B|$ (in the case where $|B| > |A|$, the input matrix could be transposed).

[1] All schemes were run on an Ubuntu Windows Subsystem for Linux with an AMD Radeon RX 6750 XT GPU and 13th generation i7-13700K CPU.

First we will analyze the case where $|A| = |B|$. Note that the runtime analysis would indicate that the old algorithm will be $O(|A|^3)$ time and the new algorithm will be $O(|A|^2\sqrt{|A|})$ time and $O(|A|^3)$ work.

1. The results indicate that the old Python implementation takes approximately 10 times as long to run when $|A|$ is doubled, and so is certainly superlinear time (w.r.t. number of edges). It appears to slow down significantly in later iterations as DFS slows down when most of the edges are gone.

2. The new Python implementation takes approximately 5 times as long to run when $|A|$ is doubled, and so is empirically close to linear, but still superlinear w.r.t. edges.

3. The C++ implementation takes approximately 4 times as long to run when $|A|$ is doubled, and so is empirically linear w.r.t. edges.

Now we will look at the case where $|B| = 16384$. We expect from runtime analysis that the old algorithm will be quadratic time and the new algorithm will be linear time and work.

1. The old Python implementation takes approximately 5 times as long to run when $|A|$ is doubled, and so is at least quadratic empirically.

2. The new Python implementation takes almost exactly 2 times as long to run when $|A|$ is doubled, and so is empirically linear.

3. The C++ implementation is very slow for small $|A|$ due to it having higher overhead than the other methods, but soon becomes the fastest method and is just under linear.

As a more concrete example, for $|A| = 128, |B| = 8192$, the old Python implementation takes an average of 5764 seconds to run, the new Python implementation takes an average of

7.716 seconds to run (7.411 to remove butterflies, 0.258 to remove cycles, and 0.047 to round the remainder), and the C++ implementation takes 3.658 seconds to run (0.343 to remove butterflies, 2.495 to remove cycles, 0.82 to round the remainder). This is a 747 times speedup from the old Python method to the new Python method and a 2 times speedup to the C++ method. One easy thing to notice is that the C++ method is much faster at rounding butterflies, but takes much longer to round cycles. This is because the C++ method has a slightly different method of selecting butterflies and always leaves significantly more edges unrounded than the Python method, which will be briefly discussed below.

3.4.1 Importance of Butterfly Removal Order

It became very clear during testing that the order of butterfly scanning was very important for how many edges remained. We found that an older serial algorithm implementation regularly rounded slightly more butterflies, which resulted in the cycle rounding step being much smaller (around 10 edges, as opposed to up to tens of thousands). This is very consequential when noting that the cycle removal step most often takes significantly more work per edge removed than the other steps, both theoretically and empirically. We can see in the results that rounding butterflies dominates the runtime of serial methods while rounding the remaining cycles takes very little time. In comparison, parallel methods take very little time to round butterflies, while the rounding cycles step takes a far longer period of time than in the serial method. Further investigation into why there is such a big difference in remaining edges is warranted. Not only does this have a large impact on runtime, but there are important theoretical distinctions such as differences in concentration bounds that may result from alternate rounding methods. For

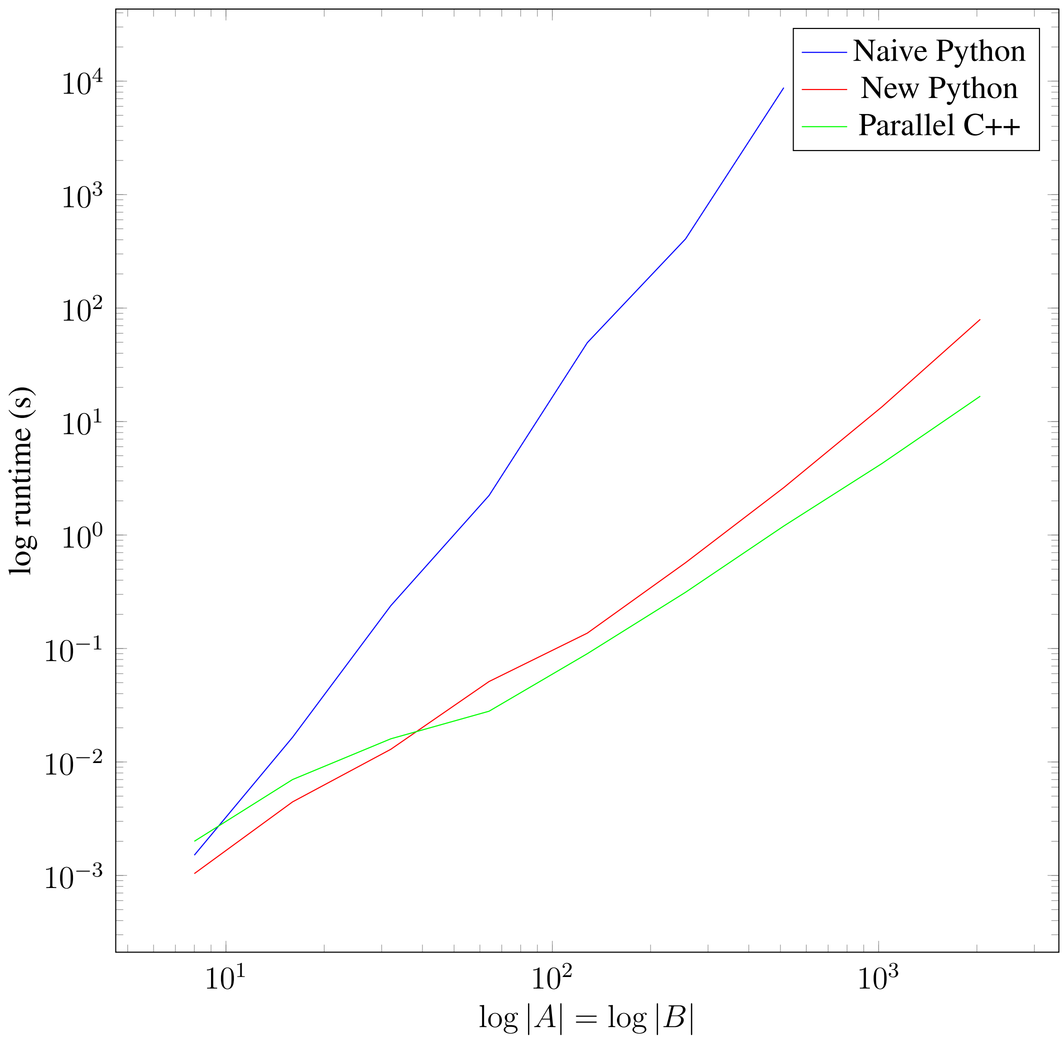

Figure 3.6: Runtime for $|A| = |B|$. Notice that the new python method results in runtime roughly linear with the number of edges. The parallel C++ method does result in some speedup (sublinear empirical runtime).

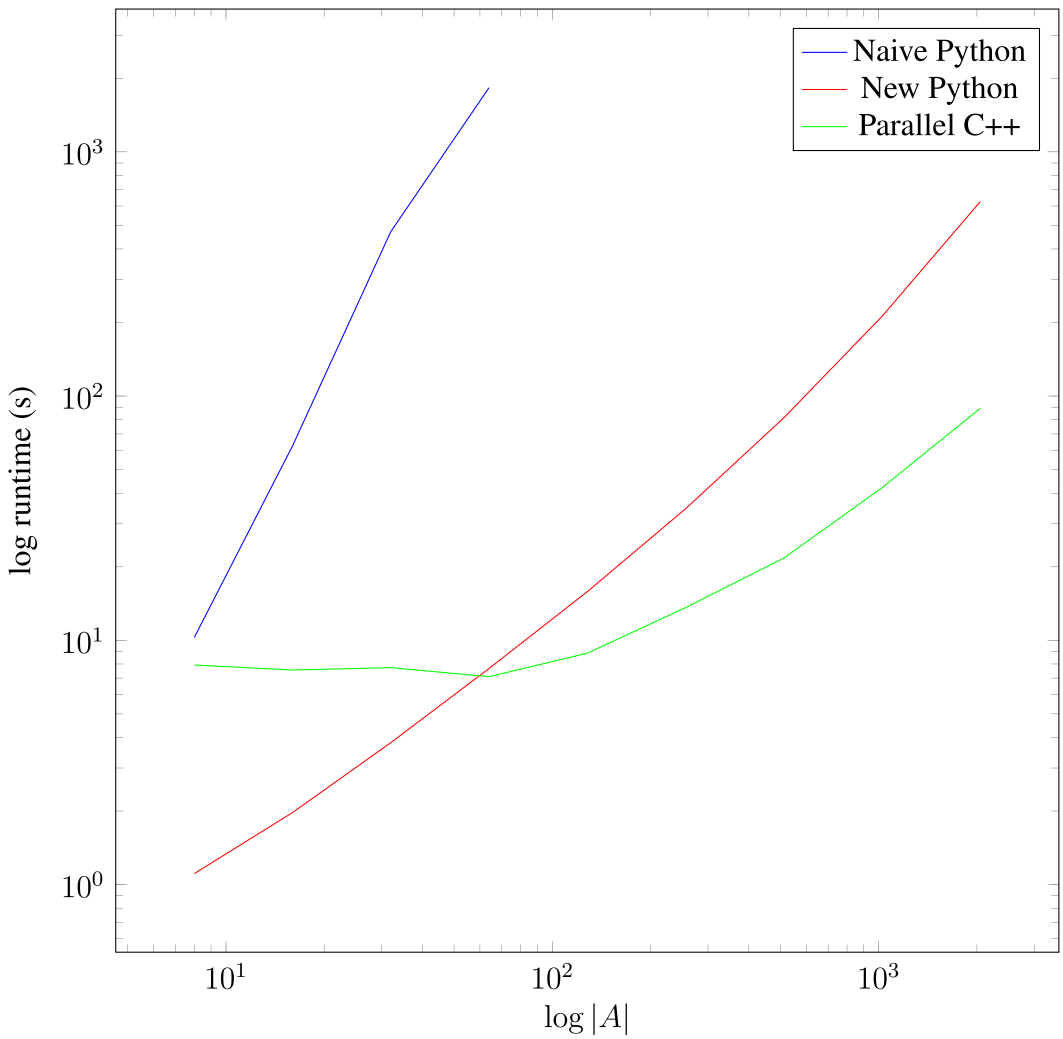

Figure 3.7: Runtime for $|B| = 16384$. The new Python method appears to be just above linear. The C++ implementation has large overhead and so it is unsurprising that speedups do not show until very large size graphs.

example, while dependent rounding always has the negative dependence property, it has been proven that the strongly Rayleigh property [19] holds in the case where dependent rounding is performed on a one-to-many bipartite graph (i.e. one of the partitions is size 1) [20]. It is also possible that more methods of speeding up the cycle rounding stage can be explored in order to alleviate the issue.

Table 3.2: Floating edges removed in cycle step vs implementation type.

	$\|A\|$	$\|B\|$	Edges - no Butterflies	Edges - no Cycles	% Removed in this Step
Serial	32	32	71	63	0.78%
	32	256	293	287	0.07%
	32	2048	2086	2079	0.01%
	32	16384	16426	16415	$< 0.01\%$
	256	256	529	511	$< 0.01\%$
	256	2048	2311	2303	$< 0.01\%$
	256	16384	16652	16639	$< 0.01\%$
	2048	2048	4101	4095	$< 0.01\%$
	2048	16384	18440	18431	$< 0.01\%$
Parallel	32	32	110	63	4.59%
	32	256	409	287	1.49%
	32	2048	2241	2079	0.25%
	32	16384	16588	16415	0.03%
	256	256	1439	511	1.42%
	256	2048	4857	2303	0.49%
	256	16384	22149	16639	0.13 %
	2048	2048	17434	4095	0.31%
	2048	16384	50267	18431	0.09%

Table 3.3: Average running time for different implementations split by steps in seconds. $|A| = 128, |B| = 8192$

	Original Method	Best Serial Method	Best Parallel Method
Rounding Butterflies	N/A	7.411	0.343
Rounding Cycles	N/A	0.258	2.495
Rounding Forest	N/A	0.047	0.820
Total	5764	7.716	3.658

Chapter 4: Applications of Dependent Rounding in Machine Learning

4.1 Motivation

We have two broad methods to apply dependent rounding to machine learning. The first is directly as a rounding method. We believe that, because dependent rounding keeps more information when rounding than stochastic rounding does (via degree preservation), dependent rounding can get improved results in machine learning methods that require maintaining information. An example of such a method is machine learning in low precision regimes, where most information is removed in order to simplify a neural network. With high levels of sparsity or low levels of precision, it is important to keep as much information as possible that is allowed given those constraints.

The second idea is to use dependent rounding as a sampling method. In this case, we take any set of generated probabilities to sample from and use dependent rounding to convert these probabilities to 1's or 0's to determine whether those points will be sampled or not. We know from above that degree preservation, which is inherent to dependent rounding, will be useful to make sampling more consistent. For example, say we have a large set of N data points each with probability $1/N$ of being sampled. Obviously, we could just pick a random number to get one of them with 100% probability, but this method will fail to work for sampling matrices. With stochastic rounding, there is a $(1 - \frac{1}{N})^N$ chance ($\approx \frac{1}{e}$) that every single number is rounded to 0 and no sample remains. In order to move up to two dimensions, we may need to use a method similar to the Fisher-Yates shuffle [21], but this will fail when the distribution is nonuniform.

We also believe that negative dependence is a useful property for rounding in machine learning contexts. Negative dependence leads to values tending to be different from their neighbors, which may manifest as more variety in the edge weights. This is almost certainly superior to positive dependence, where the values all tend to be similar, which can create redundancy and reduce the effective number of weights used (i.e. if the weights coming out of two vertices are very similar vectors, then one of them is likely to not be useful).

Now that we have several methods with which to apply dependent rounding, we need to determine which way to best apply dependent rounding to machine learning weights based on architecture. Whenever the weights for a layer are one- or two-dimensional, dependent rounding is very simple, as any one- or two-dimensional matrix can be converted into a bipartite graph (first dimension is the left partition, second dimension or single vertex is the right partition, values in matrix are the weights of edges between) and then dependent rounding can be applied directly.

The problem grows in difficulty with more complicated architectures. For example, convolutional neural networks (CNN) typically have 4-dimensional weight tensors. CNNs are designed so that the weights map (in a similar manner to standard fully connected layers) from 2-D features to new 2-D features. Because a CNN layer is similar to fully connected layers with vertices replaced with 2-D filters, we believe that it is more appropriate to iterate through each position in the filters and perform dependent rounding as if the remaining two dimensions are a fully connected layer. Alternative options are to perform dependent rounding on each of the filters or flatten the whole tensor into two dimensions and do one dependent rounding step for the whole layer. Other architectures such as feed-forward layers can have odd cycles and therefore would need to be modified for dependent rounding to be used.

4.2 SignSGD

An example of using dependent rounding directly as a method of rounding is to modify SignSGD [8]. SignSGD is a method where instead of doing standard stochastic gradient descent (SGD), the gradient data is taken and rounded to the nearest sign (or remains zero if already zero). The method is typically also momentum-based and uses an increased step size to potentially perform better than SGD in terms of loss and/or variance. What this does is significantly reduce the burden of data transfer, as the change from floats to bits (or perhaps 2 bits) per gradient element can enable distributed learning.

We suggest an improvement to SignSGD where instead of doing standard independent rounding on the SGD data, we will instead try to run dependent rounding, which we will call dependent SGD. We hope that this will, by preserving marginal sums, increase the amount of information kept and increase the convergence rate. Hopefully, the improved convergence bounds inherent to dependent rounding also may be used to decrease the variance of the technique as well.

The extent to which we have studied this problem is the creation of a toy example. In this example, we start at a random N-dimensional vector, receive noisy gradient data, and search for the zero vector by using the magnitude of the vector as loss. The gradient data will be the current vector (as that is the gradient of vector magnitude), but noise is added to emulate stochastic gradient data. We ran this program using several SGD-based methods using the same step size, decreasing with the square root of the step number.

This experiment indicates that dependent SGD can perform slightly better than SignSGD.

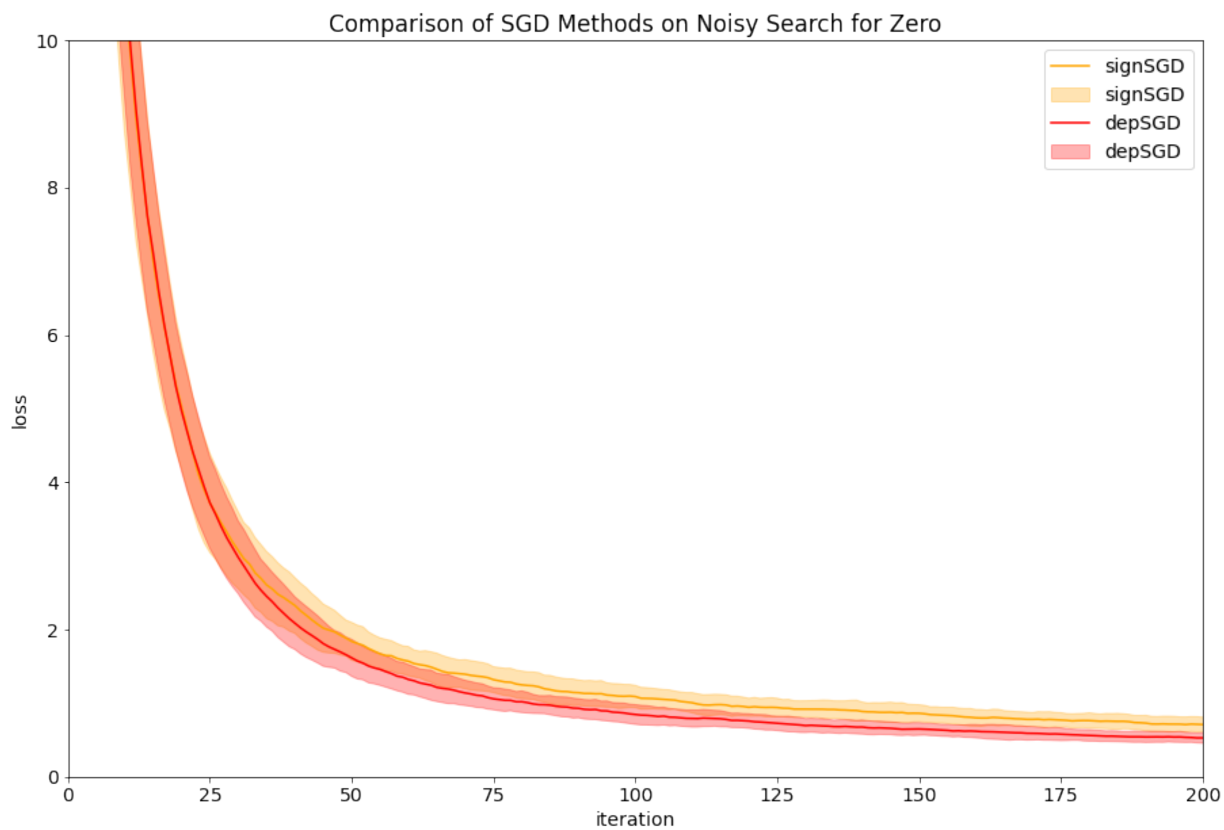

Figure 4.1: This is a comparison of SignSGD to our new dependent SGD method. The graph displays the mean (solid line) and variance (shaded area is ± 1 standard deviation) of both methods. It is obvious in this toy example that DependentSGD is both an improvement in convergence and variance.

We can see 4.1 that the dependent rounding techniques in red led to faster convergence, lower loss, and decreased variance when compared to standard SignSGD. This is a strong empirical indication that dependent SGD used in this context has potential. However, for any sizeable machine learning problem, the runtime of dependent rounding will still result in dependent SGD taking a longer time to get to the same loss as SignSGD. That is unless the gradient is difficult to calculate. Also, we must note that this is a simple one-dimensional example, where the function is simple, smooth, and concave. So, it is possible that dependent rounding causes the optimization problem to settle and remain in local minima. In conclusion, we believe that SignSGD has promise in improving optimization techniques, especially ones where gradient calculation is difficult.

4.3 Lottery Ticket Hypothesis

Another rounding method we explored is applying dependent rounding to the lottery ticket hypothesis. The idea behind the lottery ticket hypothesis is that when selecting random initial weights, there is some special pruned subnetwork of these weights that is all that is needed to reach comparable accuracy after some training. So, the original paper [14] proposed the following steps:

1. Initialize weights w_0
2. Train the neural network (not necessarily to convergence) to get new weights w_1
3. Generate a pruning mask based on the weights w_1 (originally just setting the smallest specific percentage of the weights w_1 to zero)

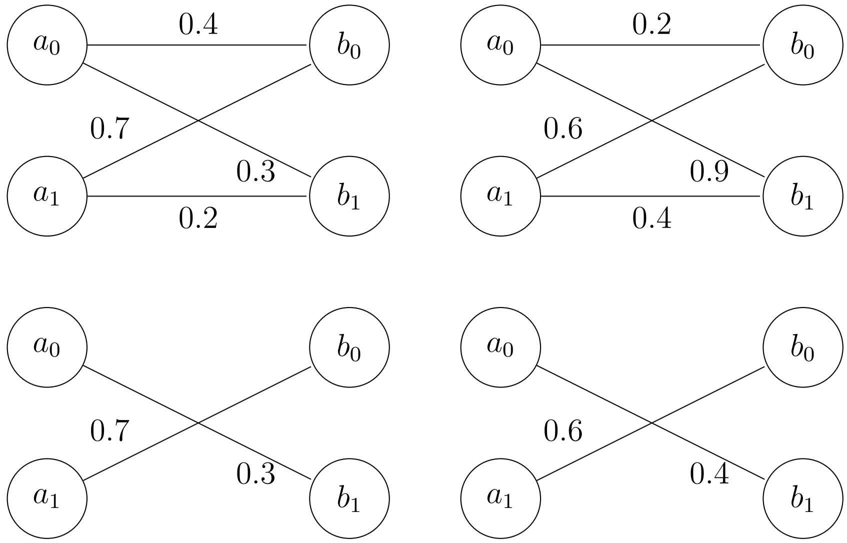

Figure 4.2: Toy example of lottery ticket hypothesis steps. Step 1 (top left): random initialization. Step 2 (top right): post training values. Steps 3 and 4 (bottom left): bottom two post training values removed and top two values replaced with original values. Step 5 (bottom right): continued training.

4. Replace the remaining nonzero values of w_1 with their original values w_0

5. Continue training to get weights w_2

This process can be done repeatedly or in one shot. The result is that the neural network with weights w_2 is comparable in accuracy to w_1 but contains significantly fewer nonzero weights and therefore has reduced storage complexity and increased training speed. The end goal of this idea is to iteratively train and prune a network until there are very few weights remaining, all at a speed comparable to simply training the full network.

It has also been suggested that we do not need to keep the original weights w_1, but only the signs of the original weights $sign(w_1)$ [15]. This has a high success rate on small neural networks, but the original authors of the lottery ticket hypothesis suggest that the idea breaks down for larger problems and the original weights w_1 are necessary to achieve similar accuracy. We propose that there may be a way to interpolate this idea: on small networks, we only need

$sign(w_1)$, on big networks we need w_1, but in between, it may be possible to reach optimal accuracy with levels of precision in between $sign(w_1)$ and w_1. In other words, we would like to determine if there is a relationship between the size of the network and the level of precision necessary to maintain accuracy. We would also like to see what the effects of different types of rounding are, to see if dependent rounding can improve the process. To that effect, we propose two modifications to the above steps:

- Instead of weight magnitude pruning in step 3, attempt alternate methods of pruning (perhaps dependent rounding-based)
- Instead of using w_1 or $sign(w_1)$ in step 4, use several rounding methods to round w_1 to varying powers of two (e.g. $2, 1, \frac{1}{2}, \frac{1}{4}, \dots$)

We have some preliminary results that indicate that dependent rounding is better than both stochastic independent rounding and standard independent rounding, especially at low levels of precision. We used the second modification to the lottery ticket hypothesis described above and attempted standard independent, stochastic, and dependent rounding to see the effects. The results indicate that on average, dependent rounding has the best performance, which is unsurprising as more data is preserved via the marginal sums.

We can see that stochastic and standard rounding both trend towards an extremely low accuracy (the equivalent of guessing 0 for all of the labels) due to layer collapse, where most or all of a layer's weights are set to 0, which removes the effect of almost all nodes. The lowest level of precision seen here is where the values are rounded to ± 1, except in the case of stochastic rounding, which used a different method to showcase its tendency towards layer collapse. Because both dependent and standard rounding round the values to ± 1 and never 0, many more weights

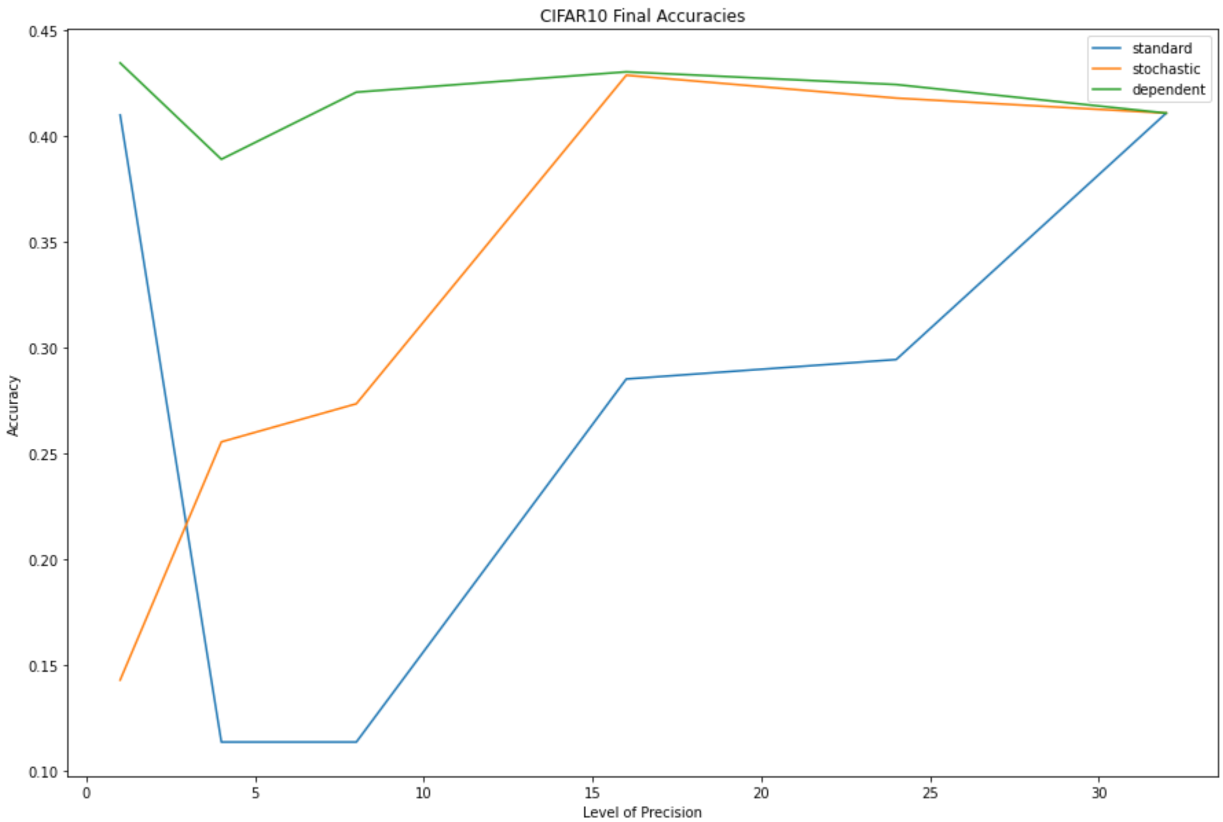

Figure 4.3: These are the preliminary results on the CIFAR10 imageset for dependent rounding applied to the lottery ticket hypothesis. The y axis indicates accuracy and the x axis indicates the level of precision, which is roughly the number of bits which would be used to represent an edge post-rounding. We can see that dependent rounding dominates both stochastic and standard rounding. The lowest level of precision is an anomaly because it is where weights were rounded to ± 1, whereas values could be rounded to zero for higher precision levels.

are still left to be trained afterward, leading to a higher accuracy when compared to higher levels of precision. Note that this is not a fair comparison due to the pruning present at other levels of precision, so we can think of it more as a baseline with no additional edges removed.

A similar idea we had is an experiment to see how dependent rounding at different levels of precision applied directly to the post-training values affects the accuracy. This is similar to the lottery ticket hypothesis idea, but steps 3 and 4 are replaced with only a single rounding step.

4.4 Dependent Sampling

Now we will briefly discuss our ideas on sampling problems that can be tied to dependent rounding. Again we must keep in mind the important properties of degree preservation and negative dependence. We believe that dependent rounding can serve as an improved method of sampling from bipartite graphs.

As a brief example, let V be a vector size N where each member is sampled with probability $\frac{1}{N}$. If we do independent stochastic rounding, there is a chance that the entire vector is eliminated, leading to layer collapse. This chance is $(1 - \frac{1}{N})^N \to \frac{1}{e}$, which is far from insignificant. When layer collapse happens, a neural network layer (and therefore the entire network) will return 0 for all values regardless of input, reducing the result of the neural network to a constant function. Note that this problem can be easily circumvented by instead comparing a random value between 0 and 1 to the prefix sums of the sampling probabilities, and picking the appropriate single point where the value is between two prefix sums (in the simple example, pick only index i when the random value is between $\frac{i}{N}$ and $\frac{i+1}{N}$). However, this method fails in the two-dimensional case. If we choose to pick one partition and iterate through the vertices in that partition, sampling one edge between the current vertex and the other partition. If we have a bipartite graph where both partitions are size N, and all edges are sampled with probability $\frac{1}{N}$, there is a $(1-\frac{1}{N})^N$ chance for each vertex in the second partition to no longer have an incident edge.

Another way to entirely avoid the possibility of layer collapse is to use dependent rounding on the sampling probabilities. In the one-dimensional case, this is just a slower equivalent to the method above, as we are guaranteed to sample exactly one value using both methods. However, the real strength of dependent rounding lies in the two-dimensional case, where every single vertex is guaranteed to have exactly one incident edge remaining. This means that vertices will not be randomly dropped from relevance and there is very precise control over edge pruning. For each vertex, we can guarantee any integer number of edges will remain incident after sampling.

Therefore, dependent rounding can be used to consistently sample edges from a bipartite graph of probabilities in one step. What we can do with this is take any method of generating probabilities for pruning machine learning, and replace the sampling method with dependent rounding. For example, there are several coreset-based algorithms [11, 12] that probabilistically prune the weights of a graph based on their potential impact on the result of a neural network. To sample, the probabilities are normalized and then a single value is sampled m times. We can replace this with dependent rounding by multiplying the normalized probabilities by m and doing a single rounding step. This allows for higher probability edges to be sampled multiple times (and it is possible that edges with a high enough probability are guaranteed to be selected), while sampling in exactly one step.

We have noticed earlier work indicates that weight magnitude pruning and related methods are generally better performers than any randomized method of pruning [5]. This is surprising given the convenient properties of probabilistic methods such as unbiased rounded variables. We hope that the performance gap is at least partially due to the shortfalls of the sampling methods (especially gradient collapse and inconsistency with high levels of pruning) and that future work can be done to improve probabilistic methods to the point where they are at least competitive with weight magnitude pruning. Dependent rounding and/or other methods that enforce more consistency may be the key to making randomized pruning methods feasible.

Chapter 5: Conclusion and Next Steps

We have achieved a speedup on the dependent rounding of several orders of magnitude. Based on new techniques alone, the new algorithm can have a performance improvement of more than 100x. Additional speedup from parallelism is as much as 5x. This is a massive speedup that results mostly from rethinking the problem to use small cycles whenever possible. There is still some work to be done on improving the algorithm, briefly summarized below:

- The cycle rounding step is slow and there are likely some improvements in the area.
- The number of cycles remaining after removing butterflies varies significantly depending on how an algorithm searches for butterflies (fewer cycles remaining is preferable). Further investigation is needed to create a fast algorithm with as few cycles remaining as possible.
- We may also be able to explore how much of an impact the original sparsity of the bipartite graph has on runtime, and if there are ways to improve performance conditioned on that sparsity. Tooling the algorithm to be able to use sparse matrix representations may also be possible.
- There is no current method for the algorithm to handle certain size graphs without exhausting memory. It should be possible to segment the graph and cache data in order to handle larger graphs than are currently possible.

We also have presented some of the possible applications of dependent rounding to machine learning techniques, with some support, as follows:

- Modifying SignSGD by using dependent rounding on the gradient.
- Applying dependent rounding to the starting weights for the lottery ticket hypothesis.
- Using dependent rounding as a method of sampling by rounding sampling probabilities produced by machine learning pruning techniques.

We hope that this research transitions the use of dependent rounding from a theoretical technique to a functioning algorithm.

Bibliography

[1] Armen S Asratian, Tristan MJ Denley, and Roland Häggkvist. *Bipartite graphs and their applications*, volume 131. Cambridge university press, 1998.

[2] Rajiv Gandhi, Samir Khuller, Srinivasan Parthasarathy, and Aravind Srinivasan. Dependent rounding and its applications to approximation algorithms. *Journal of the ACM (JACM)*, 53(3):324–360, 2006.

[3] Jarosław Byrka, Thomas Pensyl, Bartosz Rybicki, Aravind Srinivasan, and Khoa Trinh. An improved approximation for k-median and positive correlation in budgeted optimization. *ACM Transactions on Algorithms (TALG)*, 13(2):1–31, 2017.

[4] Anupam Gupta and Jochen Könemann. Approximation algorithms for network design: A survey. *Surveys in Operations Research and Management Science*, 16(1):3–20, 2011.

[5] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[6] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations, 2020.

[7] Mohsen Ghaffari, Christoph Grunau, and Jiahao Qu. Nearly work-efficient parallel dfs in undirected graphs. *arXiv preprint arXiv:2304.09774*, 2023.

[8] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signsgd: Compressed optimisation for non-convex problems, 2018.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[10] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9194–9203, 2018.

[11] Cenk Baykal, Lucas Liebenwein, Igor Gilitschenski, Dan Feldman, and Daniela Rus. Data-dependent coresets for compressing neural networks with applications to generalization bounds, 2019.

[12] Ben Mussay, Daniel Feldman, Samson Zhou, Vladimir Braverman, and Margarita Osadchy. Data-independent structured pruning of neural networks via coresets, 2020.

[13] Ari S. Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers, 2019.

[14] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[15] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. *Advances in neural information processing systems*, 32, 2019.

[16] Robert Endre Tarjan and Uzi Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium onFoundations of Computer Science, 1984.*, pages 12–20. IEEE, 1984.

[17] Steven Roman. A problem of zarankiewicz. *Journal of Combinatorial Theory, Series A*, 18(2):187–198, 1975.

[18] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib-a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 507–509, 2020.

[19] Nisheeth K Vishnoi. Zeros of polynomials and their applications to theory: a primer. In *FOCS 2013 Workshop on Zeros of Polynomials and their Applications to Theory*, pages 1–18, 2013.

[20] Petter Brändén and Johan Jonasson. Negative dependence in sampling. *Scandinavian Journal of Statistics*, 39(4):830–838, 2012.

[21] Richard Durstenfeld. Algorithm 235: random permutation. *Communications of the ACM*, 7(7):420, 1964.