Parallel Shortest Paths with Negative Edge Weights

Nairen Cao
nairen@ir.cs.georgetown.edu
Georgetown University
Washington D.C., USA

Jeremy T. Fineman
jfineman@cs.georgetown.edu
Georgetown University
Washington D.C., USA

Katina Russell
katina.russell@cs.georgetown.edu
Georgetown University
Washington D.C., USA

ABSTRACT

This paper presents a parallel version of Goldberg's algorithm for the problem of single-source shortest paths with integer (including negatives) edge weights. Given an input graph with n vertices, m edges, and integer weights $\geq -N$, our algorithms solves the problem with $\tilde{O}(m\sqrt{n}\log N)$ work and $n^{5/4+o(1)}\log N$ span, both with high probability. Our algorithm thus has work similar to Goldberg's algorithm while also achieving at least $m^{1/4-o(1)}$ parallelism. To generate our parallel version of Goldberg's algorithm, we solve two specific distance-limited shortest-path problems, both with work $\tilde{O}(m)$ and span $\sqrt{L}\cdot n^{1/2+o(1)}$, where L is the distance limit.

CCS CONCEPTS

• **Theory of computation** → **Shortest paths**; **Parallel algorithms**.

KEYWORDS

Parallel algorithm; shortest paths.

ACM Reference Format:
Nairen Cao, Jeremy T. Fineman, and Katina Russell. 2022. Parallel Shortest Paths with Negative Edge Weights. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3490148.3538583

1 INTRODUCTION

This paper presents a parallel algorithm for the single-source shortest paths (SSSP) problem on directed graphs with integer weights (both positive and negative). Here the input comprises a directed graph $G = (V, E)$, an integer edge-weight function $w : E \to \mathbb{Z}$, and a source vertex $s \in V$. The goal is to either determine that the graph contains a negative-weight cycle, or to output for each vertex $v \in V$ the shortest-path distance from s to v.

The classic sequential algorithm for SSSP with general weights is the Bellman-Ford algorithm [11], which has running time $O(nm)$ on a graph with n vertices and m edges. To date, this algorithm is the best known that tolerates (negative) real edge weights.

Throughout the paper, we use n to denote the number of vertices and m to denote the number of edges in the graph. We also adopt the soft-O notation $\tilde{O}$ to hide logarithmic factors. This notation is convenient when focusing on polynomial improvements, especially in parallel algorithms as then the specific parallel model variant does not matter—there are simulation results across many parallel models that incur only logarithmic overheads [17]. Moreover, many of the black-box results we leverage also use this notation.

For integer edge weights, there are several more efficient sequential alternatives to the Bellman-Ford algorithm. This paper focuses on Goldberg's algorithm [16], which has sequential running time $\tilde{O}(m\sqrt{n}\log N)$, where $-N$ is the most negative weight. Goldberg's algorithm is appealing because it uses a scaling technique coupled with relatively simple combinatorial algorithms designed specifically for the shortest-path problem. There are several other alternatives [2, 10, 22] relying on more sophisticated continuous-optimization methods. Some of these algorithms do achieve better bounds depending on graph density, i.e., $\tilde{O}((m + n^{3/2})\log W)$ [22] and $\tilde{O}(m^{4/3+o(1)}\log W)$ [2], where W is the largest absolute value of edge weights, but the techniques seem even more difficult to parallelize. All of these algorithms have running times depending logarithmically on the magnitude of edge weights.

Since the completion of our work on this paper, Bernstein et al. [3] have very recently discovered an algorithm for SSSP with integer weights that has $\tilde{O}(m\log W)$ sequential running time. This algorithm's running time is so good that it subsumes our main result. Nevertheless, we believe that the core distance-limited problems we solve are interesting.

Parallel algorithms. The Bellman-Ford algorithm has the advantage that it is trivial to parallelize to achieve moderate parallelism. In the binary-forking model [5], a straightforward parallel version of this algorithm has work $O(mn)$ and span $O(n\log n)$, where the ***work*** is defined as the total number of instructions executed across all processors, and the ***span*** is the length of the critical path (i.e., the length of the longest chain of sequential dependencies).[1] Although a span of $O(n\log n)$ indicates a high degree of sequential dependency in the algorithm, the work is so large that the algorithm still exhibits significant parallelism. In particular, parallel Bellman-Ford has $\Theta(n/\log n)$ parallelism, where the ***parallelism*** is defined as work over span.

The key shortcoming of parallel Bellman-Ford is that it is not work efficient as compared to any of the more-efficient sequential alternatives for integer weights [2, 10, 16, 22]. But thus far, there is no nontrivial parallelization algorithm for any of those.

1.1 Main result

The main problem we set out to solve is to parallelize Goldberg's algorithm. We provide an algorithm that solves the integer-weight (negative and positive) SSSP problem with work $\tilde{O}(m\sqrt{n}\log N)$

[1] Span is commonly called "depth" or "parallel time" in the parallel algorithms literature.

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07…$15.00
https://doi.org/10.1145/3490148.3538583

and span $n^{5/4+o(1)} \log N$, both with high probability.[2] The work of our algorithm matches Goldberg's algorithm to within logarithmic factors. When the edge weights are not too negative, e.g., all at least $-n^{O(1)}$, the work of this algorithm is nearly $\sqrt{n}$-times lower than Bellman-Ford. Much like parallel Bellman-Ford, the algorithm is still quite sequential. But it also exhibits a moderate level of parallelism; specifically the parallelism is at least $m/n^{3/4+o(1)} \geq m^{1/4-o(1)}$. We thus achieve a polynomial level of parallelism without increasing the work too much beyond Goldberg's sequential algorithm [16].

There are fundamental limits to how low a span we can hope to achieve when parallelizing Goldberg's algorithm. We provide an overview of Goldberg's algorithm in Section 5, but suffice it to say that the high-level algorithm comprises $O(\sqrt{n} \log N)$ inherently sequential iterations. The only parallelism that can be achieved in Goldberg's algorithm is thus within each iteration. But each iteration involves solving directed-graph problems that are at least as hard as (and possibly much harder than) single-source reachability. Because the current-best $\tilde{O}(m)$-work algorithm for single-source reachability has span $n^{1/2+o(1)}$ [18], we cannot hope to achieve a span better than $n^{1+o(1)}$ for a low-work parallel version of Goldberg's algorithm. Our solution with $n^{5/4+o(1)}$ span falls a bit short of this optimistic target, but this is not surprising as the core problems do indeed seem significantly harder than single-source reachability.

1.2 Background on other SSSP problems

Most classic SSSP problems that can be solved sequentially in $\tilde{O}(m)$ time are still effectively open problems in parallel algorithms. Notably, consider the problem of SSSP with nonnegative weights. This problem can be solved sequentially in $\tilde{O}(m)$ time using Dijkstra's algorithm [11], but there is yet no parallel algorithm that achieves both $\tilde{O}(m)$ work and span sublinear in n. Thus for sparse graphs, the only nearly work-efficient solutions are effectively sequential. There are several parallel solutions for this problem, but they all tradeoff higher work for lower span, for example: (1) parallel versions [7, 12] of Dijkstra's algorithm have $\tilde{O}(m)$ work and $\tilde{O}(n)$ span, which are effectively sequential for sparse graphs (2) solutions that involve repeated squaring of the distance matrix [17] have $\tilde{O}(n^3)$ work and polylogarithmic span, and (3) Klein and Subramanian's algorithm [19] for integer weights from $\{0, 1, \ldots W\}$ has $\tilde{O}(m\sqrt{n} \log W)$ work and $\tilde{O}(\sqrt{n} \log W)$ span. In fact, even for the simplest case of *unweighted* and *undirected* graphs, we are not aware of any solution with $\tilde{O}(m)$ work and span sublinear in n.

In light of this difficulty, most progress on parallel SSSP with nonnegative weights is with respect to relaxed versions of the problem. Most notably, there has been significant progress on *approximate* SSSP (ASSSP), where the goal is to output for each vertex a distance estimate that falls between the true shortest-path distance $d(s, v)$ and $(1 + \epsilon)d(s, v)$. (Note the ASSSP problem always refers to nonnegative weights.) For directed graphs, the ASSSP problem had remained open until recently when Cao et al. [8] gave an algorithm that has $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span, with high probability, assuming for conciseness that ϵ is a constant and the maximum ratio of strictly positive edge weights is $n^{O(1)}$. For *undirected* graphs, the approximate problem has a much longer history [1, 9, 13, 14, 20, 21], and the best solutions [1, 20] have $\tilde{O}(m)$ work and polylogarithmic span when subject to the same assumptions on ϵ and weights.

An alternate natural relaxation of the problem is to consider distance-limited SSSP. Here the goal is to output the exact shortest-path distance only to those vertices that are not too far from the source. The only interesting distance-limited variant we are aware of is for unweighted graphs, where the problem can be solved by parallel breadth-first search (BFS). In particular, it is straightforward to parallelize breadth-first search to achieve $\tilde{O}(m)$ work and $O(L \log n)$ span and to correctly output the exact distances to all vertices with distance at most L from the source.

1.3 Our technical contributations

This paper provides parallel algorithms for two distance-limited SSSP problems. These distance-limited solutions represent the main technical contribution of the paper, and we solve the main problem of integer SSSP with negative weights by reducing to these distance-limited problems with distance limit $L = O(\sqrt{n})$.

Distance-limited SSSP with nonnegative integer weights. For this variant of the problem, the edge weights are all nonnegative integers, but the problem is still nontrivial even if all weights are from $\{0, 1\}$. The goal is to return the correct shortest-path distance to all vertices having shortest-path distance $\leq L$ from the source, where L is part of the input to the problem. Moreover, the algorithm should also identify which vertices have shortest-path distance strictly more than L from the source.

Section 4 presents our parallel algorithm for distance-limited SSSP with nonnegative integer weights. We solve the problem assuming $L = O(n)$, as otherwise parallel Dijkstra's [7, 12] is more efficient. Our algorithm has $\tilde{O}(m)$ work and $\sqrt{L} \cdot n^{1/2+o(1)}$ span, with high probability. When $L = O(\sqrt{n})$, the span is thus $n^{3/4+o(1)}$.

We note that the main difficulty in this problem arises from the presence of 0-weight edges mixed with positive weights, as paths through 0-weight edges do not add to the distance. Without the 0s, it is not too hard to solve the problem even more efficiently using a generalization of parallel BFS.

Distance-limited DAG SSSP with weights from $\{0, -1\}$. For this variant of the problem, the input graph is a directed acyclic graph (DAG), and all weights are taken from $\{0, -1\}$. The goal is to return the correct shortest-path distance to all vertices having distance $\geq -L$ from the source, where L is specified in the input. Moreover, the algorithm should also identify which vertices have shortest-path distance strictly less than $-L$ from the source. (An equivalent formulation would be the problem of single-source *longest* paths on DAGs with $\{0, 1\}$ weights.)

Section 3 presents our parallel algorithm for this DAG SSSP problem. Our algorithm has $\tilde{O}(m)$ work and $\sqrt{L} \cdot n^{1/2+o(1)}$ span, with high probability. For $L = O(\sqrt{n})$, the span simplifies to $n^{3/4+o(1)}$.

1.4 Outline

Section 3 gives our parallel algorithm for distance-limited DAG SSSP, and Section 4 gives our solution for distance-limited SSSP with nonnegative integer weights. Section 5 gives an overview of Goldberg's algorithm, and Section 6 gives an overview of our parallel version. Section 6 also explains how to reduce the core

[2] The little-o terms throughout this paper are all $O(1/\log \log n)$. These terms are inhereted from several parallel subroutines [8, 18] that we apply as a black box.

problem solved by Goldberg's algorithm, namely finding a "$\sqrt{k}$-improvement", to the two distance-limited SSSP problems discussed above.

We emphasize that the bulk of the technical contribution of this work appears in Sections 3 and 4, which are described as self-contained problems in their respective sections. Sections 5 and Sections 6 are only necessary if the reader wishes to understand how these subroutines suffice to parallelize Goldberg's algorithm.

2 DEFINITIONS AND PRELIMINARIES

The soft-O notation is defined as follows. We say that a function $g(n) = \tilde{O}(f(n))$ if there exists a constant k such that $g(n) = O(f(n)\log^k f(n))$. When we say that an algorithm achieves some performance $O(f(n))$ ***with high probability***, we mean the following: for any particular choice of constant $c > 0$, with probability at least $1 - 1/n^c$ the algorithm achieves performance $O(f(n))$.

Consider a directed graph $G = (V, E)$. For any subset $V' \subseteq V$ of vertices, we use $G[V']$ to denote the vertex-induced subgraph of G. A ***path*** is a sequence of vertices joined by edges; we sometimes refer to the path by the sequence of vertices, and sometimes by the edges, depending on what is more convenient. In general, paths need not be simple; vertices may repeat. A ***cycle*** is a path that starts and ends at the same vertex and has at least one edge. A directed graph with no cycles is a ***directed acyclic graph (DAG)***. We say that a node u is an ***ancestor*** of v, and conversely that v is a ***descendant*** of u, if there is a directed path from u to v in G. We also say that u ***can reach*** v. Every node is an ancestor and descendant of itself. We use $Anc(v)$ and $Des(v)$ to denote the set of all nodes that are ancestors or descendants, respectively, of v.

For the following, consider a weighted directed graph $G = (V, E)$ with a function $w : E \to \mathbb{Z}$ providing edge weights. For a path $\Gamma = \langle v_0, v_1, \ldots, v_k\rangle$, the ***length*** of Γ is given by $w(\Gamma) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$, i.e., the sum of the weights of edges in the path. If Γ is a cycle (i.e., $v_0 = v_k$) and $w(\Gamma) < 0$, then we call Γ a ***negative-weight cycle***. For a pair of nodes $u, v \in V$, the ***shortest-path distance from u to v*** is the minimum length over all paths that start at u and end at v. We use $dist_w(u, v)$ to denote this shortest-path distance with respect to the weight function w. When the weight function w is clear from context, we simply write $dist(u, v)$. If there is no u-to-v path, then we define $dist(u, v) = \infty$. Similarly, if there is no shortest path (which only occurs if there is a negative-weight cycle), then $dist(u, v) = -\infty$.

Key parallel subroutines

We use solutions to the following two problems as a black box. We define the ***multisource reachability problem*** as follows. The input includes a directed graph $G = (V, E)$ and a set $S \subseteq V$ of source vertices. The goal is to compute for each vertex $v \in V$ just one source vertex that can reach it, or to conclude that none can. That is, output a function $\pi : V \to (V \cup \{\perp\})$ such that $\pi(v) \in S \cap Anc(v)$ if $S \cap Anc(v)$ is nonempty, and $\pi(v) = \perp$ otherwise. We emphasize that v need only know one source that can reach it, not all of them. We note that all of the parallel "shortcutting"-based algorithms for ***single-source reachability***, which is the case that $|S| = 1$, can trivially be extended to solve this version of the multisource problem. This is because these algorithms employ parallel BFS, which can be augmented to send information (e.g., a relevant source) along edges as vertices are discovered. The current best solution for shortcut-based single-source reachability, and hence also multisource reachability, is Jambulapati et al.'s [18] algorithm having $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span, with high probability.

The ***approximate single-source shortest paths (ASSSP) problem*** takes as input a directed graph $G = (V, E)$ and nonnegative edge-weight function w, a source vertex s, and an approximation parameter $\epsilon > 0$. A ***distance overestimate*** is a function d' such that $d'(v) \geq dist(s, v)$. The ASSSP problem is to compute a distance overestimate such that $d'(v) \leq (1 + \epsilon)dist(s, v)$ for all v. Cao et al. [8] provide an algorithm that, for the case of constant ϵ and integer weights smaller than $n^{O(1)}$ that we apply in this paper, solves this problem with $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span, with high probability. Note that the success of the algorithm is also with high probability; the output always satisfies $d'(v) \geq dist(s, v)$, but with small failure probability (at most $1/n^{\Theta(1)}$), it is possible that $d'(v) \not\leq (1 + \epsilon)dist(s, v)$.

3 SHORTEST PATHS ON ACYCLIC GRAPHS WITH {0, -1} EDGE WEIGHTS

This section gives our algorithm for distance-limited DAG shortest paths. Specifically, given a directed acyclic graph $G = (V, E)$ with edge weights in $\{0, -1\}$, source node $s \in V$, and distance limit L, the algorithm computes shortest-path distances to nodes with distance at most L from s. In particular, it outputs a distance value $d(v)$ where $d(v) = dist(s, v)$ if $d(v) \geq -L$, and $d(v) = -\infty$ otherwise. In addition, we also output for each node v a negative edge $parent(v) = (x, y)$ that satisfies the following, if one exists: (1) $w(x, y) = -1$, (2) $dist(s, x) = dist(s, v) + 1$, and (3) there is a path from y to v.

We assume throughout this section that all vertices in the graph are reachable from s, which is without loss of generality as reachability can be solved more efficiently than this problem [18].

Definitions. We call an edge (x, y) a ***negative ancestor edge*** of v if the following hold: (1) $w(x, y) < 0$, and (2) there is a directed path from y to v. If (x, y) is a negative ancestor of v, then we also call x ***negative originator*** for v.

3.1 Overview

At a high level, the algorithm is a peeling algorithm proceeding in rounds $0, 1, \ldots, L$. Round i identifies the set of vertices with distance exactly $-i$ and removes them from the graph. The challenge is to identify the set of nodes to peel by an efficient parallel algorithm.

First, consider a natural inefficient algorithm: (1) Identify the set $S \subseteq V$ of negative vertices, i.e., those with incoming negative edges. (2) Run multisource reachability with sources S, thereby identifying all vertices having a negative ancestor. (3) All vertices *not* reached in the reachability step have distance $-i$; remove them, and all of their incident edges, from the graph. An indirect approach like this seems necessary because the weights are negative; the problem is equivalent to finding *longest* paths in DAGs with weights $\{0, 1\}$.

The main problem with this algorithm is that it is not efficient, as each execution of multisource reachability has $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span [18], and we cannot afford to multiply these by L.

Our algorithm follows the same peeling approach, but we do not recompute reachability on the entire graph in each round. Instead, for each vertex v we maintain $label(v)$, which corresponds to a negative ancestor edge (u, x) if at least one exists. While u remains in the graph, v cannot be peeled. We thus do not include v in any reachability steps until u disappears. Eventually, u is peeled from the graph, rendering $label(v)$ invalid. Our "propagate" algorithm applies multisource reachability to the subgraph of such invalid vertices and finds new negative ancestors, restoring the invariant.

We now give an overview of the analysis. Imagine for the sake of argument that v always finds a negative ancestor uniformly at random. Then it is fairly easy to see that v's label changes $O(\log n)$ times. (Roughly a constant fraction of the negative ancestors need to be peeled before there is a constant probability that the sampled ancestor is one of the peeled ones.) Each vertex thus belongs to only $O(\log n)$ subgraphs on which reachability/propagation is performed, keeping the total work down to $\tilde{O}(m)$. Moreover, because $n^{1/2+o(1)}$ is concave, the worst-case for the span is that each of the L calls to reachability operate on graphs with $\tilde{O}(n/L)$ vertices. We thus get span $L \cdot (n/L)^{1/2+o(1)} = \sqrt{L}n^{1/2+o(1)}$.

Priorities. We do not know how to maintain a uniformly random negative ancestor efficiently, but we achieve roughly the effect. We assign each vertex an independently random priority chosen from a geometric distribution with a rounded tail. Specifically, for $1 \leq i < \lceil \log_2 n \rceil$, we set $priority(v) = i$ with probability $1/2^i$; and with the remaining probability $1/2^{\lceil \log_2 n \rceil}$, we set $priority(v) = \lceil \log_2 n \rceil$. Priorities never change throughout the execution. For an edge (x, y), we use $priority(x, y)$ as a shorthand for $priority(x)$. To approximate uniformly random ancestors, our algorithm ensures that nodes are labeled by a negative ancestor of maximum priority.

3.2 Algorithm details

Our algorithm for computing $\{0, -1\}$ DAG shortest paths is shown in Algorithm 1. Our style of pseudocode here reflects higher-level directives, not parallel code. With the exception of the black-box reachability subroutine, it is fairly straightforward and uninteresting to parallelize each of the steps. We thus defer a slightly more detailed discussion to Section 3.5.

As noted in the overview, the algorithm proceeds in rounds, and these rounds are ordered sequentially. In each round i, we identify the frontier F of nodes at distance $-i$ and logically peel them from the graph. Note that we do not actually remove the nodes from the graph[3]; instead, we mark these nodes as ***finalized***. We call the nodes that are not finalized ***live***. Initially, all nodes are live.

We initialize a distance $d(v) = -\infty$ for all nodes and update this value when the vertex is eventually peeled. Similarly, we maintain a negative ancestor edge $parent(v)$ that is only guaranteed to be a correct parent edge when the vertex is finalized.

The crux of the algorithm is to maintain a negative ancestor edge $label(v)$ for each vertex v. Initially, all labels are invalid, indicated by $\perp$. Labels are restored by running Propagate, which occurs first in Line 5. We shall prove that when Propagate returns, a node v has $label(v) = \perp$ only if all of v's negative originators have been finalized. The set of nodes with no label even after Propagate returns is denoted by F. We also maintain a particular inversion of these labels: namely, $SentLabel(u)$ is the set of all vertices that have any edge $(u, *)$ leaving u as their label. This *SentLabel* facilitates invalidating labels when u is finalized/peeled from the graph.

In each round of the algorithm (Lines 7–14), the current frontier F of nodes to peel is processed as follows. First, we identify the set of nodes R that have a label including one of the peeled nodes, and we invalidate their labels. Then we set the distance to all nodes in F and finalize them, which logically peels them from the graph. Next, we call propagate on the nodes R (i.e., those with invalid label) to restore their labels. Finally, we let $F \subseteq R$ be the next frontier of nodes, i.e., those with no label after calling propagate.

Propagate. The Propagate(G, V') function, see Algorithm 2, takes as input the graph $G = (V, E)$ and a subset $V' \subseteq V$ of vertices. As called, V' always corresponds to the set of vertices with invalid label. The goal is to label the nodes in V' with one of their highest-priority negative ancestor edges, or $\perp$ if none exists.

Propagate considers each of the priorities in turn, from high to low. Each iterations starts by initializing the vertices in V' by inheriting any "nearby" labels. Specifically, we define the function GetNearbyLabel(G, V', i, v) to return an edge (or $\perp$) as follows:

- If v has at least one incoming edge $(u, v) \in E$ with: (1) u is live, (2) $w(u, v) = -1$, and (3) $priority(u) = i$, then choose any one such u and return the edge (u, v).
- If v has at least one incoming edge $(u, v) \in E$ with: (1) u is live, (2) $u \notin V'$, i.e., it already has a valid label, and (3) $priority(label(u)) = i$, then choose any one such u and return the edge indicated by $label(u)$.
- If neither of the above two cases applies, return $\perp$

After initializing the nearby labels for nodes in V', those nodes in V' with labels are the set of sources S. We next run multisource reachability on the induced subgraph $G' = G[V']$ with sources S. When multisource reachability returns, for each node in $v \in V'$, $\pi(v)$ indicates a node $s \in S$ that can reach v, if one exists. If $\pi(v) \neq \perp$, then we simply update v by inheriting $\pi(v)$'s label information.

Finally, each iteration ends by removing all newly labeled nodes from V'. In this way, V' reflects the nodes that are still unlabeled.

After all iterations complete, Propagate updates the *SentLabel* of all nodes as appropriate. In particular, $SentLabel(u)$ should be updated to also include all newly label nodes that have a label $(u, *)$. Note that this update is applied across the entire graph as appropriate, not just those nodes in V'.

3.3 Correctness

This section proves the correctness of our peeling algorithm. We start by focusing on Propagate. For a live node v, we use $NA(v)$ to denote the set of ***live negative ancestors*** of v. That is

$$NA(v) = \{(x, y) \in E \mid x \text{ is live}, w(x, y) = -1, \text{and } y \in Anc(v)\}.$$

We use $maxPri(v) = \max_{(x,y) \in NA(v)} priority(x, y)$ to denote the maximum priority over all of v's live negative ancestors. For completeness, we define $maxPri(v) = 0$ if $NA(v) = \emptyset$. Finally, we say that a live node v is ***correctly labeled*** if one of the applies:

- $NA(v) = \emptyset$ and $label(v) = \perp$, or
- $label(v) \in NA(v)$ and $priority(label(v)) = maxPri(v)$.

[3] This is primarily to avoid having to discuss how to update the graph in parallel.

Algorithm 1 Peeling Algorithm

Input: Graph $G = (V, E)$, whose edge weights $w : E \to \{0, -1\}$, and a source node $s \in V$.

Output: For all v, $d(v)$ is the shortest path distance from s to v $parent(v) = (x, y)$ is a negative ancestor edge with $d(x) = d(v) + 1$.

foreach $v \in V$ **do** ▷ Initialization
  $d(v) \leftarrow -\infty$
  $label(v) \leftarrow \perp$; $parent(v) \leftarrow \perp$; $SentLabel(v) = \{\}$
  assign a (geometric) random $priority(v) \in \{1, \ldots, \lceil \log_2 n \rceil\}$
PROPAGATE(G, V)
let $F = \{u \in V \mid label(u) = \perp\}$
for $i = 0$ to L **do**
  let $R = \bigcup_{u \in F} SentLabel(u)$
  foreach $v \in R$ **do**
    $label(v) \leftarrow \perp$
  foreach $u \in F$ **do**
    $d(u) = -i$; mark the node as finalized
  PROPAGATE(G, R)
  let $F = \{u \in R \mid label(u) = \perp\}$

Algorithm 2 Propagate Algorithm

function PROPAGATE$(G = (V, E), V' \subseteq V)$
  for $i = \lceil \log_2 n \rceil$ downto 1 **do**
    foreach $v \in V'$ **do**
      $e \leftarrow$ GETNEARBYLABEL(G, V', i, v)
      $label(v) \leftarrow e$
      if $e \neq \perp$ **then** $parent(v) \leftarrow e$
    let $S = \{v \in V' \mid label(v) \neq \perp\}$
    $G' \leftarrow G[V']$
    run multisource reachability on G' with sources S and let π be the output
    for each $v \in V'$ **do**
      if $\pi(v) \neq \perp$ **then**
        $label(v) \leftarrow label(\pi(v))$
        $parent(v) \leftarrow label(v)$
    remove all nodes with $label(v) \neq \perp$ from V'
  update $SentLabel$ sets to include new label assignments

Our first goal is to show that when PROPAGATE, all live nodes are correctly labeled.

LEMMA 1. *Consider a call to* PROPAGATE$(G = (V, E), V')$*, where $V' \subseteq V$ is exactly the subset of live nodes in G that have label $\perp$. If all live nodes in $V \backslash V'$ are correctly labeled before the call, then all live nodes in V are correctly labeled after the call.*

PROOF. Because no labels are updated for any nodes outside of V', establishing the correct labeling of V' is sufficient. The proof is by induction over the iterations of the main loop in PROPAGATE. The iterations are numbered in decreasing order, from $\lceil \log_2 n \rceil$ downto 1. Our inductive claim is that after iteration i: (1) all nodes outside V' are correctly labeled, and (2) all nodes in V' have $maxPri(v) < i$. The remainder of the proof focuses on proving the inductive step. Specifically, we show that all nodes with $maxPri(v) = i$ become correctly labeled, and also that those nodes with $maxPri(v) < i$ do not get a label incorrectly.

Consider iteration i and any live node $v \in V'$ having $maxPri(v) \leq i$. Consider also any ancestor $z \in Anc(v) \cap V'$. We first claim that if z gains a label during GETNEARBYLABEL, then that label would be a valid label for v. To prove the claim, we observe that the only labels considered during this iteration have priority i by construction. Moreover, by transitivity of reachability, $NA(z) \subseteq NA(v)$ and hence $maxPri(v) \geq maxPri(z)$. Therefore, if z gains a label, that label would be a correct label for v. It follows that whenever $\pi(v) \neq \perp$ after running multisource reachability on $G[V']$, the label $label(\pi(v))$ is always a correct label for v.

We next claim that if $maxPri(v) = i$, for $v \in V'$, then there exists at least one node $z \in V'$ such that: (1) z becomes labeled during GETNEARBYLABEL, and (2) there is a path from z to v in $G[V']$. If this claim holds, we have that whenever $maxPri(v) = i$, $\pi(v) \neq \perp$. Coupled with the above claim, v becomes correctly labeled.

To prove the claim, consider any node $v \in V'$ with $maxPri(v) = i$. By definition, there exists at least one edge $(x, y) \in NA(v)$ with $priority(x, y) = i$. Consider any path Γ from y to v in G. If the entire path is in V', then Γ is a path in $G[V']$. Moreover, y has an incident negative edge with appropriate priority, so $z = y$ is indeed labeled during GETNEARBYLABEL. Otherwise, let (r, z) be the latest edge on the path with $r \notin V'$ and $z \in V'$. Then the subpath of Γ from z to v is a path in $G[V']$. Moreover, r is outside V' so it is correctly labeled by inductive assumption. We have $i = priority(x, y) \leq maxPri(r) \leq maxPri(v) = i$, so r is already labeled with priority exactly i. Thus, z is also assigned a label during GETNEARBYLABEL. □

We next turn to correctness of the higher-level algorithm. The following lemma essentially indicates that if finalizing nodes in order of distances, then all labels involving farther-away (more negative distance) nodes are still valid.

LEMMA 2. *Consider a DAG $G = (V, E)$ with edge weights from $\{0, -1\}$ and source vertex $s \in V$ that can reach all vertices.*

Consider the following generic process. A node v is initially correctly labeled with $label(v) = (x, y)$. Let $F \subset V$ be any subset of nodes nearer to the source than x, i.e., for all $f \in F$, $dist(s, f) > dist(s, x)$. Suppose that the nodes in F are finalized (or removed from the graph). Then v is still correctly labeled.

PROOF. We first observe that for all nodes $z \in Des(y)$, we have $dist(s, z) \leq dist(s, x) + w(x, y) + dist(y, z) < dist(s, x)$, because distances are all nonpositive and finite. Thus, no descendants of y are part of F, and in particular the path from y to v remains in the graph. Because no nodes higher priority nodes or paths are created, it follows that v is still correctly labeled. □

The next lemma indicates unlabeled nodes can be finalized.

LEMMA 3. *Consider a DAG $G = (V, E)$ with edge weights from $\{0, -1\}$ and source vertex $s \in V$ that can reach all vertices. Suppose all the nodes, and only the nodes, with distance $> -i$ from the source have been finalized. For any live node v, if $NA(v) = \emptyset$ then $dist(s, v) = -i$.*

PROOF. By assumption, $dist(s, v) \leq -1$. Suppose for the sake of contraction that $dist(s, v) < -i$. Then there exists some shortest

path Γ from s to v. Let (x, y) be the last negative edge on the path. Then $dist(s, v) = dist(s, y) = dist(s, x) - 1$, or $-i > dist(s, v) = dist(s, x) - 1$, or $-i \geq dist(s, x)$. Therefore, neither x nor y has not been finalized, and hence $(x, y) \in NA(v)$. This contradicts the assumption that $NA(v) = \emptyset$. □

Finally, we prove that the output of the algorithm is correct.

Theorem 4. *Consider a directed acyclic graph $G = (V, E)$ with edge weights from $\{0, -1\}$, let $s \in V$ be the source node, and suppose that all nodes in the graph are reachable from s. Then Algorithm 1 correctly solves the distance-limited DAG SSSP problem.*

More precisely, on completion, for all $v \in V$ we have:

- *$d(v) = dist(s, v)$ if $dist(s, v) \geq -L$, or $d(v) = -\infty$ otherwise.*
- *$parent(v) = (x, y) \in NA(v)$ and $dist(x) = dist(v) + 1$.*

Proof. The proof is by induction over rounds of the Algorithm 1. The inductive claim is that after round i, all nodes and only nodes with distance at least $-i$ have been finalized. Moreover, their distance is marked correctly. In addition, we include in the claim that F is exactly the set of live nodes that currently have no label.

Now the inductive step. Consider the start of iteration i. By Lemma 3, all nodes in F have distance equal to $-i$. By Lemma 2, the labels are still correct for all nodes not in R. Thus, by Lemma 1, all live nodes are correctly labeled when Propagate returns. Finally, F is restored to be the live nodes with no label.

Finally, we argue that the *parent* references are correct. To do so, we simply observe that the parent only changes when the label changes. Thus, $parent(v) = (x, y)$ corresponds to the final label that v had before arriving at a label of $\perp$. Since v is finalized in round $-dist(s, v)$, v must have lost its label in the preceding round. In other words, $dist(s, x) = dist(s, v) + 1$. □

3.4 Key performance claims

This section gives the core components of the performance analysis, specifically bounding the total work and span of all multisource reachability invocations. The other straightforward details of the parallel implementation are shown next in Section 3.5.

Our main goal is to show that for each vertex v, v's label does not change too many times. We do so by first arguing v does not have too many negative originators with priority equal to $maxPri(v)$.

Lemma 5. *Consider the state of the graph at the start of a round i, i.e., when all nodes with distance $> -i$ have already been finalized. Let v be any live vertex. With high probability, v has at most $O(\log n)$ negative originators that have priority equal to $maxPri(v)$.*

Proof. We shall consider a specific priority and argue that some, but not too many, negative originators of v have at least that priority. Specifically, let β be the total number of live negative originators of v. Consider a priority $x = \lceil \lg \beta - \lg \ln(n) - \lg(c) \rceil$, for constant $c > 1$. (The Θ absorbs the rounding of x to an integer.) Observe that $c2^x \ln n \leq \beta \leq c2^{x+1} \ln n$.

We first claim that, with failure probability at most $1/n^c$, v has at least one negative originator with priority $\geq x$. For a particular negative originator u, we have probability $1/2^{x-1}$ that $priority(u) \geq x$. There are $\beta \geq c2^x \ln(n)$ negative originators, so the probability that none of them has priority $\geq x$ is at most $(1 - 1/2^{x-1})^\beta \leq ((1 - 1/2^{x-1})^{2^{x-1} c \ln n} \leq (1/e)^{c \ln n} = 1/n^c$.

We next claim that not too many negative originators have priority $\geq x$. Each negative originator has an independent priority, and the expected number of originators having at least this priority is $\beta/2^{x-1} \leq 4c \ln n$. We can thus apply a Chernoff-Hoeffding bound to conclude that with probability at most $1/n^c$, no more than say $8c \ln n$ negative originators have priority $\geq x$. □

Corollary 6. *Consider a particular node v across the execution of the algorithm, and assume that the algorithm operates correctly. Then v's label changes at most $O(\log^2 n)$ times across the entire execution, with high probability.*

Proof. Lemma 5 states that with high probability, at the start of each round, v has at most $O(\log n)$ negative originators with priority equal to $maxPri(v)$. Because which nodes are peeled is deterministic, there are not many events to take a union bound over. In particular, the probability of even one failure for any vertex across the execution is at most $n(L + 1)/n^{\Theta(c)}$.

For the remainder, assume no failures. Then Lemma 5 holds specifically each time $maxPri(v)$ changes. Since there are only $O(\log n)$ nodes with that priority, v can only get subsequent labels with the same priority $O(\log n)$ times. Multiplying this $O(\log n)$ by the number of different priorities completes the proof. □

We are now ready to bound the total cost of the calls to multisource reachability, which dominates the costs of the algorithm.

Lemma 7. *Consider the total across all invocations of multisource reachability across the execution of the algorithm. The total cost of these calls is $\tilde{O}(m)$ work and $\sqrt{L} \cdot n^{1/2+o(1)}$, with high probability.*

Proof. From Corollary 6, each node's label changes at most $O(\log^2 n)$ times, with high probability. This also bounds the number of times each node is passed to Propagate. During each call, the node and its incident edges may be built into $O(\log n)$ induced subgraphs, one for each priority. Adding up across all edges and calls, we get a total induced subgraph size of $O(n \log^3 n)$ vertices and $O(m \log^3 n)$ edges. Because the work of multisource reachability [18] is nearly linear, the total across all calls is also $\tilde{O}(m)$ no matter how the edges are divided across calls. Because the span is concave, the worst case (according to Jensen's inequality) is that all calls are on graphs of size $O(n \log^3 n/L \log n)$ which yields a total span of $L \log n \cdot (n \log^3 n/L \log n)^{1/2+o(1)} \leq \sqrt{L} n^{1/2+o(1)}$. □

Theorem 8. *For a DAG G containing integer edge weights of $\{0, -1\}$ and a source node s, there is a parallel algorithm that outputs for each node $v \in V$, $d(v) = dist(s, v)$ if $dist(s, v) \geq -L$, and $-\infty$ otherwise. The algorithm runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)} L^{1/2}$ span.*

Proof. Proof follows from Lemmas 3, 7, and 9, where the latter (in Section 3.5) includes of the remaining parallel steps. □

3.5 Remaining Details of the Peeling Algorithm

This section discusses some of the more straightforward steps for making the algorithm parallel. Our main claim is the following lemma. Proof appears at the end of the section.

Lemma 9. *Algorithm 1 (PeelingAlgorithm) runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)} L^{1/2}$ span.*

Before showing the proof, we will show a data structure that we will use.

Parallel sets. There exist implementations of parallel sets that can perform merge in $O(m \log(n/m+1))$ work and $O(\log m \log n)$ span for sets of size m and n where $n \geq m$ [4]. The parallel set can also enumerate all elements in a size n set in $O(n)$ work and $O(\log n)$ span [4].

Parallel Implementation. Now that we have parallel sets we can show the parallel implementation of the peeling algorithm. We will first discuss parallel propagate and then the main loop of the peeling algorithm. The PROPAGATE(G, R) function runs on a subgraph G' which changes during each iteration of the algorithm. We will show how to update the graph in parallel and other steps that are non-trivial. A key point is that we can afford to look at each node in the subgraph G', since as we showed in Corollary 6 that in each node is added to $O(\log^2 n)$ subgraphs across the whole algorithm. However, we cannot afford to look at the original graph G. Also, for a node v in G' we can afford v's edges in G, and not just the edges in G'. This makes building the induced subgraph trivial once we are given the nodes.

In each iteration of propagate we must update the graph by removing any nodes that get a label. Since we have $O(\log n)$ priorities, we can afford to sort all V' nodes in each iteration. By sorting we can group the nodes together that are not removed from G' in previous iterations. The cost of the sort is $\tilde{O}(V')$ work and $\tilde{O}(1)$ span.

We implement the $SentLabel(u)$ sets as parallel sets. To update the $SentLabel(u)$ sets (Line 15 in Algorithm 2), first sort all the nodes $v \in V'$ by their label, i.e. sort edges (u, v) by ordering on u. This groups all the nodes together that will be added to $SentLabel(u)$. To add these nodes to $SentLabel(u)$, we merge these nodes with the current set $SentLabel(u)$. In total, the cost is $\tilde{O}(V')$ work and $\tilde{O}(1)$ span to update $SentLabel(u)$ for all the nodes in each round of PROPAGATE(G, V'). It is fairly straightforward to implement the rest of the steps in parallel for PROPAGATE(G, V').

Before we return to the loop in the main algorithm, we sort the nodes by label, which groups together all the nodes that have no label, and thus are finalized. This has the cost of parallel sort for the number of nodes in propagate. When we return to the main loop, we need to compute the next set of nodes to run propagate on. To implement this step, in parallel each finalized node u determines the number of nodes in $SentLabel(u)$. We can then run parallel prefix sums to allocate space in an array for the nodes in R. Then the nodes can be added to the new array in parallel, which is passed to the next call to propagate. The work and span of these steps is dominated by the cost of sorting the number of elements in propagate at each round.

Proof of Lemma 9. The initialization steps can be performed in $\tilde{O}(n)$ work and $\tilde{O}(1)$ span. Previously we discussed how the *SentLabel* sets can be maintained in $O(|V'|)$ work and $\tilde{O}(1)$ span. By Lemma 7, the cost of all the runs of multisource reachability is $\tilde{O}(m)$ and $n^{1/2+o(1)}L^{1/2}$ span. Based on Corollary 6, the total number of nodes in all calls to PROPAGATE(G, V') is $\tilde{O}(n)$. We also showed that we can update the nodes in R, and perform the updates to the graph G' in total work $\tilde{O}(n)$, and $\tilde{O}(1)$ span.

Combining everything together, the algorithm runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)}L^{1/2}$ span.

4 DISTANCE-LIMITED SSSP WITH NONNEGATIVE INTEGER WEIGHTS

Given a directed graph $G = (V, E)$ with source $s \in V$, nonnegative integer edge weights, and distance $L \leq n$, the problem is to return the shortest-path distances $d(v) = dist(s, v)$ for all v with $dist(s, v) \leq L$, and $d(v) = \infty$ otherwise. We also want to report a shortest-paths tree to recover shortest paths, which can be accomplished through a postprocessing step discussed in Section 4.2.

Algorithm overview. Similar to the peeling algorithm in the previous section, this algorithm will solve shortest paths in order by distance and peel off solved nodes. Once a node is finalized, its distance is set and not considered for the rest of the algorithm. The algorithm will use a $(1+\epsilon)$-approximate shortest paths algorithm to guess the distances of nodes in the graph. Using the distance estimate, each unfinalized node v is assigned to a 2^i-sized interval for some integer i. As long as the ASSSP never returns an incorrect answer, the true shortest-path distance to v always falls within its assigned interval. Section 4.2 discusses how to cope with the possibility that the ASSSP algorithm fails to achieve the approximation. After each layer of the peeling, the algorithm refines the distance intervals, by again running ASSSP. Since the shortest paths have gotten shorter, the approximation becomes better, and so the node can be assigned to an interval of smaller length. Once the interval size is a small enough constant, the distance can be solved directly.

It is too expensive to refine all distance estimates in each round, so the algorithm must choose when to include each node.

4.1 Algorithm Description

Let D be the smallest power of 2 strictly greater than L. The algorithm operates on intervals $[d, d+2^i)$ of length 2^i, for $0 \leq i \leq \lg(2D)$, where the intervals are aligned to multiples of 2^{i-1}, i.e., $d = k2^{i-1}$. Unfininished nodes are assigned to intervals that (barring ASSSP failures) contain their true distance, and they are moved to smaller intervals as the algorithm progresses. Initially the algorithm runs 2-approximate ASSSP; all nodes with estimate $> 2D$ (and hence true distance $> D$) are finalized to a distance of ∞; all other nodes are assigned to $[0, 2D)$.

The rest of the algorithm proceeds in rounds $0, 1, \ldots, D$, where nodes with distance d are finalized during round d. In each round, the value d encroaches (reaches the left side) of some intervals. For each of these intervals $I = [d, d+2^i)$, from largest to smallest, the goal is to "refine" the distance estimates of all nodes assigned to I to smaller intervals or finalize nodes with size-1 intervals.

We compute the shortest-path tree as straightforward a postprocessing step in Section 4.2.

Refine. The function REFINE$(d, 2^i)$ takes as input the interval $[d, d+2^i)$. It builds a graph G' on nodes whose interval overlaps $[d, d+2^i)$. More details of building the graph are discussed next. The key idea is that since all unfinalized nodes have distance d, we can shift distances downward by d; i.e., distances of d in G are translated to distances of 0 in G'; roughly speaking, this allows us to

apply an algorithm with multiplicative approximation to improve the additive approximation. The next step is to run ASSSP on G', which improves the distance estimates for those nodes assigned to $[0, d + 2^i)$. Those nodes are reassigned to one of the 3 overlapping subintervals of size 2^{i-1}, as shown in Lines 14-20 of Algorithm 3.

Build Graph G'. Add each node to the V' who's interval overlaps $[d, d + 2^i)$. Create graph $G' = G[V']$. Add a source node s' to V'. For each node v with $d(v) = +\infty$, and each incoming edge (u, v), where $d(u)$ is not infinity, add an edge from s' to v with weight $d(u) + w(u, v) - d$.

4.2 Verification and Shortest Paths tree

Verification. The algorithm for parallel approximate shortest paths works with high probability, meaning that it never gives an underestimate of the shortest paths, but may fail to achieve the $(1+\epsilon)$ approximation. In this section, we will show how to verify that our algorithm is correct. After running the algorithm, first contract cycles 0-weight edges, and then look at each nodes incoming edges. For each node $v \in V$, verify that $d(v) = \min_{(u,v)}(d(u) + w(u, v))$, and if any node fails then the algorithm has failed and must be repeated.

Lemma 10. *Assume we set source node $d(s) = 0$, then for all $v \in V$, $d(v) = dist(s, v)$ if and only if for all v, we have $d(v) = \min_{(u,v)}(d(u) + w(u, v))$.*

Proof. ($\Leftarrow$) Based on our definition of shortest path, for any v, $d(v) = dist(v) = \min_{(u,v)}(dist(s, u) + w(u, v)) = \min_{(u,v)}(d(u) + w(u, v))$.

($\Rightarrow$) Proof by contradiction. We first want to show if $d(v) = \min_{(u,v)}(d(u) + w(u, v))$ for v and $d(s) = 0$, then $d(v) \geq dist(v)$. Consider the path $(v_0, v_1, ..., v_t = v)$, such that $d(v_i) = d(v_{i-1}) + w(v_{i-1}, v_i)$. Each edge $w(v_{i-1}, v_i) > 0$ is in the graph, so $d(v_0)$ to $d(v_t)$ is increasing and we can set $v_0 = s$. Notice that $d(v_t) = \sum w(v_{i-1}, v_i)$ is weight of a path from s to v, so $d(v) \geq dist(v)$.

Let S be set of node x with $d(x) \neq dist(s, x)$. Let v the the node in S with smallest $dist(v)$. Let u be the parent of v in the shortest path tree. Since $w(u, v) > 0$, we have $dist(u) < dist(v)$ and $d(u) = dist(u)$. Then, $d(v) \leq d(u) + w(u, v) = dist(u) + w(u, v) \leq dist(v)$. and this contradicts to the fact that $d(v) \geq dist(v)$ and the definition of S. □

Shortest Paths Tree. We would like to output the shortest path tree for the contracted graph (zero weight cycles are contracted). The algorithm can be extended to find pointers for all nodes in the graph. We follow a similar strategy as the verification algorithm. First contract any zero weight cycles, and then each node looks at its incoming edges to find its parent in the tree. Specifically, for each node v, set $parent(v)$ to be u such that $d(v) = d(u) + w(u, v)$.

4.3 Limited Distance Shortest Paths Analysis

Next we will show that the algorithm correctly computes the shortest paths distances for nodes up to distance D. The algorithm for approximate shortest paths that we use as a black box achieves the approximation with high probability. In the following lemma we assume the approximate shortest paths runs successfully, since in Section 4.2 we showed how to overcome the issue that approximate shortest paths does not achieve the desired approximation.

Algorithm 3 LimitedSP
Input: Graph $G = (V, E)$ and source node s
Output: Shortest path distance $d(v)$ for each node s from v up to distance L, and ∞ for nodes with distance greater than L

```
1:  function LimitedSP(G = (V, E), s)
2:      for each v ∈ V do d(v) ← +∞
3:      d(s) = 0
4:      d' ← ASSSP(G, s, ε = 1)
5:      for each v ∈ V with d'(v) ≤ 2D do add v to interval [0, 2D)
6:      for d = 0 to D: do
7:          for i = lg(2D) to 0: do
8:              if d is a multiple of 2^{i-1} then
9:                  Refine(d, 2^i)
10: function Refine(d, 2^i)
11:     Build graph G' = (V', E')
12:     d' ← ASSSP(G', s', ε < 1/4)
13:     for each v ∈ V' with d'(v) = 0 do d(v) ← d
14:     for each v ∈ V' with I(v) = [d, d + 2^i) do
15:         if d'(v) ∈ [0, 2^{i-1}) then
16:             add v to interval [d, d + 2^{i-1})
17:         else if d'(v) ∈ [2^{i-1}, 3 · 2^{i-2}) then
18:             add v to interval [d + 2^{i-2}, d + 3 · 2^{i-2})
19:         else
20:             add v to interval [d + 2^{i-1}, d + 2^i)
```

Lemma 11. *Consider a call to LimitedSP(G) for some graph $G = (V, E)$. For any node $v \in V$ with $d(v) \leq D$, the shortest path distance is correctly computed.*

Proof. For any given node v, recall $dist(s, v)$ is the shortest path distance. Let $I(v)$ be the interval v is located in. We change $I(v)$ if and only if $I(v) = [d, d + 2^i)$ when we call Refine$(d, 2^i)$. We will use induction on Refine$(d, 2^i)$ on d and i to show

- for any node v with $dist(s, v) < d$, $d(v) = dist(s, v)$
- for any node v with $dist(s, v) \leq D$, $dist(s, v) \in I(v)$.

The base case is $d = 0$ and $i = \lg(2D)$. For any node v with $dist(s, v) \leq D, I(v) = [0, 2D)$, we run approximate shortest paths on the whole graph. In this case, $d'(v) = 0$ if and only if $dist(s, v) = 0$. We set $d(v) = dist(s, v) = 0$ if $d'(v) = 0$. For any node v with $dist(s, v) \leq D$, if $d'(v) < 2^{i-1}$, we know that $dist(s, v) \leq d'(v) < 2^{i-1}$; if $2^{i-1} \leq d'(v) < 3 \cdot 2^{i-2}$, we know that $dist(s, v) \leq d'(v) \leq 3 \cdot 2^{i-2}$ and $dist(s, v) \geq d'(v)/(1+\epsilon) \geq 2^{i-2}$; if $d'(v) \geq 3 \cdot 2^{i-2}$, we know that $dist(s, v) \geq d'(v)/(1+\epsilon) \geq 2^{i-1}$. Combining these three cases, node v will be added to $I(v)$ such that $dist(s, v) \in I(v)$.

Now assume that for d and i, the claim holds, consider d and $i - 1$, for the first claim, for any node with $dist(s, v) < d$, $d(v) = dist(s, v)$. For the second claim, for any node v with $dist(s, v) \leq D$, $dist(s, v) \in I(v)$. When we call Refine$(d, 2^{i-1})$, we change $I(v)$ if and only if $I(v) = [d, d + 2^{i-1})$. Consider a node v with $I(v) = [d, d + 2^{i-1})$ and a path p from s to v. Let u be the last node on the path with $dist(s, u) < d$ and u' be the next node on this path. We will add an edge from s' to u' with weight $d(u) + w(u, u') - d$. Let l be the weight of path from u' to v. Notice that weight of p'

is $w(p') = d(u) + w(u,u') - d + l \geq dist(s,v) - d$. When p is the shortest path, for any node w on path p between u' and v, we know $d \leq dist(s,w) \leq dist(s,v) < d+2^i$. Thus $I(w)$ overlaps $[d, d+2^i)$ and we add w to the graph, and then $w(p) = dist(s,v) - d$. Combining these two points, we know in the new graph, the shortest path distance for v is $dist(s,v) - d \in [0, 2^i)$. If $d'(v) \leq 2^{i-1}$, then we know $dist(s,v) - d \leq d'(v) \leq 2^{i-1}$ and $dist(s,v) \in [d, d+2^{i-1})$; If $2^{i-1} \leq d'(v) \leq 3 \cdot 2^{i-2}$, we know $dist(s,v) - d \geq d'(v)/(1+\epsilon) \geq 2^{i-2}$ and $dist(s,v) - d \leq d'(v) \leq 3 \cdot 2^{i-2}$; Last, if $d'(v) \geq 3 \cdot 2^{i-2}$, we know that $dist(s,v) - d \geq d'(v)/(1+\epsilon) \geq 2^{i-1}$. In either case, we have $dist(s,v) \in I(v)$.

The last thing is to show for any d and $i = 0$, we set $d(v) = d$ if $dist(s,v) = d$. Notice that based on our assumption, for any node u with $dist(s,u) \leq D$, $dist(s,u) \in I(u)$. If $dist(s,v) = d$, then we add v to the graph. Similar to the above argument, in the new graph, the shortest path distance for v is 0 and thus $d'(v) = 0$ and we will set $d(v)$ to be d for any node v with $dist(s,v) = d$. □

Next we will show the work and span the of the algorithm. Each call to Refine$(d, 2^i)$ builds a graph and runs approximate shortest paths. Our main goal is to show that each node is not added to too many of these graphs. By bounding the number of graphs a node is added to, we are able to bound the total size of these graphs.

Lemma 12. *Consider a call to LimitedSP(G) for some graph $G = (V,E)$. For a node $v \in V$, while v is assigned to a particular interval X, v is added to $O(\lg D)$ graphs G' in Refine.*

Proof. Node v is only added to a graph G' in Refine$(d, 2^i)$ when interval X overlaps the interval $[d, d+2^i)$. Let the interval $X = [c2^j, (c+1)2^j)$ have length 2^j, for some integer c. Consider the following two cases of calls to Refine$(d, 2^i)$ for size 2^i intervals. The two cases are $j \leq i$, and $j > i$. In both cases we will show at most three intervals of size 2^i overlap X, which implies X overlaps at most $O(\lg D)$ intervals, and v is added to at most $O(\lg D)$ graphs in calls to Refine$(d, 2^i)$.

Case 1: $j \leq i$. In this case, there will be at most three intervals that intersects X, i.e, $[d, 2^i)$, $[d+2^{i-1}, 2^i)$ and $[d+2^i, 2^i)$ since $j \leq i$ and interval $[d, d+2^{i+1})$ must cover X if $[d, 2^i)$ intersects X.

Case 2: $j > i$. For this case we will show that at most one interval of size 2^i overlaps X. Consider the first d such that Refine$(d, 2^i)$ intersects with $X = [c2^j, (c+1)2^j)$, we have $d \leq c2^j$. That's because when we refine Refine$(d, 2^i)$, it's impossible such that for some u with interval $I(u) = [c2^j, c2^j + 2^j)$, $c2^j < d$. If such node u exists, notice that when we update interval in Refine$(d', 2^i)$, we never put a node to some interval starting earlier than d', so $I(u) = [c2^j, c2^j + 2^j)$ must be updated in some Refine$(a', 2^{j+1})$ with $a' \leq c2^j$. Then when we call Refine$(c2^j, 2^j)$, we will refine node u and it's no longer in $I(u)$.

Assume that the first time Refine$(d, 2^i)$ intersects with X, we have $d \leq c2^j$. Since $2^{i-1} < 2^{j-1}$, and both are powers of 2, each multiple of of 2^{j-1} is also a multiple of 2^{i-1}. Therefore there is a size 2^i interval starting at $c2^{j-1}$. This interval has size 2^i, and any that start later will be refined after X since the intervals are refined in order by distance and decreasing size. For intervals that start before X, there is one size 2^i interval that starts at $c2^{j-1} - 2^{i-1}$ which overlaps X. The next size 2^i interval to the left starts at $c2^{j-1} - 2 \cdot 2^{i-1} = c2^{j-1} - 2^i$, and since it has length 2^i, the interval is $[c2^{j-1} - 2^i, c2^{j-1})$, so it does not overlap X. Any intervals starting at smaller multiples of 2^{i-1} also do not overlap X, by the same reasoning. Thus X overlaps one interval of size 2^i for each $i < j$. □

Lemma 13. *Consider a call to LimitedSP(G) for some graph $G = (V,E)$. For a node $v \in V$, v is added to $O(\lg^2 D)$ graphs G' in calls to Refine$(d, 2^i)$.*

Proof. If node v has its initial distance estimate $d(v) > 2D$, then v is not added to any intervals and therefore no graphs G' either. Otherwise, v starts in the interval $[0, 2D)$. Each time it is added to a graph in Refine, either it stays in the same interval or moves to a smaller interval. By Lemma 12, v gets added to $O(\lg D)$ refinement graphs for each interval it is in. Since the interval sizes are monotonically decreasing, and there are $O(\lg D)$ of them, v is added to $O(\lg^2 D)$ refinement graphs. □

We next turn to the work and span of the algorithm. Some of this relies on the straightforward parallel details, but the bulk of the work falls in ASSSP. Analogous to Lemma 7, we can leverage Lemma 13 to argue that the in total, these calls are not too expensive.

Before proving the work and span, we will show a data structure that we will use in the implementation.

Vector of parallel sets. We construct a vector of parallel sets VS, where each set has an identifier. Assume that there are polynomial bounded elements in total across all sets. The vector of sets VS supports the following.

- Initialization, we can set up a vector, each item of the vector contains a pointer to the set. The set is identified by the identifier. All sets are set to be empty at the beginning, the work is $O(\textit{number of sets})$ and the span is $\tilde{O}(1)$.
- Given t sets, where each set contains x_i elements, we can add the elements of the t sets into VS in $O(\sum x_i)$ work and $\tilde{O}(1)$ span because we can merge each set separately.
- Given the identifiers of t sets, we can merge the elements from all t sets into a vector with $O(x)$ work and $\tilde{O}(1)$ span, where x is the total number of elements across the t sets. To copy all elements, we first compute the number of elements in each set, then we run prefix sum to compute the location of each set should be transferred and access each sets elements in parallel.
- Given the identifiers of t sets, we can empty all elements in those t sets in $O(x)$ work and $\tilde{O}(1)$ span, where x is the total number of elements across the t sets.

Note that, to manipulate on the vector of parallel sets, we must know the identifier of each set to locate the pointer to the set.

Now we can show the following lemma which bounds the work and span of the algorithm.

Lemma 14. *Algorithm 3 has $\tilde{O}(m)$ work and $n^{1/2+o(1)}L^{1/2}$ span.*

Proof. Algorithm 3 first runs the approximate shortest paths algorithm which takes $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span. Next, it assigns each node with to an interval which can be done in $O(n)$ work and $O(1)$ span. Last, it calls Refine$(d, 2^i)$ $\tilde{O}(D)$ times.

The subroutine Refine$(d, 2^i)$ first identifies all nodes whose interval intersects with $[d, d+2^i)$. Let x_i and y_i be number of nodes

in the overlapping intervals and number of edges neighboring the nodes. We will show later that the algorithm copies all nodes in $\tilde{O}(x_i + 2^i)$ work and $\tilde{O}(1)$ span. After identifying those nodes, we can construct the graph with $\tilde{O}(x_i + y_i)$ work and $\tilde{O}(1)$ span. The approximate shortest path algorithm also depends on x_i and y_i, and takes $\tilde{O}(y_i)$ work and $x_i^{1/2+o(1)}$ span. Lastly, each node in the interval $[d, d + 2^i)$, will be reassigned to a different interval. To reassign the nodes, we can sort the nodes based on d' value. Sorting takes $\tilde{O}(x_i)$ work and $\tilde{O}(1)$ span. Moving the nodes to different intervals relies on the implementation of the sets. We will later show this step can be performed in $\tilde{O}(x_i)$ work and $\tilde{O}(1)$ span. In total, each call to Refine$(d, 2^i)$ takes $\tilde{O}(y_i + 2^i)$ work and $x_i^{1/2+o(1)}$ span, where x_i and y_i are the number of nodes and edges, respectively, in G'.

Notice that for each $(d, 2^i)$, we have $d/2^i$ different calls to Refine with interval size 2^i, so the 2^i term will contribute $\tilde{O}(D)$ to the work in total. By Lemma 13, each node is added to G' at most $O(lg^2 D)$ times, so $\sum x_i = O(n \lg^2 D) = O(n \lg^2 D)$, and $\sum y_i = O(m \lg^2 D + L) = \tilde{O}(m + L) = \tilde{O}(m)$ because of the fact that $L = O(n)$. The total work is $\tilde{O}(m + D) = \tilde{O}(m)$. Notice there are at most $O(D \lg D)$ calls to Refine, and the span of the algorithm is $\sum x_i^{1/2+o(1)} = n^{1/2+o(1)} L^{1/2+o(1)} + L = n^{1/2+o(1)} L^{1/2}$.

The remaining problem is to maintain a set for each interval. We will use the vector of parallel sets data structure to maintain a vector of sets for each possible interval $[d, d + 2^i)$. Notice that the identifier for each set is the interval $[d, d + 2^i)$ for different d and i and the identifier for each set can be sorted by the starting point of the interval. If the starting point is the same, we sort by ending point of the interval. Initialization takes $\tilde{O}(D)$ work and $\tilde{O}(1)$ span. Then we add all elements to the interval $[0, 2D)$. It takes $\tilde{O}(1)$ work and span to specify the identifier of the set and $\tilde{O}(n)$ work and $\tilde{O}(1)$ span to add all elements.

In a call to Refine$(d, 2^i)$, since the interval is sorted by starting point, the interval that intersects with $[d, d+2^i)$ and starts no earlier than d is continuous and we can identify all identifiers of intervals in $\tilde{O}(2^i)$ work and $\tilde{O}(1)$ span. Once we get all the interval that intersects with $[d, d + 2^i)$, it takes $O(|V'|)$ work and $\tilde{O}(1)$ span to copy all elements in the sets. After we identify all nodes in G', the graph can be built by using parallel sort to group together the nodes that are in the subgraph. The last piece is to empty all elements for interval $[d, d + 2^i)$ and add those elements to three different intervals. This can be done in $O(|V'|)$ and $\tilde{O}(1)$ span. □

From Lemmas 11 and 14, we conclude:

Theorem 15. *There exists a parallel algorithm solving nonnegative L-distance-limited SSSP with work $\tilde{O}(m)$ and span $n^{1/2+o(1)} L^{1/2}$ span, with high probability. That is, consider a directed graph $G = (V, E)$ with nonnegative integer weights and source $s \in V$. The algorithm outputs for each node $v \in V$, $d(v) = dist(s, v)$ if $dist(s, v) \leq L$, and infinity otherwise.* □

5 OVERVIEW OF GOLDBERG'S ALGORITHM

This section summarizes Goldberg's sequential algorithms [16] for computing exact shortest paths on a graph with integer edge weights. Our description here differs slightly from Goldberg's [16], in part to facilitate our parallel extension, but these differences are not technically relevant as far as the sequential algorithm goes. The main goal of Goldberg's algorithm is to reweight the graph such that all weights are nonnegative. Then the single-source shortest paths problem can be solved efficiently by Dijkstra's algorithm.

For the reweighting of the graph to be meaningful, the new weight function must preserve shortest paths. That is, a path should be a shortest path with respect to the updated weight function if and only if it is a shortest path with respect to the original function w. Goldberg [16] achieves valid reweightings by defining the reweighting by way of a price function $p : V \to \mathbb{Z}$ over vertices. Given the price function p, the updated weights of the graph are given by $w_p(u, v) = w(u, v) + p(u) - p(v)$, where w is the original weight function. This type of price function always produces a meaningful reweighting that preserves shortest paths; see Cormen et al.'s [11] discussion of Johnson's algorithm for a general correctness argument. Moreover, given the shortest-path distance $d(v)$ from s to v with respect to weight function w_p, the shortest-path distance with respect to the input weights is simply $d(v) + p(v) - p(s)$.

We say that a price function, or the induced reweighting, is ***feasible*** if all edge weights $w_p(u, v) \geq 0$ are nonnegative. The goal is thus to produce a feasible price function if possible, or to determine that the graph has a negative-weight cycle. As noted above, a feasible price function implies a solution to SSSP.

Reducing the negativity by scaling

Goldberg's main algorithm [16] is an algorithm that produces a feasible price function for the special case that all edge weights are integers with value at least -1. (That is, the weights may take any nonnegative value, but the only negative value they may have is -1.) We call this special case the ***1-reweighting problem***. His algorithm [16] solves the 1-reweighting problem in time $O(m\sqrt{n})$. He then applies a bit-scaling technique [15, 16] to generalize the solution to arbitrary integer weights. If all weights are at least $-N$, then $O(\log N)$ repetitions of the 1-reweighting algorithm are sufficient to find a feasible price function for general integer weights [16].

Overview of 1-reweighting

Goldberg's algorithm [16] solves the 1-reweighting problem gradually, refining the price function iteratively before arriving at a final reweighting. This refinement improves monotonically in the following sense. We call a vertex a ***negative vertex*** if it has any incoming edges with negative weight.[4] We call a price function a τ***-improvement*** if it has the following properties:

(1) (Valid.) The edge weights after reweighting are all integers with value at least -1. Thus, the reweighted graph is still a valid instance of the 1-reweighting problem.
(2) (Monotonic.) If an edge has nonnegative weight before reweighting, then it has nonnegative weight after. Consequently, reweighting does not introduce negative vertices.
(3) (Progress.) At least τ negative vertices before reweighting are no longer negative vertices after reweighting. That is, all of their incoming edges are reweighted to at least 0. We also say that those τ vertices are ***improved*** or ***eliminated***.

[4] Goldberg uses the term "improvable" instead of negative vertex, but we find the latter more intuitive.

Algorithm 4 Outline of Goldberg's algorithms [16] for the 1-reweighting problem
Input: Graph $G = (V, E)$ and weight function $w : E \to \mathbb{Z}$ where all weights are integers of at least -1.
Output: a feasible price function p, or a negative cycle if none exists.

```
1:  initially p(v) = 0 for all v
2:  let w_p denote the weights w_p = w(u,v) + p(u) − p(v) with
    respect to the price function p
3:  while there are negative vertices with respect to w_p do
4:      contract cycles of 0-weight edges
5:      let k be the number of negative vertices remaining
6:      find one of the two following objects:
7:          (1) a negative cycle C
8:          (2) price function p′: a √k-improvement w.r.t. w_p
9:      if a negative cycle was found then
10:         extend C to a cycle in the uncontracted graph
11:         return the cycle C
12:     foreach vertex v in the contracted graph do
13:         foreach x in v's component do
14:             update p(x) ← p(x) + p′(v)
15: return the price function p
```

At a high level, Goldberg's algorithm [16] simply repeatedly finds "large" τ-improvements until no negative vertices remain.

Algorithm 4 gives an outline of Goldberg's algorithm for the 1-reweighting problem. The algorithm repeatedly contracts cycles of 0-weight edges and then finds τ-improvements. The reason for contraction will make more sense as we go into more detail on the algorithm.[5] But it should be obvious that (1) contraction can only reduce the number of negative vertices, and (2) all vertices in a contracted component are equivalent with respect to distance to/from all other vertices. As vertices in the contracted graph are reweighted, the same reweighting is trivially extended to vertices in the same contracted component (Line 14).

More specifically, the algorithm finds $\sqrt{k}$-improvements (Line 8), where k is the current number of negative vertices in the contracted graph. The number of negative vertices thus reduces from k to at most $k - \sqrt{k}$ in each iteration. The algorithm terminates in $O(\sqrt{K})$ iterations [16], where K is the initial number of negative vertices. Since $K \le n$, the algorithm completes in $O(\sqrt{n})$ iterations.

Contraction and finding a $\sqrt{k}$-improvement

The core of the reweighting problem is the problem of either finding a $\sqrt{k}$-improvement or a negative cycle, where k denotes the current number of negative vertices in the graph. Goldberg [16] provides an algorithm for this subroutine that has running time $O(m)$. This thus implies a running time of $O(m\sqrt{n})$ for the 1-reweighting problem, and $O(m\sqrt{n}\log N)$ for SSSP with integer weights. The remainder of this section gives an overview of Goldberg's algorithm for finding a $\sqrt{k}$ improvement.

[5] We are vague about bookkeeping details here because it does not actually matter whether the contracted graph is maintained across iterations, or whether each iteration begins from the original graph. This is because all cycles of 0's are preserved across the reweighting steps.

Much of Goldberg's algorithm operates on a subgraph of G. Specifically, let $G_{\le 0}$ denote the subgraph of G containing only those edges whose weights are 0 and -1. That is, all edges with strictly positive edge weight are removed from the graph.

Step 1: strongly connected components. Goldberg's algorithm [16] begins by finding the strongly connected components in $G_{\le 0}$. If any component contains a negative edge, i.e., $w(u, v) = -1$ and u and v are strongly connected, then report a negative-weight cycle by finding any path from v to u in $G_{\le 0}$. Otherwise, contract each strongly connected component to get the condensation $G'_{\le 0}$ of $G_{\le 0}$. The main reason for this contraction step is that the condensation $G'_{\le 0}$ is acyclic; the remaining steps rely on the graph being acyclic.

Step 2: find a large chain or independent set of negative vertices. The next step involves finding one of the following two objects:

- (Chain.) A length $\tau \ge \sqrt{k}$ sequence of negative-weight edges $\langle (u_1, v_1), (u_2, v_2), \ldots, (u_\tau, v_\tau)\rangle$ such that, for $1 \le i < \tau$, there is a path from v_i to u_{i+1} in $G'_{\le 0}$.
- (Independent set.) A set S of negative vertices, with $|S| \ge \sqrt{k}$, such that for all $u, v \in S$, there is no negative-weight path from u to v in $G'_{\le 0}$.

To find these, augment $G'_{\le 0}$ with a supersource s, add edges from s to all other vertices with weight 0, and solve SSSP on the augmented graph from source s. Because $G'_{\le 0}$ is acyclic, this SSSP problem can be solved in $O(m)$ time (see, e.g., [11]). Partition the vertices into sets $V_0, V_1, V_2, \ldots$ by distance, where V_i denotes the set of vertices having shortest-path distance $-i$ from the supersource.

If any vertex has distance $-\tau \le -\sqrt{k}$, then any shortest path to that vertex contains τ negative edges — the chain is the sequence of negative edges along any such shortest path.

Otherwise, all negative vertices are in $V_1, V_2, \ldots, V_{\lceil\sqrt{k}\rceil-1}$. Let $S_i \subseteq V_i$ be the negative vertices in V_i. Select the largest S_i as the independent set.

Step 3: improve the chain or independent set. It is important to note that although the chain or independent set is found in $G'_{\le 0}$, the improvement/reweighting is applied to the original graph G.

Reweighting to improve all vertices in an independent set is straightforward. Let S be the independent set, and let V^R be the set of vertices reachable in $G'_{\le 0}$ from any vertex in S. For each $v \in V^R$, set $p(v) = -1$. For all other vertices, set $p(v) = 0$. If the independent set is found by the algorithm above, identifying V^R requires no additional work: for $S = S_i$, we have $V^R = V_i \cup V_{i+1} \cup \cdots \cup V_{\lceil\sqrt{k}\rceil-1}$.

Improving the vertices on the chain is harder because the price function takes on many values. Goldberg solves that problem in two ways [16], one of which involves reducing the problem to nonnegative single-source shortest paths. We describe a slightly different reduction in Section 6. In contrast to the independent-set case, improvement here may not be possible if there is a negative-weight cycle in the full graph G. Thus, the algorithm may report a negative-weight cycle instead of performing the improvement.

Summary of key subroutines. Goldberg's algorithm [16] for $\sqrt{k}$ improvement uses the following subroutines: strongly connected components, SSSP in a directed acyclic graph with negative weights, and SSSP in a general graph with nonnegative integer weights.

Each of these subroutines can be solved easily in $\tilde{O}(m)$ sequential running time using classic algorithms [11]. In fact, they can be solved in $O(m)$ time with some care [16].

6 OVERVIEW OF PARALLEL ALGORITHM

As discussed in Section 5, Goldberg's algorithm for SSSP is a scaling algorithm that performs multiple iterations of 1-reweighting and $\sqrt{k}$-improvement. The correctness argument [16] relies on these iterations being performed one at a time, (i.e., sequentially). We thus apply the same sequential scaling technique. Moreover, we use the same sequential loop as given in Algorithm 4. The difference is that we use a parallel subroutine for the relevant contractions and the $\sqrt{k}$-improvement, we use a parallel version of Dijkstra's for the final SSSP output, and that the obvious trivial steps (e.g., updating the price function) are performed in parallel. With no changes to the main structure, correctness follows from Goldberg [16].

This section outlines our algorithm assuming no negative-weight cycles. We discuss reporting cycles in Appendix A.2.

The main technical contribution of this paper are the new distance-limited SSSP problems we use for $\sqrt{k}$-improvement. The resulting performance for $\sqrt{k}$-improvement is summarized by the following theorem; this is a direct consequence of our efficient distance-limited SSSP algorithms.

Theorem 16. *There exists a randomized parallel algorithm having work $\tilde{O}(m)$ and span $n^{3/4+o(1)}$, with high probability, for the $\sqrt{k}$-improvement problem. Specifically, the algorithm takes as input a directed graph with n vertices, m edges, and integer weights of at least -1. The algorithm contracts all cycles of 0-weight edges and either finds a price function giving a $\sqrt{k}$-improvement, where k is the number of vertices with incoming negative edges in the contracted graph, or it determines that a negative-weight cycle exists.*

Assuming correctness of our algorithm for improving the chain (see Lemma 19 in Appendix A.1), correctness follows from Goldberg's [16] version of the algorithm. It remains to analyze the work and span of the algorithm. As noted previously, Step 1 can be completed with $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span. Step 2 is dominated by the $\sqrt{k}$-distance-limited $\{0,-1\}$-weight acyclic SSSP. In Section 3, we give an algorithm for that problem that has work $\tilde{O}(m)$ and span $n^{3/4+o(1)}$ for distance limit $L = \left\lceil \sqrt{k} \right\rceil = O(\sqrt{n})$. Finally, Step 3 is dominated by the problem of improving the chain, which we have reduced to $\sqrt{k}$-distance-limited nonnegative SSSP. Section 4 gives our solution to that problem, which also has work $\tilde{O}(m)$ and span $n^{3/4+o(1)}$ when the distance is limited to $O(\sqrt{n})$.

Finally, we must also consider the cost of determining whether the algorithm should terminate due to the presence of a negative-weight cycle. (Finding the actual cycle is harder, but that only occurs once; just determining that such a cycle exists is easier.) Goldberg's algorithm [16] and our extension only terminate due to negative-weight cycles in two spots: in Step 1 and in Step 3, specifically when trying to eliminate the chain. Testing for a negative-weight cycle in Step 1 involves simply testing whether a negative-weight edge falls within a component, which is trivial to perform in $O(m)$ work and $O(\log n)$ span. Lemma 19 gives us a way of testing for a negative-weight cycle—simply perform the reweighting and see if all v_i on the chain are indeed improved. If not, then there must be a negative-weight cycle. Again, this test is trivial to perform in $O(m)$ work and $O(\log n)$ span.

Adding the work and span across all steps yields the claimed bounds.

Given our efficient parallel algorithm for $\sqrt{k}$-improvement, we get our main result on SSSP as a simple corollary.

Theorem 17. *There exists a randomized parallel algorithm for the problem of SSSP with integer edge weights with the following characteristics: for an input graph with n vertices, m edges, integer weights of at least $-N$, and no negative-weight cycles, the algorithm returns the shortest-path distance from the source to all vertices with $\tilde{O}(m\sqrt{n}\log N)$ work and $n^{5/4+o(1)}\log N$ span, both with high probability. Moreover, the algorithm can be augmented to find and return a negative-weight cycle, if one exists, in the same work and span.*

Proof. Goldberg's main algorithm performs $O(\log N)$ iterations of scaling and $O(\sqrt{n})$ iterations of $\sqrt{k}$ improvement [16] to reweight the graph. Assuming no negative-weight cycle, the cost to find a feasible reweighting of the original graph can thus be obtained by multiplying the bounds of Theorem 16 by $O(\sqrt{n}\log N)$, which matches the claimed bounds.

After finding a feasible reweighting, we also need to solve the nonnegative SSSP problem. There exist several parallel versions [7, 12] of Dijkstra's algorithm having work $\tilde{O}(m)$ and span $\tilde{O}(n)$, which falls within the target bounds.

As discussed in Section A.2, a negative cycle can be reported $\tilde{O}(m)$ work and $\tilde{O}(n)$ span. □

6.1 Parallel contraction and $\sqrt{k}$-improvement

We now discuss each of the main steps of $\sqrt{k}$-improvement for the parallel version. This remainder of this section describes how each of the steps differ from those given in Section 5. Steps 2 and 3 are where we apply new algorithms, namely those given in Sections 3 and 4, respectively.

Step 1: strongly connected components. We use existing algorithms as a black box. Notably Blelloch et al. [6] reduce strongly connected components to single-source reachability (with logarithmic overheads), and Jambulapati et al. [18] solve single-source reachability with work $\tilde{O}(m)$ and span $n^{1/2+o(1)}$, with high probability. Given the strongly connected components, constructing the contracted graph is an easy reduction to sorting, for which there are many parallel algorithms with $\tilde{O}(m)$ work and polylogarithmic span [17].

Step 2: find a large chain or independent set of negative vertices. Section 5 reduces this problem to that of single-source shortest paths in an acyclic graph with edge weights from $\{0,-1\}$. The observation we make here is that it is only important to find the exact distances to vertices with small distance. In particular, the goal is to identify sets $V_0, V_1, V_2, \ldots, V_L$, where V_i is the set of all vertices with shortest-path distance exactly $-i$ from the source vertex. We need only compute these sets up to $L = \lceil\sqrt{k}\rceil$. For the remaining vertices, i.e., those at distance strictly less than $-L$, we do not need their exact distances.

Section 3 provides our parallel algorithm for this $\{0,-1\}$ weight distance-limited shortest-path problem. To facilitate reporting the chain, our algorithm also reports negative-edge ancestors along the

shortest paths. Specifically, for each vertex $v \in V_i$ with $1 \le i \le L$, we also identify an ancestor negative edge (x, y) such that (1) $x \in V_{i-1}$, and (2) there is a path from y to v in $G'_{\le 0}$. To find a chain, simply start with any vertex $x \in V_L$; the last edge on the chain is x's negative ancestor (u_L, v_L); each preceding edge can be obtained sequentially/iteratively by taking u_i's negative ancestor Finding the chain thus takes $O(L) = O(\sqrt{n})$ work and span.

Step 3: improve the chain or independent set. Improving an independent set follows the process outlined in Section 5. This process is already trivial to parallelize by considering all vertices in parallel.

In the case of a chain, we adapt Goldberg [16]'s reduction to non-negative SSSP, except that our reduction also works for distance-limited SSSP. Because Goldberg's [16] reduction is not distance limited, we present our reduction and correctness here. Nevertheless, the ideas are the same. Here we present our reduction to distance-limited SSSP. For completeness, we include a correctness argument in Appendix A.1. Section 4 provides our solution to the distance-limited SSSP problem, which is the hard part.

Suppose we have a length-L chain $\langle (u_1, v_1), (u_2, v_2), \ldots, (u_L, v_L) \rangle$ of negative edges. (For correctness below, it does not matter what algorithm is used to find a chain.) To improve the chain, construct a graph $\hat{G}$ as follows. Start with $\hat{G} = G$. (Note we are working with the contracted version of the full graph G, not $G_{\le 0}$.) For all existing edges, use the weight function $\hat{w}(x, y) = \max\{0, w_p(x, y)\}$. The weights thus reflect the current reweighting of G but with all negative edges rounded up to 0. Add a supersource s to $\hat{G}$, and add the edges (s, x) for each vertex $x \in V$ in the graph. For each v_i in the chain, set $\hat{w}(s, v_i) = L - i$. For all other vertices, set $\hat{w}(x, v_i) = L$. Observe that all weights $\hat{w}$ are, by construction, nonnegative. Next, for all x, compute the shortest path distance from s to x, denoted by $d(x)$. Note that by construction, all such shortest-path distances are at most L, so using a distance-limited SSSP algorithm suffices. Finally, for each vertex x, set $p'(x) = d(x) - L$.

ACKNOWLEDGEMENTS

This research was supported in part by NSF grants CCF-1918989 and CCF-2106759.

REFERENCES

[1] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd ACM Symposium on Theory of Computing*, 2020.

[2] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 93–104. IEEE, 2020.

[3] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in almost-linear time. *CoRR*, abs/2203.03456, 2022.

[4] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 253–264, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, July 2020.

[6] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 467–478, 2016.

[7] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1):4–21, February 1998.

[8] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Efficient construction of directed hopsets and parallel approximate shortest paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, page 336–349, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, January 2000.

[10] Michael B. Cohen, Aleksander Mądry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in Õ(m10/7 log w) time: (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 752–771, USA, 2017. Society for Industrial and Applied Mathematics.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[12] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, November 1988.

[13] M. Elkin and O. Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 128–137, Los Alamitos, CA, USA, oct 2016. IEEE Computer Society.

[14] Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in rnc. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 333–341, New York, NY, USA, 2019. ACM.

[15] Harold N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, 1985.

[16] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[17] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[18] Arun Jambulapati, Yang P. Liu, and Aaron Sidford. Parallel reachability in almost linear work and square root depth. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1664–1686. IEEE Computer Society, 2019.

[19] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, November 1997.

[20] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd ACM Symposium on Theory of Computing*, 2020.

[21] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 192–201, 2015.

[22] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 919–930. IEEE, 2020.

A ELIMINATE CHAIN, REPORTING A NEGATIVE CYCLE, AND MAIN THEOREMS

This section presents some details omitted from Section 6. Specifically, we prove that the reduction for eliminating a chain is correct, and we discuss how to report a specific negative-weight cycle as needed.

A.1 Correctness of eliminate chain

Consider the algorithm for reweighting the chain discussed in 6.1. We first argue that this reweighting is always at least a 0-improvement. We then argue that, if there are no negative-weight cycles in the graph, then it also improves all v_i on the chain, making it an L-improvement.

Lemma 18. *Consider a graph $G = (V, E)$ with integer weights $w_p(x, y) \ge -1$ for all edges $(x, y) \in V$. Suppose that the above algorithm for eliminating a chain is applied to the graph on some chain. Then the resulting weights $w_{p'}(x, y) = w_p(x, y) + p'(x) - p(y)$ satisfy the following:*

- *(Valid.) $w_{p'}(x, y)$ are integers, and $w_{p'}(x, y) \ge -1$.*
- *(Monotonic.) $w_{p'}(x, y) < 0$ only if $w_p(x, y) < 0$.*

Proof. All edge weights in $\hat{G}$ are integers, so all shortest-path distances are also integers. It follows that p' is always an integer, and hence so is $w_{p'}$.

Consider any edge (x, y) in $\hat{G}$. By the triangle inequality of SSSP, we have $d(y) \leq d(x) + \hat{w}(x, y)$, or $d(y) - L \leq d(x) - L + \hat{w}(x, y)$. Substituting in p' we get $p'(y) \leq p'(x) + \hat{w}(x, y)$, or $p'(x) - p'(y) \geq -\hat{w}(x, y)$. It follows that the reweighting in G satisfies $w_{p'}(x, y) = w_p(x, y) + p'(x) - p'(y) \geq w_p(x, y) - \hat{w}(x, y)$. In the event that $w_p(x, y) \geq 0$, $\hat{w}(x, y) = w_p(x, y)$, and we have $w_{p'}(x, y) \geq 0$, satisfying the two conditions for this edge. Otherwise, $w_p(x, y) = -1$ (the edge is already negative) and $\hat{w}(x, y) = 0$, now yielding $w_{p'}(x, y) \geq -1$, which again satisfies the conditions. □

Lemma 19. *Consider the algorithm for eliminating a chain applied to a chain $\langle (u_1, v_1), (u_2, v_2), \ldots, (u_L, v_L) \rangle$ on graph $G = (V, E)$. If the graph G contains no negative-weight cycles, then the reweighting p' improves all v_i along the chain, i.e., these vertices have no incoming negative edges w.r.t. $w_{p'}$.*

Proof. The only aspect missing from Lemma 18 is to show that the specific vertices are actually improved.

We first argue that if the distance from s to v_i in $\hat{G}_i$ is $d(v_i) < L - i$, then there is a negative-weight cycle in the graph G. (Note $d(v_i) \leq L - i$ due to the direct edge from the source, so no negative-weight cycle in G implies $d(v_i) = L - i$.) Consider a shortest path Γ_i from s to v_i in $\hat{G}$. This path Γ_i must start with some edge (s, v_j), for $j > i$, as these are the only edges leaving s with weight strictly less than $L - i$. Let $\Gamma_{j \rightsquigarrow i}$ be the subpath of Γ_i from v_j to v_i. The total length of Γ_i is thus $\hat{w}(\Gamma_i) = \hat{w}(\Gamma_{j \rightsquigarrow i}) + \hat{w}(s, v_j) = \hat{w}(\Gamma_{j \rightsquigarrow i}) + (L - j)$. For $L - i > \hat{w}(\Gamma_i)$, we have $L - i - 1 \geq \hat{w}(\Gamma_{j \rightsquigarrow i}) + (L - j)$, or $\hat{w}(\Gamma_{j \rightsquigarrow i}) \leq j - i - 1$. Edge weights only increase in $\hat{w}$, so $w_p(\Gamma_{j \rightsquigarrow i}) \leq j - i - 1$ in G as well. Finally, let $\Gamma_{i \rightsquigarrow j}$ be any shortest path in G from v_i to v_j. Because these vertices fall along the chain, we have $w_p(\Gamma_{i \rightsquigarrow j}) \leq i - j$. The cycle formed by linking $\Gamma_{j \rightsquigarrow i}$ with $\Gamma_{i \rightsquigarrow j}$ thus has total length at most $(j - i - 1) + (i - j) \leq -1$.

For the remainder, suppose that $d(v_i) = L - i$ for all i, and hence $p'(v_i) = -i$. Consider any edge (x, v_i). We now argue that if $w_{p'}(x, v_i) < 0$, then the graph contains a negative-weight cycle. It follows that if there are no negative-weight cycles, then all of v_i's incoming edges are reweighted to nonnegative weight. To start, by Lemma 18 the reweighted edge is only negative if $w_{p'}(x, v_i) = w_p(x, v_i) = -1$. Thus, we must have $p'(x) = p'(v)$, which means that $p'(x) = L - i$. We can now complete the proof as above: a shortest path from s to x in $\hat{G}$ must contain a subpath from some v_j to x with length at most $j - i$. Similarly, the shortest path length from v_i to v_j in G is at most $i - j$. Because the edge (x, v_i) itself has negative length, the total cycle length is at most -1. □

A.2 Reporting a negative-weight cycle

This section addresses the problem of identifying a specific negative-weight cycle when our parallel version of Goldberg's algorithm (Section 6) terminates early. Because a negative-weight cycle is only reported once, there is no reason to optimize the work and span bounds of this algorithm beyond those stated in Theorem 17. We thus only shoot for $\tilde{O}(m)$ work and $\tilde{O}(n)$ span here.

First, suppose a negative-weight cycle is detected in Step 1. That is, there is a negative-weight edge (x, y) where x and y are in the same strongly connected component of $G_{\leq 0}$. Then finding the cycle entails simply finding any path from y to x. This can be achieved by running any parallel version of breadth-first search having work $\tilde{O}(m)$ and span $\tilde{O}(n)$.

Suppose instead that the cycle is detected in Step 3 for the chain $\langle (u_1, v_1), (u_2, v_2), \ldots, (u_L, v_L) \rangle$. Then we need identify the vertices on a negative-weight cycle. Observe that the proof of Lemma 19 actually gives us a way of identifying a specific cycle constructively. The main takeaway is that we need to find two shortest paths: (1) a shortest path to some vertex x in $\hat{G}$, where either $x = v_i$ from some v_i along the chain, or (x, v_i) is a negative edge in G, and (2) a shortest path in $G'_{\leq 0}$ from v_i to some v_j with $j > i$.[6]

Our algorithm for nonnegative distance-limited SSSP (Section 4) returns a shortest-path tree in the form of predecessor pointers, so a shortest path in $\hat{G}$ can be recovered in $O(n)$ sequential time, and hence $O(n)$ work and span, by tracing backwards.

Our algorithm for producing the chain (Section 3) does not output a shortest path tree, but recovering such a shortest path in $G'_{\leq 0}$ is also not hard. Observe that by construction for the edges (u_i, v_i) on the chain, $1 < i \leq L$: (1) $u_i \in V_{i-1}$, (2) $v_{i-1} \in V_{i-1}$, and (3) there exists a path from v_{i-1} to u_i in $G_{\leq 0}$. The goal is to recover specific subpaths from v_{i-1} to u_i to fill-in the complete shortest path. Observe that because both $v_{i-1}, u_i \in V_{i-1}$, it follows that every shortest path from v_{i-1} to u_i uses only 0-weight edges in the subgraph of $G'_{\leq 0}$ induced by V_{i-1}. So we need only look for any path from v_{i-1} to u_i in this induced subgraph, e.g., by using parallel BFS. Because the V_i's are disjoint, the total work across all L BFSes is at most $\tilde{O}(m)$. Moreover, all L BFSes can be performed in parallel, keeping the span to $\tilde{O}(n)$.

To conclude, we note that the paths in $G'_{\leq 0}$ and $\hat{G}$ are both with respect to the contracted graph. Extending these paths to the original graph is also straightforward. For each component, we just need to find a path from the endpoint of the entering edge to the endpoint of the exiting edge. Because the components are disjoint, we can again apply parallel BFS to each such component in parallel.

[6] Specifically, if there is any v_i with $d(v_i) < L - i$ in $\hat{G}$, then choose $x = v_i$. Otherwise, choose any unimproved incoming edge (x, v_i). Finally, the vertex v_j is the first hop on a shortest-path from s to x in $\hat{G}$.