Single-Source Shortest Paths with Negative Real Weights in $\tilde{O}(mn^{8/9})$ Time

Jeremy T. Fineman
Georgetown University
WASHINGTON, USA
jf474@georgetown.edu

ABSTRACT

This paper presents a randomized algorithm for single-source shortest paths on directed graphs with real (both positive and negative) edge weights. Given an input graph with n vertices and m edges, the algorithm completes in $\tilde{O}(mn^{8/9})$ time with high probability. For real-weighted graphs, this result constitutes the first asymptotic improvement over the classic $O(mn)$-time algorithm variously attributed to Shimbel, Bellman, Ford, and Moore.

CCS CONCEPTS

• **Theory of computation → Shortest paths**.

KEYWORDS

shortest paths, randomized algorithms

ACM Reference Format:
Jeremy T. Fineman. 2024. Single-Source Shortest Paths with Negative Real Weights in $\tilde{O}(mn^{8/9})$ Time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24), June 24–28, 2024, Vancouver, BC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3618260.3649614

1 INTRODUCTION

This paper considers the problem of single-source shortest paths (SSSP) with possibly negative real weights. The input to the SSSP problem is a directed graph $G = (V, E, w)$ with real edge weights given by the function $w : E \to \mathbb{R}$ and a designated source vertex s. If the graph does not contain negative-weight cycles, then the goal is to output the shortest-path distance from the source s to every vertex $v \in V$. If there is a negative-weight cycle in the graph, then the algorithm should instead report the presence of such a cycle.

The classic algorithm for SSSP with real weights, due to Shimbel [20], Ford [11], Bellman [2], and Moore [18], henceforth called the Bellman-Ford algorithm, has a running time of $O(mn)$ on a graph with m edges and n vertices. With no further restrictions to graph topology or weights, this algorithm remains the best known algorithm for SSSP.

If weights are all *nonnegative* reals, Dijkstra's algorithm applies, which can be made to run in $O(m + n \log n)$ time using Fibonacci heaps [12]. For undirected graphs, Duan et al. [9] obtain a running time of $O(m\sqrt{\log n \cdot \log\log n})$ using a randomized algorithm.

For the case of *integer* weights (negative and positive), there has been significant further progress [1, 5, 6, 13–15, 21], culminating in nearly linear-time algorithms [3, 4]. These integer-weight solutions apply scaling or optimization techniques, and they all include at least a $\log W$ term in their running times, where $-W$ is the most-negative weight in the graph. Because the number of arithmetic operations performed depends on the magnitude of the weights and not just the size of the graph, these algorithms are all *weakly polynomial*. The $O(mn)$-time Bellman-Ford algorithm remains the best strongly polynomial runtime known.

The main result of this paper is captured by the following theorem. The model used throughout is the Real RAM (see, e.g., [10]), which augments the word RAM with unit-cost arithmetic operations on real numbers.

Theorem 1.1. *There exists a (Las Vegas) randomized algorithm that solves the SSSP problem for real-weighted graphs in $\tilde{O}(mn^{8/9})$ time, with high probability, where m is the number of edges and n is the number of vertices in the graph.*

The algorithm in this paper uses the Real RAM in a "reasonable" way. Notably, the only arithmetic operations on reals employed herein are addition, subtraction, negation, and comparison. As such, the usage of the Real RAM here is consistent with the "comparison-addition model" used in prior work [9, 19]. Moreover, all intermediate real numbers computed throughout the execution correspond to the sum/difference of at most a polynomial number of edge weights. (Proof of this claim and is deferred to the full version of the paper.) As such, the result also immediately translates to integer weights in the word RAM, and hence the algorithm is strongly polynomial.

Note that efficiently handling arithmetic on *rational* numbers in the word RAM presents different challenges that are not addressed herein. (These challenges exist even in the case of nonnegative edge weights.) Most notably, intermediate rational numbers may have significantly larger binary representations than the input weights. Karczmarz et al. [17] provide SSSP algorithms designed specifically for rational weights in the word RAM.

1.1 Preliminaries

The $\tilde{O}$ denotes the soft-O notation. Formally, $f(x) = \tilde{O}(g(x))$ if there exists an integer k such that $f(x) = O(g(x) \cdot \log^k(g(x)))$.

For the remainder, consider a graph $G = (V, E, w)$, let $m = |E|$ and $n = |V|$. For a vertex $v \in V$, $out(v)$ denotes the set of v's outgoing edges. For a subset $X \subseteq V$ of vertices, $out(X)$ denotes the set of outgoing edges from X, i.e., $out(X) = \{(x, y) \in E | x \in X\}$.

STOC '24, June 24–28, 2024, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0383-6/24/06
https://doi.org/10.1145/3618260.3649614

For a path p, the total ***weight*** of the path is given by $w(p) = \sum_{e \in p} w(e)$. The ***size*** of the path is the number of edges on the path, denoted by $|p|$. A ***cycle*** C is a path that starts and ends at the same vertex, and a negative-weight cycle is one where $w(C) < 0$. A path p from u to v is a ***shortest path*** if all u-to-v paths p' satisfy $w(p) \le w(p')$. If there exists a shortest path p from u to v, then we define the ***shortest-path distance*** from u-to-v as $dist_G(u,v) = w(p)$; if there is no u-to-v path, then $dist_G(u,v) = \infty$; if there is a path but no shortest path (i.e., there is a negative-weight cycle), then $dist_G(u,v) = -\infty$. When G is clear from context, we often write $dist(u,v)$ in place of $dist_G(u,v)$.

For a subset $S \subseteq V$ of vertices, the shortest-path distance from any vertex in S to v, denoted by $dist_G(S,v)$, is defined as

$$dist_G(S,v) = \min_{u \in S} (dist_G(u,v)) \ .$$

The problem of computing $dist_G(S,v)$ for all $v \in V$ corresponds to that of solving SSSP on a slightly augmented graph: create a "super source" vertex s, for all $u \in S$ add edges (s,u) with $w(s,u) = 0$ to the graph, and finally solve SSSP from the super source s in the augmented graph. Johnson's algorithm [16] uses this same graph augmentation with $S = V$.

Simplifying assumptions (without loss of generality). We assume throughout that every vertex has degree at most $O(m/n)$; thus, a subgraph on n/r vertices has $O(m/r)$ edges. This assumptions is without loss of generality as it can be obtained from an arbitrary input graph via a simple graph transformation without increasing the size of the graph by more than a constant factor.

The following two assumptions serve to simplify the statement of performance bounds. First, we assume that $m = \Omega(n)$. Second, we assume that at least a constant fraction of the edges have nonnegative weight. As such, the nonnegative-weight edges dominate the size of the graph.

Hop-limited shortest paths. It is a simple exercise to construct a SSSP algorithm that runs in $\tilde{O}(hm)$ time when shortest paths are limited to $h \ge 1$ negative-weight edges or "hops." (Section 2 introduces corresponding notation and briefly summarizes such an algorithm.) The novel algorithm in this paper applies hop-limited SSSP as a black-box subroutine.

Price functions. As with most of the integer-weight algorithms for SSSP, the algorithm in this paper relies on price functions introduced by Johnson [16] to transform the graph to an equivalent one without negative weights; then Dijkstra's algorithm can be used to solve the SSSP problem on the reweighted graph. In more detail, a ***price function*** is a function $\phi : V \to \mathbb{R}$. Given a price function ϕ, define $w_\phi(u,v) = w(u,v) + \phi(u) - \phi(v)$ and $G_\phi = (V, E, w_\phi)$. Modifying the weights in this way has the following key properties [16]: (1) a path p is a shortest path in G_ϕ if and only if it is a shortest path in G, and (2) every cycle C has the same weight in both G and G_ϕ, so negative-weight cycles are preserved. More precisely, all u-to-v paths p satisfy $w_\phi(p) = w(p) + \phi(u) - \phi(v)$; if p is a cycle then $\phi(u) = \phi(v)$ and hence $w_\phi(p) = w(p)$. Price functions also compose in the natural way, i.e., $(w_{\phi_1})_{\phi_2}(u,v) = w_{\phi_1+\phi_2}(u,v)$.

We call ϕ or w_ϕ a ***valid reweighting*** if w_ϕ does not cause any edge weights to become negative. That is, if $\forall e \in E((w(e) \ge 0) \implies (w_\phi(e) \ge 0))$. We call a vertex a ***negative vertex*** if it has an outgoing edge with negative weight initially. We say that ϕ or w_ϕ ***eliminates*** a negative vertex v if for all $e \in out(v)$, $w_\phi(e) \ge 0$.

Johnson [16] shows that, assuming no negative-weight cycles, the problem of eliminating *all* negative vertices can be accomplished by setting $\phi(v) = dist(V,v)$. Using Bellman-Ford to solve the super-source problem, the running time is $O(mn)$. When there are $k \ll n$ negative vertices, applying hop-limited SSSP is better, giving a running time of $\tilde{O}(km)$.

See Section 2 for further discussion of negative edges, negative vertices, and valid reweightings. Importantly, these classifications are decided at the beginning of each invocation of the elimination algorithm, discussed next, with no change until the next one.

1.2 Main Result

This paper solves the problem of efficiently computing a reweighting that eliminates a significant number of negative vertices. We say that an algorithm is an ***$f(k)$-elimination algorithm*** if, when given an input graph $G = (V, E, w)$ with k negative vertices, i.e., k vertices having negative-weight outgoing edges, the algorithm (1) computes a valid reweighting that eliminates at least $f(k)$ of these negative vertices[1], or (2) correctly determines that the graph contains a negative-weight cycle. Given an $f(k)$-elimination algorithm $\mathcal{A}$, SSSP can be solved by repeatedly applying $\mathcal{A}$ until no negative vertices remain, and then applying Dijkstra's algorithm. This strategy of gradually eliminating negative vertices is reminiscent of Goldberg's algorithm [15] for integer-weighted graphs.

Theorem 1.2. *There exists a randomized $\Theta(k^{1/3})$-elimination algorithm for real-weighted graphs that has running time $\tilde{O}(mk^{2/9})$, with high probability, where m and k are the number of edges and negative vertices in the input graph, respectively.*

Theorem 1.1 is a corollary of Theorem 1.2. A similar argument occurs in [15], so the full proof is omitted here. The main idea is that $O(k^{2/3})$ repetitions of $\Theta(k^{1/3})$-elimination suffice to reduce the number of negative vertices by a constant factor. The total running time of these repetitions is $\tilde{O}(mk^{8/9}) = \tilde{O}(mn^{8/9})$ to reduce $k \le n$ by a constant factor. And $O(\log n)$ of these constant-factor reductions are enough to eliminate all negative vertices.

Sketch of algorithm. The remainder of this paper focuses on solving the problem of $\Theta(k^{1/3})$-elimination, thereby proving Theorem 1.2. At a very high level, the algorithm reweights the graph so that $\Theta(k^{1/3})$ of the negative vertices are "remote" or "far away" from most of the graph. (In particular, only an $O(1/k^{1/9})$ fraction of the graph is "nearby" these vertices.) Then, it reweights the graph again to eliminate these $\Theta(k^{1/3})$ negative vertices using Johnson's strategy. Because these remote vertices are far from most of the graph, it turns out that it is possible to eliminate them in $\tilde{O}(k^{1/3} \cdot (m/k^{1/9})) = \tilde{O}(k^{2/9}m)$ time, which improves over the straightforward but insufficient $\tilde{O}(k^{1/3}m)$ bound by a $k^{1/9}$ factor.

A key challenge is, of course, to establish this remote subset of negative vertices. The algorithm performs several gradual reweighting steps to ensure remoteness, each applying hop-limited shortest paths. In slightly more detail, the first reweighting selects a random

[1] The reweighted graph G_ϕ thus has at most $k - f(k)$ negative vertices.

sample of vertices and use hop-limited shortest paths "spread out" the graph. Next, the algorithm searches for a large subset of negative vertices that are relatively "close together," or failing that finds a large subset that are "independent." (Resolving the latter case is easier.) The goal now is only to eliminate this subset of negative vertices, so all other negative edges (those not incident on these vertices) are removed from consideration. A subsequent reweighting moves most of the graph away from these close-together vertices, which renders them remote. Then a final reweighting step is performed to eliminate these now remote vertices; this last reweighting is the only one guaranteed to eliminate any negative vertices.

Outline. Before giving any further detail of the algorithm, Section 2 establishes useful notations and definitions to formalize these types of manipulations. Section 3 then gives an overview of the algorithm with some intuition. Finally, Sections 4–7 provide details of each step of the algorithm and the analysis.

2 PRELIMINARIES

This section provides basic definitions and notation. In addition, this section discusses one of the main black-box subroutines: hop-limited shortest path. There are various definitions introduced later in the paper as well, but most of those represent novel insights into the structure of an efficient solution. This section also includes several useful claims for which the proofs are all simple exercises and hence omitted.

Transpose graph and STSP. For a fixed target t, the problem of computing $dist_G(u,t)$ for all $u \in V$ is called the ***single-target shortest-paths (STSP)*** problem. This problem can be solved by solving SSSP from t in the transpose graph. The ***transpose graph*** is the graph obtained by reversing all the edges. That is, the transpose graph is a graph $G^T = (V, E^T, w^T)$ where $E^T = \{(v,u)|(u,v) \in E\}$ and $w^T(v,u) = w(u,v)$.

Negative vertices, negative edges, nonnegative edges, and the input graph. The ***input graph*** refers to the graph G on which the main algorithm of Theorem 1.2 is called, possibly with a modified weight function. We shall always denote the input graph by $G = (V, E^+ \cup E^-, w)$, where the edge set has been partitioned into the ***nonnegative edges*** E^+ and the ***negative edges*** E^-. Initially, $E^+ = \{e \in E | w(e) \geq 0\}$ and $E^- = \{e \in E | w(e) < 0\}$, where $E = E^+ \cup E^-$ is the full edge set. For every edge $(u,v) \in E^-$, the vertex u is called a ***negative vertex***. Throughout, let $n = |V|$, $m = |E|$, and k denote the number of negative vertices.

As a slight abuse of notation, the $\cup$ symbol in $G = (V, E^+ \cup E^-, w)$ is not simply a union, but also signifies which edges are classified as negative edges (those in E^-), and which are nonnegative (those in E^+). As the algorithm progresses, the weight function changes, but the classification of edges does not. Thus, having a negative edge $(u,v) \in E^-$ with $w(e) \geq 0$ is allowed; that edge is still called a negative edge, and u is still called negative vertex. In contrast, because the algorithm only produces valid price function, it shall always be the case that $w(e) \geq 0$ for all $e \in E^+$. Because this last premise always holds, it is omitted from most lemma/theorem statements for the sake of readability.

Whenever the partition is not provided, e.g., if referring to an auxiliary graph $H = (V', E', w')$, then implicitly the term "negative edges" refers to those edges whose weight is negative.

(Negative)-hop-limited paths and distances. A path p is an ***h-hop path*** if at most h of the edges on the path are negative edges. Nonnegative edges do not count towards the number of hops. Paths need not be simple, and each occurrence of a negative edge contributes to the hop count.

The h-hop distance, denoted

$$dist^h_G(u,v) = \min\{w(p) | p \text{ is an } h\text{-hop path from } u \text{ to } v \text{ in } G\} ,$$

is the weight of a shortest h-hop path from u to v; define $dist^h_G(u,v) = \infty$ if there is no path from u to v. We also extend the distance notation for distance from a set of vertices (as in Section 1). Specifically, for any $S \subseteq V$, define $dist^h_G(S,v) = \min_{u \in S}\left(dist^h_G(u,v)\right)$. When G is clear from context, we often write $dist^h$ instead of $dist^h_G$.

Just as with normal distance, it is easy to see that h-hop distances obeys the following modified triangle inequality, which has been adjusted to incorporate the hop counts.

Lemma 2.1 (Triangle inequality). *For all integers $h_1, h_2 \geq 0$ and all vertices x, y, z, we have*

$$dist^{h_1+h_2}(x,z) \leq dist^{h_1}(x,y) + dist^{h_2}(y,z) .$$

For any nonnegative edge (y,z), $dist^{h_1}(x,z) \leq dist^{h_1}(x,y) + w(y,z)$.

If $dist^h_G(u,v) < 0$ or $dist^h_G(v,u) < 0$, then we say that u and v are ***negatively h-hop related***. The ***negative h-hop reach*** of a vertex u is the set of vertices that can be reached by a negative-weight h-hop path. More generally, for a set subset $S \subseteq V$ of vertices, the negative h-hop reach of S is

$$R^h_G(S) = \left\{v \in V | dist^h_G(S,v) < 0\right\} .$$

The ***size*** of the reach is its cardinality. As with distance, the subscript G may be dropped when G is clear from context.

Reweighting and invariance of h-hop paths. The algorithm performs several steps that each partially reweight the graph by way of a sequence of price functions ϕ. The notation $G_\phi = (V, E^+ \cup E^-, w_\phi)$ denotes the reweighted graph, i.e., the input graph reweighted by price function ϕ. When G is clear from context, we use the subscript ϕ as a shorthand for G_ϕ in all notations where the subscript specifies the graph of concern, e.g., $dist^h_\phi$ means $dist^h_{G_\phi}$.

The classification of edges as negative or nonnegative does not change when the graph is reweighted, and the validity of the price function is defined with respect to the initial classification. Specifically, a price function ϕ is ***valid*** if for all $e \in E^+$, $w_\phi(e) \geq 0$. When going from price function ϕ to ϕ', function ϕ' may still be valid even if $w_\phi(e) \geq 0$ and $w_{\phi'}(e) < 0$ as long as $e \in E^-$.

Importantly, since the classification of edges does not change, h-hop paths in the input graph are invariant across reweighting. That is, a path p is an h-hop path in $G_\phi = (V, E^+ \cup E^-, w_\phi)$ if and only if it is an h-hop path in $G = (V, E^+ \cup E^-, w)$. Ensuring this invariant is the primary reason negative edges were defined in the specific manner above. This invariant allows us to more-cleanly reason about paths and distances when the algorithm performs

several reweighting steps, avoiding any issues that could arise if the algorithm "accidentally" makes an edge weight nonnegative. Specifically, we immediately have the following.

Lemma 2.2. *Consider the input graph $G = (V, E^+ \cup E^-, w)$, and let ϕ be a price function. Then for all $u, v \in V$, we have*

$$dist^h_\phi(u,v) = dist^h(u,v) + \phi(u) - \phi(v) .$$

Computing h-hop distances. Given a source vertex s, the problem of computing h-hop distances from s to all other vertices is called the ***h-limited SSSP problem***. There is a natural solution for h-limited SSSP that combines Bellman-Ford and Dijkstra's algorithm, called BFD here.[2] BFD interleaves $(h+1)$ full executions of Dijkstra's algorithm (but without reinitializing distances) on the nonnegative edges and h "rounds" of Bellman-Ford on the negative edges.[3] The running time of BFD is thus $O(hm \log n)$ when $h \geq 1$.

Lemma 2.3 (Follows from, e.g., [3, 8]). *Consider a graph $G = (V, E^+ \cup E^-, w)$, let $n = |V|$ and $m = |V|$, and let k denote the number of negative vertices. BFD solves the h-limited SSSP problem in time $O((h+1)(m+n \log n))$. That is, given source vertex s and integer $h \geq 0$, it returns $d_h(v) = dist^h_G(s,v)$ for all $v \in V$. Moreover, the algorithm can also return all smaller-hop distances $d_{h'}(v) = dist^{h'}_G(s,v)$ for all $h' \in \{0, 1, 2, \ldots, h\}$ with the same running time.*

For $h = k+1$, BFD solves the regular SSSP problem.

More generally, given a set $S \subseteq V$, it is possible to compute the distances $d_{h'}(v) = dist^{h'}_G(S,v)$ for all $v \in V$ and $h' \leq h$ with the same time complexity. In addition, for all $v \in V$, the algorithm can be augmented to return $s(v) \in S$ such that $d_h(v) = dist^h_G(s(v), v)$.

Note that some textbook descriptions of Bellman-Ford (e.g., CLRS [7]) update distance estimates in place, which when extended to BFD would only guarantee $d_h(v) \leq dist^h_G(s,v)$. The inequality may be problematic when reasoning about hop-limited paths. We instead require the return values to be exactly the h-hop distances.

Subgraphs of negative edges. For a subset $X \subseteq V$ of negative vertices on the input graph, we use $out^-(X) = E^- \cap out(X)$ to denote the negative edges outgoing from X. We use G^X to denote the subgraph $G^X = (V, E^+ \cup out^-(X), w)$, i.e., the subgraph with all negative edges except those leaving X removed. Moreover, G^X_ϕ denotes the reweighted subgraph $G^X_\phi = (V, E^+ \cup out^-(X), w_\phi)$. Because negative vertices are defined to be those vertices with outgoing negative edges, X is the set of all negative vertices in G^X.

Since all of the nonnegative edges are included in G^X, it should be obvious that for any price function ϕ, if w_ϕ is a valid reweighting of G^X then it is also a valid reweighting of G. Moreover, since all of X's outgoing edges appear in G^X, if w_ϕ eliminates X in G^X then it also eliminates those vertices in G. Working with subgraphs G^X thus suffices to solve the problem. Specifically, the algorithm shall eventually reach a subgraph G^X with $|X| = \Theta(k^{1/3})$ and find a reweighting that eliminates X from G^X.

[2] See, e.g., [8], for a deeper discussion of one variant of this algorithm. Bernstein et al. [3] apply an optimized version of BFD that does not reconsider a vertex in the next round unless its distance has improved; their algorithm for integer-weight SSSP leverages a tighter bound for the case that most shortest paths have few hops.

[3] A "round" of Bellman-Ford means "relaxing" all the edges once. A full execution of Bellman-Ford is n rounds.

Algorithm 1: Algorithm for eliminating $\Theta(k^{1/3})$ negative vertices. Negative-weight cycles may be discovered inside steps (1), (2), or (4), causing early termination.

input: A graph $G = (V, E^+ \cup E^-, w)$ with $w(e) \geq 0$ for $e \in E^+$ and $w(e) < 0$ for $e \in E^-$

let k be the number of negative vertices and let $r = \Theta(k^{1/9})$

1 (Section 3.2, 4) perform betweenness reduction on G with $\beta = r+1$ and $\tau = r$

let ϕ_1 be the price function computed by this step

2 (Section 3.4, 5) find a size-$\Omega(k^{1/3})$ negative sandwich (x, U, y) or independent set I in G_{ϕ_1}

if *this step discovers an independent set* **then**

  (Section 3.5) find p.f. ϕ that eliminates I in $G^I_{\phi_1}$

  return $\phi + \phi_1$

else arbitrarily remove vertices from U until $|U| = \Theta(k^{1/3})$

3 (Section 3.3, 6) reweight the graph G_{ϕ_1} to try to make U become r-remote

let ϕ_2 be the price function computed by this step

if $\left|R^r_{\phi_1+\phi_2}(U)\right| > n/r$ **then** restart Algorithm 1

4 (Section 3.1, 7) use the hop-reduction technique on $G^U_{\phi_1+\phi_2}$ to eliminate U

let ϕ be the price function computed by this step

return $\phi + \phi_1 + \phi_2$

3 ALGORITHM OVERVIEW

This sections provides an overview of the algorithm for $\Theta(k^{1/3})$ elimination. This section includes some intuition of correctness for each of the main components of the algorithm, but the details and most of the proofs are deferred to Sections 4–7 or the full paper.

The main goal of the algorithm is to find either a large (i.e., size-$\Theta(k^{1/3})$) r-remote set or a large 1-hop independent set of negative vertices, both defined next, and then to eliminate that set. (We shall eventually set $r = \Theta(k^{1/9})$).

Definition 3.1. Consider a graph $G = (V, E^+ \cup E^-, w)$, let $n = |V|$, and let X be a subset of negative vertices. If the negative r-hop reach of X has size at most n/r, i.e., $|R^r(X)| \leq n/r$, then X is an ***r-remote set***. We also call the subgraph induced by the negative r-hop reach of X an ***r-remote subgraph***.

Definition 3.2. Consider a graph $G = (V, E^+ \cup E^-, w)$. Let I be a subset of negative vertices. We say that I is a ***1-hop independent set*** if $\forall x, y \in I$, x and y are not negatively 1-hop related in G.

Algorithm 1 outlines the algorithm. Note that some of the terminology will be revealed later in this section. Nevertheless, the reader may wish to refer to this psuedocode to see how steps fit together. Each of the main steps is marked with the corresponding sections that explain them. For expository reasons, these steps are presented out of order in this overview section (but in order later in the paper). The algorithm produces a sequence of price functions through several steps. Each step computes the next price function relative to the current weighting of the graph. Thus, the actual weight is obtained by composing (adding) all of the price functions.

Roughly speaking, there are two main components in the algorithm. The first component is an efficient algorithm to either find a large r-remote set (also with large r) or, failing that, to find a large 1-hop independent set. Unfortunately, neither may exist with the original weight function of the graph—it is not hard to construct graphs where (1) every pair of negative vertices is negatively 1-hop related, and (2) every negative vertex has large 1-hop reach, i.e., $\left|R^1(\{u\})\right| = \Omega(n)$. The first component of the algorithm thus entails not simply finding such a set, but also adjusting the weight function to ensure that such a set exists. This component spans all but the last numbered step in the pseudocode.

The second component is an efficient algorithm that eliminates all of outgoing edges from the r-remote or 1-hop-independent set. The second problem is easier, and it also helps to motivate why r-remote sets are useful. Thus, this section addresses the second component first. (Efficiently eliminating a 1-hop independent set is almost trivial, so that is deferred to Section 3.5.)

3.1 Hop Reduction: Eliminate Remote Vertices

Recall that Johnson's strategy [16] for eliminating negative vertices entails solving SSSP. If there are $\hat{k}$ negative vertices, then the running time is $\tilde{O}(m\hat{k})$ using BFD. The goal is to accelerate this SSSP computation for the case that the negative vertices are remote.

To illustrate the approach, consider first a graph $G = (V, E^+ \cup N, w)$, where X is the set of negative vertices, but X is not known to be remote. (Notationally, the use of N and $\hat{k} = |X|$ here serve to emphasize that this step is applied to a subgraph; however, because this entire section concerns the same subgraph and not the original graph, we write G instead of G^X to simplify the notation.)

The goal is to produce a new auxiliary graph $H = (V_H, E_H, w_H)$ such that (1) $V \subseteq V_H$, and (2) for all hop counts $h \geq 0$ and $u, v \in V$, $dist^h_G(u,v) \geq dist^{\lceil h/r \rceil}_H(u,v) \geq dist_G(u,v)$. That is to say, all h-hop paths in G correspond to $\lceil h/r \rceil$-hop paths in H. We say that H is an ***r*-hop reduction of *G***. If there are no negative-weight cycles in G, then shortest paths are simple and have at most $\hat{k}$ hops where $\hat{k}$ is the number of negative vertices. Thus, we can compute SSSP for G by instead computing SSSP in H with a cost of $\tilde{O}((\hat{k}/r) \cdot m_H)$, where $m_H = |E_H|$ is the size of H. As we shall see next, there is a fairly straightforward construction of an $O(rm)$-size r-hop reduction of G. Unfortunately, the running time of SSSP remains $\tilde{O}((\hat{k}/r) \cdot (rm)) = \tilde{O}(\hat{k}m)$. But given an r-remote set, it is possible to improve this construction and running time.

The construction of H is roughly as follows. First, for each vertex $v \in V$, add $r+1$ copies $v = v_0, v_1, \ldots, v_r$ to V_H. Add the nonnegative edges to each layer of the graph, i.e., for each edge $(u,v) \in E^+$ and each $0 \leq i \leq r$, add the edge (u_i, v_i) to E_H. As for the negative edges $(u,v) \in N$, create the edges (u_i, v_{i+1}) for $0 \leq i < r$ to E_H. Each copy of the negative edge thus moves from the i-th layer of the graph to the $(i+1)$th layer. Finally, add edges (v_i, v_0) for all v and i to allow a way to get back to the 0th layer.

It remains to specify the weight function w_H. The goal is to ensure that only the edges (v_i, v_0) have negative weight, and thus an r-hop subpath in G can be simulated by a 1-hop path in H that moves through copies $0, 1, 2, \ldots r, 0$. This goal can be accomplished by roughly running Johnson's reweighting limited to r hops, i.e., computing i-hop SSSP from V for all $i \leq r$, and setting $w_H(u_i, v_j) = w(u,v) + dist^i_G(V,u) - dist^j_G(V,v)$. For each $(u,v) \in N$, it follows that $w_H(u_i, v_{i+1}) \geq 0$ because $dist^{i+1}_G(V,v) \leq dist^i_G(V,u) + w(u,v)$.

The graph H has size $m_H = O(rm)$ by construction. Moreover, from Lemma 2.3 the SSSP distances and hence weights w_H can be calculated in $\tilde{O}(rm)$ time.

Now let us improve the construction if X is an r-remote set. Consider a vertex $u \in V$ that falls outside the r-remote subgraph. Then r-remoteness implies that $dist^i_G(V,u) = 0$ for all $i \leq r$ as there is no negative-weight path and there is a 0-weight path (the empty path from u). There is thus no reason to include multiple copies of this vertex in H as each copy's incident edges would be weighted identically—it suffices to keep the single copy $u = u_0$, or equivalently to contract all copies into u and remove any redundant edges. In summary, when given an r-remote subgraph, H comprises r copies of the remote subgraph plus a single copy of the original graph. Applying the assumption that the maximum degree is $O(m/n)$, the total size of H now becomes $m_H = O(r \cdot (n/r) \cdot (m/n) + m) = O(m)$. Moreover, H still constitutes an r-hop reduction of G. We are thus left with the following lemma; (the second term in the runtime is the cost of constructing w_H).

Lemma 3.3. *Consider a (sub)graph $G = (V, E^+ \cup N, w)$. Let X be the set of negative vertices, $m = \left|E^+\right|$, and $\hat{k} = |X|$. If X is r-remote, then there exists an $\tilde{O}((\hat{k}/r)m + rm)$-time deterministic algorithm that either (1) correctly determines that G contains a negative-weight cycle, or (2) computes a valid reweighting that eliminates X.*

3.2 Betweenness Reduction

We are left with the more difficult problem of uncovering an r-remote set or 1-hop independent set, which as previously noted entails some reweighting. But it is not clear how to attack this problem directly. Roughly speaking, one of the challenges is that even though the classification as negative edges is invariant across reweighting, changing the price of a vertex affects distances. In particular, if $\phi(u) \ll \phi(v)$, it is possible that $dist^h_G(u,v) \geq 0$ and $dist^h_{G_\phi}(u,v) < 0$, i.e., new negative h-hop relationships may be introduced. It thus seems difficult to argue that a particular reweighting causes the number of h-hop relationships to decrease.

The key insight here is to think in terms of "betweenness" instead of direct distances. We can then later translate to an r-remote set, but that transformation is more restricted so easier to reason about.

Definition 3.4. For the following, consider a graph G, vertices u, x, and v, and integer $\beta \geq 0$.

The ***β*-distance from *u* to *v* through *x***** is defined as

$$thru^\beta_G(u,x,v) = dist^\beta_G(u,x) + dist^\beta_G(x,v)\ .$$

We say that x is ***β*-between** u and v if $thru^\beta_G(u,x,v) < 0$. The ***β*-betweenness** of u and v is the number of vertices β-between u and v, denoted $BW^\beta_G(u,v) = \left|\left\{x \in V | thru^\beta_G(u,x,v) < 0\right\}\right|$.

For all of these notations, the G may be dropped if clear from context, and ϕ is used as shorthand for G_ϕ.

The goal is to find a price function ϕ so that for given parameter τ, all pairs $u, v \in V$ have $BW^\beta_\phi(u,v) \leq n/\tau$. (We will use $\tau = \beta - 1 = r$, but the algorithm is described with parameters β and τ.) The algorithm is fairly simple. Sample a size-$\Theta(\tau \log n)$ subset of vertices.

Then find any reweighting for which all β-hop distances to or from the sampled vertices are nonnegative, or determine that the graph contains a negative-weight cycle. Roughly speaking, the reweighting entails computing $\Theta(\beta\tau \log n)$-limited SSSP (because we want $\Theta(\beta)$-hop subpaths between each of the $\Theta(\tau \log n)$ samples). There are many relatively straightforward ways to achieve the desired reweighting, and the details are deferred to Section 4.

We are left with a question: does reweighting in this way ensure that $BW^{\beta}_{\phi}(u,v) \le n/\tau$? Consider the distance from u to v through a vertex x. It follows from Lemma 2.2 that $thru^{\beta}_{\phi}(u,x,v) = thru^{\beta}(u,x,v) + \phi(u) - \phi(v)$, which importantly does not depend on $\phi(x)$. The u-to-v distances through other vertices thus compare in the same way before and after reweighting. Therefore, if any sampled vertex y has $thru^{\beta}(u,y,v) \le thru^{\beta}(u,x,v)$, then it follows that x is not β-between u and v in G_ϕ because y is not either. With high probability, there is a sample y taken from the smallest $1/\tau$-fraction of through distances, and hence at most a $1/\tau$ fraction of vertices is β-between u and v in G_ϕ. Thus, we obtain the following, with proof in Section 4:

Lemma 3.5. *Consider input graph $G = (V, E^+ \cup E^-, w)$ and let $m = |E^+|$. Then there exists an $\tilde{O}(\beta\tau m + \tau^2 n)$-time (Monte Carlo) randomized algorithm that always satisfies one of the following three cases, and it falls in one of the first two with high probability: (1) it correctly determines the graph contains a negative-weight cycle, (2) it finds valid price function ϕ such that $BW^{\beta}_{\phi}(u,v) \le n/\tau$ for all $u, v \in V$, or (3) it returns a valid price function, but the betweenness goal is not achieved.*

3.3 From Sandwiches to r-Remoteness

Consider a graph $G = (V, E^+ \cup E^-, w)$. The goal is to argue that if G has low betweenness, then it is not too hard to reweight G so that there is an r-remote subset. To do so, we apply a new object called a negative sandwich.

Definition 3.6. A ***negative sandwich*** is a triple (x, U, y) with the following properties.

- U is a subset of negative vertices,
- $x \in V$ and $dist^1(x,u) < 0$ for all $u \in U$, and
- $y \in V$ and $dist^1(u,y) < 0$ for all $u \in U$.

The ***size*** of the sandwich is the cardinality of U.

For now, let us ignore the task of finding such a sandwich. The goal here is only to argue that a negative sandwich is useful.

Given a negative sandwich (x, U, y) and hop count β, consider the price function $\phi(v) = \min(0, \max(dist^{\beta}(x,v), -dist^{\beta}(v,y)))$. Roughly speaking, there are two main goals of this price function: (1) for all $u \in U$, $\phi(u) = 0$, and (2) for most other vertices v, $\phi(v) \le 0$ and $\phi(v) \le dist^{\beta}(x,v)$. Because the 1-hop distance from x to u is negative (by definition of a negative sandwich), these together would imply that the $(\beta-1)$-hop distance from u to v in the reweighted graph becomes positive. In general, however, ensuring (1) in a way that also gives a valid reweighting somewhat interferes with (2). This is why the price function here uses $dist^{\beta}(v,y)$ to limit how negative $\phi(v)$ can get. It is not hard to see that (1) is ensured because in a negative sandwich $dist^{\beta}(u,y) < 0$ for all $u \in U$. The price function also ensures (2) because when v is *not* β-between x and y, $dist^{\beta}(x,v) + dist^{\beta}(v,y) \ge 0$ or $dist^{\beta}(x,v) \ge -dist^{\beta}(v,y)$; the implication is that $\max(dist^{\beta}(x,v), -dist^{\beta}(v,y)) = dist^{\beta}(x,v)$ as desired. That is to say, the only vertices that remain in the $(\beta-1)$-hop reach of U in G_ϕ are (a subset of) those vertices that are β-between x and y in G. It follows that if x and y have β-betweenness at most n/τ, then U becomes $\min(\tau, \beta-1)$-remote.

The following lemma formalizes these ideas and also proves that the reweighting is valid. That the reweighting is valid may not be obvious, but the proof (in Section 6) essentially amounts to applying the triangle inequality.

Lemma 3.7. *Consider a graph $G = (V, E^+ \cup E^-, w)$, a negative sandwich (x, U, y), and an integer $\beta > 1$. Let ϕ be the price function*

$$\phi(v) = \min(0, \max(dist^{\beta}_{G}(x,v), -dist^{\beta}_{G}(v,y))) .$$

Then we have the following:

1. *ϕ is a valid reweighting, i.e., $w_\phi(e) \ge 0$ for all $e \in E^+$.*
2. *For every $v \in V$: if $thru^{\beta}_{G}(x,v,y) \ge 0$ (i.e., v is not β-between x and y), then $v \notin R^{\beta-1}_{G_\phi}(U)$.*

Choosing $\beta = r+1$ and $\tau = r$ as parameters in the betweenness reduction (i.e., Lemma 3.5), we obtain the following corollary:

Corollary 3.8. *Suppose we are given a negative sandwich (x, U, y) and that $BW^{r+1}(x,y) \le n/r$ for integer $r \ge 1$. Then there is an $O(rm \log n)$-time deterministic algorithm that finds a valid price function ϕ such that U is r-remote in G_ϕ.*

Proof. Lemma 3.7 gives the following properties of ϕ: (1) ϕ is valid, and (2) $\left|R^{(r+1)-1}_{G_\phi}(U)\right| \le BW^{r+1}(x,y) \le n/r$. Thus, U's r-hop reach has size at most n/r, which means that U is r-remote. The price function ϕ can be computed by solving $(r+1)$-limited SSSP from x and $(r+1)$-limited STSP to y. Applying the running time for BFD (Lemma 2.3) completes the proof. □

This step may fail to make U become r-remote only if the Monte Carlo betweenness reduction failed to ensure that x and y have low betweenness; in this case, the entire algorithm must be restarted.

3.4 Finding a Sandwich or Independent Set

The final problem is that of finding a negative sandwich or 1-hop independent set. The main tool is given by the following lemma, proved in Section 5.

Lemma 3.9. *Consider an input graph $G = (V, E^+ \cup E^-, w)$; let $n = |V|$ and $m = |E^+|$. Let U_0 be any subset of negative vertices in G, let $\hat{k} = |U_0|$, and let ρ be an integer parameter with $1 \le \rho \le \hat{k}$.*

There exists a (Las Vegas) randomized algorithm whose running time is $O(m \log^2 n)$, with high probability, that takes as input G, U_0, and ρ and always does one of the following:

1. *correctly determines that G contains a negative-weight cycle,*
2. *returns a subset $U \subseteq U_0$ of negative vertices and vertex y such that $|U| = \Omega(\hat{k}/\rho)$ and for all $u \in U$, $dist^1(u,y) < 0$, or*
3. *returns a 1-hop independent set $I \subseteq U_0$ with $|I| = \Omega(\rho)$.*

Given Lemma 3.9, we immediately obtain the following corollary by running the algorithm twice with $\rho = \Theta(k^{1/3})$. The first

execution is on the original graph with U_0 being all negative vertices, which finds y but reduces the number of negative vertices to $\Omega(k^{2/3})$. The second execution uses the transpose graph, finding x and reducing the number of negative vertices to $\Omega(k^{1/3})$.

Corollary 3.10. *Consider an input graph $G = (V, E^+ \cup E^-, w)$; let $n = |V|$, $m = |E^+|$, and let k be the number of negative vertices.*

There exists a (Las Vegas) randomized algorithm whose running time is $O(m \log^2 n)$, with high probability, that always does one of the following:

(1) *correctly determines that G contains a negative-weight cycle,*
(2) *returns negative sandwich (x, U, y) with $|U| = \Omega(k^{1/3})$, or*
(3) *returns a 1-hop independent set I with $|I| = \Omega(k^{1/3})$.*

We are left with the problem of constructively proving Lemma 3.9. Let $C(U_0, v) = \left|\left\{u \in U_0 | dist^1_G(u, v) < 0\right\}\right|$ denote the number of vertices in U_0 that can reach v with a negative-weight 1-hop path.

The algorithm for Lemma 3.9 is roughly as follows, with details in Section 5. The first task is to estimate $C(U_0, v)$ for all $v \in V$. More precisely, the goal is to partition U_0 into two subsets H and L (for heavy and light, respectively), such that $\forall v \in H, C(U_0, v) = \Omega(\hat{k}/\rho)$ and $\forall v \in L, C(U_0, v) = O(\hat{k}/\rho)$. This task can be accomplished by randomly sampling each vertex $u \in U_0$ with probability $\rho/\hat{k}$ into a subset U', then computing $R^1(U')$. If $C(U_0, v) \gg \hat{k}/\rho$, then it is reasonably likely that $v \in R^1(U')$. Conversely, if $C(U_0, v) \ll \hat{k}/\rho$, then it is likely that $v \notin R^1(U')$. Repeating this process $\Theta(\log n)$ times and applying a Chernoff bound allows us to correctly partition the vertices, with high probability.

If H is nonempty, then select any y in H and run STSP to compute $U = \left\{u \in U_0 | dist^1(u, y) < 0\right\}$. Finally, verify that $|U| = \Omega(\hat{k}/\rho)$ just in case the estimation procedure failed.

If instead H is empty, then $L = U_0$, and all vertices $v \in U_0$ should have $C(U_0, v) = O(\hat{k}/\rho)$. Then it is straightforward to construct a large random independent set. Select a uniformly random subset $I' \subseteq U_0$ with $|I'| = \Theta(\rho)$. Then, set $I = I' - R^1(I')$, where "$-$" here denotes set subtraction, which ensures that the set I is independent. For each vertex in $v \in I'$, there is only a constant probability that there is another vertex $u \in I', u \neq v$ such that $dist^1(u, v) < 0$. Thus, as long as there no negative-weight cycles, there is at least a constant probability that $|I| = \Omega(\rho)$. Repeating $\Theta(\log n)$ times gives high probability of successfully finding an independent set.

3.5 The Full Algorithm

Assuming the lemmas stated in this section, we have almost all the components necessary to prove Theorem 1.2. The only remaining pieces are eliminating an independent set and determining the appropriate value for r.

Eliminating an independent set. Let I be a 1-hop independent set of negative vertices in the graph $G = (V, E^+ \cup E^-, w)$. Consider the subgraph G^I. Then simply use the price function $\phi(v) = dist^1_{G^I}(V, v)$, which can be computed by running 1-hop BFD in $O(m \log n)$ time. It is not too hard to see that this price function accomplishes the task. (Proof is in the full paper.)

Lemma 3.11. *Consider the input graph G and let I be any 1-hop independent set of negative vertices. Then the price function given by $\phi(v) = dist^1_{G^I}(V, v)$ is a valid price function that eliminates all negative vertices from G^I.*

Algorithm 2: Algorithm for betweenness reduction

input : A graph $G = (V, E^+ \cup E^-, w)$
input : Parameters τ and β and constant $c > 1$, with $\beta \geq 1$ and $1 \leq \tau \leq |V|$

1 let $n = |V|$
2 let $T \subseteq V$ be a uniformly random subset of $c\tau\lceil \ln n \rceil$ vertices
3 **foreach** $x \in T$ **do**
4 run β-hop SSSP and STSP, computing β-hop distances from and to x, respectively
5 construct a new graph $H = (V, E_H, w_H)$ as follows:
 $E_H = (T \times V) \cup (V \times T)$
 $w_H(u, v) = dist^\beta_G(u, v)$ using precomputed distances to/from vertices in T
6 let $\ell = 2|T|$ (which equals $2c\tau\lceil \ln n \rceil$)
7 compute super-source distances $d(v) = dist^\ell_H(V, v)$ and $d'(v) = dist^{\ell+1}_H(V, v)$ for all $v \in V$
8 **if** $\exists v$ *such that* $d'(v) < d(v)$ **then** terminate algorithm and report "cycle"
9 **else return** *price function* $\phi = d$

Choosing r to minimize runtime. Fixing $\beta - 1 = \tau = r$, there are two components that dominate the running time of the algorithm: the betweenness reduction, with a running time of $\tilde{O}(r^2 m)$ (Lemma 3.5), and eliminating the r-remote subset using hop reduction, with a running time of $\tilde{O}((k^{1/3}/r)m + rm)$ (Lemma 3.3). The total running time is thus $\tilde{O}(m \cdot (r^2 + k^{1/3}/r))$, which is minimized by setting $r = \Theta(k^{1/9})$, yielding $\tilde{O}(k^{2/9} m)$, as per Theorem 1.2.

Proving Theorem 1.2. The proof of Theorem 1.2 is not interesting—it primarily entails tracing through the steps of Algorithm 1 and applying the appropriate lemmas. Specifically, use Lemma 3.5 for Step 1, Corollary 3.10 and Lemma 3.11 for Step 2, Lemma 3.5 and Corollary 3.8 for Step 3, and Lemma 3.3 for Step 4. Due to space constraints, the full proof is deferred to the full version of the paper. The one potentially interesting point is that Lemma 3.5 gives a Monte Carlo algorithm, and a failure is not observed until Step 3.

4 BETWEENNESS REDUCTION

This section expands on the betweenness reduction introduced in Section 3.2, with the goal of proving Lemma 3.5. Throughout, let $G = (V, E^+ \cup E^-, w)$ denote the input graph and let $n = |V|$ and $m = |E^+|$. The variables β and τ denote the parameters for betweenness reduction, with $\beta \geq 1$ and $1 \leq \tau \leq |V|$. Recall that the goal is to find a price function ϕ such that for all vertices u, v, we have $BW^\beta_\phi(u, v) \leq n/\tau$. The algorithm is parameterized by a constant $c \geq 3$ used to adjust the probability of success.

Algorithm 2 presents the algorithm for betweenness reduction. The algorithm begins by sampling a subset T of vertices with $|T| = c\tau\lceil \ln n \rceil$ vertices. The remainder of the algorithm is devoted to reweighting the graph so that all β-hop distances to or from vertices in T become nonnegative.

There are many straightforward ways to accomplish the goal of nonnegative β-hop distances to/from T; Algorithm 2 is just one concrete example. Algorithm 2 proceeds by computing all β-hop distances from each vertex in T and all β-hop distances to each vertex in T, using SSSP and STSP, respectively. Then, an auxiliary graph H is constructed. The graph H contains all edges of the form (x, v) and (v, x) where $x \in T$ and $v \in V$. Thus, all edges in H are, by construction, incident on a vertex in T. The weights of these edges are the corresponding β-hop distances in G that have already been computed. The final step of the algorithm is to apply Johnson's strategy [16] to H. That is, compute distances to each vertex using super-source shortest paths. Because all edges are incident on a vertex in T, the computation stops at $2|T| + 1$ hops, at which point either the algorithm has discovered a negative-weight cycle, or the $2|T|$-hop distances are the actual shortest path distances in H. Finally, these distances are returned as a price function for G.

There are two main aspects of correctness to prove. (1) The algorithm finds a price function ϕ such that all β-hop distances to/from $x \in T$ in G_ϕ are nonnegative. The idea here is that from Johnson's strategy [16], the shortest-path distances in H constitute a valid price function ϕ that eliminates all negative edges in H. These edges in H correspond to β-hop paths in G to/from vertices in T. Thus, applying ϕ to G ensures that these β-hop paths have nonnegative weight. (2) The algorithm reduces the betweenness of all pairs to at most n/τ, as discussed in Section 3.2. The claims, along with running time, are proved next.

Lemma 4.1. *Consider an execution of Algorithm 2 on input graph $G(V, E^+ \cup E^-, w)$ starting from line 3 with any arbitrary subset $T \subseteq V$. (That is, this claim does not rely on any randomness of the sample.) Then we have the following:*

- *If the algorithm reports a negative-weight cycle, then G contains a negative-weight cycle.*
- *Otherwise, the algorithm returns a price function ϕ such that for all $v \in V$ and $x \in T$: $dist^\beta_{G_\phi}(x, v) \geq 0$ and $dist^\beta_{G_\phi}(v, x) \geq 0$. Moreover, if the initial weight satisfies $w(e) \geq 0$ for all $e \in E^+$, then the price function is valid.*

Proof. Let us start with the following observation: all simple paths in H have size at most $2|T|$, which follows from the fact that all edges in H are incident on vertices in T. (If the path is larger, some vertex in T has at least 2 incoming or outgoing edges, and hence the path is not simple.) Simple paths therefore also have at most $2|T|$ negative-weight edges. Thus, H has a negative-weight cycle if and only if there exists a vertex v such that $dist^{\ell+1}_H(V, v) < dist^\ell_H(V, v)$, where $\ell = 2|T|$. We have thus established that a cycle is reported if and only if H has a negative-weight cycle. Moreover, if no cycle is reported, then $d(v)$ is the actual super-source distance in H, so the standard (not hop-limited) triangle inequality applies to d.

Next, suppose that H has a negative-weight cycle C. Then it is easy to see that G does as well: replace each edge in C with the corresponding h-hop path in G, which by construction has the same weight. Therefore, when the algorithm reports a negative-weight cycle, that result is correct.

For the remainder, suppose that there is no negative-weight cycle in H, so a price function is returned. Here we prove the claim that the distance to/from each sample is nonnegative. By the standard triangle inequality, for all $x \in T$ and $v \in V$ (and hence $(x, v) \in E_H$), we have $d(v) \leq d(x) + w_H(x, v) = d(x) + dist^\beta_G(x, v)$, or $dist^\beta_G(x, v) + d(x) - d(v) \geq 0$. Setting $\phi = d$ and using Lemma 2.2, we thus have $dist^\beta_{G_\phi}(x, v) = dist^\beta_G(x, v) + \phi(x) - \phi(v) = dist^\beta_G(x, v) + d(x) - d(v) \geq 0$. Similarly, by the symmetric argument now considering the edge $(v, x) \in E_H$, we have $d(x) \leq d(v) + w_H(v, x) = d(v) + dist^\beta_G(v, x)$, or $dist^\beta_G(v, x) + d(v) - d(x) \geq 0$. Thus $dist^\beta_{G_\phi}(v, x) = dist^\beta_G(v, x) + \phi(v) - \phi(x) \geq 0$.

Finally, let us address the validity of the price function $\phi = d$. We shall again prove this using the triangle inequality. The only issue here is that H does not include all edges in E^+, so we cannot directly apply the triangle inequality on computed distances to these edges. Start by noting that $d(v) \leq 0$ for all $v \in V$ from the empty path. Now consider any edge $(u, v) \in E^+$ and suppose $w(u, v) \geq 0$. If $d(u) = 0$ then trivially $w_\phi(u, v) = w(u, v) - d(v) \geq w(u, v) \geq 0$. Suppose instead that $d(u) < 0$. Then a shortest path to u in H is nonempty and must end with a last edge $(x, u) \in E_H$ for some $x \in T$; that is, $d(u) = d(x) + w_H(x, u)$. By Lemma 2.1 on G, for $(u, v) \in E^+$ we have $dist^\beta_G(x, v) \leq dist^\beta_G(x, u) + w(u, v)$, and hence $w_H(x, v) \leq w_H(x, u) + w(u, v)$. Thus, using the triangle inequality in H we have $d(v) \leq d(x) + w_H(x, v) \leq d(x) + w_H(x, u) + w(u, v) = d(u) + w(u, v)$, or $w_\phi(u, v) = w(u, v) + d(u) - d(v) \geq 0$. □

Applying Lemma 2.3 for the hop-limited SSSP computations, it is not hard to prove the following bound on the running time. (Proof is deferred to the full paper.)

Lemma 4.2. *Consider input graph $G = (V, E^+ \cup E^-, w)$ and let $m = |E^+|$ and $n = |V|$. Then there is a realization of Algorithm 2 that runs in $O(\beta\tau \log n(m + n \log n) + \tau^2 n \log^2 n)$ time.*

When $\beta - 1 = \tau = \Theta(r)$, this bound simplifies to $\tilde{O}(r^2 m)$.

Lemma 4.3. *Consider an execution of Algorithm 2 on input graph $G = (V, E^+ \cup E^-, w)$ and let $n = |V|$. Then with probability at least $1 - 1/n^{c-2}$, the algorithm either*

- *correctly reports a negative-weight cycle, or*
- *returns a p.f. ϕ such that for all $u, v \in V$, $BW^\beta_\phi \leq n/\tau$.*

Proof. Fix any pair $u, v \in V$. The proof shows the claim holds with high probability for this pair. Then taking a union bound across all n^2 pairs proves the lemma. All distances in this proof are distance in G or G_ϕ, so the subscript G is omitted.

Number and order all the vertices in V as $x_1, x_2, \ldots, x_n$ such that $thru^\beta(u, x_1, v) \leq thru^\beta(u, x_2, v) \leq \cdots \leq thru^\beta(u, x_n, v)$. Now let $y = x_j$ be the sampled vertex with lowest index/rank in the numbering. If the algorithm reports a cycle, then by Lemma 4.1 this reporting is correct. For the remainder, suppose instead that the algorithm returns a price function ϕ.

By Lemma 4.1, ϕ is such that $dist^\beta_\phi(u, y) \geq 0$ and $dist^\beta_\phi(y, v) \geq 0$ and hence $thru^\beta_\phi(u, y, v) \geq 0$. From Lemma 2.2, for all $x \in V$, we have $thru^\beta_\phi(u, x, v) = dist^\beta(u, x) + \phi(u) - \phi(x) + dist^\beta(x, v) + \phi(x) - \phi(v) = thru^\beta(u, x, v) + \phi(u) - \phi(v)$. Moreover, for all x_i with $i \geq j$, we have $thru^\beta(u, x_i, v) \geq thru^\beta(u, y, v)$, and hence $thru^\beta_\phi(u, x_i, v) = thru^\beta(u, x_i, v) + \phi(u) - \phi(v) \geq thru^\beta(u, y, v) +$

Algorithm 3: Algorithm for heavy/light partition

input : A graph $G = (V, E^+ \cup E^-, w)$
input : Subset U_0 of negative vertices and integer ρ with $1 \le \rho \le |U_0|$
output: A partition $\langle H, L = U_0 - H\rangle$ of U_0

HL-Partition($G = (V, E^+ \cup E^-, w)$, U_0, ρ)

```
let k̂ = |U_0|
foreach v ∈ V do count(v) ← 0
for c⌈ln n⌉ times do
    generate set U' by sampling each vertex in U_0 with probability ρ/k̂
    compute R = R^1_G(U')
    foreach v ∈ R do count(v) ← count(v) + 1
H ← {u ∈ U_0 | count(u) ≥ (c/2)⌈ln n⌉}
L ← U_0 − H
return ⟨H, L⟩
```

Algorithm 4: Algorithm to find a random 1-hop independent set

input : A graph $G = (V, E^+ \cup E^-, w)$
input : Subset U_0 of negative vertices and integer ρ with $1 \le \rho \le |U_0|$
output: A 1-hop independent set $I \subseteq U_0$

RandIS($G = (V, E^+ \cup E^-, w)$, U_0, ρ)

```
let I' be a uniformly random size-⌈ρ/4⌉ subset of U_0
solve the super-source problem to compute
  d(v) = dist^1_G(I', v) and also a corresponding starting
  vertex s(v) ∈ I' such that d(v) = dist^1_G(s(v), v)
foreach u ∈ I' do
    if d(u) < 0 and s(u) = u then terminate algorithm
      and report "cycle"
R ← {v | d(v) < 0}
I ← I' − R
return I
```

$\phi(u) - \phi(v) = thru^{\beta}_{\phi}(u, y, v) \ge 0$. Thus, $BW^{\beta}_{\phi}(u, v) \le j - 1$, where x_j is the lowest-rank sampled vertex. As long as $j \le \lceil n/\tau \rceil$, we have $j - 1 < n/\tau$ and hence $BW^{\beta}_{\phi}(u, v) < n/\tau$.

A failure event (the algorithm neither reports a cycle nor hits the betweenness guarantee) can thus only occur if $j > \lceil n/\tau \rceil$. The last step of the proof is to bound this probability. For j to be this large, each sample must be drawn from the $b = n - \lceil n/\tau \rceil$ other vertices. If $b < |T|$, then there is never a failure. Otherwise, the failure probability is given by $\left(\frac{b}{n}\right)\left(\frac{b-1}{n-1}\right)\left(\frac{b-2}{n-2}\right)\cdots\left(\frac{b-|T|+1}{n-|T|+1}\right) \le \left(\frac{b}{n}\right)^{|T|} = \left(1 - \frac{\lceil n/\tau \rceil}{n}\right)^{|T|} \le \left(1 - \frac{1}{\tau}\right)^{|T|} \le (1 - 1/\tau)^{c\tau \ln n} \le (1/n)^c$. □

Proof of Lemma 3.5. Since $w(e) \ge 0$ for all $e \in E^+$, Lemma 4.2 can be applied, and the algorithm always meets the promised running time. Moreover, by Lemma 4.1, the algorithm always either correctly reports a cycle or returns a valid price function. Finally, Lemma 4.3 states that algorithm is successful with high probability, in which case it reports a cycle or a price function with the desired β-betweenness guarantee. □

5 FINDING A NEGATIVE SANDWICH

This section expands on the problem of finding a negative sandwich or independent set, as introduced in Section 3.4. The bulk of this section is devoted to proving Lemma 3.9. Recall that the input comprises the graph $G = (V, E^+ \cup E^-, w)$, a subset U_0 of negative vertices with $\hat{k} = |U_0|$, and integer parameter ρ with $1 \le \rho \le \hat{k}$.

As outlined in Section 3.4, the first task of Lemma 3.9 is to partition the negative vertices in U_0 into a heavy and light set.

The partitioning algorithm is given by Algorithm 3. The algorithm is parameterized by a constant $c \ge 6$ that controls the probability of failure. The algorithm is straightforward. Sample each vertex in U_0 independently with probability $\rho/\hat{k}$ to get a random subset U'. For each vertex in the 1-hop reach of U_0, increment a counter. Repeat this process $c\lceil \ln n \rceil$ times. Finally, the set H is the set of vertices in U_0 with counts at least $(c/2)\lceil \ln n \rceil$.

To prove the algorithm works, recall the notation $C(U_0, v) = \left|\left\{u \in U_0 | dist^1(u, v) < 0\right\}\right|$. Vertex v is ***heavy*** if $C(U_0, v) \ge 2\hat{k}/\rho$ and ***light*** if $C(U_0, v) \le (1/8)\hat{k}/\rho$. (Some vertices are neither.)

Lemma 5.1. *Consider an execution of Algorithm 3 with input G, U_0, ρ. Then with probability at least $1 - 1/n^{c/3-1}$, the partition is such that all heavy vertices in U are in H and all light vertices are in L. Equivalently, with high probability: $\forall v \in H, C(U_0, v) > (1/8)\hat{k}/\rho$ and $\forall v \in L, C(U_0, v) < 2\hat{k}/\rho$.*

Proof. Consider heavy vertex $v \in U_0$. Let X_i indicate whether $count(v)$ increases in the ith iteration of the loop, and let $X = count(v) = \sum_{i=1}^{c\lceil \ln n \rceil} X_i$. In each loop iteration, $X_i = 0$ is the event that none of the vertices that can reach v are sampled. This gives $\Pr(X_i = 0) \le (1 - \rho/\hat{k})^{C(U_0,v)} \le (1 - \rho/\hat{k})^{2\hat{k}/\rho} \le 1/e^2$. Let $p = E[X_i]$. Then $p = \Pr(X_i = 1) \ge (1 - 1/e^2) > 6/7$. Because the X_i's are independent identically distributed indicators, a Chernoff-Hoeffding bound applies, giving $\Pr(X \le (1/2)c\lceil \ln n \rceil)$. Let $\epsilon = p - 1/2$ or $1/2 = p - \epsilon$. Then we have $\Pr(X \le (1/2)c\lceil \ln n \rceil) = \Pr(X \le (p - \epsilon)c\lceil \ln n \rceil) \le \left(\left(\frac{p}{1/2}\right)^{1/2}\left(\frac{1-p}{1/2}\right)^{1/2}\right)^{c\lceil \ln n \rceil} \le (1/e)^{(1/3)c \ln n} = 1/n^{c/3}$ when $p \ge 6/7$.

The case of a light vertex, which is similar, appears in the full version. Taking the union bound across all vertices in U_0, the probability that any heavy or light vertex is misclassified is at most $1/n^{c/3-1}$. This bound is only meaningful if c is strictly larger than 3. □

Now let us turn to finding an independent set in the event that the returned partition has $H = \emptyset$. The algorithm is given by Algorithm 4. The algorithm is simple: sample a uniformly random size-$\lceil \rho/4 \rceil$ subset I' of U_0, and then remove from I' any vertices than can be reached by negative-weight 1-hop paths from any other vertex in I'. It is easy to see that this set is now a 1-hop independent set.

The argument that I is likely to be large is roughly as follows. Suppose that U_0 has no heavy vertices. Consider a vertex $v \in I'$.

There are only $C(U_0, v) < 2\hat{k}/\rho$ other vertices that can "knock out" v from I', each of which is only included in I' with probability roughly $(\rho/4)/\hat{k} = \rho/(4\hat{k})$. Thus, v is not too likely to be knocked out by other sampled vertices. Applying Markov's inequality allows us to conclude that I has a reasonable chance of being large.

There is one issue: if there is a negative-weight 1-hop cycle from v to itself, then v is *never* included in I. Thus, the algorithm also checks whether any of the shortest paths computed by the black-box subroutine correspond to negative-weight cycles. In particular, recall that for the super-source version of the problem, Lemma 2.3 states that BFD (and indeed any relaxation-based SSSP algorithms) can be augmented to return some vertex $s(v) \in I'$ such that $dist^1_G(I', v) = dist^1_G(s(v), v)$. If $s(v) = v$ and the distance to v is negative, then a negative-weight cycle is reported. Once a cycle is reported, the entire algorithm terminates.

The following lemma states that there is a constant probability that either I is large or a negative-weight cycle is discovered. The proof, deferred to the full version, formalizes the preceding ideas.

Lemma 5.2. *Consider an execution of Algorithm 4 with input G, U_0, ρ. The algorithm correctly reports a negative-weight cycle (i.e., only if G has one) or returns a 1-hop independent set $I \subseteq U_0$.*

Suppose that there are no heavy vertices in U_0. Then the probability that the algorithm returns an independent set with $|I| < \rho/16$ is at most $5/6$. Conversely, with probability at least $1/6$: the algorithm correctly reports a cycle or returns an independent set with $|I| \geq \rho/16$.

With all the tools in place, we are ready to complete the algorithm for Lemma 3.9, which is described in Algorithm 5. This algorithm is parameterized by a constant $c' \geq 4$, which controls the failure probability. The process matches the outline in Section 3.4. First partition the negative vertices U_0 into subsets H and L, where H should contain the heavy vertices and L should contain the light vertices, using Algorithm 3. If H is nonempty, then choose any vertex y and identify the set of negative vertices $U = \left\{u \in U_0 | dist^1_G(u, y) < 0\right\}$. This can be accomplished by computing 1-hop STSP to y using BFD. As this is supposed to be a Las Vegas algorithm, the next step is to verify that U is large enough. If so, return y and U. If not (some vertex was misclassified), restart the algorithm. If instead H is empty, then the algorithm instead searches for a large independent set $I \subseteq U_0$ by calling Algorithm 4 a total of $c'\lceil \lg n \rceil$ times, stopping when either a cycle is reported or a large independent set is found. This step may also fail either because we are unlucky or because some heavy vertices were misclassified in L. Thus, after $c'\lceil \lg n \rceil$ failed attempts, the algorithm is restarted.

Proof of Lemma 3.9. First, we consider the return values. By Lemma 5.2, if Algorithm 4 reports a cycle, then that reporting is always correct. Also by Lemma 5.2, the set I is always a 1-hop independent set. Thus, if Algorithm 5 returns I, then I is a 1-hop independent set with $|I| \geq \rho/16$. Finally, by construction, $U = \left\{u \in U_0 | dist^1_G(u, y) < 0\right\}$, and the algorithm only returns U and y if $|U| \geq (1/8)\hat{k}/\rho$.

We next consider the running time. `HL-Partition` (Algorithm 3) is dominated by $\Theta(\log n)$ iterations of 1-hop SSSP, or $O(m \log^2 n)$ time by Lemma 2.3. There is the potential for a partition failure event: that some vertex is misclassified in L or H, but Lemma 5.1 indicates the failure probability is at most $1/n^{c/3-1}$.

Algorithm 5: Algorithm of Lemma 3.9: find a sandwich crust or independent set

input : A graph $G = (V, E^+ \cup E^-, w)$
input : Subset U_0 of negative vertices and integer ρ with $1 \leq \rho \leq |U_0|$
output: A 1-hop independent set $I \subseteq U_0$ or a vertex y and set $U \subseteq U_0$ such that $dist^1_G(u, y) < 0$ for all $u \in U$. A negative-weight cycle may be reported inside a call to `RandIS`, terminating the full algorithm.

```
let k̂ = |U_0|
⟨H, L⟩ ← HL-Partition(G, U_0, ρ)
if H ≠ ∅ then
    choose arbitrary y ∈ H
    run STSP with target y to compute
      U = {u ∈ U_0 | dist^1_G(u, y) < 0}
    if |U| < (1/8)k̂/ρ then restart Algorithm 5
    else return y and U
/* we now have H = ∅ and L = U_0                    */
for c'⌈lg n⌉ attempts do
    I ← RandIS(G, U_0, ρ)
    if |I| ≥ ρ/16 then return I
/* no large independent set found                   */
restart Algorithm 5
```

Suppose that there is no partition failure. If H is not empty, then the algorithm verifies $C(U_0, y)$ with one 1-hop STSP. If $H = \emptyset$, the algorithm instead proceeds to finding an independent set. Each call to `RandIS` (Algorithm 4) entails computing 1-hop SSSP and scanning through the vertices once, so $O(m \log n)$ time. There are $c'\lceil \lg n \rceil$ such calls, so the running time is again $O(m \log^2 n)$. By Lemma 5.2, each call to `RandIS` leads to a probability of $5/6$ that Algorithm 5 completes, either finding a large-enough independent set or reporting a cycle and terminating. Thus, conditioned on no partition failure, the probability that the algorithm *does not* complete by the end of the loop is at most $(5/6)^{c'\lceil \lg n \rceil} = 1/n^{c' \lg(6/5)} < 1/n^{c'/4}$.

To conclude, the Algorithm 5 completes in $O(m \log^2 n)$ time unless there is a partition failure or there is an unlucky outcome with independent sets, either of which may result in the algorithm restarting. Adding up the failure probabilities gives a failure probability of at most $1/n^{c'/4} + 1/n^{c/3-1}$. Choosing, for example, $c = 9$ and $c' = 8$ gives a failure probability of at most $2/n^2$. □

6 REWEIGHTING A NEGATIVE SANDWICH

This section proves Lemma 3.7. Recall that the lemma states that given input graph G and negative sandwich (x, U, y), (1) the specific reweighting ϕ is valid, and (2) the only vertices in $R^{\beta-1}_{\phi}(U)$ after reweighting are those v for which $thru^{\beta}(x, v, y) < 0$ before.

Proof of Lemma 3.7. Throughout the proof, we use *dist* for the distance in G, i.e., with weight function w, and $dist_{\phi}$ for the distance in G_{ϕ}, i.e., with weight function w_{ϕ}. The latter only occurs at one point in the proof of (2).

To prove (1), consider any nonnegative edge $(u,v) \in E^+$. We then have three cases.

Case 1: $\phi(u) = 0$. We always have $\phi(v) \le 0$. So $w_\phi(u,v) = w(u,v) + \phi(u) - \phi(v) = w(u,v) + 0 - \phi(v) \ge w(u,v) \ge 0$.

For the remaining two cases, observe first the following

$$\max(dist^\beta(x,v), -dist^\beta(v,y)) \ge \phi(v) \tag{1}$$

$$(\phi(u) \ne 0) \implies ((\phi(u) \ge dist^\beta(x,u)) \wedge (\phi(u) \ge -dist^\beta(u,y))) \tag{2}$$

Case 2: $\phi(u) \ne 0$ and $\max(dist^\beta(x,v), -dist^\beta(v,y)) = dist^\beta(x,v)$. By the triangle inequality (Lemma 2.1), $dist^\beta(x,v) \le dist^\beta(x,u) + w(u,v)$ or equivalently $dist^\beta(x,u) \ge dist^\beta(x,v) - w(u,v)$. Putting everything together

$$\begin{aligned} \phi(u) &\ge dist^\beta(x,u) && \text{Equation 2} \\ &\ge dist^\beta(x,v) - w(u,v) && \text{triangle inequality} \\ &\ge \phi(v) - w(u,v) && \text{Equation 1} \end{aligned}$$

$\therefore\ w(u,v) + \phi(u) - \phi(v) \ge 0$.

Case 3: $\phi(u) \ne 0$ and $\max(dist^\beta(x,v), -dist^\beta(v,y)) = -dist^\beta(v,y)$. By the triangle ineq. (Lemma 2.1), $dist^\beta(u,y) \le w(u,v) + dist^\beta(v,y)$ or equivalently $-dist^\beta(u,y) \ge -w(u,v) - dist^\beta(v,y)$. Thus,

$$\begin{aligned} \phi(u) &\ge -dist^\beta(u,y) && \text{Equation 2} \\ &\ge -w(u,v) - dist^\beta(v,y) && \text{triangle inquality} \\ &\ge -w(u,v) + \phi(v) && \text{Equation 1} \end{aligned}$$

$\therefore\ w(u,v) + \phi(u) - \phi(v) \ge 0$.

Finally, let us prove (2). Consider any $u \in U$ and v that is not β-between x and y. The goal is to argue that $dist_\phi^{\beta-1}(u,v) \ge 0$. We proceed by breaking the proof into two smaller claims, namely (i) $\phi(u) = 0$ and (ii) $-\phi(v) > -dist^{\beta-1}(u,v)$. Assuming these claims hold, Lemma 2.2 gives us that $dist_\phi^{\beta-1}(u,v) = dist^{\beta-1}(u,v) + \phi(u) - \phi(v) > dist^{\beta-1}(u,v) + 0 - dist^{\beta-1}(u,v) = 0$ as desired.

Claim (i) follows from definition of a negative sandwich and ϕ. That is, $dist^\beta(u,y) \le dist^1(u,y) < 0$. Therefore, we have that $\max(dist^\beta(x,u), -dist^\beta(u,y)) \ge -dist^\beta(u,y) > 0$, and $\phi(u) = 0$.

For claim (ii), start with the definition of β-betweenness. By assumption, v is not β-between x and y, so $dist^\beta(x,v) + dist^\beta(v,y) \ge 0$. Therefore, $\phi(v) = \min(0, dist^\beta(x,v)) \le dist^\beta(x,v)$. By the triangle inequality (Lemma 2.1), $\phi(v) \le dist^\beta(x,v) \le dist^1(x,u) + dist^{\beta-1}(u,v)$. Because of the negative sandwich $dist^1(x,u) < 0$, and hence $\phi(v) < dist^{\beta-1}(u,v)$, which completes the proof of (ii). □

7 HOP REDUCTION

This section proves Lemma 3.3, expanding on the hop-reduction technique of Section 3.1. We use the notation $G = (V, E^+ \cup N, w)$ to refer to the subgraph being considered, where X is the set of negative vertices in the subgraph. Algorithm 6 provides pseudocode. Recall that the crux of the algorithm is building a new graph $H = (V_H, E_H, w_H)$ so that h-hop paths in G correspond to $\le \lceil h/r \rceil$-hop paths in H. Hence the SSSP distances can be computed efficiently by instead computing distances in H.

Aside from the graph construction, the algorithm is straightforward. First, compute distances $\delta_j(v) = dist_G^j(V,v)$ in G for $0 \le j \le r$, which by Lemma 2.3 corresponds to one r-limited SSSP computation. Next, use these distances to construct the graph H, discussed more below. Finally, compute $\lceil \hat{k}/r \rceil$ and $(\lceil \hat{k}/r \rceil + 1)$-hop distances in H. If these are different, the algorithm terminates by reporting a cycle. If these are the same, then the price function for $v \in V$ is given by $\phi(v) = dist_H^{\lceil \hat{k}/r \rceil}(V,v)$.

Algorithm 6: Algorithm of Lemma 3.3: eliminate a remote subset by hop reduction

input : Integer $r \ge 1$
input : A (sub)graph $G = (V, E^+ \cup N, w)$. Let X be the set of negative vertices.
output: A valid price function ϕ that eliminates X. The algorithm may instead terminate by reporting a negative-weight cycle.

let $\hat{k} = |X|$
compute super-source distances $\delta_j(v) = dist_G^j(V,v)$ for all vertices v and all j, $0 \le j \le r$
$R \leftarrow \{v | \delta_r(v) < 0\}$
construct a new graph $H = (V_H, E_H, w_H)$ (see text)
let $\kappa = \lceil \hat{k}/r \rceil$
compute super-source distances $d(v) = dist_H^\kappa(V,v)$ and $d'(v) = dist_H^{\kappa+1}(V,v)$ for all $v \in V_H$
if $\exists v \in V_H$ *such that* $d'(v) < d(v)$ **then** terminate algorithm and report "cycle"
else return *price function* $\phi : V \to \mathbb{R}$ *with* $\phi(v) = d(v)$

Vertices V_H. For all of the following, let $R = \{v | \delta_r(v) < 0\}$. All of the vertices in V are also in V_H; define $v_0 = v$, so when referring to a vertex $v \in V$ in the context of the graph H, we may use either v_0 or v.[4] In addition, for each vertex $v \in R$, V_H contains r additional copies $v_1, v_2, \ldots, v_r$ of the vertex. The subscript ℓ in v_ℓ is called the ***layer*** of the vertex. Layer 0 is the original vertices.

Edges E_H. For the edges, there are several cases depending on whether the endpoints are in R or not, i.e., whether the endpoints occur in more than one layer. Let us consider the nonnegative edges $(u,v) \in E^+$ first. The number of corresponding edges in H is determined by whether $u \in R$, and the target of the edges depends on whether $v \in R$. If $u, v \in R$, then there are $r+1$ copies of each endpoint, and there are $r+1$ corresponding copies $(u_0, v_0), (u_1, v_1), \cdots, (u_r, v_r)$ of the edge included in E_H. These edges are each within a single layer. If $u \in R$ but $v \notin R$, then there are still $r+1$ copies of the edge, but they are all directed at v_0 in layer 0, i.e., the edges have the form (u_j, v_0) for $0 \le j \le r$. If instead $u \notin R$ then u only occurs in layer 0, and hence there is only a single copy of the edge (u_0, v_0) in E_H. Notice that for all edges $(u,v) \in E^+$, the corresponding edges in E_H have the form (u_j, v_j) or (u_j, v_0)—these edges are never directed toward a higher layer. Moreover, for each $(u,v) \in E^+$, each $u_i \in V_H$ has exactly one such outgoing edge.

Now consider the negative edges $(u,v) \in N$. Again, the number of edges is dictated by whether $u \in R$, and the target depends on whether $v \in R$. If $u, v \in R$, then there are r corresponding

[4] The notation v_0 is generally used when considering distances or weights of edges in H, and the notation v is generally used when relating the distances back to G.

copies $(u_0, v_1), (u_1, v_2), \ldots, (u_{r-1}, v_r)$ of the edge in E_H; here, each (u_j, v_{j+1}) progresses from layer j to layer $j+1$, which is the key difference in the construction for negative edges and nonnegative edges. If $u \in R$ but $v \notin R$, then there are still r copies of the edge, but they all directed at layer-0 vertex v_0, i.e., the edges have the form (u_j, v_0) for $0 \le j < r$. If instead $u \notin R$, then there is only one copy of the edge in E_H: if $v \in R$, then the edge is (u_0, v_1); if $v \notin R$, then the edge is (u_0, v_0). Unlike the nonnegative case, these edges may be directed toward a higher layer, but it is always at most one higher. Specifically, for $(u, v) \in N$, the corresponding edges all have the form (u_j, v_{j+1}) or (u_j, v_0). Moreover, for $(u, v) \in N$, each $u_i \in V_H$ with $i < r$ has exactly one outgoing edge of the form (u_i, v_j) (and moreover $j \in \{0, i+1\}$). The copy of u_r in the r-th layer has no corresponding outgoing edge as there is no layer $r+1$.

For $u \in R$, E_H also includes the self edges (u_j, u_{j+1}) for $0 \le j < r$ and (u_r, u_0). These edges form a cycle on copies of u, and the weights will be set so that this is a 0-weight cycle. These edges serve two purposes. First, the edges (u_r, u_0) provide routes from layer-r to layer-0. Second, the other edges in the cycle simplify the reasoning about distances in H.

Weights w_H. For each edge $(u_i, v_j) \in E_H$, the weight is simply $w_H = w(u, v) + \delta_i(u) - \delta_j(v)$, where for notational convenience we define $w(u, u) = 0$ for all $u \in V$.

Analysis Overview. Proof of Lemma 3.3, which is deferred to the full version, uses the tools outlined below. The goal is to show that κ-hop paths in H are enough to realize Johnson's strategy [16] on G. The running time follows from the fact that X is r-remote, and hence $|R| = n/r$, so $|V_H| = O(n)$ and $|E_H| = O(m/n) \cdot |V_H| = O(m)$.

The first observation is that most edges in H have nonnegative weight. In particular, the negative edges in H are limited to the self edges (u_r, u_0) from layer r to layer 0. The proof amounts to applying the triangle inequality (Lemma 2.1) to several cases.

Lemma 7.1. *Consider the graph $G = (V, E^+ \cup N, w)$ and auxiliary graph $H = (V_H, E_H, w_H)$ as constucted by Algorithm 6. The only edges $e \in E_H$ with $w_H(e) < 0$ are the edges $e \in \{(u_r, u_0)\}$*

The next lemmas show a correspondence between paths in H and paths in G. The first, which is simpler, shows that paths between vertices in V in the graph H can be simulated by in G, and moreover those paths have the same weight. The second roughly shows the converse, but it also bounds the number of hops. Together, these imply that the distances computed in H can be used to compute distances in G. The proofs of these claims appear in the full version of the paper. All of these proofs proceed by induction on the length of the path, showing constructively how to simulate the path in the other graph, but there are several cases depending edge type.

Lemma 7.2. *Consider any $s_i, v_j \in V_H$. Let p_H be any s_i-to-v_j path in H. Then there is an s-to-v path p in G with $w(p) = w_H(p_H) - \delta_i(s) + \delta_j(v)$.*

Lemma 7.3. *Let p be any h-hop s-to-v path in G, for any $s, v \in V$. Then there is an h_H-hop s_0-to-v_j path p_H in H, for some layer $0 \le j \le r$, with the following two properties: (1) $w_H(p_H) = w(p) + \delta_0(s) - \delta_j(v)$, and (2) $rh_H + j \le h$.*

Corollary 7.4. *Let p be any h-hop s-to-v path in G, for any $s, v \in V$. Then for all layers i with $v_i \in V_H$, there is an $\lceil h/r \rceil$-hop s_o-to-v_i path p_H in H with weight $w_H(p_H) = w(p) + \delta_0(s) - \delta_i(v)$.*

ACKNOWLEDGMENTS

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

REFERENCES

[1] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. 2020. Circulation Control for Faster Minimum Cost Flow in Unit-Capacity Graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science*. 93–104. https://doi.org/10.1109/FOCS46700.2020.00018

[2] Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16, 1 (1958).

[3] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. 2022. Negative-Weight Single-Source Shortest Paths in Near-linear Time. In *63rd IEEE Annual Symposium on Foundations of Computer Science*. 600–611. https://doi.org/10.1109/FOCS54457.2022.00063

[4] Karl Bringmann, Alejandro Cassis, and Nick Fischer. 2023. Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster!. In *64th IEEE Annual Symposium on Foundations of Computer Science*. 515–538. https://doi.org/10.1109/FOCS57990.2023.00038

[5] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *63rd IEEE Annual Symposium on Foundations of Computer Science*. 612–623. https://doi.org/10.1109/FOCS54457.2022.00064

[6] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. 2017. Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in $O(m^{10/7} \log W)$ time. In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms*. 752–771. https://doi.org/10.1137/1.9781611974782.48

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.

[8] Yefim Dinitz and Rotem Itzhak. 2017. Hybrid Bellman-Ford-Dijkstra Algorithm. *J. of Discrete Algorithms* 42, C (jan 2017), 35–44.

[9] Ran Duan, Jiayi Mao, Xinkai Shu, and Longhui Yin. 2023. A Randomized Algorithm for Single-Source Shortest Path on Undirected Real-Weighted Graphs. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*. 484–492. https://doi.org/10.1109/FOCS57990.2023.00035

[10] Jeff Erickson, Ivor van der Hoog, and Tillmann Miltzow. 2020. Smoothing the gap between NP and ER. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, Sandy Irani (Ed.). 1022–1033. https://doi.org/10.1109/FOCS46700.2020.00099

[11] Lester R. Ford. 1956. Paper P-923. *Network Flow Theory* (1956).

[12] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (1987), 596–615. https://doi.org/10.1145/28869.28874

[13] Harold N. Gabow. 1983. Scaling Algorithms for Network Problems. In *24th Annual Symposium on Foundations of Computer Science*. 248–257. https://doi.org/10.1109/SFCS.1983.68

[14] Harold N. Gabow and Robert Endre Tarjan. 1989. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.* 18, 5 (1989), 1013–1036.

[15] Andrew V. Goldberg. 1995. Scaling Algorithms for the Shortest Path Problem. *SIAM J. Comput.* 24, 3 (1995), 494–504.

[16] Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24, 1 (jan 1977), 1–13.

[17] Adam Karczmarz, Wojciech Nadara, and Marek Sokołowski. 2024. Exact Shortest Paths with Rational Weights on the Word RAM. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms*. 2597–2608. https://doi.org/10.1137/1.9781611977912.92

[18] Edward F. Moore. 1959. The Shortest Path Through a Maze. In *Proceedings of the International Symposium on the Theory of Switching*. 285–292.

[19] Seth Pettie and Vijaya Ramachandran. 2005. A Shortest Path Algorithm for Real-Weighted Undirected Graphs. *SIAM J. Comput.* 34, 6 (2005), 1398–1431. https://doi.org/10.1137/S0097539702419650

[20] Alfonso Shimbel. 1955. Structure in Communication Nets. In *Proceedings of the Symposium on Information Networks*. 199–203.

[21] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2020. Bipartite Matching in Nearly-linear Time on Moderately Dense Graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science*. 919–930. https://doi.org/10.1109/FOCS46700.2020.00090

Received 07-NOV-2023; accepted 2024-02-11