Negative-Weight Single-Source Shortest Paths in Near-linear Time

Aaron Bernstein* Danupon Nanongkai† Christian Wulff-Nilsen‡

Abstract

We present a randomized algorithm that computes single-source shortest paths (SSSP) in $O(m \log^8(n) \log W)$ time when edge weights are integral and can be negative.[1] This essentially resolves the classic negative-weight SSSP problem. The previous bounds are $\tilde{O}((m+n^{1.5}) \log W)$ [BLNPSSSW FOCS'20] and $m^{4/3+o(1)} \log W$ [AMV FOCS'20]. Near-linear time algorithms were known previously only for the special case of planar directed graphs [Fakcharoenphol and Rao FOCS'01].

In contrast to all recent developments that rely on sophisticated continuous optimization methods and dynamic algorithms, our algorithm is simple: it requires only a simple graph decomposition and elementary combinatorial tools. In fact, ours is the first combinatorial algorithm for negative-weight SSSP to break through the classic $\tilde{O}(m\sqrt{n} \log W)$ bound from over three decades ago [Gabow and Tarjan SICOMP'89].

*Rutgers University

†Max Planck Institute for Informatics and KTH. Work done while at BARC, University of Copenhagen.

‡Work done while at BARC, University of Copenhagen.

[1]Throughout, n and m denote the number of vertices and edges, respectively, and $W \geq 2$ is such that every edge weight is at least $-W$. $\tilde{O}$ hides polylogarithmic factors.

Contents

1 Introduction

We consider the single-source shortest paths (SSSP) problem with (possibly negative) integer weights. Given an m-edge n-vertex directed weighted graph $G = (V, E, w)$ with integral edge weight $w(e)$ for every edge $e \in E$ and a source vertex $s \in V$, we want to compute the distance from s to v, denoted by $\text{dist}_G(s, v)$, for every vertex in v.

Two textbook algorithms for SSSP are Bellman-Ford and Dijkstra's algorithm. Dijkstra's algorithm is near-linear time ($O(m + n \log n)$ time), but restricted to *nonnegative* edge weights.[2] With negative weights, we can use the Bellman-Ford algorithm, which only requires that there is no *negative-weight cycle* reachable from s in G; in particular, the algorithm either returns $\text{dist}_G(s, v) \neq -\infty$ for every vertex v or reports that there is a cycle reachable from s whose total weight is negative. Unfortunately, the runtime of Bellman-Ford is $O(mn)$.

Designing faster algorithms for SSSP with negative edge weights (denoted negative-weight SSSP) is one of the most fundamental and long-standing problems in graph algorithms, and has witnessed waves of exciting improvements every few decades since the 50s. Early works in the 50s, due to Shimbel [Shi55], Ford [For56], Bellman [Bel58], and Moore [Moo59] resulted in the $O(mn)$ runtime. In the 80s and 90s, the scaling technique led to a wave of improvements (Gabow [Gab85], Gabow and Tarjan [GT89], and Goldberg [Gol95]), resulting in runtime $O(m\sqrt{n} \log W)$, where $W \geq 2$ is the minimum integer such that $w(e) \geq -W$ for all $e \in E$.[3] In the last few years, advances in continuous optimization and dynamic algorithms have led to a new wave of improvements, which achieve faster algorithms for the more general problems of transshipment and min-cost flow, and thus imply the same bounds for negative-weight SSSP (Cohen, Madry, Sankowski, Vladu [CMSV17]; Axiotis, Madry, Vladu [AMV20]; BLNPSSSW [BLN+20, BLL+21, BLSS20]). This line of work resulted in an near-linear runtime ($\tilde{O}((m + n^{1.5}) \log W)$ time) on moderately dense graphs [BLN+20] and $m^{4/3+o(1)} \log W$ runtime on sparse graphs [AMV20].[4] For the special case of planar directed graphs [LRT79, HKRS97, FR06, KMW10, MW10], near-linear time complexities were known since the 2001 breakthrough of Fakcharoenphol and Rao [FR06] where the best current bound is $O(n \log^2(n)/ \log \log n)$ [MW10]. No near-linear time algorithm is known even for a somewhat larger class of graphs such as bounded-genus and minor-free graphs (which still requires $\tilde{O}(n^{4/3} \log W)$ time [Wul11]). This state of the art motivates two natural questions:

1. *Can we get near-linear runtime for all graphs?*
2. *Can we achieve efficient algorithms without complex machinery?*

For the second question, note that currently all state-of-the-art results for negative-weight SSSP are based on min-cost flow algorithms, and hence rely on sophisticated continuous optimization methods and a number of complex dynamic algebraic and graph algorithms (e.g. [SW19, NSW17, CGL+20, BBP+20, NS17, Wul17]). It would be useful to develop simple efficient algorithms that are specifically tailored to negative-weight SSSP, and thus circumvent the complexity currently inherent in flow algorithms; the best known bound of this kind is still the classic $O(m\sqrt{n} \log(W))$ from over three decades ago [GT89, Gol95]. A related question is whether it is possible to achieve efficient

[2] In the word RAM model, Thorup improved the runtime to $O(m + n \log \log(C))$ when C is the maximal edge weight [Tho04] and to linear time for *undirected* graphs [Tho99].

[3] The case when n is big and W is small can be improved by the $O(n^{\omega} W)$-time algorithms of Sankowski [San05], and Yuster and Zwick [YZ05].

[4] $\tilde{O}$-notation hides polylogarithmic factors. The dependencies on W stated in [AMV20, BLN+20] are slightly higher than what we state here. These dependencies can be reduced by standard techniques (weight scaling, adding dummy source, and eliminating high-weight edges).

algorithms for the problem using combinatorial tools, or whether there are fundamental barriers that make continuous optimization necessary.

1.1 Our Result

In this paper we resolve both of the above questions for negative-weight SSSP: we present a simple combinatorial algorithm that reduces the running time all the way down to near-linear.

Theorem 1.1. *There exists a randomized (Las Vegas) algorithm that takes $O(m \log^8(n) \log(W))$ time with high probability (and in expectation) for an m-edge input graph G_{in} and source s_{in}. It either returns a shortest path tree from s_{in} or returns a negative-weight cycle.*

Our algorithm relies only on basic combinatorial tools; the presentation is self-contained and only uses standard black-boxes such as Dijkstra's and Bellman-Ford algorithms. In particular, it is a scaling algorithm enhanced by a simple graph decomposition algorithm called *Low Diameter Decomposition* which has been studied since the 80s; our decomposition is obtained in a manner similar to some known algorithms (see Section 1.2 for a more detailed discussion). Our main technical contribution is showing how low-diameter decomposition—which works only on graphs with non-negative weights—can be used to develop a recursive scaling algorithm for SSSP with negative weights. As far as we know, all previous applications of this decomposition were used for parallel/distributed/dynamic settings for problems that do not involve negative weights, and our algorithm is also the first to take advantage of it in the classical sequential setting; we also show that in this setting, there is a simple and efficient algorithm to compute it.

Perspective on Other Problems: While our result is specific to negative-weight SSSP, we note that question *(2)* above in fact applies to a much wider range of problems. The current landscape of graph algorithms is that for many of the most fundamental problems, including ones taught in undergraduate courses and used regularly in practice, the state-of-the-art solution is a complex algorithm for the more general min-cost flow problem: some examples include negative-weight SSSP, bipartite matching, the assignment problem, edge/vertex-disjoint paths, s-t cut, densest subgraph, max flow, transshipment, and vertex connectivity. This suggests a research agenda of designing simple algorithms for these fundamental problems, and perhaps eventually their generalizations such as min-cost flow. We view our result on negative-weight SSSP as a first step in this direction.

Independent and Follow-Up Results. Independently from our result, the recent major breakthrough by Chen, Kyng, Liu, Peng, Probst Gutenberg, and Sachdeva [CKL+22] culminates the line of works based on continuous optimization and dynamic algorithms (e.g. [DS08, Mad13, LS14, Mad16, CMSV17, CLS19, Bra20, LS20, AMV20, BLSS20, BLN+20, BLL+21]) and achieves an almost-linear time bound[5] for min-cost flow. The authors thus almost match our bounds for negative-weight SSSP as a special case of their result: their runtime is $m^{1+o(1)} \log(W)$ versus our $O(m \cdot \text{polylog}(n) \log(W))$ bound. The two results are entirely different, and as far as we know there is no overlap in techniques.

The above landmark result essentially resolves the running-time complexity for a wide range of fundamental graph problems, modulo the extra $m^{o(1)}$ factor. We believe that this makes it a natural time to pursue question *(2)* for these problems, outlined above.

In this paper, we prioritize simplicity and modularity, and not optimizing the logarithmic factors. The follow-up work by Bringmann, Cassis, and Fischer [BCF23] greatly optimizes our framework to reduce the running time to $O(m \log^2(n) \log(nW) \log\log n)$. There is also follow-up work by

[5] $\tilde{O}(m^{1+o(1)} \log^2 U)$ time when vertex demands, edge costs, and upper/lower edge capacities are all integral and bounded by U in absolute value.

Ashvinkumar *et al.* showing how the framework can be applied to the parallel and distributed models of computation [ABC+23].

1.2 Techniques

Our main contribution is a new recursive scaling algorithm called ScaleDown: see Section 4, including an overview in Section 4.1. In this subsection, we highlight other techniques that may be of independent interest.

Low-Diameter Decomposition. One of our key subroutines is an algorithm that decomposes any directed graph with *non-negative* edge weights into strongly-connected components (SCCs) of small diameter. In particular, the algorithm computes a small set of edges E^{rem} such that all SCCs in the graph $G \setminus E^{rem}$ have small weak diameter. Although the lemma below only applies to non-negative weights, we will show that it is in fact extremely useful for our problem.

Lemma 1.2. *There is an algorithm* LowDiamDecomposition(G, D) *with the following guarantees:*

- *INPUT: an m-edge, n-vertex graph $G = (V, E, w)$ with non-negative integer edge weight function w and a positive integer D.*
- *OUTPUT: A set of edges E^{rem} with the following guarantees:*
  - *each SCC of $G \setminus E^{rem}$ has weak diameter at most D; that is, if u, v are in the same SCC, then $\text{dist}_G(u, v) \leq D$ and $\text{dist}_G(v, u) \leq D$.*
  - *For every $e \in E$, $\Pr[e \in E^{rem}] = O\left(\frac{w(e) \cdot \log^2 n}{D} + n^{-10}\right)$. These probabilities are not guaranteed to be independent.*[6]
- *RUNNING TIME: The algorithm has running time $O(m \log^2 n + n \log^3 n)$.*

The decomposition above is similar to other low-diameter decompositions used in both undirected and directed graphs, though the precise guarantees vary a lot between papers [Awe85, AGLP89, AP92, ABCP92, LS93, Bar96, BGK+14, MPX13, PRS+18, FG19, CZ20, BPW20, FGdV21]. The closest similarity is to the algorithm Partition of Bernstein, Probst-Gutenberg, and Wulff-Nilsen [BPW20]. The main difference is that the algorithm of [BPW20] needed to work in a dynamic setting, and as a result their algorithm is too slow for our purposes. Our decomposition algorithm follows the general framework of [BPW20], but with several key differences to ensure faster running time; our algorithm is also simpler, since it only applies to the static setting. For the reader's convenience, we present the entire algorithm from scratch in Section 6.

No Negative-Weight Cycle Assumption via a Black-Box Reduction. Although it is possible to prove Theorem 1.1 directly, the need to return the negative-weight cycle somewhat complicates the details. For this reason, we focus most of our paper on designing an algorithm that returns correct distances when the input graph contains no negative-weight cycle and guarantees nothing otherwise. See the description of subroutine SPmain in Theorem 3.4 as an example. We can focus on the above case because we have a black-box reduction from Theorem 1.1 to the above case that incurs an extra $O(\log^2(n))$ factor in the runtime. This reduction is essentially the same as the reduction from the minimum cost-to-profit ratio cycle problem to the negative cycle detection problem [Law66, Law76]. We include our reduction in Section C for completeness. Note that the more general cost-to-profit ratio cycle problem can be solved in near-linear time via a similar reduction [Law66, Law76].

[6] The 10 in the exponent suffices for our application but can be replaced by an arbitrarily large constant.

2 Preliminaries

Throughout, we only consider graphs with integer weights. For any weighted graph $G = (V, E, w)$, define $V(G) = V, E(G) = E$, and

$$E^{neg}(G) := \{e \in E \mid w(e) < 0\}.$$

Define $W_G := \max\{2, -\min_{e\in E}\{w(e)\}\}$; that is, W_G is the most negative edge weight in the graph[7]. Given any set of edges $S \subseteq E$ we define $w(S) = \sum_{e\in S} w(e)$. We say that a cycle C in G is a negative-weight cycle if $w(C) < 0$. We define $\text{dist}_G(u, v)$ to be the shortest distance from u to v; if there is a negative-weight cycle on some uv-path then we define $\text{dist}_G(u, v) = -\infty$.

Consider graph $G = (V, E, w)$ and consider subsets $V' \subseteq V$ and $E' \subseteq E$. We define $G[V']$ to be the subgraph of G induced by V'. We slightly abuse notation and denote a subgraph of G as $H = (V', E', w)$ where the weight function w is restricted to edges in H. We define $G\setminus V' = G[V\setminus V']$ and $G \setminus E' = (V, E \setminus E', w)$; i.e. they are graphs where we remove vertices and edges in V' and E' respectively. We sometimes write $G \setminus v$ and $E \setminus e$ instead of $G \setminus \{v\}$ and $G \setminus \{e\}$, respectively, for any $v \in V$ and $e \in E$. We say that a subgraph H of G has *weak* diameter D if for any $u, v \in V(H)$ we have that $\text{dist}_G(u, v) \leq D$. We always let G_{in} and s_{in} refer to the main input graph/source of Theorem 1.1.

Assumption 2.1 (Properties of input graph G_{in}; justified by Lemma 2.2)**.** We assume throughout the paper that the main input graph $G_{in} = (V, E, w_{in})$ satisfies the following properties:

1. $w_{in}(e) \geq -1$ for all $e \in E$ (thus, $W_{G_{in}} = 2$).
2. Every vertex in G_{in} has constant out-degree.

Lemma 2.2. *Say that there is an algorithm as in Theorem 1.1 for the special case when the graph G_{in} satisfies the properties of Assumption 2.1, with running time $T(m, n)$. Then there is algorithm as in Theorem 1.1 for any input graph G_{in} with integral weights that has running time $O(T(m, m)\log(W_{G_{in}}))$.*

Proof. Let us first consider the first assumption, i.e that $w_{in}(e) \geq -1$. The scaling framework of Goldberg [Gol95] shows that an algorithm for this case implies an algorithm for any integer-weighted G at the expense of an extra $\log(W_G)$ factor.[8]

For the assumption that every vertex in G_{in} has constant out-degree, we use a by-now standard technique of creating $\Theta(\text{out-degree}(v))$ copies of each vertex v, so that each copy has constant out-degree; the resulting graph was $O(E)$ vertices and $O(E)$ edges.[9] □

Dummy Source and Negative Edges. The definitions below capture a common transformation we apply to negative weights and also allow us to formalize the number of negative edges on a shortest path. Note that most of our algorithms/definitions will not refer to the input source s_{in}, but instead to a dummy source that has edges of weight 0 to every vertex.

[7] We set $W_G \geq 2$ so that we can write $\log(W_G)$ in our runtime

[8] Quoting [Gol95]: "Note that the basic problem solved at each iteration of the bit scaling method is a special version of the shortest paths problem where the arc lengths are integers greater or equal to -1."

[9] One way to do this is to replace every vertex with a directed zero-weight cycle whose size is the in-degree plus out-degree of the vertex and then attach the adjacent edges to this cycle.

Definition 2.3 (G_s,w_s,G^B,w^B,G^B_s,w^B_s)**.** Given any graph $G = (V, E, w)$, we let $G_s = (V \cup \{s\}, E \cup \{(s, v)\}_{v \in V}, w_s)$ refer to the graph G with a dummy source s added, where there is an edge of weight 0 from s to v for every $v \in V$ and no edges into s. Note that G_s has a negative-weight cycle if and only if G does and that $\text{dist}_{G_s}(s, v) = \min_{u \in V} \text{dist}_G(u, v)$.

For any integer B, let $G^B = (V, E, w^B)$ denote the graph obtained by adding B to all negative edge weights in G, i.e. $w^B(e) = w(e) + B$ for all $e \in E^{neg}(G)$ and $w^B(e) = w(e)$ for $e \in E \setminus E^{neg}(G)$. Note that $(G^B)_s = (G_s)^B$ so we can simply write $G^B_s = (V \cup \{s\}, E \cup \{(s, v)\}_{v \in V}, w^B_s)$.

Definition 2.4 ($\eta_G(v)$, $P_G(v)$)**.** For any graph $G = (V, E, w)$ and $s \in V$ such that s can reach all vertices in V, define

$$\eta_G(v; s) := \begin{cases} \infty & \text{if } \text{dist}_G(s, v) = -\infty \\ \min\{|E^{neg}(G) \bigcap P| : P \text{ is a shortest } sv\text{-path in } G\}; & \text{otherwise.} \end{cases}$$

Let $\eta(G; s) = \max_{v \in V} \eta_G(v; s)$. When $\text{dist}_G(s, v) \neq -\infty$, let $P_G(v; s)$ be a shortest sv-path on G such that

$$|E^{neg}(G) \bigcap P_G(v)| = \eta_G(v). \tag{1}$$

We omit s when s is the dummy source in G_s defined in Definition 2.3; i.e. $\eta_G(v) = \eta_{G_s}(v; s)$, $\eta(G) = \eta(G_s; s)$, and $P_G(v) = P_{G_s}(v; s)$. Further, the graph G is omitted if the context is clear.

2.1 Price Functions and Equivalence

Our algorithm heavily relies on price functions, originally introduced by Johnson [Joh77]

Definition 2.5 (Price Function)**.** Consider a graph $G = (V, E, w)$ and let ϕ be any function: $V \to \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. Then, we define w_ϕ to be the weight function $w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v)$ and we define $G_\phi = (V, E, w_\phi)$. We will refer to ϕ as a *price* function on V. Note that $(G_\phi)_\psi = G_{\phi+\psi}$.

Definition 2.6 (Graph Equivalence)**.** We say that two graphs $G = (V, E, w)$ and $G' = (V, E, w')$ are *equivalent* if **(1)** any shortest path in G is also a shortest path in G' and vice-versa and **(2)** G contains a negative-weight cycle if and only if G' does.

Lemma 2.7 ([Joh77])**.** *Consider any graph $G = (V, E, w)$ and price function ϕ. For any pair $u, v \in V$ we have $\text{dist}_{G_\phi}(u, v) = \text{dist}_G(u, v) + \phi(u) - \phi(v)$, and for any cycle C we have $w(C) = w_\phi(C)$. As a result, G and G_ϕ are equivalent. Finally, if $G' = (V, E, w)$, $G' = (V, E, w')$ and $w' = c \cdot w(e)$ for some positive c, then G and G' are equivalent.*

The overall goal of our algorithm will be to compute a price function ϕ such that all edge weights in G_ϕ are non-negative (assuming no negative-weight cycle); we can then run Dijkstra on G_ϕ. The lemma below, originally used by Johnson, will be one of the tools we use.

Lemma 2.8 ([Joh77])**.** *Let $G = (V, E)$ be a directed graph with no negative-weight cycle and let s be any vertex in G that can reach all vertices in V. Let $\phi(v) = \text{dist}_G(s, v)$ for all $v \in V$. Then, all edge weights in G_ϕ are non-negative. (The lemma follows trivially from the fact that $\text{dist}(s, v) \leq \text{dist}(s, u) + w(u, v)$.)*

3 The Framework

In this section we describe the input/output guarantees of all the subroutines used in our algorithm, as well as some of the algorithms themselves.

3.1 Basic Subroutines

Lemma 3.1 (Dijkstra)**.** *There exists an algorithm* Dijkstra(G, s) *that takes as input a graph G with non-negative edge weights and a vertex $s \in V$ and outputs a shortest path tree from s in G. The running time is $O(m + n\log(n))$.*

It is easy to see that if G is a DAG (Directed Acyclic Graph), computing a price function ϕ such that G_ϕ has non-negative edge weights is straightforward: simply loop over the topological order $v_1, ..., v_n$ and set $\phi(v_i)$ so that all incoming edges have non-negative weight. The lemma below generalizes this approach to graphs where only the "DAG part" has negative edges.

Lemma 3.2 (FixDAGEdges)**.** *There exists an algorithm* FixDAGEdges$(G, \mathcal{P})$ *that takes as input a graph G and a partition $\mathcal{P} := \{V_1, V_2, \ldots\}$ of vertices of G such that*

1. *for every i, the induced subgraph $G[V_i]$ contains no negative-weight edges, and*
2. *when we contract every V_i into a node, the resulting graph is a DAG (i.e. contains no cycle).*

The algorithm outputs a price function $\phi : V \to \mathbb{Z}$ such that $w_\phi(u, v) \geq 0$ for all $(u, v) \in E$. The running time is $O(m + n)$.

Proof sketch. The algorithm is extremely simple: it loops over the SCCs V_i in topological order, and when it reaches V_i it sets the same price $\phi(v)$ for every $v \in V_i$ that ensures there are no non-negative edges entering V_i; since all $\phi(v)$ are the same, this does not affect edge-weights inside V_i. We leave the pseudocode and analysis for Section B in the appendix. □

The next subroutine shows that computing shortest paths in a graph G can be done efficiently as long as $\eta(v)$ is small on average (see Definition 2.4 for $\eta(v)$). Note that this subroutine is the reason we use the assumption that every vertex has constant out-degree (Assumption 2.1).

Lemma 3.3. (ElimNeg) *There exists an algorithm* ElimNeg(G, s) *that takes as input a graph $G = (V, E, w)$ in which all vertices $v \neq s$ have constant out-degree and a source $s \in V$ that can reach all vertices in V. The algorithm outputs a price function ϕ such that $w_\phi(e) \geq 0$ for all $e \in E$ and has running time $O(\log(n) \cdot (n + \sum_{v \in V} \eta_G(v; s)))$ (Definition 2.4); note that if G contains a negative-weight cycle then $\sum_{v \in V} \eta_G(v; s) = \infty$ so the algorithm will never terminate and hence not produce any output.*

Proof sketch. By Lemma 2.7, in order to compute the desire price function, it suffices to describe how ElimNeg(G) computes $\text{dist}_G(s, v)$ for all $v \in V$, where s is the input source to ElimNeg(G, s).

The algorithm is a straightforward combination of Dijkstra's and Bellman-Ford's algorithms. The algorithm maintains distance estimates $d(v)$ for each vertex v. It then proceeds in multiple iterations, where each iteration first runs a Dijkstra Phase that ensures that all non-negative edges are relaxed and then a Bellman-Ford Phase ensuring that all negative edges are relaxed. Consider a vertex v and let P be a shortest path from s to v in G with $\eta_G(v; s)$ edges of $E^{neg}(G)$. It is easy to see that $\eta_G(v; s) + 1$ iterations suffice to ensure that $d(v) = \text{dist}_G(s, v)$. In each of these iterations, v is extracted from and added to the priority queue of the Dijkstra Phase only $O(1)$ times. Furthermore, after the $\eta_G(v; s) + 1$ iterations, v will not be involved in any priority queue operations. Since the bottleneck in the running time is the queue updates, we get the desired time bound of $O(\log(n) \cdot \sum_{v \in V} (\eta_G(v; s) + 1)) = O(\log(n) \cdot (n + \sum_{v \in V} \eta_G(v; s)))$.

This completes the proof sketch. The full proof can be found in Appendix A. □

3.2 The Interface of the Two Main Algorithms

Our two main algorithms are called ScaleDown and SPmain. The latter is a relatively simple outer shell. The main technical complexity lies in ScaleDown, which calls itself recursively.

Theorem 3.4 (SPmain)**.** *There exists an algorithm* SPmain(G_{in}, s_{in}) *that takes as input a graph* G_{in} *and a source* s_{in} *satisfying the properties of Assumption 2.1. If the algorithm terminates, it outputs a shortest path tree* T *from* s_{in}*. The running time guarantees are as follows:*

- *If the graph* G_{in} *contains a negative-weight cycle then the algorithm never terminates.*
- *If the graph* G_{in} *does not contain a negative-weight cycle then the algorithm has expected running time* $\mathcal{T}_{spmain} = O(m \log^5(n))$*.*

Theorem 3.5 (ScaleDown)**.** *There exists the following algorithm* ScaleDown$(G = (V, E, w), \Delta, B)$*.*

1. *INPUT REQUIREMENTS:*
   (a) *B is positive integer, w is integral, and $w(e) \geq -2B$ for all $e \in E$*
   (b) *If the graph G does not contain a negative-weight cycle then the input must satisfy $\eta(G^B) \leq \Delta$; that is, for every $v \in V$ there is a shortest sv-path in G_s^B with at most Δ negative edges (Definitions 2.3 and 2.4)*
   (c) *All vertices in G have constant out-degree*
2. *OUTPUT: If it terminates, the algorithm returns an integral price function ϕ such that $w_\phi(e) \geq -B$ for all $e \in E$*
3. *RUNNING TIME: If G does not contain a negative-weight cycle, then the algorithm has expected runtime $O\left(m \log^3(n) \log(\Delta)\right)$. Remark: If G contains a negative-weight cycle, there is no guarantee on the runtime, and the algorithm might not even terminate; but if the algorithm does terminate, it always produces a correct output.*

Remark: Termination and Negative-Weight Cycles. Note that for both algorithms above, if G_{in} contains a negative-weight cycle then the algorithm might simply run forever, i.e. not terminate and not produce any output. In fact the algorithm SPmain *never* terminates if G_{in} contains a negative-weight cycle. The algorithm ScaleDown may or may not terminate in this case: our guarantee is only that if it does terminate, it always produces a correct output.

In short, neither algorithm is required to produce an output in the case where G_{in} contains a negative-weight cycle, so we recommend the reader to focus on the case where G_{in} does not contain a negative-weight cycle.

Proof sketch of Theorem 1.1. Algorithm SPmain is almost what we need for the main result in Theorem 1.1, but the guarantees of SPmain are slightly weaker: it is Monte Carlo (rather than Las Vegas), and it cannot return a negative cycle. We show in Section C that a combination of simple probabilistic bootstrapping and binary search can be used to transform SPmain into the main result promised in Theorem 1.1.

4 Algorithm ScaleDown (Theorem 3.5)

We start by describing the algorithm ScaleDown, as this contains our main conceptual contributions; the much simpler algorithm SPmain is described in the following section. Full pseudocode of ScaleDown is given in Algorithm 1. The algorithm mostly works with graph $G^B = (V, E, w^B)$. For the analysis, the readers may want to familiarize themselves with, e.g., G_s^B, w_s^B, $P_{G^B}(v)$ and $\eta(G^B)$ from Definitions 2.3 and 2.4. In particular, throughout this section, source s *always* refers to a dummy source that has outgoing edges to every vertex in V and has no incoming edges.

Algorithm 1: Algorithm for ScaleDown$(G = (V, E, w), \Delta, B)$

// Input/Output: See Theorem 3.5

1 **if** $\Delta \leq 2$ **then**

2 Let $\phi_2 = 0$ and jump to Phase 3 (Line 10)

3 Let $d = \Delta/2$. Let $G^B_{\geq 0} := (V, E, w^B_{\geq 0})$ where $w^B_{\geq 0}(e) := \max\{0, w^B(e)\}$ for all $e \in E$

// Phase 0: Decompose V to SCCs $V_1, V_2, \ldots$ with weak diameter dB in G

4 $E^{rem} \leftarrow$ LowDiamDecomposition$(G^B_{\geq 0}, dB)$ (Lemma 1.2)

5 Compute Strongly Connected Components (SCCs) of $G^B \setminus E^{rem}$, denoted by $V_1, V_2, \ldots$

// Properties: (Lemma 4.3) For each $u, v \in V_i$, $\mathrm{dist}_G(u, v) \leq dB$.

// (Lemma 4.4) If $\eta(G^B) \leq \Delta$, then for every $v \in V_i$, $E[P_{G^B}(v) \cap E^{rem}] = O(\log^2 n)$

// Phase 1: Make edges inside the SCCs $G^B[V_i]$ non-negative

6 Let $H = \bigcup_i G[V_i]$, i.e. H only contains edges inside the SCCs.

// (Lemma 4.5) If G has no negative-weight cycle, then $\eta(H^B) \leq d = \Delta/2$.

7 $\phi_1 \leftarrow$ ScaleDown$(H, \Delta/2, B)$ // (Corollary 4.6) $w_{H^B_{\phi_1}}(e) \geq 0$ for all $e \in H$

// Phase 2: Make all edges in $G^B \setminus E^{rem}$ non-negative

8 $\psi \leftarrow$ FixDAGEdges$(G^B_{\phi_1} \setminus E^{rem}, \{V_1, V_2, \ldots\})$ (Lemma 3.2)

9 $\phi_2 \leftarrow \phi_1 + \psi$ // (Lemma 4.7) All edges in $(G^B \setminus E^{rem})_{\phi_2}$ are non-negative

// Phase 3: Make all edges in G^B non-negative

10 $\psi' \leftarrow$ ElimNeg$((G^B_s)_{\phi_2}, s)$ (Lemma 3.3) // (Theorem 4.2) expected time $O(m \log^3 m)$.
(To define $(G^B_s)_{\phi_2}$ here, we define $\phi_2(s) = 0$.)

11 $\phi_3 = \phi_2 + \psi'$ // (Theorem 4.1) All edges in $G^B_{\phi_3}$ are non-negative.

12 **return** ϕ_3 ; // Since $w^B_{\phi_3}(e) \geq 0$, we have $w_{\phi_3}(e) \geq -B$

Note that $m = \Theta(n)$ since the input condition requires constant out-degree for every vertex. So, we use m and n interchangeably in this section. We briefly describe the ScaleDown algorithm and sketch the main ideas of the analysis in Section 4.1, before showing the full analysis in Section 4.2.

4.1 Overview

The algorithm runs in phases, where in the last phase it calls ElimNeg$((G^B_s)_{\phi_2}, s)$ for some price function ϕ_2 (Line 10). Recall (Lemma 3.3) that if ElimNeg terminates, it returns price function ψ' such that all edge weights in $(G^B_s)_{\phi_2+\psi'}$ are non-negative. Consequently, all edge weights in $G^B_{\phi_3}$ are non-negative for $\phi_3 = \phi_2 + \psi'$ (since $G^B_{\phi_2}$ is a subgraph of $(G^B_s)_{\phi_2}$). We thus have $w_{\phi_3}(e) \geq -B$ for all $e \in E$ as desired (because $w^B(e) \leq w(e) + B$). This already proves the output correctness of ScaleDown (Item 2 of Theorem 3.5). (See Theorem 4.1 for the detailed proof.) Thus it remains to bound the runtime when G contains no negative-weight cycle (Item 3). *In the rest of this subsection we assume that G contains no negative-weight cycle.*

Bounding the runtime when $\Delta \leq 2$ is easy: The algorithm simply jumps to Phase 3 with $\phi_2 = 0$ (Line 1 in Algorithm 1). Since $\eta(G^B) \leq \Delta \leq 2$ (the input requirement; Item 1b), the runtime of ElimNeg$((G^B_s)_{\phi_2}, s)$ is $O((m + \sum_{v \in V} \eta_{G^B}(v)) \log m) = O(m\Delta \log m) = O(m \log m)$.

For $\Delta > 2$, we require some properties from Phases 0-2 in order to bound the runtime of Phase 3. In Phase 0, we partition vertices into strongly-connected components (SCCs)[10] $V_1, V_2, \ldots$ such that

[10]Recall that a SCC is a *maximal* set $C \subseteq V$ such that for every $u, v \in V$, there are paths from u to v and from v

each V_i has weak diameter $dB = B\Delta/2$ in G. We do this by calling $E^{rem} \leftarrow \text{LowDiamDecomposition}(G^B_{\geq 0}, dB)$, where $G^B_{\geq 0}$ is obtained by rounding all negative weights in G^B up to 0; we then let $V_1, V_2, \ldots$ be the SCCs of $G^B \setminus E^{rem}$. (We need $G^B_{\geq 0}$ since LowDiamDecomposition can not handle negative weights.[11]) See Lemma 4.3 for the formal statement and proof.

The algorithm now proceeds in three phases. In Phase 1 it computes a price function ϕ_1 that makes the edges inside each SCC V_i non-negative; in Phase 2 it computes ϕ_2 such that the edges between SCCs in $G^B \setminus E^{rem}$ are also non-negative; finally, in Phase 3 it makes non-negative the edges in E^{rem} by calling ElimNeg.

(Phase 1) Our goal in Phase 1 is to compute ϕ_1 such that $w^B_{\phi_1}(e) \geq 0$ for every edge e in $G^B[V_i]$ for all i. To do this, we recursively call ScaleDown$(H, \Delta/2, B)$, where H is a union of all the SCCs $G[V_i]$. The main reason that we can recursively call ScaleDown with parameter $\Delta/2$ is because we can argue that, when G does not contain a negative-weight cycle,

$$\eta(H^B) \leq d = \Delta/2.$$

As a rough sketch, the above bound holds because if any shortest path P from dummy source s in some $(G^B[V_i])_s$ contains more than d negative-weight edges, then it can be shown that $w(P) < -dB$; this is the step where we crucially rely on the difference between $w^B(P)$ and $w(P)$. Combining $w(P) < -dB$ with the fact that $G^B[V_i]$ has weak diameter at most dB implies that G contains a negative-weight cycle. See Lemma 4.5 for the detailed proof.

(Phase 2) Now that all edges in $G^B_{\phi_1}[V_i]$ are non-negative, we turn to the remaining edges in $G^B \setminus E^{rem}$. Since these remaining edges (i.e. those not in the SCCs) form a directed acyclic graph (DAG), we can simply call FixDAGEdges$(G^B_{\phi_1} \setminus E^{rem}, \{V_1, V_2, \ldots\})$ (Lemma 3.2) to get a price function ψ such that all edges in $(G^B_{\phi_1} \setminus E^{rem})_\psi = G^B_{\phi_2} \setminus E^{rem}$ are non-negative. (See Lemma 4.7.)

(Phase 3) By the time we reach this phase, the only negative edges remaining in $G^B_{\phi_2}$ are the ones in E^{rem}; that is, $E^{neg}(G^B_{\phi_2}) \subseteq E^{rem}$. We call ElimNeg$((G^B_s)_{\phi_2}, s)$ to eliminate these negative edges. We are now ready to show that the runtime of Phase 3, which is $O((m + \sum_{v \in V} \eta_{(G^B_s)_{\phi_2}}(v; s)) \log m)$ (Lemma 3.3), is $O(m \log^3 m)$ in expectation. We do so by proving that for any $v \in V$,

$$E\left[\eta_{(G^B_s)_{\phi_2}}(v; s)\right] = O(\log^2 m).$$

(See Equation (5) near the end of the next subsection.) A competitive reader might want to try to prove the above via a series of inequalities: $\eta_{(G^B_s)_{\phi_2}}(v) \leq |P_{G^B}(v) \cap E^{neg}(G^B_{\phi_2})| + 1 \leq |P_{G^B}(v) \cap E^{rem}| + 1$, and also, the guarantees of LowDiamDecomposition (Lemma 1.2) imply that after Phase 0, $E\left[|P_{G^B}(v) \cap E^{rem}|\right] = O(\log^2 m)$. (Proved in Lemma 4.4.)

Finally, observe that there are $O(\log \Delta)$ recursive calls, and the runtime of each call is dominated by the $O(m \log^3 m)$ time of Phase 3. So, the total expected runtime is $O(m \log^3(m) \log \Delta)$

Remark. Our sequence of phases 0-3 is reminiscent of the sequencing used by Bernstein, Probst-Gutenberg, and Saranurak in their result on dynamic reachability [BGS20], although the actual work within each phase is entirely different, and the decompositions have different guarantees. The authors of [BGS20] decompose the graph into a DAG of *expanders* plus some separator edges (analogous to our phase 0); they then handle reachability inside expanders (phase 1), followed by reachability using the DAG edges (phase 2), and finally incorporate the separator edges (phase 3).

to u. See, e.g., Chapter 22.5 in [CLRS09].

[11] One can also use $G_{\geq 0}$ instead of $G^B_{\geq 0}$. We choose $G^B_{\geq 0}$ since some proofs become slightly simpler.

4.2 Full Analysis

Theorem 3.5 follows from Theorems 4.1 and 4.2 below. We start with Theorem 4.1 which is quite trivial to prove.

Theorem 4.1. ScaleDown$(G = (V, E, w), \Delta, B)$ *either does not terminate or returns* $\phi = \phi_3$ *such that* $w_\phi(e) \geq -B$ *for all* $e \in E$.

Proof. Consider when we call ElimNeg$((G^B_s)_{\phi_2}, s)$(Lemma 3.3) in Phase 3 for some integral price function ϕ_2. Either this step does not terminate or returns an integral price function ψ' such that $(G^B_{\phi_2})_{\psi'} = G^B_{\phi_2+\psi'} = G^B_{\phi_3}$ contains no negative-weight edges; i.e. $w^B_{\phi_3}(e) \geq 0$ for all $e \in E$. Since $w^B(e) \leq w(e) + B$, we have $w_{\phi_3}(e) \geq w^B_{\phi_3}(e) - B \geq -B$ for all $e \in E$. □

Theorem 4.1 implies that the output condition of ScaleDown (item 2 in Theorem 3.5) is always satisfied, regardless of whether G contains a negative-weight cycle or not. It remains to show that if G does not contain a negative-weight cycle, then ScaleDown$(G = (V, E, w), \Delta, B)$ has expected runtime of $O(m \log^3(m) \log(\Delta))$. It suffices to show the following.

Theorem 4.2. *If* G *does not contain a negative-weight cycle, then the expected time complexity of Phase 3 is* $O(m \log^3 m)$.

This suffices because, first of all, it is easy to see that Phase 0 requires $O(m \log^3(m))$ time (by Lemma 1.2) and other phases (except the recursion on Line 7) requires $O(m + n)$ time. Moreover, observe that if G contains no negative-weight cycle, then the same holds for H in the recursion call ScaleDown$(H, \Delta/2, B)$ (Line 7 of Algorithm 1); thus, if G contains no negative-weight cycle, then all recursive calls also get an input with no negative-weight cycle. So, by Theorem 4.2 the time to execute a single call in the recursion tree is $O(m \log^3 m)$ in expectation. Since there are $O(\log \Delta)$ recursive calls, the total running time is $O(m \log^3(m) \log(\Delta))$ by linearity of expectation.

Proof of Theorem 4.2. The rest of this subsection is devoted to proving Theorem 4.2. From now on, we consider any graph G that does not contain a negative-weight cycle. (We often continue to state this assumption in lemma statements so that they are self-contained.)

Base case: $\Delta \leq 2$. This means that for every vertex v, $\eta_{G^B}(v) \leq \eta(G^B) \leq \Delta \leq 2$ (see the input requirement of ScaleDown in Item 1b of Theorem 3.5). So, the runtime of Phase 3 is

$$O\left(\left(m + \sum_{v \in V} \eta_{G^B}(v)\right) \log m\right) = O\left(m\Delta \log m\right) = O\left(m \log m\right).$$

We now consider when $\Delta > 2$ and show properties achieved in each phase. We will use these properties from earlier phases in analyzing the runtime of Phase 3.

Phase 0: Low-diameter Decomposition. It is straightforward that the SCCs $G[V_i]$ have weak diameter at most dB (this property will be used in Phase 1):

Lemma 4.3. *For every* i *and every* $u, v \in V_i$, $\text{dist}_G(u, v) \leq dB$.

Proof. For every $u, v \in V_i$, we have $\text{dist}_G(u, v) \leq \text{dist}_{G^B_{\geq 0}}(u, v) \leq dB$ where the first inequality is because $w(e) \leq w^B_{\geq 0}(e)$ for every edge $e \in E$ and the second inequality is by the output guarantee of LowDiamDecomposition (Lemma 1.2). □

Another crucial property from the decomposition is this: Recall from Definition 2.4 that $P_{G^B}(v)$ is the shortest sv-path in G^B_s with $\eta_{G^B}(v)$ negative-weight edges. We show below that in expectation $P_{G^B}(v)$ contains only $O(\log^2 n)$ edges from E^{rem}. This will be used in Phase 3.

Lemma 4.4. *If $\eta(G^B) \le \Delta$, then for every $v \in V$, $E\left[|P_{G^B}(v) \cap E^{rem}|\right] = O(\log^2 m)$.*

Proof. Consider any $v \in V$. The crux of the proof is the following bound on the weight of $P_{G^B}(v)$ in $G^B_{\ge 0}$:

$$w^B_{\ge 0}(P_{G^B}(v)) \le \eta_{G^B}(v) \cdot B \tag{2}$$

where we define $w^B_{\ge 0}(s,u) = 0$ for every $u \in V$. Recall the definition of of w^B_s from Definition 2.3 and note that $w^B_s(P_{G^B}(v)) \le 0$ because there is an edge of weight 0 from s to every $v \in V$. We thus have Equation (2) because

$$\begin{aligned} w^B_{\ge 0}(P_{G^B}(v)) &\le w^B_s(P_{G^B}(v)) + |P_{G^B}(v) \cap E^{neg}(G^B)| \cdot B && \text{since } w^B(e) \ge -B \text{ for all } e \in E \\ &\le |P_{G^B}(v) \cap E^{neg}(G^B)| \cdot B && \text{since } w^B_s(P_{G^B}(v)) \le 0 \\ &= \eta_{G^B}(v) \cdot B && \text{by definition of } P_{G^B}(v) \end{aligned}$$

Recall from the output guarantee of LowDiamDecomposition (Lemma 1.2) that $\Pr[e \in E^{rem}] = O(w^B_{\ge 0}(e) \cdot (\log n)^2/D + n^{-10})$, where in our case $D = dB = B\Delta/2$. This, the linearity of expectation, and (2) imply that

$$\begin{aligned} E[P_{G^B}(v) \cap E^{rem}] &= O\left(\frac{w^B_{\ge 0}(P_{G^B}(v)) \cdot (\log n)^2}{B\Delta/2} + |(P_{G^B}(v))| \cdot n^{-10}\right) \\ &\overset{(2)}{=} O\left(\frac{2\eta_{G^B}(v) \cdot (\log n)^2}{\Delta} + n^{-9}\right) \end{aligned}$$

which is $O(\log^2 n)$ when $\eta(G^B) \le \Delta$. ∎

Phase 1: Make edges inside the SCCs $G^B[V_i]$ non-negative. We argue that ScaleDown$(H, \Delta/2, B)$ is called with an input that satisfies its input requirements (Theorem 3.5). The most important requirement is $\eta(H^B) \le \Delta/2$ (Item 1b) which we prove below (other requirements are trivially satisfied). Recall that we set $d := \Delta/2$ in Line 3.

Lemma 4.5. *If G has no negative-weight cycle, then $\eta(H^B) \le d = \Delta/2$.*

Proof. Consider any vertex $v \in V$. Let $P := P_{H^B}(v) \setminus s$; i.e. P is obtained by removing s from a shortest sv-path in H^B_s that contains $\eta_{H^B}(v)$ negative weights in H^B_s. Let u be the first vertex in P. Note three easy facts:

(a) $w_{H^B}(e) = w_H(e) + B$ for all $e \in E^{neg}(H^B)$,
(b) $|E^{neg}(H^B) \cap P| = |E^{neg}(H^B) \cap P_{H^B}(v)| = \eta_{H^B}(v)$, and
(c) $w_{H^B}(P) = w_{H^B_s}(P_{H^B}(v)) \le w_{H^B_s}(s,v) = 0$,

where (b) and (c) are because the edges from s to u and v in H^B_s have weight zero. Then,[12]

$$\begin{aligned} \mathrm{dist}_G(u,v) \le w_H(P) &\overset{(a)}{\le} w_{H^B}(P) - |E^{neg}(H^B) \cap P| \cdot B \\ &\overset{(b)}{=} w_{H^B}(P) - \eta_{H^B}(v) \cdot B \overset{(c)}{\le} -\eta_{H^B}(v) \cdot B. \end{aligned} \tag{3}$$

[12]The "$\overset{(a)}{\le}$" part in (3) is not equality because there can be an edge in $E^{neg}(H) \cap P$ that is not in $E^{neg}(H^B) \cap P$.

Note that u and v are in the same SCC V_i;[13] thus, by Lemma 4.3:

$$\text{dist}_G(v, u) \leq dB. \tag{4}$$

If G contains no negative-weight cycle, then $\text{dist}_G(u, v) + \text{dist}_G(v, u) \geq 0$ and thus $\eta_{H^B}(v) \leq dB \cdot (1/B) = d$ by Equations (3) and (4). Since this holds for every $v \in V$, Lemma 4.5 follows. □

Consequently, ScaleDown (Theorem 3.5) is guaranteed to output ϕ_1 as follows.

Corollary 4.6. *If G has no negative-weight cycle, then all edges in $G^B_{\phi_1}[V_i]$ are non-negative for every i.*

Phase 2: Make all edges in $G^B \setminus E^{rem}$ non-negative. Now that all edges in $G^B_{\phi_1}[V_i]$ are non-negative, we turn to the remaining edges in $G^B \setminus E^{rem}$. Intuitively, since these remaining edges (i.e. those not in the SCCs) form a directed acyclic graph (DAG), calling FixDAGEdges($G^B_{\phi_1} \setminus E^{rem}, \{V_1, V_2, \ldots\}$) (Lemma 3.2) in Phase 2 produces the following result.

Lemma 4.7. *If G has no negative-weight cycle, all weights in $G^B_{\phi_2} \setminus E^{rem}$ are non-negative.*

Proof. Clearly, $G^B_{\phi_1} \setminus E^{rem}$ and $\{V_1, V_2, \ldots\}$ satisfy the input conditions of Lemma 3.2, i.e. (1) $(G^B \setminus E^{rem})_{\phi_1}[V_i]$ contains no negative-weight edges for every i (this is due to Corollary 4.6), and (2) when we contract every V_i into a node, the resulting graph is a DAG (since the V_i are precisely the (maximal) SCCs of $G^B_{\phi_1} \setminus E^{rem}$).[14] Thus, FixDAGEdges($(G^B \setminus E^{rem})_{\phi_1}, \{V_1, V_2, \ldots\}$) returns ψ such that $(G^B_{\phi_1} \setminus E^{rem})_\psi = G^B_{\phi_2} \setminus E^{rem}$ contains no negative-weight edges. □

Phase 3 runtime. Now we are ready to prove Theorem 4.2, i.e. the runtime bound of ElimNeg($(G^B_s)_{\phi_2}, s$) in Phase 3 when G contains no negative-weight cycle. We start by clarifying the nature of the graph $(G^B_s)_{\phi_2}$. We start with the graph G^B; we then get G^B_s by adding a dummy source with outgoing edges of weight 0 to every $v \in V$; finally we get $(G^B_s)_{\phi_2}$ by applying price function ϕ_2 to G^B_s, where we define $\phi_2(s) = 0$. The reason we apply ϕ_2 at the end is to ensure that G^B_s and $(G^B_s)_{\phi_2}$ are equivalent. Note that after we apply ϕ_2, the edges incident to s can have non-zero weight (positive or negative); but s can still reach every vertex, so we can still call ElimNeg with s as the source.

Recall (Lemma 3.3 and definition 2.4) that the runtime of ElimNeg($(G^B_s)_{\phi_2}, s$) is

$$O\left(\left(m + \sum_{v \in V} \eta_{(G^B_s)_{\phi_2}}(v; s)\right) \log m\right)$$

Fix any $v \in V$, observe that

$$\begin{aligned}\eta_{(G^B_s)_{\phi_2}}(v; s) &= \min\{|P \cap E^{neg}((G^B_s)_{\phi_2})| : P \text{ is a shortest } sv\text{-path in } (G^B_s)_{\phi_2}\} \quad \text{(Definition 2.4)} \\ &\leq |P_{G^B}(v) \cap E^{neg}((G^B_s)_{\phi_2})|\end{aligned}$$

where the inequality is because, for any price function ϕ_2, $P_{G^B}(v)$ is a shortest sv-path in $(G^B_s)_{\phi_2}$ (because $(G^B_s)_{\phi_2}$ and G^B_s are equivalent and $P_{G^B}(v)$ is a shortest sv-path in G^B_s by definition). By Lemma 4.7, all negative-weight edges in $G^B_{\phi_2}$ are in E^{rem}, i.e. $E^{neg}(G^B_{\phi_2}) \subseteq E^{rem}$; so,

$$\eta_{(G^B_s)_{\phi_2}}(v; s) \leq |P_{G^B}(v) \cap E^{rem}| + 1$$

[13] in fact all vertices in P are in the same SCC V_i, because we define $H = \bigcup_i G[V_i]$.

[14] See, e.g., Lemma 22.13 in [CLRS09].

where the "+1" term is because the edge incident to s in $P_{G^B}(v)$ can also be negative. By Lemma 4.4 and the fact that $\eta(G^B) \leq \Delta$ (input requirement Item 1b in Theorem 3.5),[15]

$$E\left[\eta_{(G_s^B)_{\phi_2}}(v;s)\right] \leq E\left[|P_{G^B}(v) \cap E^{rem}|\right] + 1 = O(\log^2 m). \tag{5}$$

(The last equality is by Lemma 4.4.) Thus, the expected runtime of ElimNeg$((G_s^B)_{\phi_2}, s)$ is

$$O\left(\left(m + E\left[\sum_{v \in V} \eta_{(G_s^B)_{\phi_2}}(v;s)\right]\right)\log m\right) = O\left(m \log^3 m\right).$$

5 Algorithm SPmain (Theorem 3.4)

In this section we present algorithm SPmain(G_{in}, s_{in}) (Theorem 3.4), which always runs on the main input graph G_{in} and source s_{in}.

Description of Algorithm SPmain(G_{in}, s_{in})**.** See Algorithm 2 for pseudocode. Recall that if G_{in} contains a negative-weight cycle, then the algorithm is not supposed to terminate; for intuition, we recommend the reader focus on the case where G_{in} contains no negative-weight cycle.

The algorithm first creates an equivalent graph $\bar{G}$ by scaling up edge weights by $2n$ (Line 1), and also rounds B (Line 2), all to ensure that everything remains integral. It then repeatedly calls ScaleDown until we have a price function ϕ_t such that $w_{\phi_t}(e) \geq -1$ (See for loop in Line 4). The algorithm then defines a graph $G^* = (V, E, w^*)$ with $w^*(e) = w_{\phi_t}(e) + 1$ (Line 7). In the analysis, we will argue that because we are dealing with the scaled graph $\bar{G}$, the additive $+1$ is insignificant and does not affect the shortest path structure (Claim 5.3), so running Dijkstra on G^* will return correct shortest paths in G (Lines 8 and 9).

Algorithm 2: Algorithm for SPmain$(G_{in} = (V, E, w_{in}), s_{in})$

1 $\bar{w}(e) \leftarrow w_{in}(e) \cdot 2n$ for all $e \in E$, $\bar{G} \leftarrow (V, E, \bar{w})$, $B \leftarrow 2n$. `// scale up edge weights`
2 Round B up to nearest power of 2 `// still have` $\bar{w}(e) \geq -B$ `for all` $e \in E$
3 $\phi_0(v) = 0$ for all $v \in V$ `// identity price function`
4 **for** $i = 1$ *to* $t := \log_2(B)$ **do**
5 $\quad \psi_i \leftarrow \text{ScaleDown}(\bar{G}_{\phi_{i-1}}, \Delta := n, B/2^i)$
6 $\quad \phi_i \leftarrow \phi_{i-1} + \psi_i$ `// (Claim 5.1)` $w_{\phi_i}(e) \geq -B/2^i$ `for all` $e \in E$
7 $G^* \leftarrow (V, E, w^*)$ where $w^*(e) \leftarrow \bar{w}_{\phi_t}(e) + 1$ for all $e \in E$.
`// Observe:` G^* `in above line has non-negative weights`
8 Compute a shortest path tree T from s using Dijkstra(G^*, s) (Lemma 3.1)
`// (Claim 5.3) Will Show: any shortest path in` G^* `is also shortest in` G
9 **return** shortest path tree T .

Correctness We focus on the case where the algorithm terminates, and hence every line is executed. First we argue that weights in G^* (Line 7) are non-negative.

[15]The expectation in (5) is over the random outcomes of the low-diameter decomposition in Phase 0 and the recursion in Phase 1. Note that both $\eta_{(G_s^B)_{\phi_2}}(v;s)$ and $|P_{G^B}(v) \cap E^{rem}|$ are random variables. Since we always have $\eta_{(G_s^B)_{\phi_2}}(v;s) \leq |P_{G^B}(v) \cap E^{rem}| + 1$, we also have $E\left[\eta_{(G_s^B)_{\phi_2}}(v;s)\right] \leq E\left[|P_{G^B}(v) \cap E^{rem}|\right] + 1$.

Claim 5.1. *If the algorithm terminates, then for all $e \in E$ and $i \in [0, t := \log_2(B)]$ we have that $\bar{w}_i$ is integral and that $\bar{w}_i(e) \geq -B/2^i$ for all $e \in E$. Note that this implies that $\bar{w}_t(e) \geq -1$ for all $e \in E$, and so the graph G^* has non-negative weights.*

Proof. We prove the claim by induction on i. For base case $i = 0$, the claim holds for $\bar{G}_{\phi_0} = \bar{G}$ because $w_{in}(e) \geq -1$ (see Assumption 2.1), so $\bar{w}(e) \geq -2n \geq -B$ (see Lines 1 and 2).

Now assume by induction that the claim holds for $\bar{G}_{\phi_{i-1}}$. The call to ScaleDown$(\bar{G}_{\phi_{i-1}}, \Delta := n, B/2^i)$ in Line 5 satisfies the necessary input properties (See Theorem 3.5): property 1a holds by the induction hypotheses; property 1b holds because we have $\eta_G(v) \leq n$ for any graph G with no negative-weight cycle; property 1c holds because G_{in} has constant out-degree, and the algorithm never changes the graph topology. Thus, by the output guarantee of ScaleDown we have that $(\bar{w}_{\phi_{i-1}})_{\psi_i}(e) \geq (B/2^{i-1})/2 = B/2^i$. The claim follows because as noted in Definition 2.5, $(\bar{w}_{\phi_{i-1}})_{\psi_i} = \bar{w}_{\phi_{i-1}+\psi_i} = \bar{w}_{\phi_i}$. ☐

Corollary 5.2. *If G_{in} contains a negative-weight cycle then the algorithm does not terminate.*

Proof. Say, for contradiction, that the algorithm terminates; then by Claim 5.1 we have that $\bar{w}_{\phi_t}(e) \geq -1$. Now, let C be any negative-weight cycle in G_{in}. Since all weights in $\bar{G}$ are multiples of $2n$ (Line 1), we know that $\bar{w}(C) \leq -2n$. But we also know that $\bar{w}_{\phi_t}(C) \geq -|C| \geq -n$. So $\bar{w}(C) \neq \bar{w}_{\phi_t}(C)$, which contradicts Lemma 2.7. ☐

Now we show that the algorithm produces a correct output.

Claim 5.3. *Say that G_{in} contains no negative-weight cycle. Then, the algorithm terminates, and if P is a shortest sv-path in G^* (Line 7) then it is also a shortest sv-path in G_{in}. Thus, the shortest path tree T of G^* computed in Line 9 is also a shortest path tree in G_{in}.*

Proof. The algorithm terminates because each call to ScaleDown$(\bar{G}_{\phi_{i-1}}, \Delta := n, B/2^i)$ terminates, because $\bar{G}_{\phi_{i-1}}$ is equivalent to G_{in} (Lemma 2.7) and so does not contain a negative-weight cycle. Now we show that

$$P \text{ is also a shortest path in } \bar{G}_{\phi_t} \tag{6}$$

which implies the claim because $\bar{G}_{\phi_t}$ and G_{in} are equivalent. We assume that $s \neq v$ because otherwise the claim is trivial. Observe that since all weights in $\bar{G}$ are multiples of $2n$ (Line 1), all shortest distances are also multiples of $2n$, so for any two sv-paths P_{sv} and P'_{sv}, $|\bar{w}(P_{sv}) - \bar{w}(P'_{sv})|$ is either 0 or $> n$. It is easy to check that by Lemma 2.7, we also have that

$$|\bar{w}_{\phi_t}(P_{sv}) - \bar{w}_{\phi_t}(P'_{sv})| \text{ is either 0 or } > n. \tag{7}$$

Moreover, by definition of G^* we have

$$\bar{w}_{\phi_t}(P_{sv}) < w^*(P_{sv}) = \bar{w}_{\phi_t}(P_{sv}) + |P_{sv}| < \bar{w}_{\phi_t}(P_{sv}) + n. \tag{8}$$

Now, we prove (6) . Assume for contradiction that there was a shorter path P' in $\bar{G}_{\phi_t}$. Then,

$$\bar{w}_{\phi_t}(P) - \bar{w}_{\phi_t}(P') \overset{(7)}{>} n. \tag{9}$$

So, $w^*(P') \overset{(8)}{<} \bar{w}_{\phi_t}(P') + n \overset{(9)}{<} \bar{w}_{\phi_t}(P) \overset{(8)}{<} w^*(P)$, contradicting P being shortest in G^*. ☐

Running Time Analysis: By Corollary 5.2, if G_{in} does not contain a negative-weight cycle then the algorithm does not terminate, as desired. We now focus on the case where G_{in} does not contain a negative-weight cycle. The running time of the algorithm is dominated by the $\log(B) = O(\log(n))$ calls to ScaleDown$(\bar{G}_{\phi_{i-1}}, \Delta := n, B/2^i)$. Note that all the input graphs $\bar{G}_{\phi_{i-1}}$ are equivalent to G_{in}, so they do not contain a negative-weight cycle. By Theorem 3.5, the expected runtime of each call to ScaleDown is $O(m\log^3(n)\log(\Delta)) = O(m\log^4(n))$. So, the expected runtime of SPmain is $O(m\log^5(n))$.

6 Algorithm for Low-Diameter Decomposition

In this section, we prove Lemma 1.2 which we restate here for convenience. Through this section, we often shorten LowDiamDecomposition(G, D) to LDD(G, D).

Lemma 1.2. *There is an algorithm* LowDiamDecomposition(G, D) *with the following guarantees:*

- *INPUT: an m-edge, n-vertex graph $G = (V, E, w)$ with non-negative integer edge weight function w and a positive integer D.*
- *OUTPUT: A set of edges E^{rem} with the following guarantees:*
  - *each SCC of $G \setminus E^{rem}$ has weak diameter at most D; that is, if u, v are in the same SCC, then $\text{dist}_G(u, v) \leq D$ and $\text{dist}_G(v, u) \leq D$.*
  - *For every $e \in E$, $\Pr[e \in E^{rem}] = O\left(\frac{w(e)\cdot\log^2 n}{D} + n^{-10}\right)$. These probabilities are not guaranteed to be independent.*[16]
- *RUNNING TIME: The algorithm has running time $O(m\log^2 n + n\log^3 n)$.*

We start with some basic definitions.

Definition 6.1 (balls and boundaries)**.** Given a directed graph $G = (V, E)$, a vertex $v \in V$, and a distance-parameter R, we define $\text{Ball}^{\text{out}}_G(v, R) = \{u \in V \mid \text{dist}(v, u) \leq R\}$. We define $\mathsf{boundary}(\text{Ball}^{\text{out}}_G(v, R)) = \{(x, y) \in E \mid x \in \text{Ball}^{\text{out}}_G(v, R) \wedge y \notin \text{Ball}^{\text{out}}_G(v, R)\}$. Similarly, we define $\text{Ball}^{\text{in}}_G(v, R) = \{u \in V \mid \text{dist}(u, v) \leq R\}$ and we define $\mathsf{boundary}(\text{Ball}^{\text{in}}_G(v, R)) = \{(x, y) \in E \mid y \in \text{Ball}^{\text{in}}_G(v, R) \wedge x \notin \text{Ball}^{\text{in}}_G(v, R)\}$. We often use Ball^{*}_G to denote that a ball can be either an out-ball or an in-ball, i.e., $* \in \{in, out\}$.

Definition 6.2 (Geometric Distribution)**.** Consider a coin whose probability of heads is $p \in (0, 1]$. The geometric distribution $\text{Geo}(p)$ is the probability distribution of the number X of independent coin tosses until obtaining the first heads. We have $\Pr[X = k] = p(1-p)^{k-1}$ for every $k \in \{1, 2, 3, ...\}$.

Remark 6.3. *In Lemma 1.2 and throughout this section, n always refers to the number of vertices in the main graph G_{in}, i.e. graph in which we are trying to compute shortest paths. This is to ensure that "high probability" is defined in terms of the number of vertices in G_{in}, rather than in terms of potentially small auxiliary graphs. Note that whenever our shortest path algorithm executes* LDD(G, D) *we always have $|V(G)| \leq n$ (we never add new vertices).*

The algorithm. The pseudocode for our low-diameter decomposition algorithm can be found in Algorithm 3. Roughly, in Phase 1 each vertex is marked as either *in-light*, *out-light*, or *heavy*. In Phase 2 we repeatedly "remove" balls centered at either in-light or out-light vertices: Let v be any in-light vertex (the process is similar for out-light vertices). Consider a ball $\text{Ball}^{\text{in}}_G(v, R_v)$ with radius R_v selected randomly from the geometric distribution $\text{Geo}(p)$, with $p = \min\{1, 80\log(n)/D\}$ (Lines 12 and 13)[17]. We add edges pointing into this ball (i.e. edges in $\mathsf{boundary}(\text{Ball}^{\text{in}}_G(v, R_v))$)

[16] The 10 in the exponent suffices for our application but can be replaced by an arbitrarily large constant.

[17] Throughout this section, $\log(n)$ always denotes $\log_2(n)$

Algorithm 3: Algorithm for LDD$(G = (V, E), D)$

Let n be the global variable in Remark 6.3
$G_0 \leftarrow G, E^{rem} \leftarrow \emptyset$

// Phase 1: mark vertices as light or heavy
$k \leftarrow c \ln(n)$ for large enough constant c
$S \leftarrow \{s_1, \ldots, s_k\}$, where each s_i is a random node in V // possible: $s_i = s_j$ for $i \neq j$
For each $s_i \in S$ compute $\text{Ball}^{\text{in}}_{\text{G}}(s_i, D/4)$ and $\text{Ball}^{\text{out}}_{\text{G}}(s_i, D/4)$
For each $v \in V$ compute $\text{Ball}^{\text{in}}_{\text{G}}(v, D/4) \bigcap S$ and $\text{Ball}^{\text{out}}_{\text{G}}(v, D/4) \bigcap S$ using Line 5
foreach $v \in V$ **do**
    If $|\text{Ball}^{\text{in}}_{\text{G}}(v, D/4) \bigcap S| \leq .6k$, mark v *in-light* // whp $|\text{Ball}^{\text{in}}_{\text{G}}(v, D/4)| \leq .7|V(G)|$
    Else if $|\text{Ball}^{\text{out}}_{\text{G}}(v, D/4) \bigcap S| \leq .6k$, mark v *out-light* // whp $|\text{Ball}^{\text{out}}_{\text{G}}(v, D/4)| \leq .7|V(G)|$
    Else mark v *heavy* // w.h.p $\text{Ball}^{\text{in}}_{\text{G}}(v, D/4) > .5|V(G)|$ and $\text{Ball}^{\text{out}}_{\text{G}}(v, D/4) > .5|V(G)|$

// Phase 2: carve out balls until no light vertices remain
while G *contains a node* v *marked* $*$*-light for* $* \in \{in, out\}$ **do**
    Sample $R_v \sim \text{Geo}(p)$ for $p = \min\{1, 80 \log_2(n)/D\}$.
    Compute $\text{Ball}^{*}_{\text{G}}(v, R_v)$.
    $E^{boundary} \leftarrow \text{boundary}(\text{Ball}^{*}_{\text{G}}(v, R_v))$ // add boundary edges of ball to E^{rem}.
    If $R_v > D/4$ or $|\text{Ball}^{*}_{\text{G}}(v, R_v)| > .7|V(G)|$ then return $E^{rem} \leftarrow E(G)$ and terminate // Pr[terminate] $\leq 1/n^{20}$
    $E^{recurse} \leftarrow \text{LDD}(G[\text{Ball}^{*}_{\text{G}}(v, R_v)], D)$ // recurse on ball
    $E^{rem} \leftarrow E^{rem} \bigcup E^{boundary} \bigcup E^{recurse}$.
    $G \leftarrow G \setminus \text{Ball}^{*}_{\text{G}}(v, R_v)$ // remove ball from G

// Clean Up: check that remaining vertices have small weak diameter in initial input graph G_0.
Select an arbitrary vertex v in G.
If $\text{Ball}^{in}_{G_0}(v, D/2) \not\supseteq V(G)$ or $\text{Ball}^{out}_{G_0}(v, D/2) \not\supseteq V(G)$ then return $E^{rem} \leftarrow E(G)$ and terminate // Pr[terminate] $\leq 1/n^{20}$. If this does not terminate, then all remaining vertices in $V(G)$ have weak diameter $\leq D$
Return E^{rem}

to $E^{boundary}$, which will be later added to E^{rem}. We recurse the algorithm on this ball (Line 16) which may add more edges to E^{rem} (via $E^{recurse}$). Finally, we remove this ball from the graph and repeat the process. Note that the algorithm may also terminate prematurely with $E^{rem} = E(G)$ in Lines 15 and 20. We will show that this happens with very low probability (Section 6.3).

Analysis. The remainder of this section is devoted to analyze the above algorithm. In Section 6.1, we make a few simple observations. Then, in Section 6.2, we analyze the runtime of the algorithm. The next two subsections are to bound the probability that an edge is in E^{rem}: in Section 6.3 we prove that the probability that the algorithm terminates prematurely on Lines 15 or 20 is very small, and complete the bound of $Pr[e \in E^{rem}]$ in Section 6.4. Finally, in Section 6.5, we prove the diameter bound, completing the proof of Lemma 1.2.

6.1 Observations

Observation 6.4. *Consider the execution of an initial call* $\text{LDD}(G, D)$ *and let* $\text{LDD}(G_i, D)$ *denote all the lower-level recursive calls that follow from it. The following always holds:*

1. *For each vertex v, there are at most $O(\log(n))$ calls $\text{LDD}(G_i, D)$ such that G_i contains v.*
2. *The total number of recursive calls $\text{LDD}(G_i, D)$ is $O(n \log(n))$.*

Proof. The second property follows from the first because G has at most n vertices, and each call $\text{LDD}(G_i)$ contains at least one of those vertices. For the first property, note that **(i)** recursion only occurs in Line 16, **(ii)** if a vertex participates in the recursion in Line 16, it will be removed from G on Line 18, **(iii)** each time the algorithm recurses, the number of vertices in the child-graph is at most 7/10 the number of vertices in the parent-graph because otherwise the algorithm terminates in Line 15 without recursing, and **(iv)** the algorithm always terminates when $|V(G)| = 1$ because every vertex is heavy in this case, so the while loop is never executed and no recursion occurs. Thus, a single vertex can participate in only $\log_{10/7}(n) = O(\log(n))$ recursive calls. □

Observation 6.5. *Consider any call* $\text{LDD}(G, D)$ *and say that the algorithm makes some recursive call* $\text{LDD}(G[\text{Ball}^*_G(v, R_v)], D)$ *in Line 16. Then, once the call* $\text{LDD}(G, D)$ *terminates, if vertex $x \in \text{Ball}^*_G(v, R_v)$ and vertex $y \notin \text{Ball}^*_G(v, R_v)$, then x and y are not in the same SCC in $G \setminus E^{rem}$.*

Proof. Say that the recursive call is on an out-ball $\text{Ball}^{\text{out}}_G(v, R_v)$; the argument for in-balls is the same. Note that all the edges of $\text{Ball}^{\text{out}}_G(v, R_v)$ are added to $E^{boundary}$ (Line 14) and later to E^{rem} (in Lines 15 or 17). Thus, in $G \setminus E^{rem}$ there are no edges leaving $\text{Ball}^{\text{out}}_G(v, R_v)$, so clearly vertices x and y cannot be in the same SCC. □

6.2 Running Time Analysis

Lemma 6.6. *The total running time of* $\text{LDD}(G, D)$ *is* $O(|V(G)| \log^3(n) + |E(G)| \log^2(n))$.

Proof. Let us first consider the non-recursive work, i.e. everything except Line 16. The balls in Line 5 are computed via $k = O(\log(n))$ executions of Dijkstra's algorithm in total time $O(|V(G)| \log^2(n) + |E(G)| \log(n))$. The sets in Line 6 are computed from the information in Line 5.

Every time the algorithm executes the While loop in Line 11 it first samples $R_v \sim \text{Geo}(p)$ from the geometric distribution; this can be done in $O(\log(n))$ time using e.g. [BF13].[18] It then computes some ball $\text{Ball}^*_G(v, R_v)$ (Line 13). All vertices in this ball are then removed from G (Line 18), so

[18] By Theorem 5 in [BF13] with $p = \log(n)/D$ and word size $w = \Omega(\log \log D)$, the expected time is $O(\log(\min\{D/\log n, n\})/\log w) = O(\log n)$. Note that there might be other simpler methods since we do not need an exact algorithm.

at this level of recursion each vertex participates in at most one ball. Each ball is computed via Dijkstra's algorithm, which requires $O(\log(n))$ time per vertex explored and $O(1)$ time per edge explored.[19] So, the total time to execute the while loop is $O(|V(G)|\log(n)+|E(G)|)$.

Finally, checking the diameter in Line 20 only requires a single execution of Dijkstra's algorithm. Thus the overall non-recursive work is $O(|V(G)|\log^2(n)+|E(G)|\log(n))$. By Observation 6.4, each vertex (and thus each edge) participates in $O(\log(n))$ calls, completing the lemma. □

6.3 Bounding termination probabilities

We now show that it is extremely unlikely for the algorithm to terminate in Lines 15 or 20. When we discuss some call $\text{LDD}(G,D)$ in this subsection, we will use G_0 to the denote the initial input to $\text{LDD}(G,D)$, in order to differentiate it from the graph G from which vertices can be deleted over the course of the algorithm (Line 18). Note that we always have $G \subseteq G_0$.

We start with a useful auxiliary claim.

Claim 6.7. *Consider a single call* $\text{LDD}(G,D)$ *(ignoring all future recursive calls). Then, with probability* $\geq 1-O(1/n^{19})$*, the following holds:*

- *For any vertex* v *marked in-light we have* $|\text{Ball}^{\text{in}}_{G_0}(v,D/4)| \leq .7|V(G_0)|$
- *For any vertex* v *marked out-light we have* $|\text{Ball}^{\text{out}}_{G_0}(v,D/4)| \leq .7|V(G_0)|$
- *For any vertex* v *marked heavy we have* $|\text{Ball}^{\text{in}}_{G_0}(v,D/4)| > .5|V(G_0)|$ *AND* $|\text{Ball}^{\text{out}}_{G_0}(v,D/4)| > .5|V(G_0)|$

Proof. Consider any single instance of a vertex v being marked during the recursive call. We will show that for this vertex the claim holds with probability $\geq 1-O(1/n^{20})$. A union bound over all $|V(G_0)| \leq n$ vertices completes the lemma.

We first show that for any vertex v such that $|\text{Ball}^{\text{in}}_{G_0}(v,D/4)| > .7|V(G_0)|$, we have $\Pr[v \text{ is marked in-light }] \leq 1/n^{20}$: Since $|\text{Ball}^{\text{in}}_{G_0}(v,D/4)| > .7|V(G_0)|$ and since the $k = c\log(n)$ vertices from S are sampled randomly, we know that $\mathbb{E}[|\text{Ball}^{\text{in}}_{G_0}(v,D/4) \bigcap S|] > .7k$. But for v to be marked in-light we must have $|\text{Ball}^{\text{in}}_{G_0}(v,D/4) \bigcap S| \leq .6k$; a Chernoff bounds shows such a high deviation from expectation to be extremely unlikely.[20]

By similar proofs, it also holds that (a) for any vertex v such that $|\text{Ball}^{\text{out}}_{G_0}(v,D/4)| > .7|V(G_0)|$, we have $\Pr[v \text{ is marked out-light }] \leq 1/n^{20}$ and (b) for any vertex v such that $|\text{Ball}^{\text{in}}_{G_0}(v,D/4)| \leq .5|V(G_0)|$ or $|\text{Ball}^{\text{out}}_{G_0}(v,D/4)| \leq .5|V(G_0)|$, we have $\Pr[v \text{ is marked heavy}] \leq 1/n^{20}$. □

Claim 6.8. *Consider a single call* $\text{LDD}(G,D)$ *(ignoring all future recursive calls). With probability* $\geq 1-O(1/n^{19})$*,* $\text{LDD}(G,D)$ *does not terminate in Lines 15 or 20.*

Proof. Let us assume that the statement of Claim 6.7 holds. Under this assumption the algorithm clearly cannot terminate in the second condition in Line 15 (i.e. $\text{Ball}^*_G(v,R_v)$ being too large), because $\text{Ball}^*_G(v,R_v) \subseteq \text{Ball}^*_{G_0}(v,R_v)$. We now show that the algorithm also cannot terminate in

[19] A standard Dijkstra's implementation requires $O(|V(G)|)$ time to initialize the priority queue and vertices' labels (e.g. [CLRS09]). This can be easily modified to avoid the initialization time (e.g. [TZ05]). One way to do this is to initialize the priority queue and labels *once* in the beginning of Phase 2. After we use them to compute each ball $\text{Ball}^*_G(v,R_v)$ on Line 13, we reinitialize them at the cost of the number of explored vertices.

[20] We use the following Chernoff's bound. For any independent 0/1 random variables $X_1,\ldots,X_k$, if $X_{avg} := (1/k)\sum_{i=1}^k X_i$, then $Pr[X_{avg} < \mathbb{E}[X_{avg}]-\epsilon] \leq e^{-2k\epsilon^2}$. Here, we use $\epsilon = 0.1$ and define $X_i = 1$ iff $s_i \in \text{Ball}^*_{G_0}(v,D/4)$ (so, $\mathbb{E}[X_{avg}] = \frac{|\text{Ball}^*_{G_0}(v,R_v)|}{|V(G_0)|} > 0.7$).

Line 20. Note that by the time we get to Line 20, the only vertices remaining in G must have been marked heavy, because otherwise the while loop in Line 11 would continue. We now show that if x and y are marked heavy then $\text{dist}_{G_0}(x,y) \leq D/2$, which implies that the algorithm does not terminate in Line 20. To see that $\text{dist}_{G_0}(x,y) \leq D/2$, recall that we are assuming the statement of Claim 6.7, so $\text{Ball}^{in}_{G_0}(x, D/4) > |V(G_0)|/2$ and $\text{Ball}^{out}_{G_0}(x, D/4) > |V(G_0)|/2$, so there is some vertex w in the intersection of the two balls, so $\text{dist}_{G_0}(x,y) \leq \text{dist}_{G_0}(x,w) + \text{dist}_{G_0}(w,y) \leq D/4 + D/4 = D/2$. (Note that w might not be marked heavy and also might have been removed from G, which is why we only guarantee weak diameter.)

Thus, since Claim 6.7 holds with probability $\geq 1 - O(1/n^{19})$, we know that with this same probability the algorithm does not terminate in Line 20, and does not terminate in the the second condition of Line 15.

We now need to bound the probability of terminating in the first condition of Line 15 (i.e. $R_v > D/4$). Recall that $p = \min\{1, 80\log(n)/D\}$ and the definition of the geometric distribution (Definition 6.2). Every time the algorithm samples $R_v \sim \text{Geo}(p)$ in Line 12 we have that $\Pr[R_v > D/4] = \Pr[\text{the first D/4 coins are all tails}] = (1-p)^{D/4} \leq (1 - \frac{80\log(n)}{D})^{D/4} < 1/n^{20}$. Since each variable R_v is associated with a specific vertex v, a union bound over all $|V(G_0)| \leq n$ vertices completes the proof. □

6.4 Bounding $Pr[e \in E^{rem}]$

Observe that $e \in E^{rem}$ if in one of the recursive calls it is either (i) added to $E^{boundary}$ on Line 14, or (ii) added to E^{rem} before a recursive call terminates on Lines 15 and 20. The latter case was shown in the previous subsection (Claim 6.8) to happen with probability $O(n^{-19})$ in each recursive call, thus with probability $O(n^{-18}\log(n))$ over all $O(n\log n)$ recursive calls (Observation 6.4). Below (Corollary 6.10), we show that the former case happens with probability $O\left(\frac{w(e)\cdot(\log n)^2}{D}\right)$ over all recursive calls. So, the probability that e is added to E^{rem} in any of the recursive calls is $O\left(\frac{w(e)\cdot(\log n)^2}{D} + \frac{\log(n)}{n^{18}}\right) = O\left(\frac{w(e)\cdot(\log n)^2}{D} + n^{-10}\right)$ as claimed.

Lemma 6.9. *In a single call of Algorithm 3 (ignoring all future recursive calls),*

$$Pr[e \in E^{boundary}] = O\left(\frac{w(e)\cdot(\log n)}{D}\right).$$

Proof intuition. We provide a formal proof of Lemma 6.9 in Appendix D. For an intuition, note that since R_v is the number of coin tosses until obtaining the first head (Definition 6.2), the algorithm can be viewed as the following *ball-growing process:* Start with a ball $\text{Ball}^*_G(v,1)$ of radius 1 at some vertex v and $* \in \{in, out\}$. We flip a coin that turns head with probability p. Every time we get a tail, we increase the radius of the ball by one. We stop when we get a head. We then put all edges in $\mathsf{boundary}(\text{Ball}^*_G(v, R_v))$ into $E^{boundary}$ and cut out the ball. We may then repeat the ball growing process from a new vertex.

To analyze $Pr[e \in E^{boundary}]$, consider any edge (x,y). Consider the first time that x is contained in a ball $B^*_G(v,r)$ (for some r) during the ball-growing process above. Observe that if the next $w(x,y)$ coin tosses all return tails, then (x,y) will *not* be in $E^{boundary}$ (because y is either in the ball $B^*_G(v, r+w(x,y))$ or is no longer in G). In other words, $Pr[e \in E^{boundary}]$ is at most the probability that one of the next $w(x,y)$ coin tosses returns head. This is at most $p\cdot w(x,y) = O\left(\frac{w(e)\cdot(\log n)}{D}\right)$. □

Corollary 6.10. *For any edge e, the probability that* one *of the recursive calls adds e to $E^{boundary}$ is $O\left(\frac{w(e)\cdot(\log n)^2}{D}\right)$.*

Proof. By Observation 6.4, for every edge e, there are $O(\log n)$ calls LDD(G_i, D) such that G_i contains e. The corollary follows by Union Bound. □

6.5 Diameter Analysis

We now show that the set E^{rem} returned by LDD(G, D) satisfies the first output property of Lemma 1.2. The proof is essentially trivial, since the low-diameter condition is ensured by Line 20.

Lemma 6.11. *For any graph G,* LDD(G, D) *returns E^{rem} such that each SCC of $G \setminus E^{rem}$ has weak diameter $\leq D$.*

Proof. The proof will be induction on $|V(G)|$. The base case where $|V(G)| = 1$ trivially holds. We now assume by induction that the Lemma holds for all LDD(G', D) with $|V(G')| < |V(G)|$. Note that in the remaining of this proof we let G denote the initial input to LDD(G, D) and not the graph that changes throughout the execution of the algorithm.

Consider any two vertices $x, y \in V(G)$. We want to show that LDD(G, D) always returns E^{rem} such that

$$\textit{if } x \textit{ and } y \textit{ are in the same SCC in } G \setminus E^{rem}\textit{, then } \text{dist}_G(x, y) \leq D \textit{ and } \text{dist}_G(y, x) \leq D. \quad (10)$$

Note that if a vertex is in $V(G')$ for recursive call LDD(G', D) in Line 16 then it is immediately removed from the graph G (Line 18) and never touched again during the execution of the current call LDD(G, D). There are thus three possible cases regarding x and y.

Case 1 There is some recursive call LDD(G', D) in Line 16 such that one of x, y is in G' and the other is not.

Case 2 There is some recursive call LDD(G', D) in Line 16 such that both x and y are in G'.

Case 3 Neither x or y participate in any recursive call.

In Case 1, we know by Observation 6.5 that x and y cannot end up in the same SCC of $G \setminus E^{rem}$, so the property (10) holds.

In Case 2, note that for recursive call LDD(G', D) we have that G' is an induced subgraph of G, and also that $V(G') \leq .7V(G)$, because otherwise the algorithm would have terminated in Line 15. Thus, by the induction hypothesis, the Lemma holds for LDD(G', D). In particular, there are two options: **(1)** If x and y are in the same SCC of $G' \setminus E^{rem}$, then $\text{dist}_G(x, y) \leq \text{dist}_{G'}(x, y) \leq D$ and $\text{dist}_G(y, x) \leq \text{dist}_{G'}(y, x) \leq D$ and **(2)** if x and y are not in the same SCC in $G' \setminus E^{rem}$, then they are also not in the same SCC of $G \setminus E^{rem}$, because by Observation 6.5, none of the vertices of $V(G) \setminus V(G')$ can be in the same SCC as either x or y in $G \setminus E^{rem}$.

Finally, Case 3 holds because if x and y don't participate in any recursive calls, then they are not removed from graph G. In Line 20 the algorithm checks that all vertices which weren't removed have weak diameter $\leq D$. If the check fails then the algorithm sets $E^{rem} \leftarrow E(G)$, in which case the Lemma trivially holds because all SCCs of $G \setminus E^{rem}$ will be singletons. □

i

Acknowledgement

We thank Sepehr Assadi, Joakim Blikstad, Thatchaphol Saranurak, and Satish Rao for their comments on the early draft of the paper and the discussions. We thank Mohsen Ghaffari, Merav Parter, Satish Rao, and Goran Zuzic for pointing out some literature on low-diameter decomposition. A discussion with Mohsen Ghaffari led to some simplifications of our low-diameter decomposition algorithm. We also thank Vikrant Ashvinkumar, Nairen Cao, Christoph Grunau, and Yonggang Jiang for pointing out an earlier error in the paper: in Phase 3, we originally ran ElimNeg on graph $(G^B_{\phi_2})_s$ and claimed that this graph is equivalent to G^B_s; but this is not true because the price function is applied at the wrong time in the order of operations. We thus instead run ElimNeg on $(G^B_s)_{\phi_2}$, which is indeed equivalent to G^B_s.

Funding: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 715672. Nanongkai was also partially supported by the Swedish Research Council (Reg. No. 2019-05622).

Wulff-Nilsen was supported by the Starting Grant 7027-00050B from the Independent Research Fund Denmark under the Sapere Aude research career programme.

Bernstein is supported by NSF CAREER grant 1942010.

References

[ABC+23] Vikrant Ashvinkumar, Aaron Bernstein, Nairen Cao, Christoph Grunau, Bernhard Haeupler, Yonggang Jiang, Danupon Nanongkai, and Hsin Hao Su. Parallel and distributed exact single-source shortest paths with negative edge weights, 2023. 3

[ABCP92] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast network decomposition (extended abstract). In *PODC*, pages 169–177. ACM, 1992. 3

[AGLP89] Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *FOCS*, pages 364–369. IEEE Computer Society, 1989. 3

[AMV20] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. In *FOCS*, pages 93–104. IEEE, 2020. 1, 2

[AP92] Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. *SIAM J. Discret. Math.*, 5(2):151–162, 1992. 3

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985. Announced at STOC'84. 3

[Bar96] Yair Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 184–193. IEEE Computer Society, 1996. 3

[BBP+20] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. *CoRR*, abs/2004.08432, 2020. 1

[BCF23] Karl Bringmann, Alejandro Cassis, and Nick Fischer. Negative-weight single-source shortest paths in near-linear time: Now faster! *CoRR*, abs/2304.05279, 2023. 2

[Bel58] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. 1

[BF13] Karl Bringmann and Tobias Friedrich. Exact and efficient generation of geometric random variates and random graphs. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, volume 7965 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2013. 17

[BGK+14] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory Comput. Syst.*, 55(3):521–554, 2014. 3

[BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1123–1134. IEEE, 2020. 9

[BLL+21] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and $_1$-regression in nearly linear time for dense instances. In *STOC*, pages 859–869. ACM, 2021. 1, 2

[BLN+20] 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 *FOCS*, pages 919–930. IEEE, 2020. 1, 2

[BLSS20] Jan van den Brand, Yin Tat Lee, Aaron Sidford, and Zhao Song. Solving tall dense linear programs in nearly linear time. In *STOC*. https://arxiv.org/pdf/2002.02304.pdf, 2020. 1, 2

[BPW20] Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. Near-optimal decremental SSSP in dense weighted digraphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1112–1122. IEEE, 2020. 3

[Bra20] Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In *SODA*, pages 259–278. SIAM, 2020. 2

[CGL+20] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *FOCS*, 2020. https://arxiv.org/pdf/1910.08025.pdf. 1

[CKL+22] Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. March 2022. 2

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. 9, 12, 18

[CLS19] Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. In *STOC*, 2019. https://arxiv.org/pdf/1810.07896. 2

[CMSV17] Michael B Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $O(m^{10/7} \log W)$ time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–771. SIAM, 2017. 1, 2

[CZ20] Shiri Chechik and Tianyi Zhang. Dynamic low-stretch spanning trees in subpolynomial time. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 463–475. SIAM, 2020. 3

[DS08] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *STOC*, pages 451–460. ACM, 2008. 2

[FG19] Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 377–388. ACM, 2019. 3

[FGdV21] Sebastian Forster, Martin Grösbacher, and Tijn de Vos. An improved random shift algorithm for spanners and low diameter decompositions. In *OPODIS*, volume 217 of *LIPIcs*, pages 16:1–16:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 3

[For56] R. Ford. *Paper P-923*. The RAND Corperation, Santa Moncia, California, 1956. 1

[FR06] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006. Announced at FOCS'01. 1

[Gab85] Harold N. Gabow. Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168, 1985. announced at FOCS'83. 1

[Gol95] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995. Announced at SODA'93. 1, 4

[GT89] Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989. 1

[HKRS97] Monika Rauch Henzinger, Philip N. Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997. Announced at STOC'94. 1

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977. 5

[KMW10] Philip N. Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. *ACM Trans. Algorithms*, 6(2):30:1–30:18, 2010. Announced at SODA'09. 1

[Law66] Eugene L. Lawler. Optimal cycles in doubly weighted linear graphs, theory of graphs: International symposium, 1966. 3

[Law76] Eugene L. Lawler. *Combinatorial optimization - networks and matroids*. Holt, Rinehart and Winston, New York, 1976. 3

[LRT79] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979. 1

[LS93] Nathan Linial and Michael E. Saks. Low diameter graph decompositions. *Comb.*, 13(4):441–454, 1993. 3

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ(sqrt(rank)) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433, 2014. 2

[LS20] Yang P Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In *STOC*. https://arxiv.org/pdf/1910.14276.pdf, 2020. 2

[Mad13] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *FOCS*, pages 253–262. IEEE Computer Society, 2013. 2

[Mad16] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016. 2

[Moo59] E. F. Moore. The Shortest Path Through a Maze. In Proceedings of the International Symposium on the Theory of Switching, pages 285–292, 1959. 1

[MPX13] Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In Guy E. Blelloch and Berthold Vöcking, editors, *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 196–203. ACM, 2013. 3

[MW10] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n\log^2 n/\log\log n)$ time. In *ESA (2)*, volume 6347 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 2010. 1

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $O(n^{1/2-\epsilon})$-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1122–1129, 2017. 1

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017. 1

[PRS+18] Jakub Pachocki, Liam Roditty, Aaron Sidford, Roei Tov, and Virginia Vassilevska Williams. Approximating cycles in directed graphs: Fast algorithms for girth and roundtrip spanners. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1374–1392. SIAM, 2018. 3

[San05] Piotr Sankowski. *Algorithms – ESA 2005: 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005. Proceedings.* chapter Shortest Paths in Matrix Multiplication Time, pages 770–778. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. 1

[Shi55] A. Shimbel. Structure in Communication Nets. In *In Proceedings of the Symposium on Information Networks*, pages 199–203, Brooklyn, 1955. Polytechnic Press of the Polytechnic Institute of Brooklyn. 1

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *SODA*, pages 2616–2635. SIAM, 2019. 1

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999. Announced at FOCS'97. 1

[Tho04] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.*, 69(3):330–353, 2004. Announced at STOC'03. 1

[TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. 18

[Wul11] Christian Wulff-Nilsen. Separator theorems for minor-free and shallow minor-free graphs with applications. In *FOCS*, pages 37–46. IEEE Computer Society, 2011. 1

[Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1130–1143, 2017. 1

[YZ05] Raphael Yuster and Uri Zwick. Answering distance queries in directed graphs using fast matrix multiplication. pages 389–396, 2005. 1

Appendix A Proof of Lemma 3.3 (ElimNeg)

Lemma 3.3. (ElimNeg) *There exists an algorithm* ElimNeg(G, s) *that takes as input a graph* $G = (V, E, w)$ *in which all vertices* $v \neq s$ *have constant out-degree and a source* $s \in V$ *that can reach all vertices in* V*. The algorithm outputs a price function* ϕ *such that* $w_\phi(e) \geq 0$ *for all* $e \in E$ *and has running time* $O(\log(n) \cdot (n + \sum_{v \in V} \eta_G(v; s)))$ *(Definition 2.4); note that if* G *contains a negative-weight cycle then* $\sum_{v \in V} \eta_G(v; s) = \infty$ *so the algorithm will never terminate and hence not produce any output.*

Algorithm 4: Algorithm for ElimNeg(G)

```
1  Set d(s) ← 0 and d(v) ← ∞ for v ≠ s
2  Initialize priority queue Q and add s to Q.
3  Initially, every vertex is unmarked
   // Dijkstra Phase
4  while Q is non-empty do
5      Let v be the vertex in Q with minimum d(v)
6      Mark v
7      foreach edge (v, x) ∈ E \ E^neg(G) do
8          if d(v) + w(v, x) < d(x) then
9              if x ∉ Q, add x to Q
10             d(x) ← d(v) + w(v, x) and decrease-key(Q, x, d(x))
11     Extract v from Q
   // Bellman-Ford Phase
12 foreach marked vertex v do
13     foreach edge (v, x) ∈ E(G) do
14         if d(v) + w(v, x) < d(x) then
15             if x ∉ Q, add x to Q
16             d(x) ← d(v) + w(v, x) and decrease-key(Q, x, d(x))
17     Unmark v
18 If Q is empty: return d(v) for each v ∈ V        // labels do not change so we have
     correct distances.
19 Go to Line 4                                    // Q is non-empty.
```

At a high level, we implement ElimNeg(G, s) to compute distance estimates $d(v)$ so that at termination, $d(v) = \text{dist}_G(s, v)$ for each $v \in V$; we will then return $\phi(v) = \text{dist}_G(s, v)$. The pseudocode is given in Algorithm 4. We use priority queue Q that supports finding an element with a minimum key, extracting an element, and decreasing the key of some element x to k (decrease-key(Q, x, k)). It can be implemented as a binary heap to support each queue in $O(\log n)$ time. (This suffices for our needs, although there exist better implementations, such as the Fibonacci heap.) Note that, instead of extracting v on Line 11, one can do so at the same time when v is marked on Line 6. Also, on Line 13, it suffices to only consider edges $(v, x) \in E^{neg}(G)$. We instead consider all edges in $E(G)$ because this implementation simplifies some of the proofs.

A.1 Correctness

Throughout the execution of the algorithm, We define an edge (v, x) of G to be *active* if $d(v) + w(v, x) < d(x)$ and *inactive* otherwise. We define an edge (v, x) for which $d(v) = d(x) = \infty$ to be inactive.

Invariant A.1. *At any point during the execution of Algorithm 4, if an edge (v, x) is active, then v is in Q or marked. If (v, x) is active and $(v, x) \in E \setminus E^{neg}(G)$, then v is in Q.*

Proof. The statements hold initially: For all $v \in V \setminus \{s\}$, any edge (v, x) is inactive since $d(v) = \infty$ and s is in Q. The statements are affected when we (i) extract a vertex v from Q (Line 11), (ii) unmark a vertex v (Line 17), or (iii) decrease $d(x)$ for some vertex x (Lines 10 and 16), potentially making some inactive edges (x, y) active.

For (i), when a vertex v is extracted from Q on Line 11, it was already marked on Line 6, so the first statement still holds. Also, all edges $(v, x) \in E \setminus E^{neg}(G)$ were made inactive using the for-loop (Lines 7-10, by setting $d(x) \leftarrow \min\{d(x), d(v) + w(v, x)\}$); thus, the second statement still holds. For (ii), before we unmark v on Line 17, we make all edges (v, x) inactive using the previous for-loop (by setting $d(x) \leftarrow \min\{d(x), d(v) + w(v, x)\}$); thus, the statements still hold right after we unmark v. For (iii), we add x to Q before decreasing $d(x)$ on Lines 10 and 16; thus the statements still hold after Lines 10 and 16 are executed. □

Observe that the algorithm terminates only when Q is empty and every vertex is unmarked (every vertex is unmarked after the Bellman-Ford Phase). So, we have:

Corollary A.2. *When the algorithm terminates, all edges are inactive.*

For every inactive edge (v, x), $d(v) + w(v, x) - d(x) \geq 0$. Thus, d is the desired price function when the algorithm terminates.

A.2 Running time

The correctness analysis already implies that if there is a negative cycle then the algorithm never terminates (in particular, it is easy to check that if a graph has a negative-weight cycle, then for any labels d on the vertices there is always contains at least one inactive edge.) So, we assume that there is no negative cycle in this section. We analyze each iteration of the Dijkstra and Bellman-Ford phases. The initial iteration is referred to as iteration 0, the next iteration is iteration 1, and so on. To simplify notation, we let $\mathrm{dist}(v)$ and $\eta(v)$ refer to $\mathrm{dist}_G(v)$ and $\eta_G(v; s)$ for every vertex v. The key to the runtime analysis is the following lemma.

Lemma A.3. *After iteration i of the Dijkstra phase, $d(v) = \mathrm{dist}(s, v)$ for every v where $\eta(v) \leq i$.*

Proof. Iteration 0 of the Dijkstra phase is simply the standard Dijkstra's algorithm on the subgraphs induced by $E \setminus E^{neg}(G)$; thus, the statement holds for $i = 0$ by the standard analysis. We complete the proof by induction.

Fix any $i \geq 1$. Let v be any vertex such that $\eta(v) = i$. Let $P = (u_0 = s, u_1, u_2, \ldots, u_k = v)$ be the shortest sv-path with i negative-weight edges. Let u_j is the first vertex in P with $\eta(u) = i$; i.e., we define u_j to be the vertex in P such that $(u_{j-1}, u_j) \in E^{neg}(G)$ and the subpath P' of P between u_j and v contains no edge in $E^{neg}(G)$.

Consider when we just finished iteration $i - 1$ of the Dijkstra phase. By induction hypothesis, we can assume that $d(u_{j-1}) = \mathrm{dist}(s, u_{j-1})$ (since $\eta(u_{j-1}) \leq i - 1$). Observe that after the next execution of the Bellman-Ford phase (i.e. iteration $i - 1$),

$$d(u_j) \leq d(u_{j-1}) + w(u_{j-1}, u_j) \tag{11}$$

This is because, if this does not hold before the Bellman-Ford phase, then u_{j-1} must be marked (by Invariant A.1)[21], and $d(u_j)$ will be set to a value of at most $d(u_{j-1}) + w(u_{j-1}, u_j)$ on Line 16. From (11), we can conclude that after iteration i of the Bellman-Ford phase,

$$d(u_j) \leq d(u_{j-1}) + w(u_{j-1}, u_j) = \text{dist}(s, u_j). \tag{12}$$

Now consider when we finish iteration i of the Dijkstra phase. Since Q is empty and by Invariant A.1, all edges $(u_j, u_{j+1}), (u_{j+1}, u_{j+2}), \ldots, (u_{k-1}, u_k = v)$ are inactive (recall that they are all in $E \setminus E^{neg}(G)$ by definition). Thus,

$$d(v) \leq d(u_j) + w(u_j, u_{j+1}) + w(u_{j+1}, u_{j+2}) + \ldots + w(u_{k-1}, v) \overset{(12)}{=} \text{dist}(s, v)$$

It is easy to see that our algorithm always satisfies the invariant $d(v) \geq \text{dist}(s, v)$. Thus $d(v) = \text{dist}(s, v)$ as claimed. □

Invariant A.4 below is needed for proving Lemma A.5.

Invariant A.4. *The following always holds during the Dijkstra Phase: For any marked vertex v and any $x \in Q$, $d(v) \leq d(x)$.*

Proof. The invariant trivially holds when the Dijkstra phase begins (Line 4) because there is no marked vertex (the Bellman-Ford phase unmarks all vertices). The invariant can be affected when **(i)** a new vertex v is marked (Line 6), **(ii)** a new vertex is added to Q (Line 9), or **(iii)** $d(x)$ decreases for $x \in Q$ (Line 10).

For **(i)**, marking v on Line 6 does not violate the invariant because, by definition, v has the lowest $d(v)$ among vertices in Q. For **(ii)** (Line 9), note that $d(x) > d(v)$ by the if-condition on Line 8 (since $w(v, x) \geq 0$). Moreover, since $v \in Q$, we can conclude by induction that $d(u) \leq d(v)$ for every marked u; thus, $d(u) \leq d(v) < d(x)$ and so adding x to Q on Line 9 does not violate the invariant. Similarly, for **(iii)** (Line 10), the new value of $d(x)$ is such that $d(x) \geq d(v)$ (since $w(v, x) \geq 0$) and we can conclude by induction that $d(u) \leq d(v)$ for every marked u; so, decreasing $d(x)$ on Line 10 does not violate the invariant. □

Lemma A.5. *When a vertex x is added to Q on Line 9, it is not marked.*

Proof. Suppose for contradiction that there is a vertex x that is marked when it is added to Q on Line 9. This leads to the following contradiction: **(i)** $d(v) < d(x)$ (due to the if-condition on Line 8; note that $w(v, x) \geq 0$). **(ii)** $d(x) \leq d(v)$ (by Invariant A.4, where x is marked and $v \in Q$). □

Now we conclude the runtime analysis. Observe that if there are N vertices added to Q during the entire execution of the algorithm, then the algorithm's runtime is $O(N \log n)$: each vertex extracted from Q contributes $O(\log n)$ time in the Dijkstra phase and $O(1)$ time in the Bellman-Ford phase; here, we use the fact that every vertex has constant out-degree. It thus suffices to bound N, the number of vertices added to Q.

Lemma A.5 implies that every vertex is added to Q at most once in each iteration of the Dijkstra phase. It is clear from the pseudocode that every vertex is also added to Q at most once in each iteration of the Bellman-Ford phase. Lemma A.3 implies that a vertex v can be added to Q only in iterations $0, 1, \ldots, \eta(v)$ of Dijkstra and Bellman-Ford phases; thus, a vertex v can be added to Q at most $2\eta(v) + 2$ times, and the number of vertices added to Q overall is $N \leq 2\sum_{v \in V} \eta(v) + 2n$. This implies the claimed runtime of $O(\log(n) \cdot (\sum_{v \in V} \eta(v) + n))$.

[21] Note that Q is empty after the Dijkstra phase; so, for every active (x, y), vertex x must be marked after the Dijkstra phase.

Appendix B Proof of Lemma 3.2 (FixDAGEdges)

Algorithm 5: Algorithm for FixDAGEdges($G = (V, E, w), \mathcal{P} = \{V_1, V_2, \ldots\}$)

1 Relabel the sets $V_1, V_2, \ldots$ so that they are in topological order in G. That is, after relabeling, if $(u, v) \in E$, with $u \in V_i$ and $v \in V_j$, then $i \leq j$.
2 Define $\mu_j = \min\{w(u, v) \mid (u, v) \in E^{neg}(G), u \notin V_j, v \in V_j\}$; here, let $\min\{\emptyset\} = 0$.
`// ` μ_j `is min negative edge weight entering` V_j`, or 0 if no such edge exists.`
3 Define $M_1 \leftarrow \mu_1 = 0$.
4 **for** $j = 2$ *to* q **do** `// make edges into each` $V_2, \ldots, V_q$ `non-negative`
5 $M_j \leftarrow M_{j-1} + \mu_j$; `// Note:` $M_j = \sum_{k \leq j} \mu_k$
6 Define $\phi(v) \leftarrow M_j$ for every $v \in V_j$
7 **return** ϕ

See Algorithm 5 for pseudocode. Let us first consider the running time. Note that computing each μ_j requires time proportional to $O(1)$ + [the number of edges in E entering V_j]; since the V_j are disjoint, the total time to compute all of the μ_j is $O(m + n)$. Similarly, it is easy to check that the for loop in Line 4 only considers each vertex once, so the total runtime of the loop is $O(n)$.

To prove correctness, we need to show that $w_\phi(u, v) \geq 0$ for all $(u, v) \in E$. Say that $u \in V_i$ and $v \in V_j$ and note that because the algorithm labels the sets in topological order, we must have $i < j$. Moreover, by definition of μ_j we have $\mu_j \leq w(u, v)$. Thus, we have

$$w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v) = w(u, v) + M_i - M_j = w(u, v) - \sum_{k=i+1}^{j} \mu_k \geq w(u, v) - \mu_j \geq 0.$$

Appendix C Proof of Theorem 1.1 via a Black-Box Reduction

Recall from the preliminaries that W_G is defined as the absolute value of the most negative edge weight in the graph. Formally, $W_G := \max\{2, -\min_{e \in E}\{w(e)\}\}$. (We take the max with 2 so that $\log(W_G)$ is well defined.)

In this section, we show how to obtain the Las Vegas algorithm of Theorem 1.1. We start by showing how to use SPmain as a black box to obtain the following Monte Carlo algorithm.

Theorem C.1. *There exists a randomized algorithm* SPMonteCarlo(G_{in}, s_{in}) *that takes* $O(m \log^6(n) \log(W_{G_{in}}))$ *time for an* m*-edge input graph* G_{in} *and source* s_{in} *and behaves as follows:*

- *if* G_{in} *contains a negative-weight cycle, then the algorithm always returns an error message,*
- *if* G_{in} *contains* no *negative-weight cycle, then the algorithm returns a shortest path tree from* s_{in} *with high probability, and otherwise returns an error message.*

Note that the algorithm always outputs either an error message, or a (correct) shortest path tree.

Proof. Let $\mathcal{T}_{spmain}$ be the expected runtime of SPmain. The algorithm simply runs $C \log(n)$ versions of SPmain for some large constant C, where each version runs for $2\mathcal{T}_{spmain}$ time steps. If any of the versions returns distances, then SPmain guarantees that these distances are correct, so we simply return them. If all the versions never terminate, then the algorithm also returns an error. To bound the failure probability, note that if the graph contains a negative-weight cycle then returning an error message is what the algorithm is supposed to do; if it does not contain a negative-weight cycle,

then by Markov's inequality each version of SPmain terminates with probability at least 1/2, so the probability that none of the versions terminate is at most $1/2^{C\log(n)} = 1/n^C$, as desired. □

The rest of this section presents a Las Vegas algorithm SPLasVegas(G_{in}, s_{in}) that can return a negative cycle when one exists, and whose running time is $O(m\log^8(n))$ w.h.p when G_{in} satisfies the properties of Assumption 2.1. For this, we require the following definition

Definition C.2. Given any graph $G = (V, E, w)$ and any positive integer B, let $G^{+B} = (V, E, w^{+B})$ be the same as the graph G, except that for all $e \in E$ we have $w^{+B}(e) := w(e) + B$. (Note that unlike in G^B of Definition 2.3, here we add B to *all* edges, including the positive ones.)

FindThresh. The key subroutine of SPLasVegas is FindThresh, which computes the smallest integer $B \geq 0$ such that no negative cycles exist in G_{in}^{+B}. This is done using the algorithm FindThresh of the following lemma. The proof of FindThresh is deferred to Section C.3.

Lemma C.3. *Let H be an m-edge n-vertex graph with integer weights and let $s \in V(H)$. Then there is an algorithm,* FindThresh(H, s) *which outputs an integer $B \geq 0$ such that w.h.p.,*

- *If H has no negative cycles then $B = 0$, and*
- *If H has a negative cycle then $B > 0$, $H^{+(B-1)}$ contains a negative cycle, and H^{+B} does not.*

The running time of FindThresh(H, s) *is $O(m\log^6(n)\log^2(W_H))$.*

Definition C.4. If the high probability event of Lemma C.3 holds, we refer to B as the *correct* value.

C.1 The Las Vegas algorithm

We now present the algorithm SPLasVegas mentioned at the beginning of the section, using FindThresh as a black box. Recall our assumption that G_{in} satisfies $w(e) \geq -1$ for each $e \in E_{in}$. Pseudocode for SPLasVegas(G_{in}, s_{in}) can be found in Algorithm 6. For intuition when reading the pseudocode, note that we will show that w.h.p. none of the restart events occur.

Correctness of SPLasVegas(G_{in}, s_{in}) is trivial as the algorithm explicitly checks that its output is correct just prior to halting:

Lemma C.5. *If* SPLasVegas(G_{in}, s_{in}) *outputs a cycle, that cycle is negative in G_{in}. If* SPLasVegas(G_{in}, s_{in}) *outputs a tree, that tree is a shortest path tree from s_{in} in G_{in}.*

C.2 Running time

Since all edge weights in G_{in} are ≥ -1 (Assumption 2.1), we have $W_{G'} \leq n^3$. It follows that if there is no restart of the algorithm then by Theorem C.1 and Lemma C.3, its running time is $O(m\log^8(n))$ where the call to FindThresh(G', s_{in}) dominates. It remains to show that w.h.p., no restart occurs.

When $B = 0$, Lemma C.3 implies that w.h.p., G' and hence G_{in} has no negative cycles. By Theorem C.1, w.h.p. no restart occurs in Line 4.

Similarly, when $B > 0$, Lemma C.3 implies that w.h.p., $(G')^{+B}$ has no negative cycles, so by Theorem C.1, w.h.p. no restart occurs in Line 7.

To show that w.h.p., no restart occurs in Line 11, we need Corollary C.7 below which shows that $G_{\leq n}$ preserves all negative cycles from $(G')^{+(B-1)}$.

Algorithm 6: Algorithm SPLasVegas(G_{in}, s_{in})

1 Let G' be G_{in} with every edge weight multiplied by n^3
2 $B \leftarrow$ FindThresh(G', s_{in})
3 **if** $B = 0$ **then**
4 **if** SPMonteCarlo(G_{in}, s_{in}) *returns error* **then** restart SPLasVegas(G_{in}, s_{in});
5 Let T be the tree output by SPMonteCarlo(G_{in}, s_{in})
6 **return** T
7 **if** SPMonteCarlo($(G')^{+B}, s_{in}$) *returns error* **then** restart SPLasVegas(G_{in}, s_{in});
8 Let $\phi(v) = \text{dist}_{(G')^{+B}}(s_{in}, v)$ for all $v \in V$ be obtained from the tree output by SPMonteCarlo($(G')^{+B}, s_{in}$)
9 $G_{nonneg} \leftarrow ((G')^{+B})_{\phi}$ `// (Lemma 2.7) edge-weights in` G_{nonneg} `are non-negative`
10 Obtain the subgraph $G_{\leq n}$ of G_{nonneg} consisting of edges of weight at most n
11 **if** $G_{\leq n}$ *is acyclic* **then** restart SPLasVegas(G_{in}, s_{in});
12 Let C be an *arbitrary* cycle of $G_{\leq n}$
13 **if** C *is not negative in* G_{in} **then** restart SPLasVegas(G_{in}, s_{in});
14 **return** C

Lemma C.6. *If Line 8 is reached and* $(G')^{+(B-1)}$ *has a negative cycle then that cycle has weight less than* n *in* G_{nonneg}.

Proof. Edge-weights in $(G')^{+B}$ and $(G')^{+(B-1)}$ differ by at most 1, so if a cycle is negative in $(G)^{+(B-1)}$ it must have weight at most n in $(G')^{+B}$. The Lemma then follows from the fact that G_{nonneg} and $(G')^{+B}$ are equivalent, so by Lemma 2.7, the weight of any cycle is the same in G_{nonneg} and in $(G')^{+B}$. □

Corollary C.7. *If Line 8 is reached then every negative cycle in* $(G')^{+(B-1)}$ *is a cycle in* $G_{\leq n}$.

Proof. Consider any negative cycle C in $(G')^{+(B-1)}$. By the preceding lemma, we know that C has weight $\leq n$ in G_{nonneg}. We also know that all edge-weights of G_{nonneg} are non-negative. This implies the corollary. □

If a restart occurs in Line 11 then we simultaneously have that $B > 0$ and that $(G')^{+(B-1)}$ has no negative cycles by Corollary C.7, which means that the value of B is not correct; Thus, by Lemma C.3, w.h.p. no restart occurs in Line 11.

Finally, Corollary C.9 below implies that w.h.p., no restart occurs in Line 13.

Lemma C.8. *If* FindThresh(G', s_{in}) *outputs a correct value* $B > 0$ *then* $B \geq n^2$.

Proof. Since we are assuming that $B > 0$ is correct, it must be the case that G' has some negative-weight cycle C, but $(G')^{+B}$ does not. Since all edge-weights in G' are integer multiples of n^3, $w_{G'}(C) < 0$ implies that $w_{G'}(C) \leq -n^3$. We know that $w_{(G')^{+B}}(C) \geq 0$, but also by definition of $(G')^{+B}$, we have that $w_{(G')^{+B}}(C) = w_{G'}(C) + B|C| \leq w_{G'}(C) + Bn$. Combining these inequalities yields $Bn \geq n^3$, so $B \geq n^2$.

□

Corollary C.9. *If* FindThresh(G', s_{in}) *outputs a correct value* $B > 0$, *every cycle in* $G_{\leq n}$ *is a negative cycle in* G_{in}.

Algorithm 7: Algorithm for FindThresh(H, s)

$\ell \leftarrow 0$ and $r \leftarrow W_H$
while *TRUE* **do** `// Repeat loop until a value is returned`
  If $\ell = r$ **return** r
  $q \leftarrow \lfloor (\ell + r)/2 \rfloor$
  Execute SPMonteCarlo(H^{+q}, s).
  if SPMonteCarlo *returned an error* **then** `// w.h.p` H^{+q} `has a negative cycle`
    $\ell \leftarrow q + 1$
  else `//` H^{+q} `does not contain a negative cycle`
    $r \leftarrow q$

Proof. Consider a cycle C in $G_{\leq n}$. We have $w_{G_{\leq n}}(C) \leq n^2$ because every edge of $G_{\leq n}$ has weight $\leq n$. We now show that C is a negative-weight cycle in G', which also implies that it is negative in G_{in}. Observe that

$$w_{(G')^{+B}}(C) = w_{((G')^{+B})_\phi}(C) = w_{G_{nonneg}}(C) = w_{G_{\leq n}}(C),$$

where the first equality is because price functions do not change weights of cycles, the second equality is because $G_{nonneg} = ((G')^{+B})_\phi$ and the last equality is because $G_{\leq n}$ is a subgraph of G_{nonneg} with the same edge weights. It follows that

$$w_{G'}(C) = w_{(G')^{+B}}(C) - B|C| = w_{G_{\leq n}}(C) - B|C| \leq w_{G_{\leq n}}(C) - 2B \leq n^2 - 2n^2 < 0,$$

as desired. (In the inequalities, we use the facts that every cycle has at least two edges and $B \geq n^2$ by the previous lemmas.)

□

A union bound over the constant number of restart-lines now shows that w.h.p., no restart occurs. We conclude that w.h.p., SPLasVegas(G_{in}, s_{in}) runs in time $O(m \log^8(n))$.

C.3 Proof of Lemma C.3 – FindThresh

In this section, we describe algorithm FindThresh from Lemma C.3. Recall the definition of W_H from the beginning of Section C. The algorithm (Algorithm 7) is a simple binary search.

Runtime. By a standard binary-search argument, the loop is executed $O(\log(W_H))$ times. Each iteration is dominated by the call to SPMonteCarlo, which takes time $O(m \log^6(n) \log(W_H))$. The total runtime of FindThresh is thus $O(m \log^6(n) \log^2(W_H))$, as desired.

Correcntess. We now show that w.h.p. FindThresh(H, s) returns a correct value of B. The only probabilistic component of FindThresh is SPMonteCarlo(G, s). Recall the guarantees of SPMonteCarlo(G, s) from Theorem C.1: if G contains a negative-weight cycle then the algorithm *always* returns an error message; if G does not contain a negative-weight cycle then with high probability it does not. We say that an execution of SPMonteCarlo(G, s) is *bad* if G does not contain a negative-weight cycle, but the algorithm returns an error; otherwise we say the execution is *good*.

We know from Theorem C.1 that an exeuction of SPMonteCarlo is good w.h.p. Since FindThresh only executes SPMonteCarlo a total of $\log(W_H)$ times, and we will only run FindThresh on a graph with weights polynomial in n, a union bound implies that *every* execution of SPMonteCarlo is good.

We will show that as long as this holds, FindThresh returns a correct value of B. So from now on we assume that every execution of SPMonteCarlo is good.

We now show that if FindThresh(H, s) returns B, then G^{+B} does not contain a negative-weight cycle, as desired. To see this, note that the algorithm guarantees the invariant that H^{+r} contains no negative-weight cycles: this is true initially because we start with $r = W_H$, so by definition of W_H, all edges in H^{+r} are non-negative; it remains true because the algorithm only changes r in Line 9, where the else-condition guarantees that H^{+r} contains no negative-weight cycle. Since in the end the algorithm returns $B = r$, we know that G^{+B} does not contain a negative-weight cycle.

Next, we show that if H does contain a negative cycle, then FindThresh(H, s) returns $B = 0$. This is because H^{+q} will also never contain a negative-weight cycle (because q is always positive), so the algorithm will repeatedly execute the else-statement in Line 9, eventually returning $B = \ell = r = 0$.

Finally, we need to show that if H does contain a negative-weight cycle, then $H^{+(B-1)}$ contains a negative-weight cycle. Note that whenever the algorithm changes ℓ in Line 7, the preceding if-condition guarantees that $H^{+(\ell-1)}$ contains a negative cycle. The desired claim then follows from the fact that upon termination the algorithm returns $B = \ell = r$.

Appendix D Proof of Lemma 6.9

We start with some notation that sets up the main argument.

Notation:

- Throughout the proof we fix edge (u, v) and bound $\Pr[(u, v) \in E^{boundary}]$.
- Whenever the algorithm executes the While loop in Line 11, it picks some light vertex. We will assume it chooses the next vertex to process according to some arbitrary ordering $s_1, s_2, ...$ That is, there is some ordering $s_1, s_2, ...,$ of the vertices that are marked in-light or out-light (Line 7), such that every time the algorithm executes the while loop in Line 11, it does so by picking the first s_i in this ordering that has not yet been removed from G. Note that the ordering can be adversarially chosen; our analysis works with any ordering. Note also that once s_i is chosen, the direction of the ball (in-ball or out-ball) is uniquely determined because every vertex only receives one marking (see loop in Line 7).
- When the algorithm picks s_i, it grows the ball up to radius $R_i := R_{s_i}$ from s_i, where $R_i \sim Geo(p)$ is a random variable. (See Line 12). (Note: $R_1, ...R_n$ are the *only* source of randomness in this proof.)
- We now make a small technical change that does not affect the algorithm but simplifies the analysis. Let W_{max} be the heaviest edge weight in the graph and note that all shortest distances are $\leq (n-1)W_{max}$. Define $R_{max} = nW_{max}$. Note that if $R_i \geq R_{max}$ then $\text{Ball}^*_G(s_i, R_i) = \text{Ball}^*_G(s_i, R_{max})$. Thus, whenever $R_i > R_{max}$, we instead set $R_i = R_{max}$. This has zero effect on the behaviour of the algorithm or the set $E^{boundary}$ but will be convenient for the analysis because it ensures that we are now working with a *finite* probabilistic space: there are a finite number of variables R_i and each one is an integer in the bounded set $[1, R_{max}]$.
- Recall that $R_i \sim \text{Geo}(p)$ where $p = \min\{1, 40\log(n)/D\}$ (Line 12). We will use this variable p in our analysis.

- We define $G_i := G_i(R_1 ... R_{i-1})$ as follows. If s_i is already removed from G when it is considered by the while loop in Line 11 then we define G_i as the empty set. Else, we define G_i to be the graph after *every* $s_j \in \{s_1, s_2, \ldots s_{i-1}\}$ has either been processed by the while loop (i.e. a ball was grown from this s_j) or removed from G by the algorithm. Note that G_i is a random variable whose value depends on $R_1, ..., R_{i-1}$.

- Define $B_i := \text{Ball}^*_{G_i}(s_i, R_i)$, where the $*$ refers to the direction (in or out) uniquely determined by the choice of s_i. B_i is a random variable whose value depends on $R_1, ..., R_i$. Note that if G_i is empty then so is B_i.

- Define I_i to be the event that $u, v \notin B_1, ..., B_{i-1}$ AND $u \in B_i$ if the algorithm grows an out-ball $B_i = \text{Ball}^{\text{out}}_{\text{G}}(s_i, R_i)$; otherwise (if the algorithm grows an in-ball $B_i = \text{Ball}^{\text{in}}_{\text{G}}(s_i, R_i)$), I_i is defined to be the event that $u, v \notin B_1, ..., B_{i-1}$ AND $v \in B_i$. (I stands for "included".)

- Define X_i to be the event $v \notin B_i$ if the algorithm grows an out-ball $B_i = \text{Ball}^{\text{out}}_{\text{G}}(s_i, R_i)$; otherwise (if the algorithm grows an in-ball $B_i = \text{Ball}^{\text{in}}_{\text{G}}(s_i, R_i)$), X_i is defined to be the event $u \notin B_i$. (X stand for "excluded".)

Observation D.1.
- *I_i and X_i are independent from R_j for $j > i$; in other words, I_i and X_i only depend on $R_1, ..., R_{i-1}, R_i$.*
- *The events I_i are disjoint and thus $\sum \Pr[I_i] \leq 1$.*
- $\Pr[(u, v) \in E^{boundary}] = \sum_{i \geq 0} Pr_{R_1,...,R_i}[I_i \wedge X_i]$.

Proof. The first property is clear. For the second property, note that if I_i is true then $u \in B_i$, so u will be removed from the graph (Line 18) and not be present in any G_j, $j > i$, so all I_j, $j > i$ are false. For the third property, note that (u, v) is added to $E^{boundary}$ if and only if for some i, both u and v are present in G_i and (u, v) is in the boundary of ball B_i (Line 14), which is precisely captured by $I_i \wedge X_i$. □

Before proceeding with the proof, we will state a common assumption about probabilistic notation that greatly simplifies the presentation

Assumption D.2. Given probabilistic events A, B, we define $\Pr[A|B] = 0$ if $\Pr[B] = 0$.

This leads to significantly simpler notation because it allows us to write $\Pr[A \wedge B] = \Pr[B] \cdot \Pr[A|B]$ and separately bound $\Pr[A|B]$ without worrying about undefined conditional probabilities when $\Pr[B] = 0$. This notational simplification is commonly used to state the following law of total probability (which we will use later): If $\{B_1, B_2, \ldots\}$ is a finite or countably infinite partition of a sample space (i.e. it is a set of pairwise disjoint events whose union is the entire sample space) and A is another event, then

$$\Pr[A] = \sum_i \Pr[A \wedge B_i] = \sum_i \Pr[A \mid B_i] \Pr[B_i]. \tag{13}$$

(Without Assumption D.2 the sum on the right-hand side and its applications below must be only over i such that $\Pr[B_i] > 0$.) The assumption is justified because we are dealing with a finite probability space, so all zero-probability events combined still have zero probability mass.[22]

This notational assumption allows us to state the following lemma.

[22] By contrast, in an infinite space one would have to be more careful about such an assumption: it is possible for *every* individual instantiation of the random variables to have probability 0, but all infinity of them combined to have probability mass 1.

Lemma D.3. *For any i, $\Pr_{R_1,\ldots,R_i}[X_i|I_i] \le pw(u,v)$. (Recall that $p = \min\{1, 40\log(n)/D\}$.)*

We first show that Lemma D.3 completes the proof:

Proof of Lemma 6.9 given Lemma D.3. By Observation D.1 and (13) we have[23]

$$\begin{aligned}\Pr[(u,v) \in E^{boundary}] &= \sum_{i\ge 0} Pr_{R_1,\ldots,R_i}[I_i \wedge X_i] &&= \sum_{i\ge 0} \Pr[I_i]\cdot \Pr[X_i|I_i] \\ &\le pw(u,v)\cdot \sum_{i\ge 0}\Pr[I_i] &&\le pw(u,v) = O(\log(n)\cdot w(u,v)/D). \qquad \square\end{aligned}$$

To prove Lemma D.3 we use the following claim, which is the crux of the analysis. The claim allows us to fix the first $i-1$ random variables $R_1,\ldots,R_{i-1}$ and only treat R_i as random.

Claim D.4. *Fix* any *instantiations of the first $i-1$ random variables $R_1 = r_1, ..., R_{i-1} = r_{i-1}$ such that the resulting graph $G_i = G(r_1, ..., r_{i-1})$ contains both u and v. Then, $Pr_{R_i}[X_i|I_i] \le pw(u,v)$.*

Proof of Lemma D.3 assuming Claim D.4. Applying the law of total probability (13) over all possible instantiations $r_1, ..., r_{i-1}$ of $R_1, ..., R_{i-1}$ we have

$$\Pr_{R_1,\ldots,R_i}[X_i|I_i] = \sum_{r_1,\ldots,r_{i-1}} \Pr_{R_i}[X_i|I_i \wedge R_1 = r_1 \wedge \ldots \wedge R_{i-1} = r_{i-1}]\cdot \Pr[R_1 = r_1 \wedge \ldots \wedge R_{i-1} = r_{i-1}] \quad (14)$$

To bound the above, we consider two cases. If $R_1 = r_1, \ldots, R_{i-1} = r_{i-1}$ are such that $G_i = G(r_1, ..., r_{i-1})$ does not contain u or does not contain v then by definition I_i is false, so $Pr_{R_i}[X_i|I_i \wedge R_1 = r_1 \wedge \ldots \wedge R_{i-1} = r_{i-1}] = 0$. (Here we are using Assumption D.2.) The second case is that $G_i = G(r_1, ..., r_{i-1})$ contains both u and v, in which case we can apply Claim D.4. Thus, in either case, we have

$$\Pr_{R_i}[X_i|I_i \wedge R_1 = r_1 \wedge \ldots \wedge R_{i-1} = r_{i-1}] \le pw(u,v) \quad (15)$$

Combining (14) and (15) we have

$$\Pr_{R_1,\ldots,R_i}[X_i|I_i] \le pw(u,v) \sum_{r_1,\ldots,r_{i-1}} \Pr[R_1 = r_1 \wedge \ldots \wedge R_{i-1} = r_{i-1}] = pw(u,v). \qquad \square$$

All that remains is to prove Claim D.4.

Proof of Claim D.4. Recall that in this claim we have a fixed graph G_i that contains both u and v and that the only randomness now comes from $R_i \sim \text{Geo}(p)$. Note also that the LDD algorithm (Algorithm 3) is removing vertices and edges from the graph G over time, and that by when the algorithm reaches s_i, the remaining graph G that it is working on is by definition $G = G_i$.

For the rest of this proof, we will assume that the algorithm grows an out-ball $\text{Ball}^{\text{out}}_{\text{G}}(s_i, R_i)$; the case for an in-ball is exactly analogous. We thus have:

$$\begin{aligned}\Pr[X_i|I_i] &= \Pr[v \notin \text{Ball}^{\text{out}}_{\text{G}}(s_i, R_i)) \mid u \in \text{Ball}^{\text{out}}_{\text{G}}(s_i, R_i)) \\ &= \Pr[R_i < \text{dist}_G(s,v) \mid R_i \ge \text{dist}_G(s,u)] \\ &\le \Pr[R_i < \text{dist}_G(s,u) + w(u,v) | R_i \ge \text{dist}_G(s,u)]\end{aligned}$$

[23] Here we use Assumption D.2; otherwise, some of the sums must be over all i such that $\Pr[I_i] > 0$.

We can bound the last line by observing that the geometric distribution observes the memorylessness property: for any $j > i$ we have $Pr[R_i \leq i + j | R_i \geq i] \leq Pr[R_i \leq j]$. Combining this with the equation above we have

$$\Pr[X_i \mid I_i] \leq \Pr[R_i < \text{dist}_G(s,u) + w(u,v) \mid R_i \geq \text{dist}_G(s,u)] \leq \Pr[R_i \leq w(u,v)] \leq pw(u,v), \quad (16)$$

where the last inequality follows from a simple analysis of the geometric distribution: each coin is head with probability p, so $\Pr[R_i \leq w(u,v)]$ is the probability that one of the first $w(u,v)$ coins is a head, which by the union bound is at most $pw(u,v)$.

Remark D.5. *The second inequality of (16) used the memorylessness property of the geometric distribution, but we have to be a bit careful because recall that in our analysis, in order to maintain a finite probability space, we argued the algorithm sampling $R_i \sim \text{Geo}(p)$ is equivalent to the algorithm sampling $R_i \sim \text{Geo}(p)$ but then rounding down to $R_i = R_{max} = nW_{max}$ if $R_i > R_{max}$. This means that we can only apply the memorlyessness property in Equation (16) if $\text{dist}_G(s,u) + w(u,v) \leq R_{max}$. Fortunately, this is indeed the case because W_{max} is the maximum edge weight, so $w(u,v) \leq W_{max}$ and $\text{dist}_G(s,u) \leq (n-1)W_{max}$.* □

We have thus completed the proof of Lemma 6.9.