
Article:

Orlin, James B. and Végh, László A. ORCID: 0000-0003-1152-200X (2023) Directed shortest paths via approximate cost balancing. Journal of the ACM, 70 (1). ISSN 0004-5411

https://doi.org/10.1145/3565019

Directed Shortest Paths via Approximate Cost Balancing

James B. Orlin*
Sloan School of Management
Massachusetts Institute of Technology
jorlin@mit.edu

László A. Végh†
Department of Mathematics
London School of Economics and Political Science
l.vegh@lse.ac.uk

Abstract

We present an $O(nm)$ algorithm for all-pairs shortest paths computations in a directed graph with n nodes, m arcs, and nonnegative integer arc costs. This matches the complexity bound attained by Thorup [31] for the all-pairs problems in undirected graphs. The main insight is that shortest paths problems with approximately balanced directed cost functions can be solved similarly to the undirected case. The algorithm finds an approximately balanced reduced cost function in an $O(m\sqrt{n}\log n)$ preprocessing step. Using these reduced costs, every shortest path query can be solved in $O(m)$ time using an adaptation of Thorup's component hierarchy method. The balancing result can also be applied to the ℓ_∞-matrix balancing problem.

1 Introduction

Let $G = (N, A, c)$ be a directed graph with nonnegative arc costs, and $n = |N|$, $m = |A|$. In this paper, we consider the *single-source shortest paths (SSSP)* and the *all-pairs shortest paths (APSP)* problems. In the SSSP problem, the goal is to find the shortest paths from a given source node $s \in N$ to every other node; in the APSP problem, the goal is to determine the shortest path distances between every pair of nodes.

The seminal approach for SSSP is Dijkstra's 1959 algorithm [7]. An $O(m + n\log n)$ implementation of this algorithm using the Fibonacci heap data structure is due to Fredman and Tarjan [12]. Under the assumption that all of arc lengths are integral, Thorup [33] improved the running time for SSSP to $O(m + n\log\log n)$. Thorup's algorithm uses the word RAM model of computation, discussed in Section 2.1.

For the APSP problem, one can obtain $O(mn + n^2\log\log n)$ by running the SSSP algorithm of [33] n times. This has been the best previously known result for directed graphs. The main contribution of this paper is an $O(mn)$ algorithm for APSP in the word RAM model.

A breakthrough result by Thorup [31] obtained a linear time SSSP algorithm in the word RAM model for undirected graphs, implying $O(mn)$ for APSP. Our algorithm matches this bound for undirected graphs: it is based on an $O(m\sqrt{n}\log n)$ preprocessing algorithm that enables SSSP queries in $O(m)$ time.

*Supported by the ONR Grant N00014–17–1–2194.

†Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement ScaleOpt–757481).

Thorup [31] uses a *label setting* algorithm that is similar to Dijkstra's algorithm. Label setting algorithms maintain upper bounds $D(i)$ on the true shortest path distances $d(i)$ from the origin node s to each node i, and add nodes one-by-one to the set of permanent nodes S. At the time a node i is made permanent, $D(i) = d(i)$ holds—see [1, Chapter 4]. In Dijkstra's algorithm, $D(i) \leq D(j)$ is true for all nodes $j \notin S$ in the iteration when node i is made permanent. Relaxing this property is a key in further improvements

Let us define the *bottleneck costs* for nodes $i, j \in N$ as

$$b(i,j) := \min \left\{ \max_{e \in P} c(e) : P \text{ is an } i\text{–}j \text{ path in } G \right\} . \tag{1}$$

Dinitz [8] showed that label setting algorithms are guaranteed to find the shortest path distances if the following is true: whenever a node i is made permanent, $D(i) \leq D(j) + b(j,i)$ for all $j \notin S$. If an algorithm satisfies this weaker condition, then at termination it obtains distances satisfying $d(j) \leq d(i) + b(i,j)$ for all i and j, which in turn implies the shortest path optimality conditions: $d(j) \leq d(i) + c(i,j)$ for all $(i,j) \in A$—see Lemma 4.2.

Thorup's algorithm as well as the algorithm presented in this paper rely on this weaker guarantee of correctness. Both algorithms accomplish this by creating a *component hierarchy*—see Definition 2.2 for the variant used in this paper. Thorup developed this tool for SSSP on undirected networks; the hierarchy framework was subsequently extended to directed graphs in [15, 23, 24].

Our results also rely on the classical observation that shortest path computations are invariant under shifting the costs by a node potential. For a potential $\pi : N \to \mathbb{R}$, the *reduced cost* is defined as $c^\pi(u,v) := c(u,v) + \pi(u) - \pi(v)$. Computing shortest paths for c and any reduced cost c^π are equivalent: if P is a u–v path, then $c^\pi(P) = c(P) + \pi(u) - \pi(v)$.

We extend the use of reduced costs to the bottleneck costs.

$$b^\pi(i,j) := \min \left\{ \max_{e \in P} c^\pi(e) : P \text{ is an } i\text{–}j \text{ path in } G \right\} .$$

Our preprocessing step obtains a reduced cost function satisfying the following ξ-min-balancedness property for a constant $\xi > 1$.

Definition 1.1. A strongly connected directed graph $G = (N, A, c)$ with nonnegative arc costs $c \in \mathbb{R}^A_{\geq 0}$ is *ξ-min-balanced* for some $\xi \geq 1$ if for every arc $e \in A$, there exists a directed cycle $C \subseteq A$ with $e \in C$, such that $c(f) \leq \xi c(e)$ for all $f \in C$.

The importance of ξ-min-balancedness in the context of hierarchy-based algorithms arises from the near-symmetry of the bottleneck values $b(i,j)$. Lemma 2.1 below shows a graph is ξ-min-balanced if and only if $b(j,i) \leq \xi b(i,j)$ for all $i, j \in N$. Thorup's component hierarchy for undirected graphs implicitly relies on the fact that $b(i,j) = b(j,i)$ for all nodes i and j. For a ξ-balanced reduced cost function c^π, the values $b^\pi(i,j)$ and $b^\pi(j,i)$ are within a factor ξ. We can leverage this proximity to use component hierarchies essentially the same way as for undirected graphs in Thorup's original work [31], and achieve the same $O(m)$ complexity for an SSSP query, after an initial $O(m\sqrt{n}\log n)$ balancing algorithm.

This balancedness notion is closely related to the extensive literature on matrix balancing and gives an improvement for approximate ℓ_∞-balancing. We give an overview of the related literature in Section 1.1.2.

1.1 Related work

1.1.1 The SSSP and APSP problems

In the context of shortest path problems, the choice of the computational model is of high importance. The main choice is between the comparison-addition model with real costs, and variants of word RAM models with integer costs (see Section 2.1). In the comparison-addition model, additions and comparisons each take $O(1)$ time, regardless of the quantities involved. Other operations are not permitted except in so much as they can be simulated using additions and comparisons.

There is an important difference between these computational models in terms of lower bounds: sorting in the comparison-addition model requires $\Omega(n \log n)$, whereas no superlinear lower bound is known for integer sorting. Since Dijkstra's algorithm makes nodes permanent in a non-decreasing order of the shortest path distance $d(i)$ from s, the $O(m + n \log n)$ Fibonacci-heap implementation [11] is optimal for Dijkstra's algorithm in the comparison-addition model. Moreover, this is still the best known running time for SSSP in this model.

The best running time for APSP in the comparison-addition model is $O(mn + n^2 \log\log n)$ by Pettie [24]. This matches the best previous running time bounds for the integer RAM model, where the same bound was previously attained in [15, 33].

Pettie's [24] algorithm is based on the hierarchy framework. The same paper gives a lower bound that, at first glance, seems to imply that an $O(m)$ running time for the directed SSSP may not be achievable.

Let r be the ratio between the largest and the smallest nonzero arc cost. Pettie argued that if a shortest path algorithm for the directed SSSP is based on the hierarchy framework, then the running time of the algorithm is $\Omega(m + \min\{n \log r, n \log n\})$, even if the hierarchy is provided beforehand. This follows via an information-theoretic argument that is valid both in the comparison-addition as well as in the word RAM models. It uses the fact that any hierarchy approach must make node i permanent before j whenever $d(j) \geq d(i) + b(i, j)$. However, this interpretation of hierarchy frameworks for directed networks does not allow for replacing the costs by equivalent reduced costs, even though such a transformation may considerably change the bottleneck values $b(i, j)$. Therefore, his arguments do not contradict our development of an $O(m)$ time algorithm for the directed SSSP.

For undirected graphs, Pettie and Ramachandran [26] solve APSP in $O(mn \log \alpha(m, n))$ in the comparison-addition model, where $\alpha(m, n)$ is the inverse Ackermann function. After an $O(m + \min\{n \log n, n \log\log r\})$ time preprocessing step, every SSSP problem can be solved in time $O(m \log \alpha(m, n))$.

For dense graphs, that is, graphs with $m = \Omega(n^2)$ edges, the classical Floyd-Warshall algorithm [9, 34] yields $O(nm) = O(n^3)$. The first $o(n^3)$ algorithm was given by Fredman [11], in time $O(n^3/\log^{1/3} n)$. This was followed by a long series of improvements with better logarithmic factors, see references in [35]. In 2014, Williams [35] achieved a breakthrough with a randomized algorithm running in time $n^3/2^{\Omega(\sqrt{\log n})}$, by speeding up min-plus (tropical) matrix multiplication using tools from circuit complexity. A deterministic algorithm of the same asymptotic running time was obtained by Chan and Williams [4].

1.1.2 Approximate graph and matrix balancing

Our notion of ξ-min-balanced graphs is closely related to previous work on graph and matrix balancing. For $\xi = 1$, we simply say that G is min-balanced. A graph G is min-balanced if and only if for each proper subset S of nodes, the following is true: the minimum cost over arcs entering S is at equal to the minimum cost over arcs leaving S—see Lemma 2.1.

Schneider and Schneider [27] defined max-balanced graphs where for every subset S, the maximum cost over arcs entering S equals the maximum cost over arcs leaving S. For each $e \in E$, let $c'(e) = c_{\max} - c(e)$. Then G is max-balanced with respect to c' if and only if G is min-balanced with respect to c. For exact min/max-balancing, the running time $O(mn + n^2 \log n)$ by Young, Tarjan, and Orlin [36] is still the best known complexity bound. Relaxing the exactness condition, we give an $O\left(2^\rho(\rho+1)\sqrt{n}\log n\right)$ algorithm for ξ-min-balancing for any $\rho \in \mathbb{Z}_{\geq 0}$, $\xi = 1 + 1/2^{\rho-1}$.

Min-balancing is a generalization of the min-mean cycle problem: if C is a min-mean cycle, then any min-balanced residual cost function satisfies $c^\pi(e) \geq \mu$ for all $e \in E$ and $c^\pi(e) = \mu$ for $e \in C$ for some $\mu \in \mathbb{R}$. In fact, following [27], one can solve min-balancing as a sequence of min-mean cycle computations; see the discussion after Theorem 3.1. Karp's $O(mn)$ algorithm from 1978 [16] is still the best known strongly polynomial algorithm for min-mean cycle problem. Weakly polynomial algorithms that run in $O(m\sqrt{n}\log(nC))$ time were given by Orlin and Ahuja [18] and by McCormick [17]. The latter provides a scaling algorithm based on the same subroutine of Goldberg [14] that plays a key role in our balancing algorithm. The algorithms [17, 18] easily extend to finding an ε-approximate min-mean cycle in $O(m\sqrt{n}\log(n/\varepsilon))$ time. That is, finding a reduced cost c^π, a cycle C, and a value μ such that $c^\pi(e) \geq \mu$ for all $e \in E$ and $c^\pi(e) \leq (1+\varepsilon)\mu$ for all $e \in C$.

A restricted case of APSP is the problem of finding the shortest cycle in a network. Orlin and Sedeño-Noda [19] show how to solve the shortest cycle problem in $O(nm)$ time by solving a sequence of n (truncated) shortest path problems, each in $O(m)$ time. Their preprocessing algorithm was the solution of a minimum cycle mean problem in $O(nm)$ time. However—analogously to the approach in this paper—they could have relied instead on [17, 18] to find a 2-approximation of the minimum cycle mean in $O(m\sqrt{n}\log n)$ time.

We say that a graph is *weakly max-balanced* if for every node $v \in N$, the maximum cost over arcs entering v equals the maximum cost over arcs leaving v; that is, we require the property in the definition of max-balancing only for singleton sets $S = \{v\}$.

This notion corresponds to the well-studied *matrix balancing* problem: given a nonnegative matrix $M \in \mathbb{R}^{n\times n}$, and a parameter $p \geq 1$, find a positive diagonal matrix D such that in DMD^{-1}, the p-norm of the i-th column equals the p-norm of the i-th row. Given $G = (N, A, c)$, we let $M_{ij} = e^{c_{ij}}$ if $(i,j) \in A$ and $M_{ij} = 0$ otherwise. Then, balancing M in ∞-norm amounts to finding a weakly max-balanced reduced cost c^π.

Matrix balancing was introduced by Osborne [20] as a preconditioning step for eigenvalue computations. He also proposed a natural iterative algorithm for ℓ_2-norm balancing. Parlett and Reinsch [22] extended this algorithm to other norms. Schulman and Sinclair [28] showed that a natural variant of the Osborne–Parlett–Reinsch (OPR) algorithm finds an ε-approximately balanced solution in ℓ_∞ norm in time $O(n^3 \log(n\rho/\varepsilon))$, where ρ is the initial imbalance. Ostrovsky, Rabani, and Yousefi [21] give polynomial bounds for variants of the OPR algorithm for fixed finite p values, in particular, $O(m + n^2\varepsilon^{-2}\log w)$ for a weighted randomized variant, where w is the ratio of the sum of the entries over the minimum nonzero entry, and m is the number of nonzero entries. Recently, Altschuler and Parillo [3] showed an $\tilde{O}(m\varepsilon^{-2}\log w)$ bound for a simpler randomized variant of OPR. Cohen et al. [5] use second order optimization techniques to attain $\tilde{O}(m \log\kappa \log^2(nw/\varepsilon))$, where κ is the ratio between the maximum and minimum entries of the optimal rescaling matrix D; similar running times follow from [2]. The value κ may be exponentially large; the paper [5] also shows a $\tilde{O}(m^{1.5}\log(nw/\varepsilon))$ bound via interior point methods using fast Laplacian solvers.[1]

[1] In the quoted running times, $\tilde{O}(.)$ hides polylogarithmic factors. Various papers define ε-accuracy in different ways; here, we adapt the statements to ℓ_1-accuracy as in [3].

Our graph balancing problem corresponds to ℓ_∞ matrix balancing. Except for [28], the above works are applicable for finite ℓ_p norms only. Compared to [28], our approximate balancing algorithm has lower polynomial terms, but our running time depends linearly on $1/\varepsilon$ instead of a logarithmic dependence.[2]

1.2 Overview

The rest of the paper is structured as follows. Section 2 introduces notation and basic concepts, including the directed variant of component hierarchies used in this paper, and the comparison-addition and word RAM computational models. Section 3 is dedicated to the approximate min-balancing algorithm. The algorithm is developed in several steps: a key ingredient is a subroutine by Goldberg [14] that easily gives rise to a weakly polynomial algorithm. In order to achieve a strongly polynomial bound, we need a further preprocessing step to achieve an initial 'rough balancing'. An additional technical contribution is a new variant of the Union-Find data structure, called Union-Find-Increase. At the beginning of Section 3, we give a detailed overview of the overall algorithm and the various subsections.

In Section 4, we describe the shortest path algorithm for 3-min-balanced directed graphs. This is very similar to Thorup's original algorithm [31]. However, the setting is different, and we use a slightly different notion of the component hierarchy. For completeness, we include a concise description of the algorithm and the proof of correctness. Concluding remarks are given in the final Section 5.

2 Notation and preliminaries

For an integer k, we let $[k] = \{1, 2, \dots, k\}$. We let $\mathbb{Z}_{\geq 0}$ denote the nonnegative integers and let $\mathbb{Z}_{>0}$ denote the positive integers; similarly for $\mathbb{Q}_{\geq 0}$, $\mathbb{Q}_{>0}$, $\mathbb{R}_{\geq 0}$, and $\mathbb{R}_{>0}$. We let $\log x = \log_2 x$ refer to base 2 logarithm unless stated otherwise. For a vector $z \in \mathbb{R}^T$ and $S \subseteq T$, we let $z|_S \in \mathbb{R}^S$ denote the restriction of z to S.

Throughout, we let $G = (N, A, c)$ be a directed graph with nonnegative arc costs $c \in \mathbb{R}^A_{\geq 0}$. Let $m = |A|$ and let n denote the smallest integer power of 2 greater than or equal to $|N|$; we assume $n, m \geq 2$. This choice instead of $n = |N|$ will be convenient in the word RAM model. We use $C_{\min} = \min_{e \in E} c(e)$ and $C_{\max} = \max_{e \in E} c(e)$ to denote the smallest and largest values of the cost function. All graphs considered will be simple and loopless.

For a node $i \in N$, we let $A(i)$ denote the set of the outgoing arcs from i. For an arc set $F \subseteq A$, we let $N(F)$ denote the set of nodes incident to F. For a node set $X \subseteq N$, let $A[X]$ denote the set of arcs in A with both endpoints inside X.

For a node set $S \subseteq N$, let $\bar{S} = N \setminus S$ denote the complement of S. We let $(S, \bar{S}) \subseteq A$ denote the set of arcs directed from a node in S to a node in $\bar{S}$.

For a node set $Z \subseteq N$, we denote the graph obtained by contracting Z by $G/Z = (N', A', c')$. Here, $N' = (N \setminus Z) \cup \{z\}$; z represents the contracted node set. We include every arc $(i, j) \in A$ in A' with the same cost if $i, j \notin Z$. Arcs with both endpoints in Z are deleted. If $i \in Z$ or $j \in Z$, the corresponding endpoint is replaced by z. In case parallel arcs are created, we only keep one with the smallest cost. For a partition $\mathcal{P} = (P_1, P_2, \dots, P_k)$ of N, the contraction $G/\mathcal{P}$ denotes the graph obtained after contracting (in an arbitrary order) each of the sets P_i, $i \in k$ in G.

We will assume that $G = (N, A, c)$ is *strongly connected*; that is, a directed path exists between

[2] We note that, in contrast to the previous work, we consider min- rather than max-balancing. The exact min- and max-balancing problems can be transformed to each other by setting $c'(e) = c_{\max} - c(e)$; however, such a reduction does not preserve multiplicative approximation factors, and hence our result cannot be directly compared with [28]. Nevertheless, it seems that both algorithms can be adaptable to both the min and max settings. Such extensions are not included in this paper.

any two nodes. If the input is not strongly connected, then we preprocess the graph as follows. We find the strongly connected components in $O(n+m)$ time using Tarjan's algorithm [29]. We select a value M greater than the sum of all arc costs, pick one node in each strongly connected component, add a directed cycle on these nodes, and set the cost of these arcs to M. This results in a strongly connected graph $G' = (N, A', c')$ with $|A'| = O(m+n)$. Computing shortest paths in G' provides the shortest paths in G; if the shortest path distance between nodes i and j in G' is at least M, then j is not reachable from i in G.

Dijkstra's algorithm Dijkstra's algorithm [7] is the starting point of the fastest algorithms for SSSP and APSP. We now give a brief overview of the key steps. The algorithm maintains distance labels $D(i)$ for each node i that are upper bounds on $d(i)$, the shortest path distance from s. The algorithm adds nodes one-by-one to a *permanent node set* S with the property that $D(i) = d(i)$ for every $i \in S$. Further, for every $i \in N \setminus S$, $D(i)$ is the length of a shortest s–i path in the subgraph induced by the node set $S \cup \{i\}$.

These are initialized as $D(s) = 0$, $D(i) = \infty$ for $i \in N \setminus \{i\}$, and $S = \emptyset$. Every iteration adds a new node to S, selecting the node $i \in N \setminus S$ with the smallest label $D(i)$. Then, the outgoing arcs (i, j) are considered, and $D(j)$ is updated to $\min\{D(j), D(i) + c(i, j)\}$. The crucial property of the analysis is that this selection rule is correct, that is, for $i \in \arg\min\{D(j) : j \in N \setminus S\}$, we must have $D(i) = d(i)$.

Bottleneck costs in balanced graphs Our shortest path algorithm requires the input graph to be 3-min-balanced—see Definition 1.1. As shown next, the bottleneck costs are approximately balanced in such graphs.

Recall the definition of the bottleneck cost $b(i, j)$ in (1). We extend the definition to non-empty disjoint subsets $S, T \subsetneq N$ as $b(S, T) := \min\{b(i, j) : i \in S,\ j \in T\}$. Equivalently, $b(S, \bar{S}) = \min\{c(i, j) : i \in S, j \in \bar{S}\}$. By a *bottleneck* i–j path we mean an i–j path where the maximum arc cost is $b(i, j)$.

Lemma 2.1. *The following are equivalent.*

(1) G is ξ-min-balanced.

(2) For all proper subsets $\emptyset \neq S \subsetneq N$, $b(\bar{S}, S) \leq \xi b(S, \bar{S})$.

(3) For all $i \in N$ and $j \in N$, $b(j, i) \leq \xi b(i, j)$.

Proof. (1) $\Rightarrow$ (2). Suppose that G is ξ-min-balanced and $\emptyset \neq S \subsetneq N$. Choose $e \in \arg\min\{c(e) : e \in (S, \bar{S})\}$; thus, $c(e) = b(S, \bar{S})$. Let C be the bottleneck cycle containing e. Because C contains an arc f of $(\bar{S}, S)$, the following is true: $b(\bar{S}, S) \leq c(f) \leq \xi c(e) = \xi b(S, \bar{S})$.

(2) $\Rightarrow$ (3). Suppose that (2) is true. For given nodes i and j, let $S = \{k \in N : b(j, k) \leq \xi b(i, j)\}$. Clearly, $j \in S$. We show by contradiction that $i \in S$, and consequently, $b(j, i) \leq \xi b(i, j)$. Suppose that $i \in \bar{S}$. Let $e \in \arg\min\{c(e) : e \in (S, \bar{S})\}$, and suppose that $e = (h, \ell)$. Then $b(j, h) \leq \xi b(i, j)$ because $h \in S$; further, $c(h, \ell) = b(S, \bar{S}) \leq \xi b(\bar{S}, S) \leq \xi b(i, j)$ by (2) and the fact that the bottleneck path from i to j includes an arc of $(\bar{S}, S)$. Then $b(j, \ell) \leq \max\{b(j, h), c(h, \ell)\} \leq \xi b(i, j)$. But this implies that $\ell \in S$, a contradiction.

(3) $\Rightarrow$ (1). Suppose that (3) is true. Let $e = (j, i)$ be any arc of A; note that $b(j, i) \leq c(e)$. Let P be a path from i to j with arcs of length at most $b(i, j)$, and let $C = P \cup \{e\}$. Then C is a cycle, and $\max\{c(f) : f \in C\} \leq \max\{b(i, j), c(e)\} \leq \max\{\xi b(j, i), c(e)\} \leq \xi c(e)$. Thus, G is ξ-min-balanced. □

The component hierarchy We now introduce the concept of a component hierarchy. This is a variant of Thorup's [31] component hierarchy, adapted for approximately min-balanced directed graphs. The papers [15, 23, 24] also use component hierarchies for directed graphs. However, our notion exploits the ξ-min-balanced property, and will be more similar to the undirected concept [31] in that it does not impose orderings of the children of the vertices.

We use the standard terminology for a tree (V', E') rooted at $r \in V'$.

- For $v \in V' \setminus \{r\}$, the *parent* $p(v)$ of v is the first vertex after v on the unique path in the tree from v to r. All nodes in the path are called the *ancestors* of v.
- For $v \in V'$, $\text{children}(v) \subseteq V'$ is the set of nodes u such that $p(u) = v$.
- For every $v \in V'$, $\text{desc}(v) \subseteq V'$ is the set of nodes in the subtree rooted at v.
- For $u, v \in V'$, $\text{lca}(u, v)$ is the *least common ancestor* of u and v, i.e., the unique vertex on the u–v path in E' that is an ancestor of both u and v.

Definition 2.2. The tuple $(V \cup N, E, r, a)$ is called a *component hierarchy of* G for a strongly connected directed graph $G = (N, A, c)$ if

- $(V \cup N, E)$ is a tree with root $r \in V$, and N is the set of leaves.
- The vector $a : V \to \mathbb{Z}_{>0}$ is such that each $a(v)$ is an integer power of 2. For every $v \in V \setminus \{r\}$, $a(v) \leq a(p(v))/2$.
- For any $i, j \in N$ with $\text{lca}(i, j) = v$, we have $a(v) \leq b(i, j) \leq 3a(v)$; moreover, there exists an i–j path P inside $\text{desc}(v) \cap N$ such that $c(e) \leq 3a(v)$ for every arc $e \in P$.

2.1 Computational models

Our results use two different computational models. The approximate min-balancing algorithm in Section 3 can be implemented in the more restrictive comparison-addition model. However, the word RAM model is needed for constructing the component hierarchy: we require the operation of rounding down numbers to the nearest power of two in order to obtain $a(v)$ values that are powers of two. The shortest path algorithm in Section 4 uses integer arithmetic in two parts: *(1)* storing vertices and nodes in buckets, and *(2)* in the Split/FindMin data structure.

The comparison-addition model The input is a set of real numbers, and only addition and comparison operations are allowed; each takes constant time. Subtraction can be easily simulated with a constant overhead by representing numbers in the form $\alpha - \beta$. Multiplication by an integer N can be simulated by $O(\log N)$ additions. See [23, 26] for more details.

The algorithms in Section 3 also include division by a power of 2 in a restricted sense: for a value $\gamma = O(\log n)$, we require that all calculated values can be expressed as sums of quotients in the form $w = \sum_{i=0}^{\gamma} w_i/2^i$, where the w_i values can be obtained as a difference of two sums of input values. We can work with such numbers in the comparison-addition model by representing such a sum by ordered pairs $(w_1, d_1), \ldots, (w_i, d_i)$.

One can convert the sum of quotients into a single quotient with denominator $\leq 2^\gamma$ in time $O(\gamma)$. Moreover, additions, subtractions, and comparisons of sums of quotients can each be carried out in $O(\gamma + 1)$ steps. When the approximate balancing algorithms are used as preprocessing for the APSP, one can multiply the outputs by 2^γ at termination. Then the shortest path algorithms will determine the shortest path trees for the original problem as well.

The word RAM model We use the standard random access machine model, where every memory cell can store an integer of w bits. For convenience, we assume the word size is at least $\log(nC_{\max})$ so that each input and output number fits into a single word.

There is no universally accepted computational model for integer weights. We use the same model as in [15]; this is more restrictive than the one in [31], which also allows arbitrary multiplications. In our model, unit-time operations include comparison, addition, subtraction, bit shifts by an arbitrary number of positions, and bitwise boolean operations.

We do not allow multiplications and divisions in general. However, the bit shift operations enable multiplications by integer powers of 2 in $O(1)$ time. Due to the assumption that n is a power of 2, multiplying by a monomial term such as bn^k can be done in $O(1)$ time if $b, k = O(1)$. We will use divisions by powers of 2. These operations can be simulated with constant overhead, maintaining a representation $a/2^b$ of the occurring numbers. Throughout, we maintain numbers in such representation with $b = O(\log n)$.

We highlight the only two operations involving integer arithmetics that are used for constructing the component hierarchy in Section 3.3 and for the bucketing operations in Section 4.3.

(i) Given $r \in \mathbb{Z}_{\geq 0}$, compute the largest integer power of 2 smaller or equal than r; we denote this by $\lfloor r \rfloor_2$.

(ii) Given $r \in \mathbb{Z}_{\geq 0}$, and $b \in \mathbb{Z}_{\geq 0}$, compute $\lfloor r/2^b \rfloor$.

All other integer operations are only needed for Split/FindMin; we discuss this in more detail in Section 4.5.

Any running time bound obtained in the comparison-addition model is directly applicable to the word RAM model. Bounds in the comparison-addition model can be worse than in the word RAM model. In particular, restricted divisions in the comparison-addition model require $O(\gamma)$ time for numbers in the sum of quotients form $w = \sum_{i=0}^{\gamma} w_i/2^i$, in contrast to $O(1)$ in the word RAM model where bit shift operations are permitted.

3 An algorithm for approximate min-balancing

This section is dedicated to the proof of the following theorem. The algorithm asserted in the theorem is Algorithm 5 in Section 3.3.

Theorem 3.1. *Assume we are given a strongly connected directed graph $G = (N, A, c)$ with arc costs $c \in \mathbb{R}^A_{\geq 0}$, and a parameter $\rho \in \mathbb{Z}_{\geq 0}$; let $\xi := 1 + 1/2^{\rho-1}$.*

(a) There exists an $O\left(2^\rho(\rho+1) \cdot m\sqrt{n}\log n\right)$ time algorithm in the comparison-addition model that finds a potential $\pi \in \mathbb{R}^N$ such that c^π is ξ-min-balanced.

(b) For $\rho = 0$ and $\xi = 3$ and an integer input $c \in \mathbb{Z}^A_{\geq 0}$, we can obtain a potential $\pi \in \mathbb{Q}^N$ and a component hierarchy of (N, A, c^π) in time $O\left(m\sqrt{n}\log n\right)$ in the word RAM model. Further, all $\pi(v)$ values are integer multiples of $1/(4n^3)$.

It is instructive to start the overview from exact min-balancing, that is, $\xi = 1$, even though our algorithm is not applicable to this case. For $\xi = 1$, the exact max-balancing algorithms [27, 36] can be used (by negating the costs). A simple and natural algorithm (see [27]) is based on the iterative application of min-mean cycle finding. First, find all arcs that are in a min-mean cycle in the graph; let $\mu \geq 0$ denote the minimum cycle mean value, and F the set of all arcs in such cycles. Every arc $e \in F$ must have $c^\pi(e) = \mu$ if c^π is a min-balanced reduced cost function.

It is easy to see that the min-cycle mean algorithm produces a potential π such that $c^\pi(e) \geq \mu$ for all $e \in E$, and $c^\pi(e) = \mu$ for all $e \in F$. We can then contract all strongly connected components of F, and recurse on the contracted graph, by repeatedly modifying the potential π and contracting the components of min-mean cycles.

The current best running times are $O(mn + n^2 \log n)$ for min-balancing [36] and $O(mn)$ for minimum-mean cycle computation [16]. Both these running times are substantially higher than the overall running time in Theorem 3.1.

We can thus only afford to approximately compute min-mean cycles. This can be achieved faster using a subroutine in Goldberg's paper [14], originally developed for a weakly polynomial algorithm for negative cycle detection. There are some technical differences from [14]; we present the detailed description of the subroutine and the proof of correctness in Section 3.5.

The input to the subroutine Small-Cycles is a strongly connected directed graph with minimum arc cost L and a parameter $D > 0$. In time $O(m\sqrt{n})$, it finds a reduced cost c^π and strongly connected components of arcs with reduced cost in the range of $[L, L + 2D]$. At the same time, the reduced cost of every arc between different components is at least $L + D$.

If the input graph has positive arc costs, the iterative application of this subroutine yields a simple weakly polynomial algorithm with running time $O\left(2^\rho(\rho + 1) \cdot m\sqrt{n} \log\left(nC_{\max}/C_{\min}\right)\right)$, as described in Section 3.1.

In order to turn this into a strongly polynomial algorithm, in Section 3.2 we start by a preprocessing algorithm that finds a $14n^2$-min-balanced reduced cost. An important step in this algorithm is to determine the *balance values* $\beta(e)$ for all arcs $e \in E$; this is defined as the smallest value b such that G contains a cycle C with $e \in C$ and $c(f) \leq b$ for all $f \in C$. These balance values can be efficiently found using a simple recursive framework.

The strongly polynomial algorithm in Section 3.3 requires the input cost function to be $14n^2$-min-balanced. How can we benefit from this 'rough' balance of the input? The weakly polynomial algorithm consists of $O\left(2^\rho(\rho + 1) \cdot \log(nC_{\max}/C_{\min})\right)$ calls to Small-Cycles. If each call uses the entire arc set in the current contracted graph, we obtain a total running time $O\left(2^\rho(\rho + 1) \cdot m\sqrt{n} \log\left(nC_{\max}/C_{\min}\right)\right)$ as above. However, when running Small-Cycles with parameter L, it is possible to restrict attention to arcs e with $c(e) \leq 2nL$. We refer to such arcs as *active*. If the input is assumed to be a $14n^2$-min-balanced cost-function, then each arc is active for $O\left(2^\rho \log n\right)$ calls of Small-Cycles prior to being contracted. Thus each arc contributes $O\left(2^\rho(\rho + 1)\sqrt{n} \log n\right)$ to the total running time.

In the weakly polynomial algorithm, the parameter L giving a lower bound on the minimum reduced cost of non-contracted arcs increases by a factor at most $1 + 1/2^\rho$ in each iteration. To avoid the dependence on $C_{\max}/C_{\min}$ in the strongly polynomial algorithm, this value may sometimes 'jump' by large amounts in iterations with no active arcs.

An important technical detail is the maintenance of the reduced costs. In every iteration, we only directly maintain $c^\pi(e)$ for the active arcs. Querying the reduced cost of a newly activated arc is nontrivial, since one or both of its endpoints may have been part of one or more contracted cycles, each of which corresponds to a node in the contracted graph. To compute the potential of an original node i, we need to add to the potential of node i the potentials of every contracted node j that contains node i. We develop a new extension of the Union-Find data structure, called Union-Find-Increase by incorporating a new 'increase' operation. This is described in Section 3.4.

Contractions and preprocessing We use contractions several times. Whenever a set S is contracted, we let s be the contracted node, and set the potential $\pi_s = 0$. For each arc with one endpoint in S, we keep the same reduced cost as immediately before the contraction.

On multiple occasions we need the subroutine Strongly-connected(N, A) that implements

Tarjan's algorithm [29] to find the strongly connected components of the directed graph (N, A) in time $O(|N|+|A|)$. The output includes the strongly connected components $(N_1, A_1), (N_2, A_2), \ldots, (N_k, A_k)$ in the topological order, namely, for every arc $(u, v) \in A$ such that $u \in N_i$, $v \in N_j$, it must hold that $i \leq j$.

In Theorem 3.1, the input is a nonnegative cost function. For our algorithm, it is more convenient to assume a strictly positive cost function. We now show how the nonnegative case can be reduced to the strictly positive case by a simple $O(m)$ time preprocessing. Recall the notation $C_{\min} = \min_{e \in E} c(e)$ and $C_{\max} = \max_{e \in E} c(e)$.

We first call STRONGLY-CONNECTED(N, A_0) on the subgraph of 0-cost arcs A_0. We contract all strongly connected components, and keep the notation $G = (N, A)$ for the contracted graph, where the output of the subroutine gives a topological ordering $N = \{v_1, v_2, \ldots, v_n\}$ such that for every 0-cost arc (v_i, v_j), we must have $i < j$. We let C' denote the smallest nonzero arc cost, and set $\pi(v_i) = -iC'/n$. Then, it is easy to see that $c^\pi(e) \geq C'/n$ for every $e \in A$.

We then replace the cost function c by nc^π, after which we obtain a cost function c' with $C' \leq c'(e) \leq n(C' + C_{\max})$ for every $e \in A$. This preprocessing algorithm can be implemented in $O(m \log n)$ time in the comparison-addition model.

3.1 A simple weakly polynomial variant

The following subroutine is a variant of REFINE in Goldberg's paper [14].

Algorithm 1 SMALL-CYCLES

Input: A directed graph $G = (N, A, c)$ with a cost function $c \in \mathbb{R}^A$, and $L \in \mathbb{R}$, $D \in \mathbb{R}_{>0}$ such that $c(e) \geq L$ for all $e \in A$.

Output: A partition $\mathcal{P} = (P_1, P_2, \ldots, P_k)$ of the node set N and a potential vector $\pi \in \mathbb{R}^N$ such that

(i) For every $i \in [k]$, $c^\pi(e) \geq L$ for every $e \in A[P_i]$, and P_i is strongly connected in the subgraph of arcs $\{e \in A[P_i] : L \leq c^\pi(e) \leq L + 2D\}$;

(ii) $c^\pi(e) \geq L + D$ for all $e \in A \setminus \left(\cup_{i \in [k]} A[P_i]\right)$;

(iii) $-|N|D \leq \pi(v) \leq 0$, and $\pi(v)$ is an integer multiple of D for all $v \in N$.

Lemma 3.2. *The subroutine* SMALL-CYCLES(L, D, N, A, c) *can be implemented in* $O(|A|\sqrt{|N|}\log|N|)$ *time in the comparison-addition model.*

The proof adapts the argument in [14]; it is deferred to Section 3.5. We now summarize the weakly polynomial algorithm SIMPLE-MIN-BALANCE (Algorithm 2). We initialize $L_1 = C_{\min}$ and $D_1 = L_1/2^\rho$. Every iteration calls SMALL-CYCLES for the current values of L_t and D_t. In Step 5, we contract each subset (some or all of which may be singletons) in the partition $\mathcal{P}_t$ returned by the subroutine, and iterate with the returned reduced cost, setting the new value $L_{t+1} = L_t + D_t$. We update D_{t+1} to $L_{t+1}/2^\rho$ whenever t is an integer multiple of 2^ρ; otherwise, we keep $D_{t+1} = D_t$. Thus, the value of L_t doubles in every 2^ρ iterations.

We let $(\hat{N}_t, \hat{A}_t)$ denote the contracted graph at iteration t. The algorithm terminates when $\hat{N}_t$ has a single node only, at iteration $t = T$.

Uncontraction In the final step of the algorithm, we uncontract all sets in the reverse order of contractions. We start by setting $\pi = p_T$. Assume a set S was contracted to a node s in iteration t, and we have uncontracted all sets from iterations $t+1, \ldots, T$. When uncontracting S, for every $v \in S$ we set $\pi(v) = p_t(v) + \pi(s)$, i.e., the potential right before contraction, plus the potential

Algorithm 2 SIMPLE-MIN-BALANCE

Input: A strongly connected directed graph $G = (N, A, c)$ with $c \in \mathbb{R}^A_{>0}$, parameters $\rho \in \mathbb{Z}_{\geq 0}$ and $\xi = 1 + 1/2^{\rho-1}$.

Output: A potential $\pi \in \mathbb{R}^N$ such that c^π is ξ-min-balanced.

$(\hat{N}_1, \hat{A}_1, \hat{c}_1) \leftarrow (N, A, c)$; $t \leftarrow 1$;
$L_1 \leftarrow \min_{e \in A} c(e)$; $D_1 \leftarrow L_1/2^\rho$
while $|\hat{N}_t| > 1$ **do**
    $(\mathcal{P}_t, p_t) \leftarrow$ SMALL-CYCLES$(L_t, D_t, \hat{N}_t, \hat{A}_t, \hat{c}_t)$;
    $(\hat{N}_{t+1}, \hat{A}_{t+1}, \hat{c}_{t+1}) \leftarrow (\hat{N}_t, \hat{A}_t, \hat{c}_t^{p_t})/\mathcal{P}_t$;
    $L_{t+1} \leftarrow L_t + D_t$;
    if t is an integer multiple of 2^ρ **then** $D_{t+1} \leftarrow L_{t+1}/2^\rho$;
    else $D_{t+1} \leftarrow D_t$;
    $t \leftarrow t + 1$;
Uncontract $(\hat{N}_t, \hat{A}_t, \hat{c}_t)$, and compute the overall potential $\pi \in \mathbb{R}^N$;
return π.

of s accumulated during the uncontraction steps. This takes time $O(n')$ where n' is the total size of all sets contracted during the algorithm; it is easy to bound $n' \leq 2n$. Thus, the total time for uncontraction is $O(n)$.

Lemma 3.3. *Algorithm 2 finds a ξ-min-balanced cost function in time $O\left(2^\rho(\rho+1) \cdot m\sqrt{n}\log(nC_{\max}/C_{\min})\right)$ in the comparison-addition model.*

Proof. At initialization, $L_1 = C_{\min}$, and L_t increases by a factor 2 in every 2^ρ iterations. At every iteration, we can extend the cost function $\hat{c}_t$ to the original arc set A: for an arc e contracted in an earlier iteration $\tau < t$, we let $\hat{c}_t(e) = \hat{c}_\tau(e)$ represent the value right before the contraction. It is easy to see that this extension of $\hat{c}_t$ to A gives a valid reduced cost of c.

Throughout, we have that $L_t \leq \hat{c}_t(e)$ for all $e \in \hat{A}_t$, and $\hat{c}_t(e) \geq 0$ for all contracted arcs. Thus, for any cycle $C \subseteq A$ that contains some non-contracted arcs in $\hat{A}_t$, $2L_t \leq \hat{c}_t(C) = c(C) \leq nC_{\max}$ holds. Consequently, $L_t \leq nC_{\max}/2$ throughout, implying a bound $O(2^\rho \log(nC_{\max}/C_{\min}))$ on the number of iterations.

As explained above, the final uncontraction and computing π can be implemented in $O(n)$ time. To show that the final c^π is ξ-min-balanced, consider an arc $e \in A$, and assume it was contracted in iteration t, that is, $e \in A[P_j]$ for a component P_j of the partition $\mathcal{P}_t$. In particular, $c^\pi(e) = c^{p_t}(e) \geq L_t$. The set P_j is strongly connected in the subgraph of arcs of reduced cost $\leq L_t + 2D_t \leq \xi L_t$. Thus, at iteration t, P_j contains a cycle C with $e \in C$ such that $\hat{c}_t(f) \leq \xi L_t$ for all $f \in C$. This cycle may contain nodes that were contracted during previous iterations. Every component previously contracted contains a strongly connected subgraph of arcs with costs $L_{t-1} + D_{t-1} \leq \xi L_{t-1}$, noting that the arc costs do not change anymore after contraction. Thus, when uncontracting a node, we can extend C to a cycle of arc costs $\leq \xi L_t$. Hence, we can obtain a cycle C' in the original graph G with $e \in C'$ and $c^\pi(f) \leq \xi L_t \leq \xi c^\pi(e)$ for all $f \in C'$.

The algorithm only performs addition and comparison operations, and divisions by 2^ρ. Divisions only happen when setting $D_{t+1} = L_{t+1}/2^\rho$. At such iterations, we have $D_{t+1} = 2^k C_{\min}$, where $k = t/2^\rho$ is an integer. As remarked in Section 2.1, we can implement every step in $O(\rho + 1)$ time. □

3.2 A quick algorithm for rough balancing

In this section, we present the subroutine ROUGH-BALANCE(N, A, c), which finds a potential $\pi \in \mathbb{R}^N$ such that c^π is $14n^2$-min-balanced. As mentioned previously, this will be an important preprocessing step for the strongly polynomial algorithm in Section 3.3. The running time can be stated as follows. Here, $\alpha(m, n)$ is the inverse Ackermann function.

Lemma 3.4. *Let $G = (N, A, c)$ be a strongly connected directed graph with $c \in \mathbb{R}^N_{>0}$. Then, in time $O(m\alpha(m, n) \log n)$, we can find a potential $\pi \in \mathbb{R}^N$ such that c^π is $14n^2$-min-balanced, where $n = |N|$ and $m = |A|$. The algorithm can be implemented in the comparison-addition model, and every $c^\pi(e)$ value will be an integer multiple of $4n^2$.*

Given $G = (N, A, c)$ with $c \in \mathbb{R}^A_{\geq 0}$, and $r > 0$, we let $G[\leq r]$ denote the subgraph of G formed by the arcs $e \in A$ with $c(e) \leq r$. For every $e \in A$, we define $\beta(e) \in \mathbb{R}_{>0}$ as the smallest value r such that $G[\leq r]$ contains a directed cycle C with $e \in C$. We call $\beta(e)$ the *balance value* of e. Clearly, G is ξ-min-balanced if and only if $\beta(e) \leq \xi c(e)$ for every $e \in A$.

The algorithm proceeds in two stages. Section 3.2.1 presents FIND-BALANCE(N, A, c), which determines the balance value $\beta(e)$ for every arc in $e \in A$. The main algorithm ROUGH-BALANCE(N, A, c) follows in Section 3.2.2.

3.2.1 Determining the balance values

Algorithm 3 presents the recursive subroutine FIND-BALANCE(N, A, c). Let $c[1] < c[2] < \ldots < c[K]$ denote the set of different arc cost values. If $K = 1$, i.e., all arc costs are the same, then we return $\beta(e) = c(e) = c[1]$ for every arc. Otherwise, we let r and r' denote the two consecutive values in the middle. We identify the strongly connected components $(N_1, A_1), (N_2, A_2), \ldots, (N_k, A_k)$ of $G[\leq r]$, and recursively determine the $\beta(e)$ values for $e \in A_i$ by calling the algorithm for each nonsingleton component (N_i, A_i). For all arcs $e \in A[N_i] \setminus A_i$, we set $\beta(e) = c(e)$.

We then contract all components (N_i, A_i) to singletons to obtain $\hat{G} = (\hat{N}, \hat{A}, \hat{c})$. We increase each arc cost $\hat{c}(e)$ in this graph to $\max\{\hat{c}(e), r'\}$, make another recursive call to the algorithm on $\hat{G}$, and use the obtained balanced values for the pre-images of the contracted arcs.

Lemma 3.5. *Algorithm 3 correctly computes the balance values in G in time $O(m \log n)$.*

Proof. For any pair of nodes v and w in N_i, $b(w, v) \leq r$. If $e = (v, w) \in A$ and $c(e) > r$, then $\beta(e) = c(e)$ is correctly determined. If $c(e) \leq r$, then $e \in A_i$, and $\beta(e) \leq r$ can be found recursively by finding the balance values in (N_i, A_i).

Suppose instead that $e = (v, w) \in E$, where $v \in N_i$ and $w \in N_j$ for $j \neq i$. Then $\beta(e) \geq r'$, and we can replace $c(e)$ by $c'(e) = \max\{c(e), r'\}$ without changing $\beta(e)$. In addition, contracting the strongly connected components (N_i, A_i) does not affect $\beta(e)$. Thus, the algorithm correctly computes the balance numbers.

We now turn to the running time. The initial time to sort the arcs is $O(m \log m) = O(m \log n)$. Let $T(m', K')$ be the running time for the algorithm if the input has m' sorted arcs with K' different costs. The algorithm finds the median value for the arc costs and partitions the graph into subgraphs with m^* arcs and $m' - m^*$ arcs respectively, where $m^* \in [1, m' - 1]$. This takes $O(m')$ time, and leads to the following recursion:

$$T(m', K') \leq O(m') + \max_{m^* \in [1, m']} \{T(m^*, \lfloor K/2 \rfloor) + T(m - m^*, \lfloor K/2 \rfloor + 1)\} .$$

We conclude that $T(m', K') = O(m' \log(K' + 1))$ because the number of different arc values is halved in every recursive call. Hence, every arc can participate in at most $\lfloor \log(K' + 1) \rfloor$ recursive calls. □

Algorithm 3 FIND-BALANCE

Input: A strongly connected directed graph $G = (N, A, c)$ with $c \in \mathbb{Q}^A_{>0}$.
Output: A function $\beta : A \to \mathbb{Q}$ giving the balance value $\beta(e)$ of each arc $e \in A$.

Let $c[1] < c[2] < \ldots < c[K]$ denote the set of arc cost values $c(e)$;
if $K = 1$ **then** $\beta(e) \leftarrow c(e)$ for all $e \in A$;
else
  $r \leftarrow c\left[\left\lfloor \frac{K}{2} \right\rfloor\right]$; $r' \leftarrow c\left[\left\lfloor \frac{K}{2} \right\rfloor + 1\right]$;
  $\{(N_1, A_1), (N_2, A_2), \ldots, (N_k, A_k)\} \leftarrow$ STRONGLY-CONNECTED$(G[\leq r])$;
  for $i = 1, \ldots, k$ **do**
    if $|N_i| > 1$ **then**
      for $e \in A[N_i] \setminus A_i$ **do** $\beta(e) \leftarrow c(e)$;
      $\beta_i \leftarrow$FIND-BALANCE$(N_i, A_i, c|_{A_i})$;
      for $e \in A_i$ **do** $\beta(e) \leftarrow \beta_i(e)$;
  obtain $\hat{G} = (\hat{N}, \hat{A}, \hat{c})$ by contracting every set N_j, $j \in [k]$;
  for $e \in \hat{A}$ **do** $\hat{c}(e) \leftarrow \max\{\hat{c}(e), r'\}$;
  $\hat{\beta} \leftarrow$ FIND-BALANCE$(\hat{G})$;
  for $e \in A \setminus \left(\bigcup_{j=1}^{s} A[N_j]\right)$ **do** $\beta(e) \leftarrow \hat{\beta}(\hat{e})$, where $\hat{e}$ is the contracted image of e ;
return β.

3.2.2 Constructing the potential

We now describe the algorithm ROUGH-BALANCE(N, A, c). We first compute the balance values $\beta(e)$ by running FIND-BALANCE(N, A, c). We define

$$\eta(e) := \max\left\{c(e), \frac{\beta(e)}{2n}\right\} .$$

For $r \geq 0$, we let $G[\eta \leq r]$ denote the subgraph of G formed by the arcs $e \in A$ with $\eta(e) \leq r$. We say that $e \in A$ is *active* with respect to the value r if $\eta(e) \leq r$ but e is not contained in any strongly connected component of $G[\eta \leq r]$.

The ROUGH-BALANCE subroutine is shown in Algorithm 4. A value $r \geq 0$ is maintained, and the graph $\hat{G}$ denotes the contraction of the strongly connected components of $G[\eta \leq r]$; we use the η values also in $\hat{G}$ that refer to the pre-image of the arc in G. At the beginning of the first iteration, r is set as the minimum $\eta(e)$ value in G; in later iterations, we increase r by a factor $2n$, or to the minimum of the $\eta(e)$ values in the current $\hat{G}$. Each iteration computes a topological ordering of the active arcs w.r.t. r. Then, the potential π_{v_i} of the i-th node v_i in the order is decreased by $ri/(2n)$. We terminate once $\hat{G}$ becomes a single node, i.e., $G[\eta \leq r]$ is strongly connected.

We handle contractions as in Section 3.1. That is, the final reduced cost of an arc e is equal to its reduced cost immediately before its endpoints got contracted into the same node. At the end, we uncontract and obtain the overall potential in the original graph in time $O(n)$.

We now turn to the proof of Lemma 3.4. As the first step, we bound the reduced costs obtained in the algorithm. The reduced costs are defined in the contracted graph, but can be naturally mapped back to the input graph G.

Lemma 3.6. *Consider the potentials at the end of any iteration of Algorithm 4, and let $c^\pi(e)$ denote the reduced cost of any arc $e \in A$. Then, $|c^\pi(e) - c(e)| \leq 2r/3$. If e is an active arc in the current iteration, then $c^\pi(e) \geq c(e) + r/(6n)$.*

Algorithm 4 ROUGH-BALANCE

Input: A strongly connected directed graph $G = (N, A, c)$ with $c \in \mathbb{R}^A_{>0}$.
Output: A potential $\pi : V \to \mathbb{R}$ such that c^π is $14n^2$-min-balanced.

obtain the balance values $\beta(e)$ by calling FIND-BALANCE(G) ;
for $e \in A$ **do** $\eta(e) \leftarrow \max\left\{c(e), \frac{\beta(e)}{2n}\right\}$;
$r \leftarrow 0$; $\hat{G} \leftarrow G$;
for $i \in N$ **do** $\pi_i \leftarrow 0$;
while $|\hat{N}| > 1$ **do**
    $r \leftarrow \max\{2nr, \min\{\eta(e) : e \in \hat{A}\}\}$;
    contract all strongly connected components of $\hat{G}[\eta \le r]$ in $\hat{G}$;
    compute a topological ordering $\hat{N} = \{v_1, v_2, \dots, v_k\}$ of $\hat{G}[\eta \le r]$ such that $i < j$ for all $(v_i, v_j) \in \hat{A}$ with $\eta(v_i, v_j) \le r$;
    for $i = 1, \dots, k$ **do** $\pi_{v_i} \leftarrow \pi_{v_i} - \frac{ri}{2n}$;
uncontract $\hat{G}$ and map π back to the original graph G ;
return π.

Proof. The initial potential values are $\pi_i = 0$ and are monotone decreasing throughout. The current iteration decreases every potential by at most $r/2$. Since the value of r increases by at least a factor $2n \ge 4$ in every iteration, the cumulative change in all iterations thus far is at most $2r/3$. This implies the first statement.

Assume now that $e = (v_i, v_j)$ is active. Then, $c^\pi(e)$ increases by at least $r/(2n)$ in the current iteration, since π_{v_i} is decreased by a smaller amount than π_{v_j}. The second part follows, since the total change up to the previous iteration with value $r'' \le r/(2n)$ was $2r''/3 \le r/(3n)$. □

Lemma 3.7. *An arc $e \in A$ is contained in a strongly connected component of $G[\eta \le r]$ if and only if $\beta(e) \le r$. Every arc can be active in at most one iteration.*

Proof. Suppose first that $\beta(e) \le r$. By definition of $\beta(e)$, there exists a cycle C with $e \in C$ such that $c(f) \le \beta(e)$ for all $f \in C$. Consequently, $\beta(f) \le \beta(e)$ and $\eta(f) \le \beta(e)$ for all $f \in C$, showing that e is inside a strongly connected component of $G[\eta \le r]$ whenever $\beta(e) \le r$. Conversely, assume that there exists a cycle C' containing e such that $\eta(f) \le r$ for all $f \in C'$. Since $c(f) \le \eta(f)$, it follows that $\beta(e) \le r$.

Therefore, an arc e is active if and only if $\eta(e) \le r < \beta(e)$. Since $\eta(e) \ge \beta(e)/(2n)$, and r increases by at least a factor $2n$ between two iterations, it follows that each arc can be active at most once. □

Proof of Lemma 3.4. We first show that the algorithm ROUGH-BALANCE finds a $14n^2$-min-balanced cost function. Consider any arc $e \in A$, and let us pick a cycle C containing e such that $c(f), \beta(f) \le \beta(e)$ for every $f \in C$. Take the largest value of r during the algorithm such that $r < \beta(e)$; let $r' \ge \beta(e)$ denote the value in the next iteration. By Lemma 3.7, $e \in \hat{A}$ in the current iteration, and e will be contracted in the next iteration, along with the entire cycle C. Hence, $|c^\pi(f) - c(f)| \le 2r/3$ for all $f \in C$ for the final reduced cost c^π according to Lemma 3.6.

Claim 3.8. *We have $r' = 2nr$ or $r' = c(e) = \beta(e)$.*

Proof. If $r' > 2nr$, then $r' = \min\{\eta(f) : f \in \hat{A}\}$. Hence, $r' \le \eta(e) \le \beta(e) \le r'$. Equality must hold throughout, which in particular implies $\eta(e) = c(e) = \beta(e)$. □

We consider two cases.

Case I: $r < c(e)$**.** By the above claim, $\beta(e) \le r' \le 2nc(e)$. On the one hand, we have

$$c^\pi(e) \ge c(e) - \frac{2}{3}r \ge \frac{1}{3}c(e)\,.$$

On the other hand, for every $f \in C$, we have

$$c^\pi(f) \le c(f) + \frac{2}{3}r \le \beta(e) + \frac{2}{3}c(e) \le \left(2n + \frac{2}{3}\right) c(e) \le (6n+2)c^\pi(e)\,.$$

Hence, $\beta^\pi(e) \le (6n+2)c^\pi(e) < 14n^2 c^\pi(e)$.

Case II: $r \ge c(e)$**.** Since $r < r'$, Claim 3.8 yields $r' = 2nr \ge \beta(e)$, and thus $r \ge \beta(e)/(2n)$. Consequently, $r \ge \eta(e) = \max\{c(e), \beta(e)/(2n)\}$.

Since $\eta(e) \le r < \beta(e)$, by Lemma 3.6, e is an active arc, and the second part of the lemma guarantees that

$$c^\pi(e) \ge c(e) + \frac{r}{6n}\,.$$

For every $f \in C$, we have

$$c^\pi(f) \le c(f) + \frac{2}{3}r \le \beta(e) + \frac{2}{3}r \le \left(2n + \frac{2}{3}\right) r \le \left(2n + \frac{2}{3}\right) 6nc^\pi(e) \le 14n^2 c^\pi(e)$$

showing that $\beta^\pi(e) \le 14n^2 c^\pi(e)$. This completes the proof.

Running time bound The initial call to Find-Balance(N, A, c) takes $O(m \log n)$ time according to Lemma 3.5. The significant terms in the running time are computing strongly connected components of $G[\eta \le r]$ along with the topological ordering of active arcs, and updating the potentials. According to Lemma 3.7, each arc is active at most once. Hence, it is either contracted in the first iteration it appears in $G[\eta \le r]$, or the subsequent one. Therefore, the total number of these operations is $O(m)$. Maintaining the contracted graph using the Union-Find data structure is $O(m\alpha(n,m))$, see also Section 3.4. The number of operations in the final uncontraction is $O(n)$, similarly to the argument in Section 3.1.

To implement in the comparison-addition model, note that every number during the computations will be integer multiples of $(2n)^2$. Additions, subtractions, and comparisons of numbers in this form can be implemented in $O(\log n)$ time, as in Section 2.1. We also need multiplications by $i \le n$ and by $2n$ as well as divisions by $2n$; these operations also take time $O(\log n)$. Hence, the total running time can be bounded by $O(m\alpha(m,n)\log n)$. □

3.3 The strongly polynomial algorithm

We are ready to present the strongly polynomial algorithm as stated in Theorem 3.1. Given a graph $G = (N, A, c)$ with nonnegative arc costs, we preprocess it by contracting 0-cycles and changing to a strictly positive reduced cost. We then apply the subroutine Rough-balance to find a $14n^2$-min-balanced reduced cost function c^π. We can thus assume that the input of Algorithm 5 is a strictly positive and $14n^2$-min-balanced cost function c.

Algorithm 5 is similar to the weakly polynomial Algorithm 2. The two crucial differences are that *(a)* the subroutine Small-Cycles is called only for a subset of 'active' arcs; and *(b)* we may 'jump' over irrelevant values of L.

At the beginning of the algorithm, we sort the arcs in the increasing order of costs $c(e)$. At iteration t, we maintain two key parameters, the *'lower bound'* L_t and the *'step-size'* D_t, a contracted graph $(\hat{N}_t, \hat{A}_t, \hat{c}_t)$, and a set of *active arcs* $F_t \subseteq \hat{A}_t$. This is the subset of arcs with $\hat{c}_t(e) \le (n+1)\left(1 + \frac{1}{2^\rho}\right) L_t$.

Algorithm 5 MIN-BALANCE

Input: A strongly connected directed graph $G = (N, A, c)$ with a $14n^2$-balanced cost vector $c \in \mathbb{R}^A_{>0}$, parameters $\rho \in \mathbb{Z}_+$ and $\xi = 1 + 1/2^{\rho-1}$.

Output: A potential vector $\pi \in \mathbb{R}^N$ such that c^π is ξ-min-balanced.

sort all arcs in the increasing order of costs as $c(e_1) \le c(e_2) \le \ldots \le c(e_m)$;
$(\hat{N}_1, \hat{A}_1, \hat{c}_1) \leftarrow (N, A, c)$; $t \leftarrow 1$;
$L_1 \leftarrow c(e_1)$, $D_1 \leftarrow L_1/2^\rho$;
$F_1 \leftarrow \left\{e \in A : c(e) \le (n+1)\left(1 + \frac{1}{2^\rho}\right) L_1\right\}$;
while $|\hat{N}_t| > 1$ **do**
    $(\mathcal{P}_t, p_t) \leftarrow$ SMALL-CYCLES$(L_t, D_t, \hat{N}_t(F_t), F_t, \hat{c}_t)$;
    $(\hat{N}_{t+1}, \hat{F}, \hat{c}_{t+1}) \leftarrow (\hat{N}_t, F_t, \hat{c}_t^{p_t})/\mathcal{P}_t$;
    $L_{t+1} \leftarrow L_t + D_t$;
    if t is an integer multiple of 2^ρ **then**
        if $4nL_{t+1} < \min_{e \in \hat{A}_{t+1}} c(e)$ **then** $L_{t+1} \leftarrow \min_{e \in \hat{A}_{t+1}} c(e)/2$;
        $D_{t+1} \leftarrow L_{t+1}/2^\rho$;
    else $D_{t+1} \leftarrow D_t$;
    $F_{t+1} \leftarrow \hat{F} \cup \left\{e \in \hat{A}_{t+1} : (n+1)\left(1 + \frac{1}{2^\rho}\right) L_t < c(e) \le (n+1)\left(1 + \frac{1}{2^\rho}\right) L_{t+1}\right\}$;
    for $e \in F_{t+1} \setminus F_t$ **do** $\hat{c}_{t+1}(e) \leftarrow$GET-COST$(e)$;
uncontract $(\hat{N}_t, \hat{A}_t, \hat{c}_t)$, and compute the overall potential $\pi \in \mathbb{R}^N$.
return π.

As in Algorithm 2, the parameters are initialized as $L_1 = c(e_1)$, $D_1 = L_1/2^\rho$. The iterations start with a call to SMALL-CYCLES for the current value of L_t and D_t, but restricted to the graph $(\hat{N}_t(F_t), F_t)$ induced by the active arcs; this returns a partition $\mathcal{P}_t$ and potentials p_t. With a slight abuse of notation, the node potentials p_t are extended to the entire node set $\hat{N}_t$, by setting $p_t(v) = 0$ for $v \in \hat{N}_t \setminus \hat{N}_t(F_t)$. We contract each non-singleton subset in the partition $\mathcal{P}_t$; the new costs $\hat{c}_{t+1}$ represent the contractions of $\hat{c}_t^{p_t}$. However, we only maintain the $\hat{c}_{t+1}(e)$ values explicitly for the active arcs $\hat{F}$, the contracted image of F_t.

We now turn to the updates of L_t and D_t. In most iterations[3], we set $L_{t+1} = L_t + D_t$, and keep $D_{t+1} = D_t$. Exceptions are the special iterations when t is an integer multiple of 2^ρ, in which case we set $D_{t+1} = L_{t+1}/2^\rho$. In these special iterations, the update defining L_{t+1} is also different. We start by letting $L_{t+1} = L_t + D_t$, and then compare this value to $\min_{e \in \hat{A}_{t+1}} c(e)/(4n)$. If L_{t+1} is smaller, then we increase L_{t+1} to $\min_{e \in \hat{A}_{t+1}} c(e)/2$. Note that L_t increases by at least a factor 2 between any two special iterations.

After updating L_{t+1} and D_{t+1}, we update the set of active arcs by adding all arcs $e \in \hat{A}_{t+1}$ with cost $c(e) \in \left((n+1)\left(1 + \frac{1}{2^\rho}\right) L_t, (n+1)\left(1 + \frac{1}{2^\rho}\right) L_{t+1}\right]$. We emphasize that $c(e)$ here refers to the input costs and not the reduced cost. The subroutine GET-COST(e) obtains the reduced cost $\hat{c}_t(e)$ of the newly added arcs. This will be explained in Section 3.4, using the *Union-Find-Increase* data structure. We terminate once the graph is contracted to a singleton; at this point, we uncontract and obtain the output potential π in the original graph as in Algorithm 2.

Let us now turn to the analysis. We let T denote the total number of iterations.

Lemma 3.9. *Let $\tau \in [T]$ be an iteration of Algorithm 5 such that in all previous iterations $t \in [\tau]$, $\hat{c}_t(e) \ge L_t$ was valid for all $e \in F_t$. Then, $|\hat{c}_{\tau+1}(e) - c(e)| \le n\left(1 + \frac{1}{2^\rho}\right) L_\tau$ for every $e \in \hat{A}_t$.*

[3]More precisely, in $1 - 2^{-\rho}$ fraction of all iterations; there are no such iterations for $\rho = 0$.

Proof. The condition guarantees that the input to Small-Cycles at all iterations $t \le \tau$ satisfies the requirement on the arc costs. The potential p_t found by Small-Cycles has values $-|\hat{N}_t|D_t \le p_t(v) \le 0$. Therefore, for each $e \in \hat{A}_t$, $|\hat{c}_{\tau+1}(e) - c(e)| \le n\sum_{t=1}^{\tau} D_t$.

We show that $\sum_{t=1}^{\tau} D_t \le \left(1+\frac{1}{2^\rho}\right) L_\tau$. Indeed, $L_{t+1} \ge L_t + D_t$ in every iteration, implying $\sum_{t=1}^{\tau-1} D_t \le L_\tau$; and $D_\tau \le L_\tau/2^\rho$. □

Lemma 3.10. *In every iteration $t \in [T]$ of Algorithm 5, $\hat{c}_t(e) \ge L_t$ for all $e \in \hat{A}_t$. The final reduced cost function c^π is ξ-min-balanced. Further, every arc $e \in A$ with $c(e) < L_t/(14n^3)$ was contracted before iteration t.*

Proof. Let us start with the first claim. The proof is by induction. For $t = 1$, $\hat{c}_1(e) \ge L_1$ is true for every $e \in A = \hat{A}_1$ by the definition of $L_1 = c(e_1)$. Assume the claim was true for all $1 \le t' \le t$; we show it for $t+1$.

Assume first we set the value $L_{t+1} = \min_{f\in\hat{A}_{t+1}} c(f)/2$ in an iteration where t is divisible by 2^ρ. This happens if $2n(L_t + D_t) < \min_{f\in\hat{A}_{t+1}} c(f)/2$. Lemma 3.9 then implies that $\hat{c}(e) > \min_{f\in\hat{A}_{t+1}} c(f) - 2nL_t > L_{t+1}$ for every $e \in \hat{A}_{t+1}$.

Let us next assume the update was $L_{t+1} = L_t + D_t$. If $e \in \hat{F}$, i.e., the contracted image of F_t, then $\hat{c}_{t+1}(e) \ge L_t + D_t = L_{t+1}$ is guaranteed by Small-Cycles. Let $e \in \hat{A}_{t+1} \setminus F_t$, i.e., $c(e) > (n+1)\left(1+\frac{1}{2^\rho}\right) L_t$. Then, Lemma 3.9 shows $\hat{c}_{t+1}(e) > \left(1+\frac{1}{2^\rho}\right) L_t \ge L_{t+1}$.

The ξ-min-balancedness property of the final reduced cost c^π follows as in Lemma 3.3 for the weakly polynomial Algorithm 2.

Consider now an arc $e \in A$ with $c(e) < L_t/(14n^3)$. By the $14n^2$-min-balancedness of the input cost function c, there exists a cycle $C \subseteq A$ such that $c(f) \le 14n^2 c(e)$ for all $f \in C$. The final reduced cost c^π is nonnegative, and therefore

$$c^\pi(e) \le c^\pi(C) = c(C) \le 14n^3 c(e) < L_t\,.$$

Recall that the final reduced cost $c^\pi(e)$ equals $\hat{c}_{t'}(e)$ for the iteration t' when f was contracted. Since $\hat{c}_t(f) \ge L_t$ for all $f \in \hat{A}_t$, it follows that $t' < t$, as required. □

In Section 3.4 we will show that the overall running time of the operations Get-Cost(e) can be bounded as $O(m\alpha(m,n))$. We need one more claim that shows the geometric increase of L_t.

Lemma 3.11. *For every iteration $t' \ge 1$, we have $L_{t'+2^\rho} \ge 2L_{t'}$.*

Proof. Let $t = t' + 2^\rho$. Assume first $2^\rho | t' - 1$. Then, $D_{t'} = L_{t'}/2^\rho$, and we have $D_{t''} = D_{t'}$ for all $t'' \in [t', t-1]$. Consequently, $L_t \ge L_{t'} + 2^\rho D_{t'} = 2L_{t'}$. The inequality may be strict if in iteration $t-1$ we set $L_t > L_{t-1} + D_{t-1}$.

Assume now $t' = t_0 + k$ such that $2^\rho | t_0 - 1$ and $k \in [1, 2^\rho - 1]$. Then, $L_{t'} = L_{t_0}(1 + k/2^\rho)$, $L_{t_0+2^\rho} \ge 2L_{t_0}$, and $L_t = L_{t_0+2^\rho}(1 + k/2^\rho) \ge 2L_{t_0}(1 + k/2^\rho)$, thus, we again have $L_t \ge 2L_{t'}$. □

We are ready to prove Theorem 3.1.

Proof of Theorem 3.1. Part (a): the approximate min-balancing algorithm. Let us start with bounding the total number of arithmetic operations. After the $O(m \log n)$ preprocessing algorithm, we run the algorithm Rough-Balance to find a $14n^2$-balanced cost function in time $O(m \log n)$ (Lemma 3.4). We now turn the analysis of Algorithm 5. Let $m_t = |F_t|$ denote the number of active arcs in iteration t. The number of arithmetic operations in Small-Cycles in iteration t is bounded as $\max\{O(1), O((\rho+1)\cdot m_t\sqrt{n})\}$. The term $O(1)$ is needed since there may be some 'idle' iterations without any active arcs, that is, $m_t = 0$. In such a case the update rule in line 10

guarantees that new active arcs appear within the next $O(2^\rho)$ iterations. Thus, the number 'idle' iterations without active arcs can be bounded as $O(m2^\rho)$, since every arc can give the minimum value in line 10 at most once. The total running time of the 'idle' iterations is dominated by the other terms.

Let us now focus on the iterations containing active arcs. We show that

$$\sum_{t=1}^{T} m_t = O\left(2^\rho m \log n\right) . \tag{2}$$

Consider any arc $e \in A$. Let t_1 be the first and t_2 be the last iteration such that $e \in F_t$. By definition, t_1 is the smallest value such that $c(e) \leq (n+1)\left(1+\frac{1}{2^\rho}\right) L_{t_1}$, and by the last part of Lemma 3.10 $L_{t_2}/(14n^3) \leq c(e)$. Thus, $L_{t_2} \leq 28n^4 L_{t_1}$. Lemma 3.11 shows that L_t increases by a factor 2 in every 2^ρ iterations. Hence, $t_2 - t_1 \leq 2^\rho \log(28n^4)$, implying (2).

Hence, the total number of operations in the calls to Small-Cycles is bounded as $O\left(2^\rho m\sqrt{n}\log n\right)$. The time of contractions and cost updates can be bounded as $O(m\alpha(m,n))$ as shown in Section 3.4, and the final uncontraction takes $O(n)$.

Implementation in the comparison-addition model: As noted previously, Rough-Balance and Small-Cycles are both implementable in this model. Algorithm 5 uses additions, comparisons, multiplications by $4n$, divisions by 2 and by 2^ρ. Further, all numbers in the computations will be integer multiples of 2^b for $b \leq 2\rho + 1$. As noted in Section 2.1, all operations can be implemented in time $O(\rho + 1)$. The running time bound follows.

Part (b): obtaining the component hierarchy. Assume now $\rho = 0$ and $\xi = 3$; let us use the algorithm as described in Algorithm 5 with two simple modifications: we set the initial value as $L_1 = \lfloor c(e_1) \rfloor_2$ in line 3, and if $4nL_{t+1} < \min_{e \in \hat{A}_{t+1}} c(e)$, then we update L_{t+1} to $\left\lfloor \min_{e \in \hat{A}_{t+1}} c(e)/2 \right\rfloor_2$ in line 10. Thus, these values are rounded down to the nearest power of two. Such an operation is not allowed in the comparison-addition model, but can be done by a most significant bit operation in the word RAM model.

Recalling also that n is a power of 2, and that we set $D_{t+1} = L_{t+1}/2^\rho = L_{t+1}$ in every step, it follows that every L_t value is a power of 2.

The sets contracted during the algorithm can be naturally represented by a rooted tree $(V \cup N, E)$, where the nodes N correspond to the leaves and the root $r \in V$ to the final contraction of the entire node set. If the set represented by some $v \in V$ was contracted at iteration t, we set $a(v) = L_t$.

We claim that $(V \cup N, E, a)$ forms a component hierarchy of $G^\pi = (N, A, c^\pi)$. All $a(v) = L_t$ values are integer powers of 2 (this is the reason for the additional rounding steps). It is immediate that the leaves in the subtree of each $v \in V$ form a strongly connected component in G^π. Let v represent a set contracted in iteration t, that is, $v = P_i$ for a set P_i in the partition $\mathcal{P}_t$. If $\mathrm{lca}(i,j) = v$ for $i, j \in N$, that means that the nodes i and j got contracted together in iteration t. We show that $a(v) \leq \beta(i,j) \leq 3a(v)$, and that the nodes in $\mathrm{desc}(v)$ contain a path between i and j of arcs with cost at most $3a(v)$; consequently, $\beta(i,j) \leq 3a(v)$. If $t = 1$, then $L_1 = \lfloor C_{\min} \rfloor_2$, and P_i is strongly connected in the subgraph of arcs of cost at most $3L_1$. If $t > 1$, then $(\hat{N}_t, \hat{A}_t)$ contains a path between the contracted images of i and j with all arc costs between $a(v) = L_t$ and $3L_t$, and every i–j path must contain an arc of cost $\geq L_t$. We can map this back to the original graph by uncontracting the sets from previous iterations; all arc obtained in the uncontraction will have costs $\leq 3L_{t-1} < 3L_t$.

Note that for $\rho = 0$ and an integer input, the Algorithm 5 finds an integer π. This is because all D_t values are integral, and Small-Cycles changes the potential by integer multiples of D_t.

However, the input to Algorithm 5 is not the original cost but the cost obtained after the preprocessing and ROUGH-BALANCE. Preprocessing returns $nc^{\bar{\pi}}$ for a $1/n$-integral potential $\bar{\pi}$. For an integer input c, ROUGH-BALANCE returns a $1/4n^2$-integral potential. From these three steps, we can obtain a relabelling c^{π} of the original potential that is $1/(4n^3)$-integral if the original input cost was nonnegative integer. □

3.4 Union-Find-Increase: Maintaining the reduced costs

In GET-COST(e), we need to compute the current reduced cost of an arc e. Let $e = (i, j)$ in the original graph. In the current contracted graph $\hat{N}_t$, e is mapped to an arc (i', j'); that is, i is in a contracted set represented by node i', and j is in a contracted set represented by j' ($i = i'$ and $j = j'$ is possible). In the case that e is newly active (that is, it was not active at the previous iteration), we need to recover the reduced cost $\hat{c}_t(e)$. We do so by performing the uncontractions, as in the final step. Let π be the potential obtained by uncontracting all sets. To compute $\pi(i)$, we need to add up all the $p_{t'}(i[t'])$ values for every iteration $t' \leq t$, where $i[t']$ is the contracted node in $\hat{N}_{t'}$ representing i, and similarly for computing $\pi(j)$. Since there could have already been $\Omega(t) = \Omega(n)$ contractions of sets containing i and j, a naïve implementation would take $O(n)$ to compute a single reduced cost, or $O(nm)$ to obtain all current reduced costs.

We show that the time to calculate the reduced costs of newly active arcs in SMALL-CYCLES can be bounded as $O(m\alpha(m, n))$ by using an appropriate variant of the classical *Union-Find* data structure that we call *Union-Find-Increase*.

We refer the reader to [30] and [6, Chapter 21] for the description and analysis of *Union-Find*; we highlight the simple modifications only. The data structure maintains a forest F on the node set $N = \{1, 2, \ldots, n\}$, with each tree in F corresponding to a set in the partition. For each $i \in N$, let $\text{Anc}(i)$ be the ancestors of i in F (including i).

In addition, each $i \in N$ is associated with a key value $\sigma(i)$ that is initially 0, and which changes dynamically. We add two new operations to the data structure *Union-Find*: the operation $\text{Increase}(i, \delta)$ increases $\sigma(j)$ by δ for all j in the same tree as i; and the operation $\text{Value}(i)$ returns $\sigma(i)$. However, the $\sigma(i)$ values are not maintained explicitly. Instead, the algorithm maintains auxiliary values $\tau(j)$ such that the following property is satisfied for all $i \in N$:

$$\sigma(i) = \sum_{j \in \text{Anc}(i)} \tau(j)\,. \tag{3}$$

We need to modify the original operations as follows:

- Suppose a Union operation is performed on root nodes j and k, and j is made the root of the combined component. Then $\tau(k) \leftarrow \tau(k) - \tau(j)$.
- Suppose that a path compression takes place along path $j_1, \ldots, j_k$, where j_k is the root of the nodes in j_1 to j_k. The UNION-FIND algorithm sets the parent of j_i to j_k for $i \in [1, k-1]$. Let $\gamma(j_i) = \tau(j_{i+1}) + \ldots + \tau(j_{k-1})$; the time to compute the values are proportional to the length of the path. In addition to compressing the path, we set $\tau(j_i) \leftarrow \tau(j_i) + \gamma(j_i)$ for each $i \in [1, k-1]$.

Given these modifications, $\text{Increase}(i, \delta)$ can be implemented by first calling $\text{Find}(i)$ to determine the root j of the tree containing i, and increasing $\tau(j)$ by δ. To implement $\text{Value}(i)$, we first run $\text{Find}(i)$, which uses path compression so that i becomes the child of the root node node j of the tree. Thus, we can return $\sigma(i) = \tau(i) + \tau(j)$.

Clearly, the amortized complexity bound $O(\ell\alpha(\ell, n))$ for a sequence of ℓ steps for *Union-Find* is applicable for the modified data structure.

When applying *Union-Find-Increase* to implement the operations GET-COST(e), the key values $\sigma(i)$ correspond to the uncontracted potentials $\pi(i)$, and the sets to the pre-images of the nodes $v \in \hat{N}_t$ in the original node set N. We can further contract sets with the Union step. When $p_t(v)$ is changed by δ for a contracted node $v \in \hat{N}_t$, we need to update the potential of every original node represented by v; this is achieved by Increase(i, δ). Finally, GET-COST(e) for an arc $e = (i, j)$ can be implemented by calls to Value(i) and Value(j), and setting $\hat{c}_t(e) = c(e) + \pi(i) - \pi(j)$.

3.5 The adaptation of Goldberg's algorithm

In this section, we prove Lemma 3.2, showing how the subroutine SMALL-CYCLES can be implemented using a modification of Goldberg's algorithm [14].

Let $G = (N, A, c)$ be a directed graph with an integer cost function $c \in \mathbb{Z}^A$. Let $n = |N|$, $m = |A|$, and $C = \|c\|_\infty$. Goldberg developed an $O(m\sqrt{n}\log C)$ algorithm that finds a shortest path in a network or else finds a negative cost cycle. The algorithm runs in $\log C$ scaling phases. The key subroutine is REFINE; this is called at each scaling phase and takes $O(m\sqrt{n})$ time.

Algorithm 6 REFINE

Input: A directed graph $G = (N, A, c)$ with a cost function $c \in \mathbb{Z}^A$ such that $c(e) \geq -1$ for all $e \in A$.

Output: A negative cost cycle C, or a potential vector $\pi \in \mathbb{Z}^N$ such that

(i) $c^\pi(e) \geq 0$ for all $e \in A$, and

(ii) $-n + 1 \leq \pi(v) \leq 0$ for every $v \in N$.

We describe the modification BALANCED-REFINE that allows for negative cost cycles in a specific way.

Algorithm 7 BALANCED-REFINE

Input: A directed graph $G = (N, A, c)$ with a cost function $c \in \mathbb{Z}^A$ such that $c(e) \geq -1$ for all $e \in A$.

Output: A potential vector $\pi \in \mathbb{Z}^N$ and a subset of arcs $A' \subseteq A$ such that

(i) A' is the union of directed cycles, and $-1 \leq c^\pi(e) \leq 0$ for all $e \in A'$;

(ii) $c^\pi(e) \geq 0$ for all $e \in A \setminus A'$, and

(iii) $-n + 1 \leq \pi(v) \leq 0$ for every $v \in N$.

The running time of BALANCED-REFINE is also $O(m\sqrt{n})$. The subroutine SMALL-CYCLES (see Lemma 3.2) calls this for the cost function $\bar{c}(e) = \left\lfloor \frac{c(e)-L}{D} \right\rfloor - 1$. We obtain an arc set A' and a potential $\bar{\pi}$. We return the partition $\mathcal{P}$ formed by the (strongly) connected components of A', and the potential $\pi = D\bar{\pi}$. Note that $\bar{c}^{\bar{\pi}}(e) \leq 0$ implies $L \leq c^\pi(e) \leq L + 2D$, and $\bar{c}^{\bar{\pi}}(e) \geq 0$ implies $c^\pi(e) \geq L + D$. The required properties then follow.

To obtain an algorithm in the comparison-addition model, we do not need to compute the $\bar{c}(e)$ values explicitly: the only relevant information will be whether an arc cost is -1, 0, or positive. This simply corresponds to the cases $L \leq c(e) < L+D$, $L+D \leq c(e) \leq L+2D$, and $L+2D < c(e)$. We can directly update the original potentials π, subtracting Dk whenever $\bar{\pi}$ is decreased by k. This leads to an overhead $O(\log |N|)$ in the overall running time.

For completeness, we now describe the subroutines REFINE and BALANCED-REFINE in parallel; omitted parts of the analysis follow as in [14]. Both algorithms iteratively construct an integer potential $\pi \in \mathbb{Z}^N$. Throughout, $c^\pi(e) \geq -1$ for all $e \in A$. At termination, $c^\pi(e) \geq 0$ for all arcs in the contracted graph. The main difference is that REFINE terminates once a negative cycle is found. In contrast, BALANCED-REFINE adds all negative cycles to the arc set A' and contracts them.

An arc with $c^\pi(e) \leq 0$ is called *admissible*; we let $G_\pi = (N, A_\pi)$ be the subgraph formed by admissible arcs. Arcs with $c^\pi(e) = -1$ are called *improvable arcs*, and nodes with incoming improvable arcs are called *improvable nodes*; we denote this set as $I \subseteq N$.

The DECYCLE subroutine eliminates all directed cycles from G_π by contractions, using STRONGLY-CONNECTED(G_π). REFINE terminates if a negative cost cycle is found; in contrast, BALANCED-REFINE adds all such cycles to A' and proceeds with the algorithm. Contractions are carried out as described in Section 3.

A set of nodes $S \subseteq N$ is *closed* if no admissible arc leaves S. For a closed set, the subroutine CUT-RELABEL(S) decreases $\pi(u)$ by 1 for every $u \in S$. The closedness of S guarantees that no improvable arcs are created.

Assume G_π is acyclic. Let us pick any improvable node i, and let S be the set of nodes reachable from i in G_π; this is a closed set. After CUT-RELABEL(S), i is no longer improvable, and no new improvable nodes appear. In this manner, we can decrease the number of improvable nodes in $O(m)$ time. By alternating between the subroutines DECYCLE and CUT-RELABEL, one can eliminate all improvable nodes in $O(nm)$ time, resulting in a graph with nonnegative reduced costs.

Goldberg improves this to $O(m\sqrt{n})$ time by eliminating at least $\sqrt{k}$ improvable nodes in $O(m)$ time, where $k = |I|$ is the number of improvable nodes in G_π. The first step in speeding up the running time is to eliminate more than one improvable node when running CUT-RELABEL.

A set $X \subseteq I$ of improvable nodes is called an *anti-chain* in G_π if for all nodes i and j in X, there is no directed path from node i to node j in G_π. Let S be the set of nodes reachable in G_π from a node of X. After running CUT-RELABEL(S), none of the nodes in X are improvable.

In order to find a large anti-chain of improvable nodes, Goldberg's algorithm appends a source node s to (the acyclic graph) G_π and for each other node j, it adds an arc (s, j) with a cost of 0. Then for each node j in G_π, the algorithm determines the shortest path distance $d(j)$ in G_π from node s to node j; these values can be computed in linear time for an acyclic graph.

Case I: $d(j) \geq -\sqrt{k}$ for all nodes $j \in I$: For each integer q with $-\sqrt{k} \leq q \leq -1$, let $X_q = \{j \in I : d(j) = -q\}$. This gives an anti-chain partition of I; thus we have $|X_q| \geq \sqrt{k}$ for the largest one among these sets. After running CUT-RELABEL(S) for the nodes reachable from the largest anti-chain X_q, the number of improvable nodes reduces by at least $\sqrt{k}$ in $O(m)$ time.

Case II: $\min_{j \in I} d(j) < -\sqrt{k}$: In this case, there exists a directed path P in G_π that contains improvable arcs $(v_1, w_1), (v_2, w_2), \ldots, (v_t, w_t)$ for $t \geq \sqrt{k}$ in this order. We now describe the subroutine ELIMINATE-CHAIN after which none of the nodes in w_i are improvable, and no new improvable nodes are created.

We start with the original variant of the subroutine used in REFINE. The nodes w_i are processed in reverse order. For each node w_i, $i = t, t-1, \ldots 1$, find the sets S_i of nodes reachable from w_i in G_π, and run CUT-RELABEL(S_i). No new improvable arcs are created, and if $v \notin S_i$ for any improvable arc (v, w_i), then i is not improvable after the change. It is easy to see that $S_i \subsetneq S_j$ for all $1 \leq j < i \leq t$.

If $v \in S_i$ for an improvable arc (v, w_i) at any iteration, then we discover a negative cost cycle containing (v, w_i). The subroutine REFINE terminates at this point by returning this cycle. Goldberg [14] presents an efficient $O(m)$ implementation of REFINE by exploiting that the sets S_i

are nested. The implementation (temporarily) contracts the S_i sets, and maintains a data structure using priority queues.

We now describe the variant of Eliminate-Chain used in Balanced-Refine. We say that an arc (v, w) is *eligible* if (v, w) is improvable and if (v, w) is not contained in an admissible cycle. (This definition is not relevant in Refine, since that algorithm terminates if an improvable arc is in an admissible cycle.) We say that a node w is *eligible* if there is an eligible arc directed into w. Initially, G_w is acyclic, and hence w_j is eligible for all $j \in [t]$.

Now consider the iteration in which the eligible node w_i is selected. We note that w_i is not eligible after running Cut-Relabel(S_i). This is because for any arc (v, w_i) that is still improvable after Cut-Relabel(S_i), we must have $v \in S_i$, implying that (v, w_i) was not eligible.

Let us select the smallest index j such that after Cut-Relabel(S_i), w_j becomes reachable from w_i in G_π; that is, w_j enters S_i. Let us analyze the case when $j < i$; note that S_i also contains every node on the subpath in P from w_j to w_i. Thus, w_ℓ is not eligible for any $\ell \in [j, i]$ after Cut-Relabel(S_i). At the subsequent iteration of Eliminate-Chain we skip all nodes in S_i and instead select w_{j-1}, which is eligible.

After running Cut-Relabel(S_i), Eliminate-Chain temporarily contracts S_i in the same way as the version used in Refine. Thus, Balanced-Refine uses essentially the same implementation and data structures as in [14].

At the end of Eliminate-Chain, we uncontract all S_i's. Some of the w_i's may now be improvable, however, all improvable arcs incident to them are contained in directed admissible cycles. At the next call to Decycle, the algorithm would contract any improvable arc that was not eligible. Subsequently, the number of improvable nodes will have decreased by at least $\sqrt{k}$.

4 The shortest path algorithm

In this section, we assume that a 3-min-balanced directed graph $G = (N, A, c)$ is given, along with a component hierarchy $(V \cup N, E, r, a)$ for G (see Definition 2.2). The algorithm described in this section is an adaptation of Thorup's [31] result to the setting of balanced directed graphs. We use the word RAM model throughout this section.

We assume that the input cost function c is 3-min-balanced, integral, and strictly positive. This is justified by Theorem 3.1: in time $O(m\sqrt{n}\log n)$, one can obtain a strictly positive and $1/(4n^3)$-integral reduced cost c^π such that $G^\pi = (N, A, c^\pi)$ is 3-min-balanced. Since for any i–j path P, $c^\pi(P) = c(P) - \pi(i) + \pi(j)$, the set of shortest paths between any two nodes is the same in G and G^π. Integrality can be assumed after multiplying the relabelled cost by $4n^3$ (recall that n is a power of 2); this again does not change the set of shortest paths.

4.1 Upper bounds for the component hierarchy

In the component hierarchy $(V \cup N, E, r, a)$, recall that for a vertex $v \in V$, $\mathrm{desc}(v) \subseteq V \cup N$ denotes the set of descendants of v (with $v \in \mathrm{desc}(v)$). We introduce the shorthand notation $\mathrm{desc}(v, N) = \mathrm{desc}(v) \cap N$ and $\mathrm{desc}(v, V) = \mathrm{desc}(v) \cap V$. For a node $u \in V \cup N$, the *height* $h(u)$ is the length of the longest path between u and a node in $\mathrm{desc}(u)$; in particular, $h(u) = 0$ for $u \in N$.

We define the functions $\Gamma, \eta : V \to \mathbb{Q}$ recursively, in non-decreasing order of $h(u)$ as follows.

$$\begin{aligned} \Gamma(v) &:= 3a(v)(|\,\mathrm{children}(v)| - 1) + \sum_{v' \in \mathrm{children}(v) \setminus N} \Gamma(v')\,, \\ \eta(v) &:= \left\lceil \frac{\Gamma(v)}{a(v)} \right\rceil . \end{aligned} \tag{4}$$

These values will be relevant for the *buckets* in the algorithm. As shown in the next lemma, $\Gamma(v)$

is a bound on the length of a shortest path between any two nodes in $\mathrm{desc}(v, N)$; we will associate $\eta(v) + 1$ buckets with each vertex $v \in V$.

Lemma 4.1. *Let $(V \cup N, E, r, a)$ be a component hierarchy for a directed graph $G = (N, A, c)$, and let Γ, η be as in (4). For any pair of nodes $i, j \in N$ and $v = \mathrm{lca}(i, j)$, there is an i–j path P in $\mathrm{desc}(v, N)$ of length at most $\Gamma(v)$. In addition,*

$$\sum_{v \in V} \eta(v) < 7|N|\,.$$

Proof. Let $i, j \in N$ and $v = \mathrm{lca}(i, j)$. The proof is by induction on $h(v)$. Consider the i–j path P' in $\mathrm{desc}(v)$ such that $c(e) \leq 3a(v)$ for all $e \in P'$, as guaranteed by the property of the component hierarchy.

In the base case $h(v) = 1$, the bound is immediate, since P' has at most $|\,\mathrm{children}(v)| - 1$ arcs. Assume now $h(v) > 1$, and that the statement holds for any i', j' with $h(\mathrm{lca}(i', j')) < h(v)$. One can choose an i–j path P that satisfies the following property for each child u of v. If i' and j' are the first and last nodes of P that are in $\mathrm{desc}(u)$, then the subpath in P from i' to j' consists of nodes of $\mathrm{desc}(u)$. By the inductive hypothesis, for each child u of v, the length of the subpath in $\mathrm{desc}(u)$ is at most $\Gamma(u)$. There are at most $|\,\mathrm{children}(v)| - 1$ arcs in P between different $\mathrm{desc}(u)$ subpaths; their cost is at most $3a(v)(|\,\mathrm{children}(v)| - 1)$. Thus, the bound $c(P) \leq \Gamma(v)$ follows.

Let us now turn to the second statement. We analyze the contribution of each $i \in N$ to the sum $\sum_{v \in V} \Gamma(v)/a(v)$. Let $i = v_0, v_1, v_2, \ldots, v_k = r$ be the unique path in the tree $(V \cup N, E)$ from i to the root; thus, $p(v_t) = v_{t+1}$ for $t = 0, \ldots, k-1$. Then, the contribution of i to each $\Gamma(v_t)$ is less than $3a(v_1)$. Using that $a(v_{t+1}) \geq 2a(v_t)$ for each $t = 0, \ldots, k-1$, we see that

$$\sum_{v \in V} \frac{\Gamma(v)}{a(v)} < 3 \sum_{i \in N} \sum_{t=1}^{\infty} \frac{1}{2^{t-1}} < 6|N|\,.$$

The statement follows noting also that $|V| \leq |N| - 1$, since $(V \cup N, E)$ is a tree with leaves N, and $\eta(v) < 1 + (\Gamma(v)/a(v))$ for all $v \in V \setminus N$. □

4.2 Overview of the algorithm

Given the input directed graph $G = (N, A, c)$, our goal is to compute the shortest path distances from a *source node* $s \in N$ to all nodes in N. We assume that a positive integer cost function and a component hierarchy are given as above. We start with an informal overview and highlight some key ideas of the analysis.

The algorithm is a bucket-based *label setting* algorithm, similarly to a bucket-based implementation of Dijkstra's algorithm. For each node $i \in N$, we maintain an upper bound $D(i)$ on the true distance $d(i)$ from s, and gradually extend the set S of permanent nodes. Initially, $D(s) = 0$ and $D(i) = \infty$ for $i \in N \setminus \{s\}$ and $S = \{s\}$. At the iteration at which i enters S, $D(i) = d(i)$ will be guaranteed.

Recall that Dijkstra's algorithm always selects a next node j to enter S with $j \in \arg\min\{D(i) : i \in N \setminus S\}$. To obtain an $O(m)$ algorithm, we relax this condition, and always add a new node $j \in N \setminus S$ to S such that

$$D(j) \leq D(i) + b(i, j) \quad \forall i \in N \setminus S\,. \tag{5}$$

In accordance with this rule, the next lemma formulates the conditions that guarantee the correctness of our algorithm.

Lemma 4.2. *Given a directed graph $G = (N, A, c)$ with $c \in \mathbb{R}^N_{\geq 0}$ and a source node $s \in N$, assume that an algorithm proceeds by adding nodes in N one-by-one to a set S such that the following two invariants are maintained at every iteration:*

(a) For all $j \in S$ and $i \in N \setminus S$, $D(j) \leq D(i) + b(i, j)$.

(b) For all $j \in N \setminus S$, $D(j)$ is the length of a shortest path from s to j inside the node set $S \cup \{j\}$.

Further, assume that initially $D(s) = 0$ and s is the first node added to S. Then, at any point of the algorithm, for every $j \in S$, we have $D(j) = d(j)$ and S contains a shortest s–j path.

Proof. For convenience, suppose that that the nodes are relabelled such that node i is the i-th node added to S. The lemma is true for node $1 = s$, since $D(1) = d(1) = 0$. We now assume inductively that the lemma is true for nodes $\ell = 1$ to i, and we prove it for node $i + 1$.

Let P be any path from node 1 to node $i + 1$. We show $c(P) \geq D(i + 1)$; together with (b), this implies $D(i + 1) = d(i + 1)$.

Let $V(P)$ be the vertices of P. If $V(P) \subseteq \{1, \ldots, i+1\}$, then $c(P) \geq D(i+1)$ by (b). Otherwise, let j be the first vertex of P that is not in $\{1, \ldots, i + 1\}$. Let P' be the subpath of P from 1 to j. Then at the iteration in which node $i + 1$ is added to S, we have

$$c(P) \geq c(P') + b(j, i + 1) \geq D(j) + b(j, i + 1) \geq D(i + 1)\,,$$

where the second inequality follows by (a). This completes the proof. □

We rely on the component hierarchy and the use of buckets to efficiently implement the selection property (5). We will also have (possibly infinite) $D(v)$ values for certain vertices $v \in V$. Throughout, we maintain a set of *active* vertices (we describe the treatment of active vertices in more detail later). Initially, the root r is the only active vertex and all other vertices are *inactive*. At any point, the active vertices form an upper ideal (i.e., all ancestors of an active vertex are also active). Once all their descendants are added to S, vertices in V also enter S (become permanent); the algorithm terminates when r is added to S. A vertex is active during the iterations from its activation until it is made permanent. One of the active vertices will be the *current vertex*, denoted as CV and initalized as CV $= r$. This plays a special role: in particular, nodes added to S will always be among the children of CV.

A vertex v is called a *highest inactive vertex* if v is inactive and $p(v)$ is active. For an inactive vertex v, we let HIA(v) denote its *highest inactive ancestor*: HIA$(v) = v$ if v is a highest inactive vertex; otherwise, HIA(v) is v's unique ancestor that is a highest inactive vertex.

The next lemma, proved in Section 4.4, shows that for every active vertex v, $D(v)$ is a lower bound on $\min\{D(j) : j \in \mathrm{desc}(v, N)\}$, and when a node in $j \in \mathrm{desc}(v, N)$ is added to S, $D(j)$ is within $a(v)$ from $D(v)$.

Lemma 4.3. *Let $j \in N \setminus S$ and let v be an active ancestor of j. Then, $D(v) \leq D(j)$. In the iteration when j is added to S, we also have $D(j) < D(v) + a(v)$.*

Recalling the property of the component hierarchy that $b(i, j) \geq a(v)$ for $v = \mathrm{lca}(i, j)$, this immediately implies property (5).

Buckets The choice of CV and the sequence of nodes added to S is guided by the use of buckets associated with the vertices $v \in V$. The buckets of v are created when v is activated by the ACTIVATE(v) subroutine. Before activation, v was a highest inactive vertex, and for all such vertices, we maintain $D(v) = \min\{D(j) : j \in \mathrm{desc}(v, N)\}$ using the Split/FindMin data structure.

At activation, $L(v)$ is set to $a(v) \cdot \lfloor D(v)/a(v) \rfloor$. Then an array $\eta(v)+1$ buckets is created for vertex v, indexed from 0 to $\eta(v)$. The value range of the bucket with index k is $[L(v)+ka(v), L(v)+(k+1)a(v))$. We let $U(v) := L(v)+(\eta(v)+1)a(v)$ denote the upper range of the last bucket for vertex v. We place a child x of v in the bucket whose value range contains $D(x)$, or leave it unassigned if $D(x) > U(v)$.

An important feature of the algorithm is that the value range of the buckets at v, created at activation, contains the $d(i)$ values for all $i \in \mathrm{desc}(v, N)$ (Lemma 4.7). We now highlight the reason behind this. At the iteration at which v is activated, let $i = \arg\min\{D(j) : j \in \mathrm{desc}(v, N)\}$. One can show that $d(i) = D(i)$, and that $d(j) \geq d(i)$ for all $j \in \mathrm{desc}(v, N)$. After activation, we have $L(v) \leq D(i) \leq L(v) + a(v)$. By Lemma 4.1, for any other node $j \in \mathrm{desc}(v, N)$, there is a path in G with node i to node j of length at most $\Gamma(v)$. Thus, $d(j) \leq d(i)+\Gamma(v) \leq L(v)+(\eta(v)+1)a(v) = U(v)$.

The *current index* $\mathrm{CI}(v)$, initialized as 0, refers to the index of the first nonempty bucket, called the *current bucket*. We will maintain $D(v)$ as the lower endpoint of the current bucket, augmented by $a(v)$ every iteration the current bucket becomes empty. The vertex v is made permanent once $\mathrm{CI}(v) = \eta(v) + 1$, that is, all its buckets have been exhausted.

Recall also from Lemma 4.1 that the overall number of buckets for all vertices is bounded as $O(n)$; this enables an $O(n)$ running time bound on the operations involving buckets.

The trajectory of the current vertex The algorithm is guided by the movement of the current vertex CV that explores the component hierarchy. Initially, it moves down from the root r to the source node s, activating all vertices along the r–s path. As long as the current bucket at CV contains a node, we add such nodes to S. Whenever a node i is added to S, the subroutine UPDATE(i) scans over the outgoing arcs (i, j), and updates the estimates $D(j)$ to $\min\{D(j), D(i)+c(i, j)\}$ as in Dijkstra's algorithm. This requires some additional updates in the data structure, i.e., moving j to a different bucket if its parent $p(j)$ is active, or updating the $D(w)$ value of its highest inactive ancestor.

If the current bucket B at $v = \mathrm{CV}$ contains some vertices but no nodes, then CV moves down to a child vertex, and also activates it in case it had not yet been active. If B is empty and if B is not the last bucket of v, then we move the current bucket to the next one, i.e., increment $\mathrm{CI}(v)$ by 1, and increase $D(v)$ by $a(v)$. If the last bucket at v becomes empty, then we make v permanent. At this point, all nodes and vertices in $\mathrm{desc}(v)$ must have been already made permanent. The algorithm then replaces CV by $p(v)$ if $v \neq r$. The algorithm terminates once the last bucket at the root r becomes empty and r is made permanent.

After incrementing $\mathrm{CI}(v)$ in the case that B is empty, we proceed to the next bucket with no change in CV if $v = r$ or if the new $D(v)$ value is less than $D(p(v)) + a(p(v))$. On the other hand, if $D(v) \geq D(p(v)) + a(p(v))$, then the current vertex CV moves up to $p(v)$, and v is moved from the current bucket at $p(v)$ to a higher bucket. Overall, this scheme allows $D(\mathrm{CV})$ to be approximately minimal among the labels of active vertices, and thereby enabling the properties asserted in Lemma 4.3.

Finally, if the last bucket at v becomes empty, then we make v permanent; at this point, all nodes and vertices in $\mathrm{desc}(v)$ must have been already made permanent. The algorithm terminates once the last bucket at the root r becomes empty and r is made permanent.

4.3 Description of the algorithm

A more formal description of the algorithm with pseudocodes is in order. Recall the basic notation regarding component hierarchies from Section 2: $p(v)$ (parent of v); $\mathrm{children}(v)$ (children of v); $\mathrm{desc}(v)$ (descendant of v, refined as $\mathrm{desc}(v, N)$ for nodes and $\mathrm{desc}(v, V)$ for vertices); $\mathrm{lca}(u, v)$ (least common ancestor of v).

The set $S \subseteq N \cup V$ denotes the set of *permanent* nodes and vertices, initialized as $S = \emptyset$; the first node entering will be the source s. Shortest paths will be maintained using predecessor arcs: for each $i \in N \setminus \{s\}$ with $D(i) < \infty$, $\text{pred}(i) \in S$ is an in-neighbour such that $D(i) = D(\text{pred}(i)) + c(\text{pred}(i), i)$. The graph of the arcs $(\text{pred}(i), i)$ is acylic, and contains a path from the source s to every node $i \in N$ with $D(i) < \infty$.

The description of the two main subroutines, ACTIVATE and UPDATE follows.

The Activate subroutine and buckets Each vertex $v \in V$ can be *active* or *inactive*. One of the active vertices will be CV, the current vertex, initalized as $\text{CV} = r$.

The labels are defined for all nodes (initially as $D(s) = 0$ and $D(i) = \infty$ for $i \in N \setminus \{s\}$), for all active vertices, and for all highest inactive vertices. For the latter set, we maintain $D(v) = \min\{D(i) : i \in \text{desc}(v)\}$ using the Split/FindMin data structure, as detailed in Section 4.5. For all other inactive vertices, the labels $D(v)$ are undefined.

The ACTIVATE(v) subroutine (Algorithm 8) is called the first time CV is set to v. We create an array of $\eta(v)+1$ empty buckets, indexed $k = 0, \ldots, \eta(v)$, and denoted as $\text{Bucket}(v, k)$. The buckets correspond to intervals $[\text{Lower}(v, k), \text{Upper}(v, k))$ of length $a(v)$. The 0th bucket starts at $L(v)$, which equals $D(v)$ rounded down to the nearest integer multiple of $a(v)$ (recall this is an integer power of 2).

For $x \in V \cup N$, the MOVETOBUCKET(x) procedure (Algorithm 9) checks if $D(x)$ falls in the value range of a bucket at the parent $v = p(j)$, places it in such a bucket, and if it was previously in a bucket, deletes it from there.

Algorithm 8 The ACTIVATE subroutine

```
1:  procedure ACTIVATE(v)
2:      L(v) ← a(v) ⌊D(v)/a(v)⌋ ;
3:      D(v) ← L(v); CI(v) ← 0 ;
4:      for k = 0, ..., η(v) do
5:          Bucket(v, k) ← ∅ ;
6:          Lower(v, k) ← L(v) + ka(v) ;
7:          Upper(v, k) ← L(v) + (k + 1)a(v) ;
8:      U(v) ← L(v) + (η(v) + 1)a(v) ;
9:      for w ∈ children(v) ∩ V do
10:         D(w) ← min{D(i) : i ∈ desc(v)} ;        ▷ using the Split/FindMin data structure
11:         MOVETOBUCKET(w) ;
12:     for j ∈ children(v) ∩ N do
13:         MOVETOBUCKET(j) ;
```

Algorithm 9 The MOVETOBUCKET subroutine

```
1:  procedure MOVETOBUCKET(x)
2:      v ← p(x) ;
3:      if v is active and D(x) < U(v) then
4:          k ← ⌊(D(x) − L(v))/a(v)⌋ ;
5:          if x ∉ Bucket(v, k) then
6:              delete x from its current bucket (if any) ;
7:              add x to Bucket(v, k) ;
```

The Update subroutine The UPDATE subroutine (Algorithm 10) performs the label update step once a node i is made permanent, similarly to Dijkstra's algorithm. For every outgoing arc (i, j), if $D(i) + c(i, j)$ is strictly less than the current label $D(j)$, we reduce $D(j)$ to this value, and set the predecessor $\text{pred}(j)$ to i. If the parent $p(j)$ is active, we call MOVETOBUCKET(j) to update the bucket containing j. Otherwise, we update $D(w)$ for $w = \text{HIA}(j)$, i.e., the highest inactive ancestor of j, using Split/FindMin.

Algorithm 10 The UPDATE subroutine

```
1: procedure UPDATE(i)
2:     for (i, j) ∈ A(i) do
3:         if D(i) + c(i, j) < D(j) then
4:             D(j) ← D(i) + c(i, j) ; pred(j) ← i ;
5:             if p(j) is active then MOVETOBUCKET(j) ;
6:             else  w ← HIA(j) ; D(w) ← min{D(j), D(w)} ;
7:                                          ▷ using the Split/FindMin data structure
```

The overall algorithm The overall algorithm is shown in Algorithm 11. Initially, the current vertex is set as the root: $\text{CV} = r$. At any given iteration, we let $v = \text{CV}$ and let B denote the current bucket at v, i.e., $B = \text{Bucket}(w, \text{CI}(v))$.

If B contains a node $i \in N$, we make it permanent, i.e., add it to S, and call UPDATE(i) to update the labels for each out-neighbour j of i. If B contains no nodes but some vertices, we move CV to such a vertex w, and activate it if necessary.

The remaining possibility is when the bucket B becomes empty in the current iteration. We increment the counter $\text{CI}(v)$ by 1 and accordingly update $D(v)$ to $D(v) + a(v)$, the starting point of the new current bucket. In case $\text{CI}(v) = \eta(v) + 1$, i.e., if B was already the final bucket, then we make v permanent, and unless $v = r$, we move CV up to the parent $p(v)$. If $v = r$ then the algorithm terminates.

Otherwise, if $\text{CI}(v) \leq \eta(v)$, we check if the updated value $D(v) \geq D(p(v)) + a(p(v))$, i.e., if the update requires moving v to a higher bucket at $p(v)$ (assuming $v \neq r$). If this is the case, CV moves up to $p(v)$; otherwise, we proceed with $\text{CV} = v$.

4.4 Analysis

Theorem 4.4. *Algorithm 11 computes shortest paths from node $s \in N$ to all other nodes in $O(m)$.*

We prove the theorem in two parts. Lemma 4.5 shows the running time bound $O(m)$. Correctness follows using Lemma 4.2 and Lemma 4.3 stated above. To prove the latter lemma, we need one more auxiliary statement (Lemma 4.6) that relates the label of an active vertex to that of its active descendants.

Lemma 4.5. *The total running time of Algorithm 11 is bounded as $O(m)$.*

Proof. The time for initialization is $O(n)$. Let us show that the main *while* cycle is called $O(n)$ times. We consider the cases for $v = \text{CV}$ and current bucket B as *(i)* B contains a node, or *(ii)* B contains a vertex but no node, or *(iii)* B is empty.

Whenever case *(i)* occurs, a node is added to S, giving a bound of $O(n)$ for this case. In case *(iii)*, $\text{CI}(v)$ is incremented, and CV is possibly moved to $p(v)$. The number of times this can occur is equal to the total number of buckets, which is $O(n)$ by Lemma 4.1.

Let us now turn to case *(ii)*. Let τ denote the distance of the current vertex CV from the root r in the component hierarchy. Both in the first and the final iteration, $\text{CV} = r$, and thus $\tau = 0$.

Algorithm 11 SHORTEST-PATHS

Input: A directed graph $G = (N, A, c)$ with $c \in \mathbb{Z}^A_{>0}$, source node $s \in N$, a component hierarchy $(V \cup N, E, a)$ for G.

Output: Shortest path labels for each $i \in N$ from s.

$S \leftarrow \emptyset$;
$D(s) \leftarrow 0$; $D(r) \leftarrow 0$;
for $j \in N \setminus \{s\}$ **do** $D(j) \leftarrow \infty$;
for $v \in V$ **do** compute $\Gamma(v)$ and $\eta(v)$ as in (4) ;
CV $\leftarrow r$; ACTIVATE(r) ;
while $r \notin S$ **do**
    $v \leftarrow$ CV ; $B \leftarrow$ Bucket$(v, \text{CI}(v))$;
    if $B \cap N \neq \emptyset$ **then**
        select a node $i \in B \cap N$ and delete i from B ;
        $S \leftarrow S \cup \{i\}$;
        UPDATE(i) ;
    else if $B \cap V \neq \emptyset$ **then**
        select a vertex $w \in B \cap V$;
        CV $\leftarrow w$;
        if w is inactive **then** ACTIVATE(w) ;
    else ▷ $B = \emptyset$
        CI$(v) \leftarrow$ CI$(v) + 1$; $D(v) \leftarrow D(v) + a(v)$;
        if CI$(v) = \eta(v) + 1$ **then**
            $S \leftarrow S \cup \{v\}$;
            if $v \neq r$ **then** CV $\leftarrow p(v)$;
        else if $v \neq r$ and $D(v) \geq D(p(v)) + a(p(v))$ **then**
            CV $\leftarrow p(v)$;
            MOVETOBUCKET(v) ;
return labels $D(i)$: $i \in N$.

Whenever case *(ii)* occurs, τ increases by one. The only way τ can decrease is if CV is moved from a vertex to its parent in case *(iii)*. Thus, the total number of occurrences of case *(ii)* is equal to the total number of increases in τ, which equals the total number of decreases, in turn bounded by $O(n)$. Thus, each of the three cases can only occur $O(n)$ times, bounding the number of iterations of the *while* cycle.

The subroutine UPDATE(i) is called once for each $i \in N$. At each call, the arcs in $A(i)$ are scanned. The time to update $D(j)$ for $(i,j) \in A(i)$ is $O(1)$. If $p(j)$ is active, then the time to put node j in the correct bucket at $p(j)$ is $O(1)$. A potential bottleneck occurs when $p(j)$ is inactive and $D(j)$ is updated. In this case, the algorithm determines $w = \text{HIA}(j)$ and then updates $D(w)$. The amortized time to determine w and update $D(w)$ is $O(1)$ using Thorup's [31] implementation of the Split/FindMin data structure (see Section 4.5). Thus, the total time of the updates is $O(m)$.

We now consider ACTIVATE(v), which is called $O(n)$ times. The total number of buckets is $O(n)$, and each $x \in \text{children}(v)$ has to be placed in a bucket; note that $\sum_{v \in V} |\text{children}(v)| \leq 2n-1$. The overall time for creating buckets and placing the children in buckets takes $O(n)$. Further, we need to update $D(w)$ for $w \in \text{children}(v) \cap V$. For each w, this is again accomplished using the Split/FindMin data structure in amortized time $O(1)$.

The total running time of the Split/FindMin operations can be bounded as $O(m)$. The $O(n)$ bound on the *while* iterations, the total $O(n)$ on ACTIVATE and $O(m)$ on UPDATE yields the overall $O(m)$ bound. □

The next lemma will be key in proving Lemma 4.3.

Lemma 4.6. *Let v and w be active vertices such that $w \in \text{desc}(v, V)$. Then*

(i) $D(w) \leq D(v) + a(v)$; and

(ii) if $\text{CV} \in \text{desc}(w, V)$, then $D(w) + a(w) \leq D(v) + a(v)$.

Proof. We start by showing part (ii). We prove it for the case $v = p(w)$; this immediately implies the general case. We first consider the case that w has just become the current vertex and v was previously the current vertex. Since w was selected from the current bucket of v, it follows that $D(w) < D(v) + a(v)$. Moreover, $D(w)$, $D(v)$ and $a(v)$ are all integer multiples of $a(w)$. (In the case that w was just activated, its label $D(w)$ was obtained by rounding its previous label down to the nearest multiple of $a(w)$.) The claim that $D(w) + a(w) \leq D(v) + a(v)$ follows.

If w is the current vertex, then $D(w)$ may only change if the current bucket at w is empty, in which case $D(w)$ is incremented to $D'(w) = D(w) + a(w)$. If $D'(w) \geq D(v) + a(v)$, then the current vertex moves up to v, at which point $\text{CV} \notin \text{desc}(w, V)$. Otherwise, $D'(w) < D(v) + a(v)$, implying $D'(w) + a(w) \leq D(w) + a(w)$ as above.

In all other iterations when $\text{CV} \in \text{desc}(w, V)$, neither $D(v)$ nor $D(w)$ may change, and therefore the statement remains valid. This completes the proof of part (ii).

Let us now show part (i); we do not assume $v = p(w)$ for this proof. In light of part (ii), we can focus on iterations when $\text{CV} \notin \text{desc}(w, V)$. When w is activated, $w = \text{CV}$. Consider any iteration when CV leaves $\text{desc}(w, V)$; this means that CV moves from w to $p(w)$. This happens when the current bucket at w is empty and $D(w) \geq D(p(w)) + a(p(w))$; but again using divisibility this means $D(w) = D(p(w)) + a(p(w))$. By part (ii) applied to $p(w)$ and v, it follows that $D(w) = D(p(w)) + a(p(w)) \leq D(v) + a(v)$. In all subsequent iterations until w becomes the current vertex again, $D(w)$ remains unchanged, and $D(v)$ may only increase. Thus, $D(w) \leq D(v) + a(v)$ is maintained, implying (i). □

We are ready to show Lemma 4.3, restated here.

Lemma 4.3. *Let $j \in N \setminus S$ and let v be an active ancestor of j. Then, $D(v) \leq D(j)$. In the iteration when j is added to S, we also have $D(j) < D(v) + a(v)$.*

Proof. Let us start with the second statement. When j is added to S, then $w = p(j)$ must be the current vertex, and j is in the current bucket at v, that is, $D(w) \leq D(j) < D(w) + a(w)$. According to Lemma 4.6(ii), we have $D(w) + a(w) \leq D(v) + a(v)$. Thus, the second statement holds.

We now prove the first statement by induction on the number of iterations. The statement clearly holds at initialization: r is the only active vertex. $D(r) = 0$, $D(s) = 0$, and $D(i) = \infty$ for $i \in N \setminus \{s\}$. Assume $D(v) \leq D(j)$ holds at the beginning of the current iteration for every pair j and v such that $j \in N$, $v \in V$ is active, and $j \in \text{desc}(v, N)$. The label of an active vertex may only increase, and the label of a node may only decrease in the algorithm; we analyze the two cases separately.

Consider a pair of v and j such that $D(v)$ increases. $D(v)$ may only change when CV $= v$, and the current bucket at v is empty; the new value is set to $D'(v) = D(v) + a(v)$. We claim that $D'(v) \leq D(j)$ holds. If $j \in \text{children}(v)$, then this is true because the current bucket was empty. Otherwise, let $w \in \text{children}(v) \cap V$ be the vertex following v on the v–j path in the component hierarchy. Since the first bucket is empty, we must have $D(v) + a(v) \leq D(w)$. By induction, we have $D(w) \leq D(j)$; thus, $D'(v) \leq D(j)$ must still hold.

Consider now a pair v and j such that $D(j)$ decreases. This can happen in a call to UPDATE(i) such that $(i, j) \in E$ and $D'(j) = D(i) + c(i, j) < D(j)$. We need to show $D'(j) \geq D(v)$.

Let $z = \text{lca}(i, j)$; by the property of the component hierarchy, we have $c(i, j) \geq a(z)$. By induction, $D(z) \leq D(i)$, and thus $D(z) + a(z) \leq D(i) + c(i, j)$. Since z and v are both on the path from j to r, either $z \in \text{desc}(v, V)$ or $v \in \text{desc}(z, V)$.

If $v \in \text{desc}(z, V)$, then $D(v) \leq D(z) + a(z) \leq D(i) + c(i, j) = D'(j)$ using Lemma 4.6(i). If $z \in \text{desc}(v, V)$, then also $i \in \text{desc}(v, V)$, and thus $D(v) \leq D(i) < D(i) + c(i, j) = D'(j)$ by induction. This completes the proof of the first statement. □

Lemma 4.7. *For every vertex $v \in V$ and descendant $i \in \text{desc}(v)$, $d(i) < U(v)$.*

Proof. Let $D(.)$ denote the labels immediately prior to the activation of vertex v, and let $D'(.)$ be the labels immediately after activation. Then $D'(v) \leq D(v) < D'(v) + a(v)$. Let $i := \arg\min\{D(j) : j \in \text{desc}(v, N)\}$. Then $d(i) \leq D(i) = D(v)$. By Lemma 4.1, for all $j \in \text{desc}(v, N)$,

$$\begin{aligned} d(j) &\leq d(i) + \Gamma(v) \leq D(v) + \Gamma(v) \leq D(v) + \eta(v)a(v) \\ &\leq D'(v) + (\eta(v) + 1)a(v) - 1 = U(v) - 1\,. \end{aligned}$$

□

We are ready to prove Theorem 4.4.

Proof of Theorem 4.4. Lemma 4.5 provides the running time analysis. It remains to show that $D(i) = d(i)$ for every $i \in S$, and that the algorithm terminates with $N \subseteq S$.

We can use Lemma 4.2 to show that the algorithm correctly sets the labels inside S. For this, we need to verify the following two properties:

(a) For all $j \in S$ and $i \in N \setminus S$, $D(j) \leq D(i) + b(i, j)$.

(b) For all $j \in N \setminus S$, $D(j)$ is the length shortest path from s to j inside the node set $S \cup \{j\}$.

The proof of (b) follows the same argument as for Dijkstra's algorithm, see e.g. [1, Section 4.5]. Part (a) clearly holds in the first step when $S = \{r\}$. At the iteration when a node j is added to S, consider any $i \in N \setminus S$, $i \neq j$, and let $v = \mathrm{lca}(i, j)$. Then, Lemma 4.6 shows $D(j) - a(v) \leq D(v) \leq D(i)$. The claim follows since $b(i, j) \geq a(v)$ is a property of the component hierarchy.

It remains to show that $N \subseteq S$ at termination, i.e., at the iteration that sets $D(r) = U(r)$ and makes r permanent. For a contradiction, let $j \in N \setminus S$ at the this iteration. Let $P = i_1, i_2, \ldots, i_k$ (where $i_1 = s$ and $i_k = j$) be a shortest path from node s to node j. Clearly, $k \geq 2$, and without loss of generality, let us assume that each node i_t, $t \leq k - 1$ was added to S during the algorithm (or else we can replace j by the first node i_t of P not added to S).

In the iteration when $h = i_{k-1}$ was added to S, we had $D(h) = d(h)$ as shown above. Further, Update(i) updated $D(j)$ to $D(h) + c_{hj} = c(P) = d(h)$. Clearly, $D(h) = d(h)$ for the rest of the algorithm. According to Lemma 4.3, the final iteration has

$$U(r) = D(r) \leq D(h) = d(h) < U(r)\,,$$

where the first equality follows by the termination condition, and the last inequality by Lemma 4.7. This completes the proof. □

4.5 The Split/FindMin data structure

For each highest inactive vertex v, the algorithm needs to be able to compute $D(v) = \min\{D(j) : j \in \mathrm{desc}(v, N)\}$. To accomplish this, we will use the Split/FindMin data structure. Before reviewing this data structure, we note that in addition to computing $D(v)$ for highest inactive vertices, the data structure will need to be updated whenever either of the following algorithmic operations takes place:

- When a highest inactive vertex is activated, the subset $\mathrm{children}(v) \cap V$ all become highest inactive vertices.
- In step Update(i), if $D(j)$ is updated, then $D(v)$ should be updated for $v = \mathrm{HIA}(j)$.

The steps can be implemented using the Split/FindMin data structure. This was first introduced by Gabow [13] for the maximum weight matching problem, and can be stated as follows (see also [25]). The data structure is initialized with a sequence $E = \{e_1, \ldots, e_n\}$ of n weighted elements. At each iteration, there is a set $\mathcal{S}$, which is a partition of E into consecutive subsequences. For every element e_i, we maintain a key value $\kappa(e_i)$. At a given operation described below, we let $\mathcal{S}(e_i)$ denote the unique subsequence $\mathcal{S}$ that contains e_i. Note that $\mathcal{S}$ and $\mathcal{S}(e_i)$ are modified whenever a split operation is called.

The operations are as follows:

- init$(e_1, e_2, \ldots, e_n)$: Create a sequence set $\mathcal{S} \leftarrow \{(e_1, e_2, \ldots, e_n)\}$ with $\kappa(e_i) = \infty$ for all $i \in [n]$.
- split(e_i): For $\mathcal{S}(e_i) = (e_j, \ldots, e_{i-1}, e_i, \ldots, e_k)$, let $\mathcal{S} \leftarrow (\mathcal{S} \setminus \mathcal{S}(e_i)) \cup \{(e_j, \ldots, e_{i-1}), (e_i, \ldots, e_k)\}$.
- findmin(e_i): Return $\min\{\kappa(e_j) : e_j \in \mathcal{S}(e_i)\}$.
- descreasekey(e_i, w): Set $\kappa(e_i) \leftarrow \min\{\kappa(e_i), w\}$.

To use this data structure for our setting, we take the component hierarchy $(V \cup N, E, r, a)$, and impose an arbitrary ordering on the children of every vertex $v \in V$. This induces a total ordering on the set of leaves N; we index the node set $N = \{e_1, e_2, \ldots, e_n\}$ accordingly. Then, all sets $\mathrm{desc}(v)$ will correspond to contiguous subsequences of nodes. Initially, there are no active

vertices and E is the set of nodes. Then r is activated. In general, when a vertex v is activated, it corresponds to performing $|\text{children}(v)|-1$ splits on the nodes in $\text{desc}(v)$, resulting in a consecutive subsequence for each child of v. (The nodes in $\text{children}(v)$ correspond to subsequences of length 1.) Whenever $D(j)$ is updated, this corresponds to a decreasekey operation. Using the Split/FindMin data structure for the shortest path algorithm requires at most n findmin operations, at most $n-1$ splits, and at most m decreasekey operations.

For $O(n)$ split and $O(m)$ decreasekey operations with $m \geq n$, Gabow [13] gave an implementation in $O(m\alpha(m,n))$ total time in the comparison-addition model. This was improved by Thorup to $O(m)$ in the word RAM model, using the atomic heaps data structure by Fredman and Willard [10]. The original implementation of fusion trees permits all bitwise operations as well as multiplication. In a subsequent paper [32], Thorup showed how to implement fusion trees on a mild extension of the AC^0 model, thus avoiding the need for multiplication except for multiplication by powers of 2.

We note that the data Split/FindMin structure was also used in all subsequent papers on shortest path problems using the hierarchy approach [15, 23, 24, 26]. In the comparison-addition model, an improved bound $O(m \log \alpha(n,m))$ was given by Pettie [25].

5 Conclusions

In this paper, we have given an $O(mn)$ algorithm for the directed all pairs shortest paths problem with nonnegative integer weights. Our algorithm first replaces the cost function by a reduced cost satisfying an approximate balancing property in $O(m\sqrt{n}\log n)$ time. Subsequently, every shortest path computation can be done in linear time, by adapting Thorup's algorithm [31].

One might wonder if our technique may also lead to an improvement for APSP in the comparison-addition model, where the best running time is $O(mn + n^2 \log\log n)$ by Pettie [24]. This running time bound is based on multiple bottlenecks. However, as explained in Section 1.1.1, the approximate cost balancing is able to get around the sorting bottleneck of [24]. Using the $O(m \log \alpha(n,m))$ implementation of Split/FindMin, an overall $O(mn \log \alpha(n,m))$ might be achievable.

However, there is one remaining important bottleneck where our algorithm crucially relies on bit-shift operations: the operation MoveToBucket(j), which places a node/vertex in the bucket at $v = p(j)$ containing the value $D(j)$. Pettie and Ramachandran [26] show that these operations can be efficiently carried out in $O(1)$ amortized time per operation in a bucket-heap data structure, assuming the hierarchy satisfies certain *'balancedness'* property. Section 5 of the paper shows how the 'coarse hierarchy' obtainable from a minimum spanning tree and used by Thorup can be transformed to a 'balanced hierarchy'. This method does not seem to easily apply to the directed hierarchy concept used in this paper.

Our approximate min-balancing algorithm may be of interest on its own, and has strong connections to the matrix balancing literature as detailed in Section 1.1.2. For finding an $(1+\varepsilon)$-min-balanced reduced cost for $\varepsilon = O(1)$, our algorithm takes $O\left(\frac{1}{\varepsilon} m\sqrt{n}\log n\right)$ time. One might wonder if there is an algorithm with the same polynomial term $\tilde{O}(m\sqrt{n})$ but with a dependence on $\log(1/\varepsilon)$. We note that the algorithm in [28] for approximate max-balancing has a $\log(1/\varepsilon)$ dependence.

Acknowledgement The authors are very grateful to an anonymous referee. Their insightful comments lead to simplifications in some arguments and significant improvements in the presentation.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows – Theory, Algorithms and Applications*. Prentice Hall, 1993.

[2] Z. Allen-Zhu, Y. Li, R. Oliveira, and A. Wigderson. Much faster algorithms for matrix scaling. In *58th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 890–901, 2017.

[3] J. M. Altschuler and P. A. Parrilo. Near-linear convergence of the random Osborne algorithm for matrix balancing. *Mathematical Programming*, 2022. (to appear).

[4] T. Chan and R. Williams. Deterministic APSP, orthogonal vectors, and more. *ACM Transactions on Algorithms*, 17(1):1–14, 2021.

[5] M. B. Cohen, A. Madry, D. Tsipras, and A. Vladu. Matrix scaling and balancing via box constrained Newton's method and interior point methods. In *58th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 902–913, 2017.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[8] E. Dinic. Economical algorithms for finding shortest paths in a network. *Transportation Modeling Systems*, pages 36–44, 1978.

[9] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[10] M. Fredman and D. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[11] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[13] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *26th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 90–100, 1985.

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

[15] T. Hagerup. Improved shortest paths on the word RAM. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 61–72. Springer, 2000.

[16] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[17] S. T. McCormick. Approximate binary search algorithms for mean cuts and cycles. *Operations Research Letters*, 14(3):129–132, 1993.

[18] J. B. Orlin and R. K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Mathematical Programming*, 54(1-3):41–56, 1992.

[19] J. B. Orlin and A. Sedeño Noda. An $O(nm)$ time algorithm for finding the min length directed cycle in a graph. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1866–1879, 2017.

[20] E. Osborne. On pre-conditioning of matrices. *Journal of the ACM (JACM)*, 7(4):338–345, 1960.

[21] R. Ostrovsky, Y. Rabani, and A. Yousefi. Matrix balancing in l_p norms: bounding the convergence rate of Osborne's iteration. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 154–169, 2017.

[22] B. N. Parlett and C. Reinsch. Balancing a matrix for calculation of eigenvalues and eigenvectors. *Numerische Mathematik*, 13(4):293–304, 1969.

[23] S. Pettie. On the comparison-addition complexity of all-pairs shortest paths. In *International Symposium on Algorithms and Computation*, pages 32–43. Springer, 2002.

[24] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[25] S. Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. *Journal of Graph Algorithms and Applications*, 19(1):375–391, 2015.

[26] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing*, 34(6):1398–1431, 2005.

[27] H. Schneider and M. H. Schneider. Max-balancing weighted directed graphs and matrix scaling. *Mathematics of Operations Research*, 16(1):208–222, 1991.

[28] L. J. Schulman and A. Sinclair. Analysis of a classical matrix preconditioning algorithm. *Journal of the ACM (JACM)*, 64(2):1–23, 2017.

[29] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[30] R. E. Tarjan. *Data structures and network algorithms*, volume 44. SIAM, 1983.

[31] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[32] M. Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 699–707, 2003.

[33] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, 2004.

[34] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[35] R. R. Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the 46th ACM Symposium on Theory of Computing (STOC)*, pages 664–673, 2014.

[36] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.