Efficient Algorithms for Geometric Partial Matching

Pankaj K. Agarwal
Duke University, USA
pankaj@cs.duke.edu

Hsien-Chih Chang
Duke University, USA
hsienchih.chang@duke.edu

Allen Xiao
Duke University, USA
axiao@cs.duke.edu

Abstract

Let A and B be two point sets in the plane of sizes r and n respectively (assume $r \leq n$), and let k be a parameter. A matching between A and B is a family $M \subseteq A \times B$ of pairs so that any point of $A \cup B$ appears in at most one pair. Given two integers $p, q \geq 1$, we define the cost of M to be $\mathrm{cost}(M) = \sum_{(a,b)\in M} \|a - b\|_p^q$ where $\|\cdot\|_p$ is the L_p-norm. The geometric partial matching problem asks to find the minimum-cost size-k matching between A and B.

We present efficient algorithms for geometric partial matching that work for any powers of L_p-norm matching objective: An exact algorithm that runs in $O((n + k^2)\,\mathrm{polylog}\, n)$ time, and a $(1 + \varepsilon)$-approximation algorithm that runs in $O((n + k\sqrt{k})\,\mathrm{polylog}\, n \cdot \log \varepsilon^{-1})$ time. Both algorithms are based on the primal-dual flow augmentation scheme; the main improvements are obtained by using dynamic data structures to achieve efficient flow augmentations. Using similar techniques, we give an exact algorithm for the planar transportation problem that runs in $O((r^2\sqrt{n} + rn^{3/2})\,\mathrm{polylog}\, n)$ time. For $r = o(\sqrt{n})$, this algorithm is faster than the state-of-art near-quadratic time algorithm by Agarwal *et al.* [SOCG 2017].

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases partial matching, transportation, optimal transport, minimum-cost flow, bichromatic closest pair

Funding Work on this paper was supported by NSF under grants CCF-15-13816, CCF-15-46392, and IIS-14-08846, by an ARO grant W911NF-15-1-0408, and by BSF Grant 2012/229 from the U.S.-Israel Binational Science Foundation.

Acknowledgements We thank Haim Kaplan for discussion and suggestion to use Goldberg *et al.* [13] algorithm. We thank Debmalya Panigrahi for helpful discussions.

Lines 517

1 Introduction

Given two point sets A and B in $\mathbb{R}^2$, we consider the problem of finding the minimum-cost partial matching between A and B. Formally, suppose A has size r and B has size n where $r \leq n$. Let $G(A, B)$ be the undirected complete bipartite graph between A and B, and let the cost of edge (a, b) be $c(a, b) = \|a - b\|_p$, for some $1 \leq p < \infty$. A *matching* M in $G(A, B)$

The 35th International Symposium on Computational Geometry (SOCG 2019).
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is a set of edges sharing no endpoints. The *size* of M is the number of edges in M. Given $q \geq 1$, the cost of M is defined to be

$$\text{cost}(M) := \sum_{(a,b)\in M} \|a-b\|_p^q.$$

For a parameter $k \leq r$, the problem of finding the minimum-cost size-k matching in $G(A,B)$ is called the *geometric partial matching problem*. We call the corresponding problem in general bipartite graphs (with arbitrary edge costs) the *partial matching* problem.[1]

We also consider the following generalization of bipartite matching. Let $\lambda : A \cup B \to \mathbb{Z}$ be a *supply-demand function* with positive value on points of A and negative value on points of B, satisfying $\sum_{a\in A} \lambda(a) = -\sum_{b\in B} \lambda(b)$. A *transportation map* is a function $\tau : A \times B \to \mathbb{R}_{\geq 0}$ such that $\sum_{b\in B} \tau(a,b) = \lambda(a)$ for all $a \in A$ and $\sum_{a\in A} \tau(a,b) = -\lambda(b)$ for all $b \in B$. We define the cost of τ to be

$$\text{cost}(\tau) := \sum_{(a,b)\in A\times B} \|a-b\|_p^q \cdot \tau(a,b).$$

The *transportation problem* asks to compute a transportation map of minimum cost.

Related work Maximum-size bipartite matching is a classical problem in graph algorithms. Upper bounds include the $O(m\sqrt{n})$ time algorithm by Hopcroft and Karp [15] and the $O(m \min\{\sqrt{m}, n^{2/3}\})$ time algorithm by Even and Tarjan [11], where n is the number of vertices and m is the number of edges. The first improvement in over thirty years was made by Mądry [19], which uses an interior-point algorithm and runs in $O(m^{10/7} \operatorname{polylog} n)$ time.

The Hungarian algorithm [17] computes a minimum-cost maximum matching in a bipartite graph in roughly $O(mn)$ time. Faster algorithms have been developed, such as the $O(m\sqrt{n}\log(nC))$ time algorithms by Gabow and Tarjan [12] and the improved $O(m\sqrt{n}\log C)$ time algorithm by Duan *et al.* [9] assuming that the edge costs are integral; here C is the maximum cost of an edge. Ramshaw and Tarjan [21] showed that the Hungarian algorithm can be extended to compute a minimum-cost partial matching of size k in time $O(km + k^2 \log r)$ time. They also proposed a cost-scaling algorithm for partial matching that runs in time $O(m\sqrt{n}\log(nC))$, again assuming that costs are integral. By reduction to unit-capacity min-cost flow, Goldberg *et al.* [13] developed a cost-scaling algorithm for partial matching with running time $O(m\sqrt{k}\log(kC))$, again only for integral edge costs.

In geometric settings, the Hungarian algorithm can be implemented to compute an optimal perfect matching between A and B (assuming equal size) in time $O(n^2 \operatorname{polylog} n)$ [16] (see also [1, 26]). This algorithm computes an optimal size-k matching in time $O(kn \operatorname{polylog} n)$. Faster approximation algorithms have been developed for computing a perfect matching in geometric settings [4, 23, 26, 27]. For $q = 1$, the best algorithm to date by Sharathkumar and Agarwal [24] computes a $(1+\varepsilon)$-approximation to the optimal perfect matching in $O(n \operatorname{polylog} n \cdot \varepsilon^{-O(1)})$ expected time with high probability. Their algorithm can also compute a $(1+\varepsilon)$-approximate partial matching within the same time bound. For $q > 1$, the best known approximation algorithm to compute a perfect matching runs in $O(n^{3/2} \operatorname{polylog} n \log(n/\varepsilon))$ time [23]; it is not obvious how to extend this algorithm to the partial matching setting.

There is also some work on computing an optimal or near-optimal partial matching when B is fixed but A is allowed to translate and/or rotate [3, 5, 8, 22]. Here, the goal is to (i)

[1] Partial matching is also called *imperfect matching* or *imperfect assignment* [13, 21].

compute a (near-)optimal matching over all possible transformations of A, or (ii) to compute a set $\mathcal{M}$ of matchings, such that for any translation/rotation $\mathcal{T}$ of A, a (near-)optimal matching of $\mathcal{T}(A)$ and B in $\mathcal{M}$.

The transportation problem can also be formulated as a minimum-cost flow problem in a graph. The strongly polynomial uncapacitated min-cost flow algorithm by Orlin [20] solves the transportation problem in $O((m + n \log n)n \log n)$ time. Lee and Sidford [18] give a weakly polynomial algorithm that runs in $O(m\sqrt{n}\,\mathrm{polylog}(n, U))$ time, where U is the maximum amount of vertex supply-demand. Agarwal *et al.* [2] showed that Orlin's algorithm can be implemented to solve 2D transportation in time $O(n^2 \,\mathrm{polylog}\, n)$. It is not known whether the 2D transportation problem can be solved in $O(rn \,\mathrm{polylog}\, n)$ time. By adapting the Lee-Sidford algorithm, they developed a $(1 + \varepsilon)$-approximation algorithm that runs in $O(n^{3/2}\varepsilon^{-2} \,\mathrm{polylog}(n, U))$ time. They also gave a Monte-Carlo algorithm that computes an $O(\log^2(1/\varepsilon))$-approximate solution in $O(n^{1+\varepsilon})$ time with high probability.

Our results There are three main results in this paper. First in Section 2 we present an efficient algorithm for computing an optimal partial matching in $\mathbb{R}^2$.

▶ **Theorem 1.1.** *Given two point sets A and B in $\mathbb{R}^2$ each of size at most n, a minimum-cost matching of size k between A and B can be computed in $O((n + k^2) \,\mathrm{polylog}\, n)$ time.*

We use *bichromatic closest pair (BCP)* data structures to implement the Hungarian algorithm efficiently, similar to Agarwal *et al.* and Kaplan *et al.* [1, 16]. But unlike their algorithms which take $\Omega(n)$ time to find an augmenting path, we show that after $O(n \,\mathrm{polylog}\, n)$ preprocessing, an augmenting path can be found in $O(k \,\mathrm{polylog}\, n)$ time. The key is to recycle (rather than rebuild) our data structures from one augmentation to the next. We refer to this idea as the *rewinding mechanism*.

Next in Sections 3 and 4, we obtain a $(1 + \varepsilon)$-approximation algorithm for the geometric partial matching problem in $\mathbb{R}^2$ by providing an efficient implementation of the unit-capacity min-cost flow algorithm by Goldberg *et al.* [13].

▶ **Theorem 1.2.** *Given two point sets A and B in $\mathbb{R}^2$ each of size at most n, a $(1 + \varepsilon)$-approximate min-cost matching of size k between A and B can be computed in $O((n + k\sqrt{k}) \,\mathrm{polylog}\, n \cdot \log \varepsilon^{-1})$ time.*

The main challenge here is the set of *null vertices* which do not play any role in the augmentations, but still contribute to the size of the graph. Instead, we run the unit-capacity min-cost flow algorithm on a *shortcut graph*, circumventing all null vertices. The shortcut graph itself may have $\Omega(n^2)$ edges, but we can represent it implicitly and use a data structure to explore this graph. As such, we can implement each augmentation in the Goldberg *et al.*algorithm in time proportional to the size of the *flow support*, which turns out to be of size $O(k)$.

Finally in Section 5 we present a faster algorithm for the transportation problem in $\mathbb{R}^2$ when the two point sets are unbalanced.

▶ **Theorem 1.3.** *Given two point sets A and B in $\mathbb{R}^2$ of sizes r and n respectively along with supply-demand function $\lambda : A \cup B \to \mathbb{Z}$, an optimal transportation map between A and B can be computed in $O((r^2\sqrt{n} + rn^{3/2}) \,\mathrm{polylog}\, n)$ time.*

Our result improves over the $O(n^2 \,\mathrm{polylog}\, n)$ time algorithm in [2] when $r = o(\sqrt{n})$. The algorithm uses the strongly polynomial uncapacitated minimum-cost flow algorithm

by Orlin [20], adapted for geometric costs as in Agarwal *et al.* [2]. Unlike in the case of matchings, the flow support for the transportation problem may have size $\Omega(n)$ even when r is a constant; so naïvely we can no longer charge the execution time to flow support size. However, we show that most of the support arcs are of degree one and thus can be partitioned into *stars* centered at vertices of A. We describe a data structure that processes these stars in amortized $O((r^2/\sqrt{n} + r\sqrt{n}) \operatorname{polylog} n)$ time per augmentation.

2 Minimum-Cost Partial Matchings using Hungarian Algorithm

In this section, we solve the geometric partial matching problem and prove Theorem 1.1 by implementing the Hungarian algorithm for partial matching in $O((n + k^2) \operatorname{polylog} n)$ time.

The dual to the standard linear program for partial matching has dual variables for each vertex, called *potentials* π. Given potentials π, we can define the *reduced cost* on the edges to be $c_\pi(v, w) \coloneqq c(v, w) - \pi(v) + \pi(w)$. Potentials π are *feasible* if the reduced costs are nonnegative for all edges in G. We say that an edge (v, w) is *admissible* under potentials π if $c_\pi(v, w) = 0$.

A vertex v is *matched* by matching $M \subseteq E$ if v is the endpoint of some edge in M; otherwise v is *unmatched*. Given a matching M, an *augmenting path* $\Pi = (a_1, b_1, \ldots, a_\ell, b_\ell)$ is an odd-length path with unmatched endpoints (a_1 and b_ℓ) that alternates between edges outside and inside of M. The symmetric difference $M \oplus \Pi$ creates a new matching of size $|M| + 1$, called the *augmentation* of M by Π.

Fast implementation of Hungarian search The Hungarian algorithm is initialized with $M = \emptyset$ and $\pi = 0$. Each iteration of the Hungarian algorithm augments M with an admissible augmenting path Π, discovered using a procedure called the *Hungarian search*. The algorithm terminates after k augmentations, when $|M| = k$; Ramshaw and Tarjan [21] showed that M is guaranteed to be an optimal partial matching.

The Hungarian search grows a set of *reachable vertices* S *from* all unmatched $v \in A$ *using* augmenting paths of admissible edges. Initially, S is the set of unmatched vertices in A. Let the *frontier* of S be the edges in $(A \cap S) \times (B \setminus S)$. S is grown by *relaxing* an edge (a, b) in the frontier: adding b into S, and also modifying potentials to make (a, b) admissible, preserve c_π on other edges within S, and keep π feasible on edges outside of S. Specifically, the algorithm relaxes the minimum-reduced-cost frontier edge (a, b), and then raises $\pi(v)$ by $c_\pi(a, b)$ for all $v \in S$. If b is already matched, then we also relax the matching edge (a', b) and add a' into S. The search finishes when b is unmatched, and an admissible augmenting path now can be recovered.

2.1 Fast implementation of Hungarian search

In the geometric setting, we find the min-reduced-cost frontier edge using a dynamic *bichromatic closest pair* (BCP) data structure, as observed in [2, 26]. Given two point sets P and Q in the plane and a weight function $\omega : P \cup Q \to \mathbb{R}$, the BCP is two points $a \in P$ and $b \in Q$ minimizing the additively weighted distance $c(a, b) - \omega(a) + \omega(b)$. Thus, a minimum reduced-cost frontier edge is precisely the BCP of point sets $P = A \cap S$ and $Q = B \setminus S$, with $\omega = \pi$. Note that the "state" of this BCP is parameterized by S and π.

The dynamic BCP data structure by Kaplan *et al.* [16] supports point insertions and deletions in $O(\operatorname{polylog} n)$ time and answers queries in $O(\log^2 n)$ time for our setting. Outside of potential updates, each relaxation in the Hungarian search requires one query, one deletion, and at most one insertion. As $|M| \le k$ throughout, there are at most $2k$ relaxations in

each Hungarian search, and the BCP can be used to implement each Hungarian search in $O(k \operatorname{polylog} n)$ time. Explained shortly, there is an existing technique to handle potential updates without performing BCP updates for each one.

Rewinding mechanism. We cannot afford to take $O(n \operatorname{polylog} n)$ time to initialize the BCP data structure at the beginning of every Hungarian search beyond the first one. To resolve the issue, observe that exactly one vertex of A is newly matched after an augmentation. Thus (modulo potential changes), given the initial state of the BCP at the i-th Hungarian search, we can obtain the initial state for the $(i+1)$-th with a single BCP deletion operation. Suppose we log the sequence of points added to S in the i-th Hungarian search. Then, at the start of the $(i+1)$-th Hungarian search, we can *rewind* this log by undoing each BCP update operation in reverse order of the log to obtain the initial state of the i-th BCP. With one additional BCP delete, we have the initial state for the $(i+1)$-th BCP. The number of points in the log is $O(k)$, bounded by the number of relaxations per Hungarian search. Thus, in $O(k \operatorname{polylog} n)$ time we can recover the initial BCP data structure for each Hungarian search beyond the first. We refer to this procedure as the *rewinding mechanism*.

As for potential updates, we adapt an update batching trick from Vaidya [26] to work under the rewinding mechanism, so that the time spent on potential updates per Hungarian search is $O(k)$. See the full version for details. Putting everything together, our implementation of the Hungarian algorithm runs in $O((n+k^2) \operatorname{polylog} n)$ time. This proves Theorem 1.1.

3 Approximating Min-Cost Partial Matching through Cost-Scaling

In this section we present a $(1+\varepsilon)$-approximation algorithm for computing a geometric partial matching that runs in time $O((n+k\sqrt{k}) \operatorname{polylog} n \log(1/\varepsilon))$.

After introducing the necessary terminologies in Section 3.1, we reduce the partial matching problem to computing an approximate minimum-cost flow on a unit-capacity reduction network in Section 3.2. In Section 3.3 we prove a high-level overview of the cost-scaling algorithm, executed on the reduction network. We postpone the fast implementation using dynamic data structures to Section 4.

3.1 Preliminaries on Network Flows

Due to the space restriction, we omit the definitions of standard network flow theory terminologies from the main text. For a reference see Appendix B.1, or any texts on network flows [13, 20]. We emphasize that a directed graph $G = (V, E)$ is augmented by edge costs c and capacities u, and a supply-demand function $\phi : V \to \mathbb{Z}$. We call the positive values of $\phi(v)$ *supply*, and the negative values of $\phi(v)$ *demand*. We assume that the sum of supplies and demands is equal to zero. For Sections 3 and 4, we assume the capacity of every edge in G is 1, i.e. G is a unit capacity graph. A *network* $N = (V, \vec{E})$ turns each edge in E into a pair of *arcs* $v{\to}w$ and $w{\to}v$ in arc set $\vec{E}$. With the unit-capacity assumption on the network, all the pseudoflows in this section take integer values. The *support* of a pseudoflow f in N, is the set of arcs with positive flows: $\operatorname{supp}(f) := \{v{\to}w \in \vec{E} \mid f(v{\to}w) > 0\}$. Given a pseudoflow f, we define the *imbalance* of a vertex to be $\phi_f(v) := \phi(v) + \sum_{w\to v\in\vec{E}} f(w{\to}v) - \sum_{v\to w\in\vec{E}} f(v{\to}w)$. If all vertices are *balanced*, the pseudoflow is a *circulation*. The *cost* of a pseudoflow is defined to be

$$\operatorname{cost}(f) := \sum_{v\to w\in\operatorname{supp}(f)} c(v{\to}w) \cdot f(v{\to}w).$$

The *minimum-cost flow problem (MCF)* asks to find a circulation of minimum cost inside a given directed graph.

⟨⟨**Move to when used**⟩⟩ The *bottleneck value* of a flow is

$$\text{bottleneck}(f) \coloneqq \max_{v\to w \in \text{supp}(f)} c(v\to w).$$

The *minimum bottleneck value* for a network is the minimum bottleneck value over all circulations.

LP-duality and admissibility. We solve the min-cost flow problem using primal-dual algorithms. Let $G = (V, E)$ be a given directed graph with the corresponding network $N = (V, \vec{E}, c, u, \phi)$. Formally, the *potentials* $\pi(v)$ are the variables of the linear program dual to the standard linear program for the min-cost flow problem with variables $f(v\to w)$ for each directed edge in E. Assignments to the primal variables satisfying the capacity constraints extend naturally into a pseudoflow on the network N. Let $G_f = (V, \vec{E}_f)$ be the residual graph under pseudoflow f. The *reduced cost* of an arc $v\to w$ in $\vec{E}_f$ with respect to π is defined as

$$c_\pi(v\to w) \coloneqq c(v\to w) - \pi(v) + \pi(w).$$

Notice that the cost function c_π is also antisymmetric.

The *dual feasibility constraint* says that $c_\pi(v\to w) \geq 0$ holds for every directed edge (v, w) in E; potentials π which satisfy this constraint are said to be *feasible*. Suppose we relax the dual feasibility constraint to allow some small violation in the value of $c_\pi(v\to w)$. We say that a pair of pseudoflow f and potential π is *θ-optimal* [6, 25]if $c_\pi(v\to w) \geq -\theta$ for every residual arc $v\to w$ in $\vec{E}_f$. Pseudoflow f is *θ-optimal* if it is θ-optimal with respect to some potentials π; potential π is *θ-optimal* if it is θ-optimal with respect to some pseudoflow f. Given a pseudoflow f and potentials π, a residual arc $v\to w$ in $\vec{E}_f$ is *admissible* if $c_\pi(v\to w) \leq 0$. We say that a pseudoflow g in G_f is *admissible* if all support arcs of g on G_f are admissible; in other words, $g(v\to w) > 0$ holds only on admissible arcs $v\to w$. Throughout the rest of the paper we make use of the property that performing an admissible flow augmentation preserves θ-optimality. To our knowledge this is folklore; see Lemma B.1 for a proof.

3.2 Reduction to Unit-Capacity Min-Cost Flow Problem

Here, we reduce the min-cost partial matching problem to finding a θ-optimal circulation in a unit-capacity min-cost flow instance.

Additive approximation. Given a bipartite graph $G = (A, B, E_0)$ for the geometric partial matching problem with cost function c, we construct the *reduction graph* H as follows: Direct the edges in E_0 from A to B, and assign each directed edge with capacity 1. Now add a dummy vertex s with directed edges to all vertices in A, and add a dummy vertex t with directed edges from all vertices in B; each edge added has cost 0 and capacity 1. Assign vertex s with supply k and vertex t with demand $-k$; the rest of the vertices in H have zero supply-demand. We call the network corresponding to H the *reduction network* N_H. It is straightforward to show that any integer circulation f on N_H uses exactly k of the A-to-B arcs, which correspond to the edges of a size-k matching M_f. Notice that the cost of the circulation f is equal to the cost of the corresponding matching M_f.

First we show that the number of arcs used by any integer pseudoflow in N_H is bounded by the excess of the pseudoflow.

▶ **Lemma 3.1.** *The size of* $\mathrm{supp}(f)$ *is at most* $3k$ *for any integer circulation* f *in reduction network* N_H*. As a corollary, the number of residual backward arcs is at most* $3k$*.*

Using the bound on the support size, we show that a θ-optimal integral circulation gives an additive $O(k\theta)$-approximation to the MCF problem.

▶ **Lemma 3.2.** *Let* f *be a* θ*-optimal integer circulation in* N_H*, and* f^* *be an optimal integer circulation for* N_H*. Then,* $\mathrm{cost}(f) \le \mathrm{cost}(f^*) + 6k\theta$*.*

Multiplicative approximation. We employ a technique from Sharathkumar and Agarwal [23] to convert the additive approximation into a multiplicative one. The reduction does not work out of the box, as they were tackling a similar but different problem on geometric transportations. See Appendix B.2 for details.

▶ **Lemma 3.3.** *Computing* $(1+\varepsilon)$*-approximate geometric partial matching reduces to the following problem in* $O(n \operatorname{polylog} n)$ *time: Given a fixed* α*, compute an* $(\alpha\varepsilon/6k)$*-optimal circulation on a reduction network* N *with min-bottleneck value at most* $n^q\alpha$*.*

3.3 High-Level Description of Cost-Scaling Algorithm

The main body of our algorithm is an implementation of the unit-capacity min-cost flow algorithm of Goldberg *et al.* [13]. Their algorithm is based on the *cost-scaling* technique, originally due to Goldberg and Tarjan [14]. The algorithm finds θ-optimal circulations for geometrically shrinking values of θ. Each fixed value of θ is called a *cost scale*. Once θ is sufficiently small, the θ-optimal flow is a suitable approximation according to Lemma 3.3.[2]

In what follows, we use θ^* to denote the "target" scale of the cost-scaling (that is, we seek a θ^*-optimal circulation). The cost-scaling algorithm initializes the flow f and the potential π to be zero.

⟨⟨**Unclear; rewrite**⟩⟩ The initial value of the scale parameter, denoted θ_0, is set to C by the original algorithm [13], but this does not work well with the multiplicative approximation reduction (Lemma 3.3). Instead, we use

$$\theta_0 \in 2 \cdot \{x \in \mathbb{R}_{\ge 0} \mid \text{there exists a circulation supported on edges of cost} \le x\},$$

or in the terminology of Lemma 3.3, $\theta_0 = 2n^q\alpha$. We show that this is sufficient for correctness in Lemma 3.4. Choosing this type of θ_0 is important for turning the cost-scaling algorithm into a multiplicative approximation — explained in Section B.2 — where we also give the precise method for choosing α. Note that the minimum possible x is the *minimum bottleneck value* of the network.

Within each scale, the algorithm performs the following:

- Scale-Init takes the previous circulation (now 2θ-optimal) and transforms it into an θ-optimal pseudoflow with $O(k)$ excess. (For the first scale when $\theta = \theta_0$, Scale-Init does nothing.
- Refine then reduces the excess to zero while maintaining θ-optimality, turning f into a θ-optimal circulation.

[2] When the costs are integers, a θ-optimal circulation for a sufficiently small θ (say less than $1/n$) is itself an optimal solution [13, 14]. We present this algorithm without the integral-cost assumption because in the geometric partial matching setting (with respect to L_p norms) the costs are generally not integers.

If $\theta \leq \theta^*$, then f is a θ^*-optimal circulation and we are done. Otherwise, θ is halved and the next scale begins. The algorithm produces a θ^*-optimal circulation after $O(\log(\theta_0/\theta^*))$ scales. Using the reduction in Lemma 3.3, we have an initial scale $\theta_0 = 2n^q\alpha$ and target scale $\theta^* = \varepsilon\alpha/6k$. Thus, the number of cost scales after the reduction is $O(\log(n/\varepsilon))$. ⟨⟨**log(*kn*/*ε*)?**⟩⟩

Scale initialization. For Scale-Init, we use a simple extension to a scale initialization procedure proposed for partial matching [13, Section 6.1]. Namely, the potential changes are the same, but we take the additional step to remove arcs violating θ-optimality. The vertex set of H consists of two point sets A and B, as well as two dummy vertices s and t. The directed edges in H are pointed from s to A, from A to B, and from B to t. We call those arcs in N_H whose direction is consistent with their corresponding directed edges as *forward arcs*, and those arcs that points in the opposite direction as *backward arcs*.

The procedure Scale-Init transforms a 2θ-optimal circulation from the previous cost scale into a θ-optimal flow with $O(k)$ excess, by raising the potentials π of all vertices in A by θ, those in B by 2θ, and the potential of t by 3θ. The potential of s remains unchanged. Now the reduced cost of every forward arc is increased by θ, and thus all the forward arcs have reduced cost at least $-\theta$.

As for backward arcs, the procedure Scale-Init continues by setting the flow on $v \to w$ to zero for each backward arc $w \to v$ violating the θ-optimality constraint. In other words, we set $f(v \to w) = 0$ whenever $c_\pi(w \to v) < -\theta$. This ensures that all such backward arcs are no longer residual, and therefore the flow (now with excess) is θ-optimal.

Because the arcs are of unit-capacity in N_H, each arc desaturation creates one unit of excess. By Lemma 3.1 the number of backward arcs is at most $3k$. Thus the total amount of excess created is also $O(k)$. In total, the whole procedure Scale-Init takes $O(n)$ time.

Refinement. The procedure Refine is implemented using a primal-dual augmentation algorithm, which sends flows on admissible arcs to reduce the total excess. Unlike the Hungarian algorithm, it uses *blocking flows* instead of augmenting paths. We call a pseudoflow f on residual network N_g a *blocking flow* if f saturates at least one residual arc in every augmenting path in N_g. In other words, there is no admissible augmenting path in N_{f+g} from an excess vertex to a deficit vertex.

Each iteration of Refine finds an admissible blocking flow that is then added to the current pseudoflow in two stages:

1. A *Hungarian search*, which increases the dual variables π of vertices that are reachable from an excess vertex by at least θ, in a Dijkstra-like manner, until there is an excess-deficit path of admissible edges.
2. A *depth-first search* through the set of admissible arcs to construct a blocking flow. It suffices to repeatedly extract admissible augmenting paths until no more admissible excess-deficit paths remain.

The algorithm continues until the total excess becomes zero.

First we analyze the number of iterations executed by Refine. The proof follows the strategy in Goldberg *et al.* [13, Section 3.2], which proves a $O(\sqrt{n})$ bound on the number of iterations for general unit-capacity networks. They claimed an analogous $O(\sqrt{k})$ bound for partial matchings in [13, Section 6.1], which we reproduce for completeness. Due to space constraint we have moved the proofs to Appendix B.3.

In the following lemma, we prove the number of iterations of Refine required at each cost scale. This is also the (only) place where our assumption on θ_0 become important. ⟨⟨**the**

following?⟩ We require that θ_0 is sufficiently high such that a circulation with bottleneck value at most $\theta_0/2$ exists in N_H.

▶ **Lemma 3.4.** *Suppose that θ_0 is chosen so that a circulation in N_H with bottleneck value at most $\theta_0/2$ exists. At any scale, let f be a pseudoflow in N_H with $O(k)$ excess. The procedure* Refine *runs for $O(\sqrt{k})$ iterations before the excess of f becomes zero.*

4 Fast Implementation of Refinement

The goal of the section is to show that after $O(n \operatorname{polylog} n)$ time preprocessing, each Hungarian search and depth-first search can be implemented in $O(k \operatorname{polylog} n)$ time. Combined with Lemma 3.4 the procedure Refine can be implemented in $O((n + k\sqrt{k}) \operatorname{polylog} n)$ time. Together with our analysis on scale initialization and the bound on number of cost scales, this concludes the proof to Theorem 1.2.

The Hungarian search and depth-first search are similar, traversing through the residual graph using admissible arcs starting from the excess vertices. Each step of the search procedures *relaxes* a minimum-reduced-cost arc from the set of visited vertices to an unvisited vertex, until a deficit vertex is reached. At a high level, our analysis strategy is to charge the relaxation events to the support arcs of f.

4.1 Null vertices and shortcut graph

As it turns out, there are some vertices visited by a relaxation event which we cannot charge to $\operatorname{supp}(f)$. Unfortunately the number of such vertices can be as large as $\Omega(n)$. To overcome this issue, we replace the residual graph with an equivalent graph that excludes these *null vertices*, and run the Hungarian search and depth-first search on the resulting graph instead.

Null vertices. We say a vertex v in the residual graph N_f is a *null vertex* if $\phi_f(v) = 0$ and no arc of $\operatorname{supp}(f)$ is incident to v. We use $A_\emptyset$ and $B_\emptyset$ to denote the null vertices A and B respectively. Vertices that are not null are called *normal vertices*. A *null 2-path* is a length-2 path in N_f from a normal vertex to another normal vertex, passing through a null vertex. As every vertex in A has in-degree 1 and every vertex in B has out-degree 1 in the residual graph, the null 2-paths must be of the form either (s, v, b) for some vertex b in $B \setminus B_\emptyset$ or (a, v, t) for some vertex a in $A \setminus A_\emptyset$. In either case, we say that the null 2-path *passes through* null vertex v. Similarly, we define the length-3 path from s to t that passes through two null vertices to be a *null 3-path*. Because reduced costs telescope for residual paths, the reduced cost of any null path does not depend on the potentials of the null vertices it passes through.

Shortcut graph. We construct the *shortcut graph* $\tilde{H}_f$ from the reduction graph H by removing all null vertices and their incident edges, followed by inserting an arc from the head of each each null path Π to its tail, with cost equals to the sum of costs on the arcs. We call this arc the *shortcut* of null path Π, denoted as $\operatorname{short}(\Pi)$. The resulting multigraph $\tilde{H}_f$ contains only normal vertices of H_f, and the reduced cost of any path between normal vertices are preserved. We argue now that $\tilde{H}_f$ is fine as a surrogate for H_f. Let $\tilde{\pi}$ be a θ-optimal potential on $\tilde{H}_f$. Construct potentials π on H_f which extends $\tilde{\pi}$ to null vertices, by setting $\pi(a) \coloneqq \tilde{\pi}(s)$ for $a \in A_\emptyset$ and $\pi(b) \coloneqq \tilde{\pi}(t)$ for $b \in B_\emptyset$.

▶ **Lemma 4.1.** *Consider $\tilde{\pi}$ a θ-optimal potential on $\tilde{H}_f$ and π the corresponding potential constructed on H_f. Then, (1) potential π is θ-optimal on H_f, and (2) if arc $\operatorname{short}(\Pi)$ is admissible under $\tilde{\pi}$, then every arc in Π is admissible under π.*

4.2 Dynamic data structures for search procedures

Hungarian search. Conceptually, we execute the Hungarian search on the shortcut graph $\tilde{H}_f$. We describe how we can query the minimum-reduced-cost arc leaving $\tilde{S}$ in $O(\text{polylog}\, n)$ time for $\tilde{H}_f$, without constructing it explicitly. For this purpose, let S be a set of "reached" vertices maintained, identical to $\tilde{S}$ except whenever a shortcut is relaxed, we add the null vertices passed by the corresponding null path to S in addition to its (normal) endpoints. Observe that the arcs of $\tilde{H}_f$ leaving $\tilde{S}$ fall into $O(1)$ categories:

- non-shortcut backward arcs (v, w) with $(w, v) \in \text{supp}(f)$;
- non-shortcut A-to-B forward arcs;
- non-shortcut forward arcs from s-to-A and from B-to-t;
- shortcut arcs (s, b) corresponding to null 2-paths from s to $b \in (B \setminus B_\emptyset) \setminus S$;
- shortcut arcs (a, t) corresponding to null 2-paths from $a \in (A \setminus A_\emptyset) \cap S$ to t; and
- shortcut arcs (s, t) corresponding to null 3-paths.

For each category of arcs we maintain a proper data structure (either heap or BCP) to answer to the min-cost arc query.

Depth-first search. Depth-first search is similar to Hungarian search in that it uses the relaxation of minimum-reduced-cost arcs/null paths, this time to identify admissible arcs/null paths in a depth-first manner. Similar to the Hungarian search, for each category of arcs in $\tilde{H}_f$ leaving $\tilde{S}$, we maintain a proper data structure to answer the minimum-reduced cost arc leaving a *fixed* vertex in $\tilde{S}$ given by the query. Thus unlike Hungarian search which uses BCP data structures, we use dynamic nearest-neighbor data structures instead [16]. Each data structure requires $O(1)$ queries and updates per relaxation. So in collaboration each relaxation can be implemented in $O(\text{polylog}\, n)$ time [16].

Time analysis. The complete time analysis can be found in Appendix C.2, C.3, and C.4; here we sketch the ideas. First we show (in Appendix C.2) that both Hungarian search and depth-first search performs $O(k)$ relaxations before a deficit vertex is reached, by looking at shortcut and non-shortcut relaxations separately. Both types of relaxations are eventually charged to the support size of f. As for the time analysis (see Appendix C.3), using the same rewinding mechanism as in Section 2.1, the running time of the Hungarian search and depth-first search, other than the potential updates, can be charged to the number of relaxations. Again using the trick by Vaidya [26] we can charge the potential updates of normal vertices to the number of relaxations in the Hungarian search. We never explicitly maintain the potentials on the null vertices; instead they are reconstructed whenever needed, either at the end of each iteration of refinement or when an augmentation sends flow through a null vertex. We show that such updates does not happen often in Appendix C.4. This completes the time analysis, which we summarize as follows:

▶ **Lemma 4.2.** *After $O(n \,\text{polylog}\, n)$-time preprocessing, each Hungarian search and depth-first search can be implemented in $O(k \,\text{polylog}\, n)$ time.*

5 Unbalanced Transportation

In this section, we give an exact algorithm which solves the planar transportation problem in $O((r^2\sqrt{n} + rn^{3/2})\,\text{polylog}\, n)$ time, proving Theorem 1.3. Our strategy is to use the standard reduction to the uncapacitated min-cost flow problem, and provide a fast implementation

under the geometric setting for the uncapacitated min-cost flow algorithm by Orlin [20], combined with some of the tools developed in Sections 2 and 3.

For lack of space, we only sketch Orlin's strongly polynomial-time algorithm for uncapacitated min-cost flow problem [20]. See Appendix D.1 for a brief introduction, as well as the original paper. In short, Orlin's algorithm follows the *excess-scaling* paradigm under the primal-dual framework: Maintain a *scale parameter* Δ, initially set to U. A vertex v is *active* if $|\phi_f(v)| \geq \alpha\Delta$ for a fixed parameter $\alpha \in (0.5, 1)$. Repeatedly run a *Hungarian search* that raises potentials (while maintaining dual feasibility) to create an admissible augmenting excess-deficit path between active vertices, on which we perform flow augmentations. Once there are no more active excess or deficit vertices, Δ is halved. Each sequence of augmentations where Δ holds a constant value is called an *excess scale*. On top of that, the algorithm performs contraction on arcs with flow value at least $3n\Delta$ at the beginning of a scale, in which case the flow and potentials are no longer tracked, as well as aggressive Δ-lowering under circumstances.

▶ **Theorem 5.1 (Orlin [20, Theorems 2 and 3]).** *Orlin's algorithm finds a set of optimal potentials after $O(n \log n)$ scaling phases and $O(n \log n)$ total augmentations.*

The remainder of the section focuses on showing that each augmentation can be implemented in $O((r^2/\sqrt{n} + r\sqrt{n}) \operatorname{polylog} n)$ time (after preprocessing). A subtle issue is that our geometric data structures must deal with real points in the plane instead of the contracted components. A solution is provided by Agarwal *et al.* [2]; for the sake of completeness, we describe the method for maintaining contractions under dynamic data structures in Appendix D.2.

Recovering optimal flow. Using a strategy from Agarwal *et al.* [2], we can recover the optimal flow in time $O(n \operatorname{polylog} n)$. If furthermore the cost function is just the p-norm (without the qth-power), we show a structural result which is interesting on its own right: the set of admissible arcs under an optimal potential forms a planar graph. Thus, we can extract the admissible network in $O(n \operatorname{polylog} n)$ time thanks to the sparsity of planar graphs, and compute the flow using planar multiple-source multiple-sink maximum-flow algorithm by Borradaile *et al.* [7] which runs in $O(n \log^3 n)$ time. For details see Appendix D.3 and D.4.

5.1 Support stars

To find an augmenting path, we again use a Hungarian search with geometric data structures to perform relaxations quickly. Our strategy is summarized as follows:

- Discard vertices that lead to dead ends in the search (not on a path to a deficit vertex).
- Cluster parts of the flow support, such that the number of support arcs outside clusters is $O(r)$. The number of relaxations we perform is proportional to the number of support arcs outside of clusters.

Querying/updating clusters degrades our amortized time per relaxation from $O(\operatorname{polylog} n)$ to $O(\sqrt{n} \operatorname{polylog} n)$. Thus overall each augmentation takes $O(r\sqrt{n} \operatorname{polylog} n)$ time.

Support stars. The vertices of B with support degree 1 are partitioned into subsets $\Sigma_a \subset B$ by the $a \in A$ lying on the other end of their single support arc. We call Σ_a the *support star* centered at $a \in A$.

Roughly speaking, we would like to handle each support star as a single unit. When the Hungarian search reaches a or any $b \in \Sigma_a$, the entirety of Σ_a (as well as a) is also admissibly-reachable and can be included into S without further potential updates. Additionally, the

only outgoing residual arcs of every $b \in \Sigma_a$ lead to a, thus the only way to leave $\Sigma_a \cup \{a\}$ is through an arc leaving a. Once a relaxation step reaches some $b \in \Sigma_a$ or a itself, we would like to quickly update the state such that the rest of $b \in \Sigma_a$ is also reached without performing relaxation steps to each individual $b \in \Sigma_a$.

5.2 Implementation details

Before describing our workaround for support stars, we analyze the number of relaxation steps for arcs outside of support stars. To this end we need to strip of some *dead* vertices—having no incident flow support edges and not an active excess or deficit vertex—that does not affect the search. We use A_ℓ and B_ℓ to denote vertices of points in A and B that are not dead. The details for handling such vertices can be found in Appendix D.5. For a proof of the following lemma, see Appendix D.6.

▶ **Lemma 5.2.** *Suppose we have stripped the graph of dead vertices. The number of relaxation steps in a Hungarian search outside of support stars is $O(r)$.*

Relaxations outside support stars. For relaxations that don't involve support star vertices, we can once again maintain a BCP data structure to query the minimum A_ℓ-to-B_ℓ arc. To elaborate, this is the BCP between $P = A_\ell \cap S$ and $Q = (B_\ell \setminus (\bigcup_{a \in A_\ell} \Sigma_a)) \setminus S$, weighted by potentials. Since the query is outside the support stars, there is at most one update per relaxation. Backward (support) arcs are kept admissible by the invariant, so we relax them immediately when they arrive at the frontier.

Relaxing support stars. We classify support stars into two categories: *big stars* with $|\Sigma_a| > \sqrt{n}$, and *small stars* with $|\Sigma_a| \le \sqrt{n}$. Let $A_{big} \subseteq A$ denote the centers of big stars and and $A_{small} \subseteq A$ denote the centers of small stars.

- For each big star Σ_a, we use a data structure $\mathcal{D}_{big}(a)$ to maintain BCP between $P = A_\ell \cap S$ and $Q = \Sigma_a$. We query this until $a \in S$ or any vertex of Σ_a is added to S.
- All small stars are added to a single BCP data structure $\mathcal{D}_{small}$ between $P = A_\ell \cap S$ and $Q = (\bigcup_{a \in A_{\text{small}}} \Sigma_a) \setminus S$. When an $a \in A_{\text{small}}$ or any vertex of its support star is added to S, we remove the points of Σ_a from $\mathcal{D}_{\text{small}}$ using $|\Sigma_a|$ deletion operations.

We will update these data structures as each support star center is added into S. If a relaxation step adds some $b \in B_\ell$ and b is in a support star Σ_a, then we immediately relax $b \to a$, as all support arcs are admissible.

Suppose a relaxation step adds $a \in A_\ell$ to S. We must (i) remove a from every $\mathcal{D}_{\text{big}}$, (ii) remove a from $\mathcal{D}_{\text{small}}$. If $a \in A_{\text{big}}$, we also (iii) deactivate $\mathcal{D}_{\text{big}}(a)$. If $a \in A_{\text{small}}$, we also (iv) remove the points of Σ_a from $\mathcal{D}_{\text{small}}$. The operations (i–iii) can be performed in $O(\text{polylog}\, n)$ time each, but (iv) may take up to $O(\sqrt{n}\,\text{polylog}\, n)$ time. On the other hand, there are now $O(\sqrt{n})$ data structures to query during each relaxation step, which takes $O(\sqrt{n}\log^2 n)$ time in total. Together with Lemma 5.2 we bound the time for each Hungarian search.

▶ **Lemma 5.3.** *Hungarian search takes $O(r\sqrt{n}\,\text{polylog}\, n)$ time.*

Updating support stars. As the flow support changes, the membership of support stars may shift causing a big star to become small or vice versa. To efficiently support this, we introduce a soft boundary in determining whether a support star is big or small. Standard charging argument shows that the amortized update time per membership change is $O(\text{polylog}\, n)$ for big-to-small updates, and $O((r/\sqrt{n})\,\text{polylog}\, n)$ for small-to-big ones; see Appendix D.7. The

number of star membership changes by adding a single augmenting path is bounded above by twice of its length, so $O(r)$. By Theorem 5.1, the total number of membership changes is $O(rn \log n)$. The total time spent on big-to-small updates is $O(rn \operatorname{polylog} n)$, and the total time spent on small-to-big updates is $O(r^2\sqrt{n} \operatorname{polylog} n)$. Membership changes themselves can be performed in $O(\operatorname{polylog} n)$ time each.

Preprocessing time. It takes $O(rn \operatorname{polylog} n)$ time to build the very first set of data structures. There are $r \cdot |\Sigma_a|$ points to insert for each $\mathcal{D}_{\text{big}}(a)$, so the number of points to insert is $O(rn)$. At most $O(rn)$ points have to be inserted for $\mathcal{D}_{\text{small}}$. So the total preprocessing time is $O(rn \operatorname{polylog} n)$.

Between searches. After each augmentation, we reset the data structures to their initial state plus the change from augmentation using rewinding mechanism (see Section 2.1). This takes time proportional to the time for Hungarian search, which is $O(r\sqrt{n} \operatorname{polylog} n)$ by Lemma 5.3. The most recent augmentation may have deactivated $O(1)$ active excess and deficit vertices, which takes $O(\sqrt{n} \operatorname{polylog} n)$ time to update. Finally, we note that an augmenting path cannot reduce the support degree of a vertex to zero, and therefore no new dead vertices are created by augmentation.

Between excess scales. When the excess scale changes, vertices that were previously inactive may become active, and vertices that were dead may be revived. If we have the data structures built at the end of the previous scale, then we can add in each new active vertex $a \in A$ and charge the insertion to the (future) augmenting path or contraction which eventually causes the vertex being inactive or absorbed. By Theorem 5.1, there are $O(n \log n)$ such newly active vertices; each of them takes $O(\sqrt{n} \operatorname{polylog} n)$ to update. So the total time spent is $O(n^{3/2} \operatorname{polylog} n)$.

Putting it together. After $O(rn \operatorname{polylog} n)$ preprocessing, we spend $O(r\sqrt{n} \operatorname{polylog} n)$ time on relaxations each Hungarian search by Lemma 5.3, for a total of $O(rn^{3/2} \operatorname{polylog} n)$ time over the course of the algorithm. Rewinding takes the same amount of time. We spend up to $O((r^2\sqrt{n}+rn) \operatorname{polylog} n)$ time switching stars between big/small. We spend $O(n^{3/2} \operatorname{polylog} n)$ time activating and reviving vertices. Adding, the algorithm takes $O((r^2\sqrt{n}+rn^{3/2}) \operatorname{polylog} n)$ time to produce optimal potentials π^*, from which we can recover f^* in $O(n \operatorname{polylog} n)$ additional time. This completes the proof of Theorem 1.3.

References

1. Pankaj K. Agarwal, Alon Efrat, and Micha Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM J. Comput.* 29(3):912–953, 1999. ⟨https://doi.org/10.1137/S0097539795295936⟩.
2. Pankaj K. Agarwal, Kyle Fox, Debmalya Panigrahi, Kasturi R. Varadarajan, and Allen Xiao. Faster algorithms for the geometric transportation problem. *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, 7:1–7:16, 2017. ⟨https://doi.org/10.4230/LIPIcs.SoCG.2017.7⟩.
3. Pankaj K. Agarwal, Haim Kaplan, Geva Kipper, Wolfgang Mulzer, Günter Rote, Micha Sharir, and Allen Xiao. Approximate minimum-weight matching with outliers under translation. *CoRR* abs/1810.10466, 2018. ⟨http://arxiv.org/abs/1810.10466⟩.
4. Pankaj K. Agarwal and Kasturi R. Varadarajan. A near-linear constant-factor approximation for euclidean bipartite matching? *Proceedings of the 20th ACM Symposium on*

Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004, 247–252, 2004. ⟨https://doi.org/10.1145/997817.997856⟩.

5 Rinat Ben Avraham, Matthias Henze, Rafel Jaume, Balázs Keszegh, Orit E. Raz, Micha Sharir, and Igor Tubis. Partial-matching RMS distance under translation: Combinatorics and algorithms. *Algorithmica* 80(8):2400–2421, 2018. ⟨https://doi.org/10.1007/s00453-017-0326-0⟩.

6 D. Bertsekas and D. El Baz. Distributed asynchronous relaxation methods for convex network flow problems. *SIAM Journal on Control and Optimization* 25(1):74–85, 1987. ⟨https://doi.org/10.1137/0325006⟩.

7 Glencora Borradaile, Philip N. Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM J. Comput.* 46(4):1280–1303, 2017. ⟨https://doi.org/10.1137/15M1042929⟩.

8 Sergio Cabello, Panos Giannopoulos, Christian Knauer, and Günter Rote. Matching point sets with respect to the earth mover's distance. *Comput. Geom.* 39(2):118–133, 2008. ⟨https://doi.org/10.1016/j.comgeo.2006.10.001⟩.

9 Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for approximate and exact maximum weight matching. *CoRR* abs/1112.0790, 2011. ⟨http://arxiv.org/abs/1112.0790⟩.

10 Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM* 19(2):248–264, 1972. ⟨https://doi.org/10.1145/321694.321699⟩.

11 Shimon Even and Robert E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.* 4(4):507–518, 1975. ⟨https://doi.org/10.1137/0204043⟩.

12 Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.* 18(5):1013–1036, 1989. ⟨https://doi.org/10.1137/0218069⟩.

13 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, and Robert E. Tarjan. Minimum-cost flows in unit-capacity networks. *Theory Comput. Syst.* 61(4):987–1010, 2017. ⟨https://doi.org/10.1007/s00224-017-9776-7⟩.

14 Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.* 15(3):430–466, 1990. ⟨https://doi.org/10.1287/moor.15.3.430⟩.

15 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2(4):225–231, 1973. ⟨https://doi.org/10.1137/0202019⟩.

16 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, 2495–2504, 2017. ⟨https://doi.org/10.1137/1.9781611974782.165⟩.

17 Harold W Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 2(1-2):83–97. Wiley Online Library, 1955.

18 Yin Tat Lee and Aaron Sidford. Following the path of least resistance : An Õ(m sqrt(n)) algorithm for the minimum cost flow problem. *CoRR* abs/1312.6713, 2013. ⟨http://arxiv.org/abs/1312.6713⟩.

19 Aleksander Mądry. Navigating central path with electrical flows: From flows to matchings, and back. *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, 253–262, 2013. ⟨https://doi.org/10.1109/FOCS.2013.35⟩.

20 James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research* 41(2):338–350, 1993. ⟨https://doi.org/10.1287/opre.41.2.338⟩.

21 Lyle Ramshaw and Robert E. Tarjan. A weight-scaling algorithm for min-cost imperfect matchings in bipartite graphs. *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, 581–590, 2012. ⟨https://doi.org/10.1109/FOCS.2012.9⟩.

22 Günter Rote. Partial least-squares point matching under translations. *Proc. 26th European Workshop Comput. Geom. (EuroCG 2010)* 249–251, 2010.

23 R. Sharathkumar and Pankaj K. Agarwal. Algorithms for the transportation problem in geometric settings. *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, 306–317, 2012. ⟨http://portal.acm.org/citation.cfm?id=2095145&CFID=63838676&CFTOKEN=79617016⟩.

24 R. Sharathkumar and Pankaj K. Agarwal. A near-linear time ϵ-approximation algorithm for geometric bipartite matching. *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, 385–394, 2012. ⟨https://doi.org/10.1145/2213977.2214014⟩.

25 Éva Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica* 5(3):247–256, 1985. ⟨https://doi.org/10.1007/BF02579369⟩.

26 Pravin M. Vaidya. Geometry helps in matching. *SIAM J. Comput.* 18(6):1201–1225, 1989. ⟨https://doi.org/10.1137/0218080⟩.

27 Kasturi R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, 320–331, 1998. ⟨https://doi.org/10.1109/SFCS.1998.743466⟩.

A Missing Details and Proofs from Section 2

A.1 Potential updates

We modify a trick from Vaidya [26] to batch potential updates. Potentials have a *stored value*, i.e. the currently recorded value of $\pi(v)$, and a *true value*, which may have changed from $\pi(v)$. The resulting algorithm queries the minimum-reduced-cost under the true values of π and updates the stored value occasionally.

Throughout the entire Hungarian algorithm, we maintain a nonnegative scalar δ (initially set to 0) which aggregates potential changes. Vertices $a \in A$ that are added to S are inserted into BCP with weight $\omega(a) \leftarrow \pi(a) - \delta$, for whatever value δ is at the time of insertion. Similarly, vertices $b \in B$ that are added to S have $\omega(b) \leftarrow \pi(b) - \delta$ recorded ($B \cap S$ points aren't added into a BCP set). When the Hungarian search wants to raise the potentials of points in S, δ is increased by that amount instead. Thus, true value for any potential of a point in S is always $\omega(p) + \delta$. For points of $(A \cup B) \setminus S$, the true potential is equal to the stored potential. Since all the points of $A \cap S$ have weights uniformly offset from their true potentials, the minimum edge returned by the BCP does not change. ⟪**why?**⟫

Once a point is removed from S (i.e. by an augmentation or the rewinding mechanism), we update its stored potential $\pi(p) \leftarrow \omega(p) + \delta$, again for the current value of δ. Most importantly, δ is not reset at the end of a Hungarian search and persists through the entire algorithm. Thus, the initial BCP sets constructed by the rewinding mechanism have true potentials accurately represented by δ and $\omega(p)$.

We update δ once per edge relaxations; thus $O(k)$ times in total per Hungarian search. There are $O(k)$ stored values updated per Hungarian search during the rewinding process. The time spent on potential updates per Hungarian search is therefore $O(k)$.

B Missing Details and Proofs from Section 3

B.1 Preliminaries on Network Flows

Network. Let $G = (V, E)$ be a directed graph, augmented by edge costs c and capacities u, and a supply-demand function ϕ defined on the vertices. We can turn the graph G into a *network* $N = (V, \vec{E})$: For each directed edge (v, w) in E, insert two *arcs* $v{\to}w$ and $w{\to}v$ into the arc set $\vec{E}$; the *forward arc* $v{\to}w$ inherits the capacity and cost from the directed graph G, while the *backward arc* $w{\to}v$ satisfies $u(w{\to}v) = 0$ and $c(w{\to}v) = -c(v{\to}w)$. This we ensure that the graph $(V, \vec{E})$ is *symmetric* and the cost function c is *antisymmetric* on N. The positive values of $\phi(v)$ are referred to as *supply*, and the negative values of $\phi(v)$ as *demand*. We assume that all capacities are nonnegative, all supplies and demands are integers, and the sum of supplies and demands is equal to zero. A *unit-capacity* network has all its edge capacities equal to 1. In this section we assume all networks are of unit-capacity.

Pseudoflows. Given a network $N := (V, \vec{E}, c, u, \phi)$, a *pseudoflow* (or *flow* to be short) $f\colon \vec{E} \to \mathbb{Z}$ on N is an antisymmetric function on the arcs of N satisfying $f(v{\to}w) \le u(v{\to}w)$ for every arc $v{\to}w$. [3] We say that f *saturates* an arc $v{\to}w$ if $f(v{\to}w) = u(v{\to}w)$; an arc $v{\to}w$ is *residual* if $f(v{\to}w) < u(v{\to}w)$. The *support* of f in N, denoted as $\mathrm{supp}(f)$, is the set of

[3] In general the pseudoflows are allowed to take real-values. Here under the unit-capacity assumption any optimal flows are integer-valued.

arcs with positive flows:

$$\mathrm{supp}(f) := \left\{ v{\to}w \in \vec{E} \mid f(v{\to}w) > 0 \right\}.$$

Given a pseudoflow f, we define the *imbalance* of a vertex (with respect to f) to be

$$\phi_f(v) := \phi(v) + \sum_{w{\to}v \in \vec{E}} f(w{\to}v) - \sum_{v{\to}w \in \vec{E}} f(v{\to}w).$$

We call positive imbalance *excess* and negative imbalance *deficit*; and vertices with positive (resp. negative) imbalance *excess* (resp. *deficit*) *vertices*. A vertex is *balanced* if it has zero imbalance. If all vertices are balanced, the pseudoflow is a *circulation*. The *cost* of a pseudoflow is defined to be

$$\mathrm{cost}(f) := \sum_{v{\to}w \in \mathrm{supp}(f)} c(v{\to}w) \cdot f(v{\to}w).$$

The *minimum-cost flow problem (MCF)* asks to find a circulation of minimum cost inside a given network.

Residual graph. Given a pseudoflow f, the *residual network* is defined as follows. Recall that the set of *residual arcs* $\vec{E}_f$ under f are those arcs $v{\to}w$ satisfying $f(v{\to}w) < u(v{\to}w)$. In other words, an arc that is not saturated by f is a residual arc; similarly, given an arc $v{\to}w$ with positive flow value, the backward arc $w{\to}v$ is a residual arc.

Let $N = (V, \vec{E}, c, u, \phi)$ be a network constructed from graph G, with a pseudoflow f on N. The *residual graph* G_f of f has V as its vertex set and $\vec{E}_f$ as its arc set. The *residual capacity* u_f with respect to pseudoflow f is defined to be $u_f(v{\to}w) := u(v{\to}w) - f(v{\to}w)$. Observe that the residual capacity is always nonnegative. We can define residual arcs differently using residual capacities:

$$\vec{E}_f = \{ v{\to}w \mid u_f(v{\to}w) > 0 \}.$$

In other words, the set of residual arcs are precisely those arcs in the residual graph, each of which has nonzero residual capacity.

Admissible flow augmentation.

▶ **Lemma B.1.** *Let f be an ε-optimal pseudoflow in G and let f' be an admissible flow in G_f. Then $f + f'$ is also θ-optimal.*

Proof. Augmentation by f' will not change the potentials, so any previously θ-optimal arcs remain θ-optimal. However, it may introduce new arcs $v{\to}w$ with $u_{f+f'}(v{\to}w) > 0$, that previously had $u_f(v{\to}w) = 0$. We will verify that these arcs satisfy the θ-optimality condition.

If an arc $v{\to}w$ is newly introduced this way, then by definition of residual capacities $f(v{\to}w) = u(v{\to}w)$. At the same time, $u_{f+f'}(v{\to}w) > 0$ implies that $(f+f')(v{\to}w) < u(v{\to}w)$. This means that f' augmented flow in the reverse direction of $v{\to}w$, that is, $f'(w{\to}v) > 0$. By assumption, the arcs of $\mathrm{supp}(f')$ are admissible, so $w{\to}v$ was an admissible arc ($c_\pi(w{\to}v) \le 0$). By antisymmetry of reduced costs, this implies $c_\pi(v{\to}w) \ge 0 \ge -\theta$. Therefore, all arcs with $u_{f+f'}(v, w) > 0$ respect the θ-optimality condition, and thus $f + f'$ is θ-optimal. ◀

▶ **Lemma 3.1.** *The size of* $\mathrm{supp}(f)$ *is at most $3k$ for any integer circulation f in reduction network N_H. As a corollary, the number of residual backward arcs is at most $3k$.*

Proof. Because f is a circulation, $\text{supp}(f)$ can be decomposed into k paths from s to t. Each s-to-t path in N_H is of length three, so the size of $\text{supp}(f)$ is at most $3k$. As every backward arc in the residual network must be induced by positive flow in the opposite direction, the total number of residual backward arcs is at most $3k$. ◀

▶ **Lemma 3.2.** *Let f be a θ-optimal integer circulation in N_H, and f^* be an optimal integer circulation for N_H. Then, $\text{cost}(f) \leq \text{cost}(f^*) + 6k\theta$.*

Proof. By Lemma 3.1, the total number of backward arcs in the residual network N_f is at most $3k$. Consider the residual flow in N_f defined by the difference between f^* and f. Since both f and f^* are both circulations and N_H has unit-capacity, the flow $f - f^*$ is comprised of unit flows on a collection of edge-disjoint residual cycles $\Gamma_1, \ldots, \Gamma_\ell$. Observe that each residual cycle Γ_i must have exactly half of its arcs being backward arcs, and thus we have $\sum_i |\Gamma_i| \leq 6k$.

Let π be some potential certifying that f is θ-optimal. Because Γ_i is a residual cycle, we have $c_\pi(\Gamma_i) = c(\Gamma_i)$ since the potential terms telescope. We then see that

$$\text{cost}(f) - \text{cost}(f^*) = \sum_i c(\Gamma_i) = \sum_i c_\pi(\Gamma_i) \geq \sum_i (-\theta) \cdot |\Gamma_i| \geq -6k\theta,$$

where the second-to-last inequality follows from the θ-optimality of f with respect to π. Rearranging the terms we have that $\text{cost}(f) \leq \text{cost}(f^*) + 6k\theta$. ◀

B.2 Multiplicative approximation

⟨⟨**To be compressed.**⟩⟩ The reduction in Sharathkumar and Agarwal [23] uses a *single-linkage clustering* of G—the clustering induced by a low-cost subgraph of the minimum spanning tree (MST) on G—to transform the instance into a number of bounded-diameter point sets "preserving" M^*. We also perform a single-linkage clustering, but use it to find a good upper bound on the minimum bottleneck value of the reduction network instead.

Let T be the minimum spanning tree on input graph G and order its edges by increasing length as $e_1, \ldots, e_{r+n-1}$; both T and the sorted edges can be constructed in $O(n \operatorname{polylog} n)$ time using a dynamic data structure for planar nearest neighbors. ⟨⟨**HC: Don't mix construction with analysis.**⟩⟩ Let T_i denote the connected components of T induced by the first i edges $(e_1, \ldots, e_i)$ of T. Abusing notation, we also use T_i to denote the clustering of $A \cup B$ induced by the components of T_i. For each cluster $K \in T_i$, we use A_K and B_K to denote the points of $A \cap K$ and $B \cap K$ respectively.

We say a partial matching M is *intra-cluster* with respect to a clustering of $A \cup B$ if no edge of M has endpoints in two different clusters. Otherwise, the matching is *inter-cluster*. Let i^* be the smallest index such that T_{i^*} contains an intra-cluster partial matching. We can compute i^* in $O(n)$ time by traversing the edges $e_1, \ldots, e_{r+n-1}$ in ascending order, tracking the number of A and B points in each cluster defined by T_i. Specifically, for each $i \in 1, \ldots, (r + n - 1)$ let

$$\text{count}(i) \coloneqq \sum_{K \in T_i} \min(A_K, B_K),$$

which is the size of the largest intra-cluster matching possible under T_i. The first index for which $\text{count}(i) \geq k$ is i^*.

▶ **Lemma B.2.**

(i) $c(e_{i^*}) \leq \text{cost}(M^*)$

(ii) *The minimum bottleneck value for the reduction network on $A \cup B$ is at most $n^q \cdot c(e_{i^*})$.*

Proof. First, by definition of i^*, M^* is inter-cluster in T_{i^*-1}. Although M^* need not be intra-cluster in T_{i^*}, this implies that at least one edge $e \in M^*$ has $c(e) \geq c(e_{i^*})$. It follows that $c(e_{i^*}) \leq \text{cost}(M^*)$.

Second, let M be the intra-cluster k-matching purported to exist in T_{i^*}. Each component of T_{i^*} has first-power diameter at most $n\|e_{i^*}\|$, so the longest edge in M has cost at most $n^q \cdot c(e_{i^*})$. In the reduction network N, M corresponds to a circulation where the most expensive arc costs at most $n^q \cdot c(e_{i^*})$; the minimum bottleneck value of N is at most that. ◀

With Lemma B.2, we can complete the proof of Lemma 3.3.

Proof of Lemma 3.3. We can determine i^* and $c(e_{i^*})$ in $O(n \operatorname{polylog} n)$ time, as described above. Fix $\alpha = c(e_{i^*})$. By Lemma B.2.ii the minimum bottleneck value of the reduction network is at most $n^q\alpha$. Computing an $(\alpha\varepsilon/6k)$-optimal circulation produces a partial matching of cost at most

$$\text{cost}(M^*) + 6k(\alpha\varepsilon/6k) = \text{cost}(M^*) + \varepsilon \cdot c(e_{i^*}) \leq (1+\varepsilon)\,\text{cost}(M^*)$$

where the inequality follows from Lemma B.2.i. ◀

B.3 Number of iterations during refinement.

To this end we need a bound on the size of the support of f right before and throughout the execution of Refine. This bound will also be useful later for bounding the overall running time.

▶ **Lemma B.3.** *Let f be an integer pseudoflow in N_H with $O(k)$ excess. Then, the size of the support of f is at most $O(k)$.*

Proof. Observe that the reduction graph H is a directed acyclic graph, and thus the support of f does not contain a cycle. Now $\text{supp}(f)$ can be decomposed into a set of inclusion-maximal paths, each of which contributes a single unit of excess to the flow if the path does not terminate at t or if more than k paths terminate at t. By assumption, there are $O(k)$ units of excess to which we can associate to the paths, and at most k paths (those that terminate at t) that we cannot associate with a unit of excess. The length of any such paths is at most three by construction of the reduction graph H. Therefore we can conclude that the number of arcs in the support of f is $O(k)$. ◀

▶ **Corollary B.4.** *The size of $\text{supp}(f)$ is at most $O(k)$ for pseudoflow f right before or during the execution of* Refine.

▶ **Lemma 3.4.** *Suppose that θ_0 is chosen so that a circulation in N_H with bottleneck value at most $\theta_0/2$ exists. At any scale, let f be a pseudoflow in N_H with $O(k)$ excess. The procedure* Refine *runs for $O(\sqrt{k})$ iterations before the excess of f becomes zero.*

Proof. Let f_0 and π_0 be the circulation and potential at the end of the previous cost scale. For the first cost scale, let $\pi_0 := 0$ and f_0 be any circulation with bottleneck value $\leq \theta_0/2$. In both cases, f_0 is a 2θ-optimal circulation with respect to π_0. Let f and π be the current flow and the potential. Let $d(v)$ defined to be the amount of potential increase at v since π_0, measured in units of θ; in other words, $d(v) := (\pi(v) - \pi_0(v))/\theta$.

Now divide the iterations executed by the procedure Refine into two phases: The transition from the first phase to the second happens when every excess vertex v has $d(v) \geq \sqrt{k}$. At most $\sqrt{k}$ iterations belong to the first phase as each Hungarian search

increases the potential π by at least θ for each excess vertex (and thus increases $d(v)$ by at least one).

The number of iterations belonging to the second phase is upper bounded by the amount of total excess at the end of the first phase, because each subsequent push of a blocking flow reduces the total excess by at least one. We now show that the amount of such excess is at most $O(\sqrt{k})$. Consider the set of arcs $E^+ \coloneqq \{v{\to}w \mid f(v{\to}w) < f_0(v{\to}w)\}$. The total amount of excess is upper bounded by the number of arcs in E^+ that crosses an arbitrarily given cut X that separates the excess vertices from the deficit vertices, when the network has unit-capacity [13, Lemma 3.6]. Consider the set of cuts $X_i \coloneqq \{v \mid d(v) > i\}$ for $0 \le i < \sqrt{k}$; every such cut separates the excess vertices from the deficit vertices at the end of first phase. Each arc in E^+ crosses at most 3 cuts of type X_i [13, Lemma 3.1]. So there is one X_i crossed by at most $3|E^+|/\sqrt{k}$ arcs in E^+. The size of E^+ is bounded by the sum of support sizes of f and f_0; by Corollary B.4 the size of E^+ is $O(k)$. This implies an $O(\sqrt{k})$ bound on the total excess after the first phase, which in turn bounds the number of iterations in the second phase. ◀

C Missing Details and Proofs from Section 4

▶ **Lemma 4.1.** *Consider $\tilde{\pi}$ a θ-optimal potential on $\tilde{H}_f$ and π the corresponding potential constructed on H_f. Then, (1) potential π is θ-optimal on H_f, and (2) if arc* short(Π) *is admissible under $\tilde{\pi}$, then every arc in Π is admissible under π.*

Proof. Reduced costs for any arc from a normal vertex another is unchanged under either $\tilde{\pi}$ or π. Recall that a null path is comprised of one A-to-B arc, and one or two zero-cost arcs (connecting the null vertex/vertices to s and/or t). With our choice of null vertex potentials, we observe that the zero-cost arcs still have zero reduced cost. It remains to prove that an arbitrary ⟨⟨**residual?**⟩⟩ arc (a, b) ⟨⟨**arc or directed edge?**⟩⟩ satisfies the θ-optimality condition and admissibility when either a or b is a null vertex.

By construction of the shortcut graph, there is always a null path Π that contains (a, b). Observe that $c_\pi(a, b) = c_\pi(\Pi)$, independent to the type of null path. Again by construction, $c_\pi(\Pi) = c_{\tilde{\pi}}(\text{short}(\Pi))$, so we have $c_\pi(a, b) = c_{\tilde{\pi}}(\text{short}(\Pi)) \ge -\theta$. Additionally, if short(Π) is admissible under $\tilde{\pi}$, then so is (a, b) under π. This proves the lemma. ◀

C.1 Dynamic data structures for search procedures

Here we formally describe in details the set of dynamic data structure we use for the Hungarian search and depth-first search procedures.

For Hungarian search, we maintain the following for each type of outgoing arcs of $\tilde{H}_f$ leaving $\tilde{S}$:

1. Non-shortcut backward arcs (v, w) with $(w, v) \in \text{supp}(f)$. For these, we can maintain a min-heap on supp(f) arcs as each v arrives in $\tilde{S}$.
2. Non-shortcut A-to-B forward arcs. For these, we can use a BCP data structure between $(A \setminus A_\emptyset) \cap \tilde{S}$ and $(B \setminus B_\emptyset) \setminus \tilde{S}$, weighted by potential.
3. Non-shortcut forward arcs from s-to-A and from B-to-t. For s, we can maintain a min-heap on the potentials of $B \setminus \tilde{S}$, queried while $s \in \tilde{S}$. For t, we can maintain a max-heap on the potentials of $A \cap \tilde{S}$, queried while $t \notin \tilde{S}$.
4. Shortcut arcs (s, b) corresponding to null 2-paths from s to $b \in (B \setminus B_\emptyset) \setminus S$. For these, we maintain a BCP data structure with $P = A_\emptyset$, $Q = (B \setminus B_\emptyset) \setminus S$ with weights $\omega(p) = \pi(s)$

for all $p \in P$, and $\omega(q) = \pi(q)$ for all $q \in Q$. A response (a, b) corresponds to th null 2-path (s, a, b). This is only queried while $s \in S$.

5. Shortcut arcs (a, t) corresponding to null 2-paths from $a \in (A \setminus A_\emptyset) \cap S$ to t. For these, we maintain a BCP data structure with $P = (A \setminus A_\emptyset) \cap S$, $Q = B_\emptyset \setminus S$ with weights $\omega(p) = \pi(p)$ for all $p \in P$, and $\omega(q) = \pi(t)$ for all $q \in Q$. A response (a, b) corresponds to th null 2-path (a, b, t). This is only queried while $t \notin S$.
6. Shortcut arcs (s, t) corresponding to null 3-paths. For these, we maintain in a BCP data structure with $P = A_\emptyset \setminus S$, $Q = B_\emptyset \setminus S$ with weights $\omega(p) = \pi(s)$ for all $p \in P$, and $\omega(q) = \pi(t)$ for all $q \in Q$. A response (a, b) corresponds to th null 3-path (s, a, b, t). This is only queried while $s \in S$ and $t \notin S$.

For depth-first search, we maintain the following for each type of outgoing arcs of $\tilde{H}_f$ leaving $\tilde{S}$:

1. Non-shortcut backward arcs (v', w') with $(w', v') \in \mathrm{supp}(f)$. For these, we can maintain a min-heap on $(w', v') \in \mathrm{supp}(f)$ arcs for each normal $v' \in V$.
2. Non-shortcut A-to-B forward arcs. For these, we maintain a NN data structure over $P = (B \setminus B_\emptyset) \setminus \tilde{S}$, with weights $\omega(p) = \pi(p)$ for each $p \in P$. We subtract $\pi(v')$ from the NN distance to recover the reduced cost of the arc from v'.
3. Non-shortcut forward arcs from s-to-A and from B-to-t. For s, we can maintain a min-heap on the potentials of $B \setminus \tilde{S}$, queried only if $v' = s$. For B-to-t arcs, there is only one arc to check if $v' \in B$, which we can examine manually.
4. Shortcut arcs (s, b) corresponding to null 2-paths from s to $b \in (B \setminus B_\emptyset) \setminus S$. For these, we maintain a NN data structure with $P = A_\emptyset$, $Q = (B \setminus B_\emptyset) \setminus S$ with weights $\omega(p) = \pi(s)$ for all $p \in P$, and $\omega(q) = \pi(q)$ for all $q \in Q$. A response (a, b) corresponds to th null 2-path (s, a, b). This is only queried if $v' = s$.
5. Shortcut arcs (a, t) corresponding to null 2-paths from $a \in (A \setminus A_\emptyset) \cap S$ to t. For these, we maintain a NN data structure over $P = B_\emptyset \setminus S$ with weights $\omega(p) = \pi(t)$ for each $p \in P$. A response (v', b) corresponds to th null 2-path (v', b, t). We subtract $\pi(v')$ from the NN distance to recover the reduced cost of the arc from v'. This is not queried if $t \in \tilde{S}$.
6. Shortcut arcs (s, t) corresponding to null 3-paths. For these, we maintain in a NN data structure with $P = A_\emptyset \setminus S$, $Q = B_\emptyset \setminus S$ with weights $\omega(p) = \pi(s)$ for all $p \in P$, and $\omega(q) = \pi(t)$ for all $q \in Q$. A response (a, b) corresponds to th null 3-path (s, a, b, t). This is only queried while $v' = s$ and $t \notin S$.

C.2 Number of relaxations

First we bound the number of relaxations performed by both the Hungarian search and the depth-first search.

▶ **Lemma C.1.** *Hungarian search performs $O(k)$ relaxations before a deficit vertex is reached.*

Proof. ⟨⟨TO BE REWRITTEN.⟩⟩ First we prove that there are $O(k)$ non-shortcut relaxations. Each edge relaxation adds a new vertex to S, and non-shortcut relaxations only add normal vertices. The vertices of $V \setminus S$ fall into several categories: (i) s or t, (ii) vertices of A or B with 0 imbalance, and (iii) deficit vertices of A or B (S contains all excess vertices). The number of vertices in (i) and (iii) is $O(k)$, leaving us to bound the number of (ii) vertices.

An A or B vertex with 0 imbalance must have an even number of $\mathrm{supp}(f)$ edges. There is either only one positive-capacity incoming arc (for A) or outgoing arc (for B), so this quantity is either 0 or 2. Since the vertex is normal, this must be 2. We charge 0.5 to each of the two

supp(f) arcs; the arcs of supp(f) have no more than 1 charge each. Thus, the number of type (ii) vertex relaxations is $O(|\operatorname{supp}(f)|)$. By Corollary B.4, $O(|\operatorname{supp}(f)|) = O(k)$.

Next we prove that there are $O(k)$ shortcut relaxations. Recall the categories of shortcuts from the list of data structures above. We have shortcuts corresponding to (i) null 2-paths surrounding $a \in A_\emptyset$, (ii) null 2-paths surrounding $b \in B_\emptyset$, and (iii) null 3-paths, which go from s to t. There is only one relaxation of type (iii), since t can only be added to S once. The same argument holds for type (ii).

Each type (i) relaxation adds some normal $b \in B \setminus B_\emptyset$ into S. Since b is normal, it must either have deficit or an adjacent arc of supp(f). We charge this relaxation to b if it is deficit, or the adjacent arc of supp(f) otherwise. No vertex is charged more than once, and no supp(f) edge is charged more than twice, therefore the total number of type (i) relaxations is $O(|\operatorname{supp}(f)|)$. By Corollary B.4, $O(|\operatorname{supp}(f)|) = O(k)$. ◀

Similarly we can prove that there are $O(k)$ relaxations during the DFS.

▶ **Corollary C.2.** *Depth-first search performs $O(k)$ relaxations before a deficit vertex is reached.*

C.3 Time analysis

Now we complete the time analysis by showing that each Hungarian search and depth-first search can be implemented in $O(k \operatorname{polylog} n)$ time after a one-time $O(n \operatorname{polylog} n)$-time preprocessing.

There are only $O(1)$ data structures for both. The rewinding mechanism and potential update scheme of Section 2 to reset for each Hungarian search in time $O(\operatorname{polylog} n)$ times the number of relaxations, giving the following. The main difference is that we are using blocking flows rather than augmenting paths, so the initial S (i.e. excess vertices) may change by more than one element for each Refine iteration. The total number of such changes throughout Refineis $O(k)$, however, since Refine begins with $O(k)$ excess and does not gain more. We obtain the following:

▶ **Lemma C.3.** *After $O(n \operatorname{polylog} n)$-time preprocessing, each Hungarian search can be implemented in $O(k \operatorname{polylog} n)$ time.*

▶ **Lemma C.4.** *After $O(n \operatorname{polylog} n)$-time preprocessing, each depth-first search can be implemented in $O(k \operatorname{polylog} n)$ time.*

▶ **Lemma C.5.** *The rewinding mechanism takes in total $O(k \operatorname{polylog} n)$ time in* Refine.

C.4 Number of potential updates on null vertices

In our implementation of Refine, we do not explicitly construct $\tilde{H}_f$; instead we query its edges using BCP/NN oracles and min/max heaps on elements of H_f. Potentials on the null vertices are only required right before an augmentation sends a flow through a null path, making the null vertices it passes normal. We use the construction from Lemma 4.1 to obtain potential π on H_f such that the flow f is both θ-optimal and admissible with respect to π.

Size of blocking flows. Now we bound the total number arcs whose flow is updated by a blocking flow during the course of Refine. This bounds both the time spent updating the flow on these arcs and also the time spent on null vertex potential updates (Lemma C.7).

▶ **Lemma C.6.** *The support of each blocking flow found in* Refine *is of size $O(k)$.*

Proof. Let i be fixed and consider the invocation of DFS which produces the i-th blocking flow f_i. DFS constructs f_i as a sequence of admissible excess-deficit paths. Let one of these paths be P. Every arc in P is an arc relaxed by DFS, so N_i is bounded by the number of relaxations performed in DFS. Using Corollary C.2, we have $N_i = O(k)$. ◀

▶ **Lemma C.7.** *The number of end-of-*Refine *null vertex potential updates is $O(n)$. The number of augmentation-induced null vertex potential updates in each invocation of* Refine *is $O(k \log k)$.*

Proof. The number of end-of-Refine potential updates is $O(n)$. Each update due to flow augmentation involves a blocking flow sending positive flow through an null path, causing a potential update on the passed null vertex. We charge this potential update to the edges of that null path, which are in turn arcs with positive flow in the blocking flow. For each blocking flow, no positive arc is charged more than twice. It follows that the number of augmentation-induced updates is at most the size of support of the blocking flow, which is $O(k)$ by Lemma C.6. According to Lemma 3.4 there are $O(\sqrt{k})$ iterations of Refine before it terminates. Summing up we have an $O(k\sqrt{k})$ bound over the course of Refine. ◀

Now combining Lemma 3.3, Lemma 3.4, Lemma C.3, Lemma C.4, and Lemma C.7 completes the proof of Theorem 1.2.

D Missing Details and Proofs from Section 5

D.1 Uncapacitated MCF by excess scaling

We give an outline of the strongly polynomial-time algorithm for uncapacitated min-cost flow problem from Orlin [20]. Orlin's algorithm follows an *excess-scaling* paradigm originally due to Edmonds and Karp [10]. Consider the basic primal-dual framework used in the previous sections: The algorithm begins with both flow f and potentials π set to zero. Repeatedly run a *Hungarian search* that raises potentials (while maintaining dual feasibility) to create an admissible augmenting excess-deficit path, on which we perform flow augmentations. In terms of cost, f is maintained to be 0-optimal with respect to π and each augmentation over admissible edges preserves 0-optimality (by Lemma B.1). Thus, the final circulation must be optimal. The excess-scaling paradigm builds on top of this skeleton by specifying (i) between which excess and deficit vertices we send flows, and (ii) how much flow is sent by the augmentation.

The excess-scaling algorithm maintains a *scale parameter* Δ, initially set to U. A vertex v with $|\phi_f(v)| \geq \Delta$ is called *active*. Each augmenting path is chosen between an active excess vertex and an active deficit vertex. Once there are no more active excess or deficit vertices, Δ is halved. Each sequence of augmentations where Δ holds a constant value is called an *excess scale*. There are $O(\log U)$ excess scales before $\Delta < 1$ and, by integrality of supplies/demands, f is a circulation.

With some modifications to the excess-scaling algorithm, Orlin [20] obtains a strongly polynomial bound on the number of augmentations and excess scales. First, an *active* vertex is redefined to be one satisfying $|\phi_f(v)| \geq \alpha\Delta$, for a fixed parameter $\alpha \in (0.5, 1)$. Second, arcs with flow value at least $3n\Delta$ at the beginning of a scale are *contracted* to create a new vertex, whose supply-demand is the sum of those on the two endpoints of the contracted arc. We use $\hat{G} = (\hat{V}, \hat{E})$ to denote the resulting *contracted graph*, where each $\hat{v} \in \hat{V}$ is a contracted component of vertices from V. Intuitively, the flow is so high on contracted arcs that no set of future augmentations can remove the arc from $\mathrm{supp}(f)$. Third, in additional

to halving, Δ is aggressively lowered to $\max_{v \in V} \phi_f(v)$ if there are no active excess vertices and $f(v \to w) = 0$ holds for every arc $v \to w \in \hat{E}$. Finally, flow values are not tracked within contracted components, but once an optimal circulation is found on $\hat{G}$, optimal potentials π^* can be *recovered* for G by sequentially undoing the contractions. The algorithm then performs a post-processing step which finds the optimal circulation f^* on G by solving a max-flow problem on the set of admissible arcs under π^*.

D.2 Implementing contractions

⟨⟨**REWRITE**⟩⟩ Following Agarwal *et al.* [2], our geometric data structures deals with real points in the plane instead of the contracted components. We will track the contracted components described in $\hat{G}$ (e.g. with a disjoint-set data structure) and mark the arcs of $\text{supp}(f)$ that are contracted. We maintain potentials on the points A and B directly, instead of the contracted components.

When conducting the Hungarian search, we initialize S to be the set of vertices from *active excess contracted components* who (in sum) meet the imbalance criteria. ⟨⟨**unclear**⟩⟩ Upon relaxing any $v \in \hat{v}$, we immediately relax all the contracted support arcs which span $\hat{v}$. Since the input network is uncapacitated, each contracted component is strongly connected in the residual network by the admissible forward/backward arcs of each contracted arc. ⟨⟨**unparsable**⟩⟩ To relax arcs in $\hat{E}$, we relax the support arcs before attempting to relax any non-support arcs; this will guarantee that the underlying graph of the support is acyclic (see Lemma D.6). Relaxations of support arcs can be performed without further potential changes, since they are admissible by invariant.

During the augmentations, contracted residual arcs are considered to have infinite capacity, and we do not update the value of flows on these arcs. We allow augmenting paths to begin from any point $a \in \hat{v} \cap A$ in an active excess component $\hat{v}$, and end at any point $b \in \hat{w} \cap B$ in an active deficit component $\hat{w}$.

D.3 Recovering the optimal flow

We use the recovery strategy from Agarwal *et al.* [2], which runs in $O(n \operatorname{polylog} n)$ time. The main idea is that, if $\mathcal{T}$ is an undirected *spanning tree of admissible edges* under optimal potentials π^*, then there exists an optimal flow f^* with support only on arcs corresponding to edges of $\mathcal{T}$. Intuitively, $\mathcal{T}$ is a maximal set of linearly independent dual LP constraints for the optimal dual (π^*), so there exists an optimal primal solution (f^*) with support only on these arcs. To see this, we can use a perturbation argument: raising the cost of each non-tree edge by a positive amount does not change $\text{cost}(\pi^*)$ or the feasibility of π^*, but does raise the cost of any circulation f using non-tree edges. Strong duality suggests that $\text{cost}(f^*) = \text{cost}(\pi^*)$ is unchanged, therefore f^* must have support only on the tree edges.

Since the arcs corresponding to edges of $\mathcal{T}$ have no cycles, we can solve the maximum flow in linear time using the following greedy algorithm. Let $\text{par}(v)$ be the parent of vertex v in $\mathcal{T}$. We begin with $f^* = 0$ and process $\mathcal{T}$ from its leaves upwards. For a supply leaf v, we satisfy its supply by choosing $f^*(v \to \text{par}(v)) \leftarrow \phi(v)$. Otherwise if leaf v is a demand vertex, we choose $f^*(\text{par}(v) \to v) \leftarrow -\phi(v)$. Once we've solved the supplies/demands for each leaf, then we can *trim* the leaves, removing them from $\mathcal{T}$ and setting the supply/demand of each parent-of-a-leaf to its current imbalance. Then, we can recurse on this smaller tree and its new set of leaves.

▶ **Lemma D.1.** *Let $G(\mathcal{T})$ be the subnetwork of G corresponding to edges of the undirected spanning tree $\mathcal{T}$. If there exists a flow in $G(\mathcal{T})$ which satisfies every supply and demand, then*

the greedy algorithm finds the maximum flow in $G(\mathcal{T})$ in $O(n)$ time.

Proof. Observe that, for any flow f in G, $\text{supp}(f)$ has no paths of length longer than one. Thus, if a flow f^* satisfying supplies/demands exists within $G(\mathcal{T})$, then each supply vertex has flow paths that terminate at its parent/children. Similarly, each demand vertex receive all its flow from its parent/children. Since there is only one option for a supply leaf (resp. demand leaf) to send its flow (resp. receive its flow), the greedy algorithm correctly identifies the values of f^* for arcs adjoining $\mathcal{T}$ leaves. Trimming these leaves, we can apply this argument recursively for their parents. The running time of the greedy algorithm is $O(n)$, as leaves can be identified in $O(n)$ time and no vertex becomes a leaf more than once. ◀

It remains to show how we construct $\mathcal{T}$. We begin with a (spanning) *shortest path tree* (SPT) T in the residual network of f, under reduced costs and rooted at an arbitrary vertex r. For the SPT to span, we need the additional assumption that G is strongly connected. We make can make G strongly connected by adding a 0-supply vertex s with arcs $s \to a$ for all $a \in A$ and $b \to a$ for all $b \in B$, with some high cost M. Following Orlin [20], these arcs cannot appear in an optimal flow if M is sufficiently high, and we can extend π^* to include s using $\pi^*(s) = 0$ if $M > \max_{b \in B} \pi^*(b)$. This extension to π^* preserves feasibility.

The edges corresponding to arcs of T do not suffice for $\mathcal{T}$, since some SPT arcs may be inadmissible. Let $d_r(v)$ be the shortest path distance of $v \in A \cup B \cup \{s\}$ from r, and consider potentials $\pi^\# = \pi^* - d_r$.

▶ **Lemma D.2 (Orlin [20, Lemma 3]).** *Let f be a flow satisfying the optimality conditions with respect to π^*. Then, (i) f satisfies the optimality conditions with respect to $\pi^\#$, and (ii) all SPT arcs are admissible under $\pi^\#$.*

We can use this lemma to argue that $\pi^\#$ is still optimal. Recall that f has values defined only on the non-contracted residual arcs; we can apply the first part of Lemma D.2 on these arcs. For arcs within contracted components, we use a different argument. Observe that each $\hat{v} \in \hat{V}$ is spanned by a set of $\text{supp}(f)$ arcs, which are admissible by invariant. Thus, all $v \in \hat{v}$ are equidistant from r, and they will have the same value $d_r(v)$. It follows that the reduced costs of arcs with both endpoints in $\hat{v}$ do not change when replacing π^* with $\pi^\#$, so arcs contained in $\hat{v}$ that met the optimality conditions for π^* still meet them for $\pi^\#$.

From the second part of Lemma D.2, the SPT T is a spanning tree of admissible arcs under $\pi^\#$. We set $\mathcal{T}$ to be the set of undirected edges corresponding to T.

Computing the SPT. We conclude by describing the procedure for building the SPT, i.e. by running Dijkstra's algorithm in the residual network. We use a geometric implementation that is very similar to Hungarian search. We begin with $S = \{r\}$ and $d_r(r) = 0$, where r is our arbitrary root. For all other vertices, $d_r(v)$ is initially unknown. In each iteration, we relax the minimum-reduced cost arc $v \to w$ in the frontier $S \times (A \cup B) \setminus S$, adding w to S, and setting $d_r(w) = d_r(v) + c_{\pi^*}(v, w)$. Once $S = A \cup B$, the SPT T is the set of relaxed arcs.

If an either direction of an arc of $\text{supp}(f)$ enters the frontier, we relax it immediately. To detect support arcs, we build a list for each $v \in A \cup B$ of the support arcs which use v as an endpoint, and once $v \in S$ we check its list. There are $O(n)$ support arcs in total (by acyclicity of $E(\text{supp}(f))$; see Lemma D.6), so the total time spent searching these lists is $O(n)$. Such relaxations are correct for the shortest path tree, since the support edges are admissible and reduced costs are nonnegative.

Other edges appearing in the frontier can be split into three categories:

1. Forward A-to-B arcs. We query these using a BCP with $P = A \cap S$ and $Q = B \setminus S$.

2. B-to-s arcs. These will never have flow support. We can query the minimum with a max-heap on potentials of $B \cap S$. We query these while $s \in S$.
3. s-to-A arcs. These will also never have flow support. We can query the minimum with a min-heap on potentials of $A \setminus S$. We query these while $s \in S$.

We perform $O(n)$ relaxations and takes $O(\text{polylog}\, n)$ time per relaxation, for non-support relaxations. An additional $O(n)$ time is spent relaxing support edges. The total running time of Dijkstra's algorithm is $O(n \,\text{polylog}\, n)$. Combining with Lemma D.1, we obtain the following.

▶ **Lemma D.3.** *Given optimal potentials π^* and an optimal contracted flow f, the optimal flow f^* can be computed in $O(n \,\text{polylog}\, n)$ time.*

D.4 Recovering the optimal flow for sum-of-distances.

When the matching objective uses the just the p-norms (that is, when $q = 1$), we can prove that the subgraph formed by admissible arcs is in fact *planar*. Planarity gives us two things for recovery: there are only a linear number of admissible arcs, and the max-flow on them can be solved in near-linear time with a planar graph multiple-source multiple-sink max-flow algorithm. Note that this recovery algorithm is not asymptotically faster than the other, and depending on the planar max-flow algorithm, not necessarily simpler either.

Up until now, we have not placed restrictions on coincidence between A and B, but for the next proof it is useful to do so. We can assume that all points within $A \cup B$ are distinct, otherwise we can replace all points coincident at $x \in \mathbb{R}^2$ with a single point whose supply/demand is $\sum_{v \in A \cup B : v = x} \lambda(v)$. This is roughly equivalent to transporting as much as we can between coincident supply and demand, and is optimal by triangle inequality.

Without loss of generality, assume π^* is nonnegative (raising π^* uniformly on all points does not change the objective or feasibility). Recall that π^* is feasibility if for all $a \in A$ and $b \in B$

$$c_{\pi^*}(a \to b) = \|a - b\|_p - \pi^*(a) + \pi^*(b) \geq 0.$$

An arc $a \to b$ is admissible when

$$c_{\pi^*}(a \to b) = \|a - b\|_p - \pi^*(a) + \pi^*(b) = 0.$$

We note that these definitions have a nice visual: Place disks D_q of radius $\pi(q)$ at each $q \in A \cup B$. Feasibility states that for all $a \in A$ and $b \in B$, D_a cannot contain D_b with a gap between their boundaries. The arc $a \to b$ is admissible when D_a contains D_b and their boundaries are tangent.

▶ **Lemma D.4.** *Let π^* be a set of optimal potentials for the point sets A and B, under costs $c(a, b) = \|a - b\|_p$. Then, the set of admissible arcs under π^* form a planar graph.*

Proof. We assume the points of $A \cup B$ are in general position (e.g. by symbolic perturbation) such that no three points are collinear. Let $a_1 \to b_1$ and $a_2 \to b_2$ be any pair of admissible arcs under π^*. We will isolate them from the rest of the points, considering π^* restricted to the four points $\{a_1, a_2, b_1, b_2\}$. Clearly, this does not change whether the two arcs cross. Observe that we can raise $\pi^*(a_2)$ and $\pi^*(b_2)$ uniformly, until $c_\pi(a_2, b_1) = 0$, without breaking feasibility or changing admissibility of $a_1 \to b_1$ and $a_2 \to b_2$ Henceforth, we assume that we have modified π^* in this way to make $a_2 \to b_1$ admissible. Given positions of a_1, a_2, and b_1, we now try to place b_2 such that $a_1 \to b_1$ and $a_2 \to b_2$ cross. Specifically, b_2 must be placed within a region $\mathcal{F}$ that lies between the rays $\overrightarrow{a_2 a_1}$ and $\overrightarrow{a_2 b_1}$, and within the halfplane bounded by $\overleftrightarrow{a_1 b_1}$ that does not contain a_2.

Let $g_a(q) \coloneqq \|a - q\| - \pi^*(a)$ for $a \in A$ and $q \in \mathbb{R}^2$. Let the *bisector* between a_1 and a_2 be $\beta \coloneqq \{q \in \mathbb{R}^2 \mid g_{a_1}(q) = g_{a_2}(q)\}$. β is a curve subdividing the plane into two open faces, one where g_{a_1} is minimized and the other where g_{a_2} is. From these definitions, admissibility of $a_1 \to b_1$ and $a_2 \to b_1$ imply that b_1 is a point of the bisector.

We show that $\mathcal{F}$ lies entirely on the g_{a_1} side of the bisector. First, we prove that the closed segment $\overline{a_1 b_1}$ lies entirely on the g_{a_1} side, except b_1 which lies on β. Any $q \in \overline{a_1 b_1}$ can be written parametrically as $q(t) = (1-t)b_1 + ta_1$ for $t \in [0, 1]$. Consider the single-variable functions $g_{a_1}(q(t))$ and $g_{a_2}(q(t))$.

$$g_{a_1}(q(t)) = (1-t)\|a_1 - b_1\| - \pi(a_1)$$
$$g_{a_2}(q(t)) = \|(a_2 - b_1) - t(a_1 - b_1)\| - \pi(a_2)$$

At $t = 0$, these expressions are equal. Observe that the derivative with respect to t of $g_{a_1}(q(t))$ is less than $g_{a_2}(q(t))$. Indeed, the value of $\frac{d}{dt}\|(a_2 - b_1) - t(a_1 - b_1)\|$ is at least $-\|a_1 - b_1\| = \frac{d}{dt} g_{a_1}(q(t))$, which is realized if and only if $\frac{(a_2 - b_1)}{\|a_2 - b_1\|} = \frac{(a_1 - b_1)}{\|a_1 - b_1\|}$. This corresponds to $\overrightarrow{a_2 b_1}$ and $\overrightarrow{a_1 b_1}$ being parallel, but this is disallowed since a_1, a_2, b_1 are in general position. Thus, $g_{a_1}(q(t)) \le g_{a_2}(q(t))$ with equality only at b_1.

Now, we parameterize each point of $\mathcal{F}$ in terms of points on $\overline{a_1 b_1}$. Every $q \in \mathcal{F}$ can be written as $q(t') = q' + t'(q' - a_2)$ for some $q' \in \overline{a_1 b_1}$ and $t \ge 0$, i.e. $q' = \overline{a_1 b_1} \cap \overrightarrow{a_2 q}$. We call q' the *projection* of q onto $\overline{a_1 b_1}$. We can write g_{a_1} and g_{a_2} in terms of t' and observe that $\frac{d}{dt'} g_{a_1}(q(t')) \le \frac{d}{dt'} g_{a_2}(q(t'))$, as the derivative of $g_a(q(t'))$ is maximized if $(q(t') - a)$ is parallel to $(q(t') - a_2)$ and lower otherwise. Notably, $q(t')$ with projection b_1 have $\frac{d}{dt'} g_{a_1}(q(t')) < \frac{d}{dt'} g_{a_2}(q(t'))$, since a_1, a_2, b_1 are in general position. Any $q(t')$ with a different projection do not have strict inequality, but the projection itself has $g_{a_1}(q') < g_{a_2}(q')$ for $q' \neq b_1$ since it lies on $\overline{a_1 b_1}$. Therefore, for all $q \in \mathcal{F} \setminus \{b_1\}$, $g_{a_1}(q') < g_{a_2}(q')$, and $\mathcal{F}$ lies on the g_{a_1} side of the bisector except for b_1 which lies on β. We can eliminate b_1 as a candidate position for b_2, since points of B cannot coincide.

Observe that $g_{a_1}(b) < g_{a_2}(b)$ for $b \in B$ implies that $c_\pi(a_1, b) < c_\pi(a_2, b)$, and $c_\pi(a_1, b) = c_\pi(a_2, b)$ if and only if b lies on β. This holds for all $b \in \mathcal{F}$ including our prospective b_2, but then $c_\pi(a_1, b_2) < c_\pi(a_2, b_2) = 0$ since $a_2 \to b_2$ is admissible. This violates feasibility of $a_1 \to b_2$, so there is no feasible placement of b_2 which also crosses $a_1 \to b_1$ with $a_2 \to b_2$. ◀

We can construct the entire set of admissible arcs by repeatedly querying the minimum-reduced-cost outgoing arc for each $a \in A$ until the result is not admissible. By Lemma D.4 the resulting arc set forms a planar graph, so by Euler's formula the number of arcs to query is $O(n)$. We can then find the maximum flow in time $O(n \log^3 n)$ time, using the planar multiple-source multiple-sink maximum-flow algorithm by Borradaile *et al.* [7].

▶ **Lemma D.5.** *If the transportation objective is sum-of-costs, then given the optimal potentials π^*, we can compute an optimal flow f^* in $O(n \operatorname{polylog} n)$ time.*

D.5 Dead vertices

Let the *support degree* of a vertex be its degree in the graph induced by the underlying edges of $\operatorname{supp}(f)$. We call a vertex $b \in B$ *dead* if b has support degree 0 and is not an active excess or deficit vertex; call it *living* otherwise. Dead vertices are essentially equivalent to the *null vertices* of Section 3. However, since the reduction in this section does not use a super-source/super-sink, we can simply remove these from consideration during a Hungarian search — they will not terminate the search, and have no outgoing residual arcs. Like the null vertices, we ignore dead vertex potentials and infer feasible potentials when they become

live again. We use A_ℓ and B_ℓ to denote the living vertices of points in A and B, respectively. Note that being dead/alive is a notion strictly defined only for vertices, and not for contracted components.

We say a dead vertex is *revived* when it stops meeting either condition of the definition. Dead vertices are only revived after Δ decreases (at the start of a subsequent excess scale) as no augmenting path will cross a dead vertex and they cannot meet the criteria for contractions. When a dead vertex is revived, we must add it back into each of our data structures and give it a feasible potential. For revived $b \in B$, a feasible choice of potential is $\pi(b) \leftarrow \max_{a \in A}(\pi(a) - c(a,b))$ which we can query by maintaining a weighted nearest neighbor data structure on the points of A. The total number of revivals is bounded above by the number of augmentations: since the final flow is a circulation on $\hat{G}$ and a newly revived vertex v has no incident arcs in $\text{supp}(f)$ and cannot be contracted, there is at least one subsequent augmentation which uses v as its beginning or end. Thus, the total number of revivals is $O(n \log n)$.

D.6 Number of relaxations

By prioritizing the relaxation of support arcs, we also have the following lemma.

▶ **Lemma D.6 (Agarwal *et al.* [2]).** *If arcs of* $\text{supp}(f)$ *are relaxed first as they arrive on the frontier, then* $E(\text{supp}(f))$ *is acyclic.*

Proof. Let f_i be the pseudoflow after the i-th augmentation, and let T_i be the forest of relaxed arcs generated by the Hungarian search for the i-th augmentation. Namely, the i-th augmenting path is an excess-deficit path in T_i, and all arcs of T_i are admissible by the time the augmentation is performed. Let $E(T_i)$ be the undirected edges corresponding to arcs of T_i. Notice that, $E(\text{supp}(f_{i+1})) \subseteq E(\text{supp}(f_i)) \cup E(T_i)$. We prove that $E(\text{supp}(f_i)) \cup E(T_i)$ is acyclic by induction on i; as $E(\text{supp}(f_{i+1}))$ is a subset of these edges, it must also be acyclic. At the beginning with $f_0 = 0$, $E(\text{supp}(f_0))$ is vacuously acyclic.

Let $E(\text{supp}(f_i))$ be acyclic by induction hypothesis. Since T_i is a forest (thus, acyclic), any hypothetical cycle Γ that forms in $E(\text{supp}(f_i)) \cup E(T_i)$ must contain edges from both $E(\text{supp}(f_i))$ and $E(T_i)$. To give a visual analogy, we will color $e \in \Gamma$ *purple* if $e \in E(\text{supp}(f_i)) \cap E(T_i)$, *red* if $e \in E(\text{supp}(f_i))$ but $e \notin E(T_i)$, and *blue* if $e \in E(T_i)$ but $e \notin E(\text{supp}(f_i))$. Then, Γ is neither entirely red nor entirely blue. We say that red and purple edges are *red-tinted*, and similarly blue and purple edges are *blue-tinted*. Roughly speaking, our implementation of the Hungarian search prioritizes relaxing red-tinted admissible arcs over pure blue arcs.

We can sort the blue-tinted edges of Γ by the order they were relaxed into S during the Hungarian search forming T_i. Let $(v,w) \in \Gamma$ be the last pure blue edge relaxed, of all the blue-tinted edges in Γ — after (v,w) is relaxed, the remaining unrelaxed, blue-tinted edges of Γ are purple.

Let us pause the Hungarian search the moment before (v,w) is relaxed. At this point, $v \in S$ and $w \notin S$, and the Hungarian search must have finished relaxing all frontier support arcs. By our choice of (v,w), $\Gamma \setminus (v,w)$ is a path of relaxed blue edges and red-tinted edges which connect v and w. Walking around $\Gamma \setminus (v,w)$ from v to w, we see that every vertex of the cycle must be in S already: $v \in S$, relaxed blue edges have both endpoints in S, and any unrelaxed red-tinted edge must have both endpoints in S, since the Hungarian search would have prioritized relaxing the red-tinted edges to grow S before relaxing (v,w) (a blue edge). It follows that $w \in S$ already, a contradiction.

No such cycle Γ can exist, thus $E(\text{supp}(f_i)) \cup E(T_i)$ is acyclic and $E(\text{supp}(f_{i+1})) \subseteq E(\text{supp}(f_i)) \cup E(T_i)$ is acyclic. By induction, $E(\text{supp}(f_i))$ is acyclic for all i. ◀

Let $E(\Sigma_a)$ ⟪**only used once**⟫ be the underlying edges of the support star centered at a and $F := E(\text{supp}(f)) \setminus \bigcup_{a \in A} E(\Sigma_a)$. Using Lemma D.6, we can show that the number of support arcs outside support stars ($|F|$) is small.

▶ **Lemma D.7.** $|B_\ell \setminus \bigcup_{a \in A} \Sigma_a| \leq r$.

Proof. F is constructed from $E(\text{supp}(f))$ by eliminating edges in support stars, therefore all edges in F must adjoin vertices in B of support degree at least 2. By Lemma D.6, $E(\text{supp}(f))$ is acyclic and therefore forms a spanning forest over $A \cup B_\ell$, so F is also a bipartite forest. All leaves of F are therefore vertices of A.

Pick an arbitrary root for each connected component of F to establish parent-child relationships for each edge. As no vertex in B is a leaf, each vertex in B has at least one child. Charge each vertex in B to one of its children in F, which must belong to A. Each vertex in A is charged at most once. Thus, the number of B_ℓ vertices outside of support stars is no more than r. ◀

▶ **Lemma 5.2.** *Suppose we have stripped the graph of dead vertices. The number of relaxation steps in a Hungarian search outside of support stars is $O(r)$.*

Proof. If there are no dead vertices, then each non-support star relaxation step adds either (i) an active deficit vertex, (ii) a non-deficit vertex $a \in A_\ell$, or (iii) a non-deficit vertex $b \in B_\ell$ of support degree at least 2. There is a single relaxation of type (i), as it terminates the search. The number of vertices of type (ii) is r, and the number of vertices of type (iii) is at most r by Lemma D.7. The lemma follows. ◀

▶ **Lemma 5.3.** *Hungarian search takes $O(r\sqrt{n}\,\text{polylog}\,n)$ time.*

Proof. The number of relaxation steps outside of support stars is $O(r)$ by Lemma 5.2. The time per relaxation outside of support stars is $O(\sqrt{n}\,\text{polylog}\,n)$. The time spent processing relaxations within a support star is $O(\sqrt{n}\,\text{polylog}\,n)$, and at most r are relaxed during the search. The total time is therefore $O(r\sqrt{n}\,\text{polylog}\,n)$. ◀

D.7 Updating support stars

Initially, we label stars big or small according to the $\sqrt{n}$ threshold. Afterwards, a star that is currently big is turned into a small star once $|\Sigma_a| \leq \sqrt{n}/2$, and star that is currently small is turned into a big star once $|\Sigma_a| \geq 2\sqrt{n}$. We say a star which crosses one of these size thresholds is *changing state* (from small-to-big or big-to-small), and must be represented in the opposite type of data structure. Our strategy is to charge the data structure update time associated with a state change to the *membership changes* in Σ_a that preceded the state change.

A star Σ_a undergoing a big-to-small state change has size $|\Sigma_a| \leq \sqrt{n}/2$. The state change deletes $\mathcal{D}_{\text{big}}(a)$ and inserts Σ_a into $\mathcal{D}_{\text{small}}$. Thus, the time spent for a big-to-small state change is $O(\sqrt{n}\,\text{polylog}\,n)$, and there were at least $\sqrt{n}/2$ points removed from Σ_a since it last changed state. The amortized time for a big-to-small state change per star membership change is $O(\text{polylog}\,n)$.

A star Σ_a undergoing a small-to-big state change has size $|\Sigma_a| \geq 2\sqrt{n}$. We can write its size as $|\Sigma_a| = \sqrt{n} + x$ for some integer $x \geq \sqrt{n}$, so we also have $|\Sigma_a| \leq 2x$. When switching, we delete all $|\Sigma_a|$ points from $\mathcal{D}_{\text{small}}$ and construct a new $\mathcal{D}_{\text{big}}(a)$. Constructing $\mathcal{D}_{\text{big}}(a)$

requires inserting up to r points of A (into P) and the $|\Sigma_a|$ points of the star (into Q). Thus, the time spent for a small-to-big state change is $(r + 2x) \cdot O(\text{polylog}\, n)$, and there were at least x points added to Σ_a since it last changed state. The amortized time for a small-to-big state change per star membership change is $O((r/x)\,\text{polylog}\, n)$. Since $x \geq \sqrt{n}$, this is at most $O((r/\sqrt{n})\,\text{polylog}\, n)$.

Star membership can only be changed by augmenting paths passing through the vertex, therefore the total number of membership changes is $O(rn \log n)$ by Lemmas 5.1 and 5.2. Thus, the total time spent on state changes is $O((r^2\sqrt{n} + rn)\,\text{polylog}\, n)$.