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 497

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 [14] 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 [18], which uses an interior-point algorithm and runs in $O(m^{10/7} \operatorname{polylog} n)$ time.

The Hungarian algorithm [16] 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 [20] 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 $O(n^2 \operatorname{polylog} n)$ [15] (see also [1, 25]). 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, 22, 25, 26]. For $q = 1$, the best algorithm to date by Sharathkumar and Agarwal [23] 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 [22]; 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, 21]. Here, the goal is to (i)

[1] Partial matching is also called *imperfect matching* or *imperfect assignment* [13, 20].

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 [19] solves the transportation problem in $O((m + n \log n)n \log n)$ time. Lee and Sidford [17] 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, 15]. 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 **??**, 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 4 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 [19], 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.

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. 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 $c_\pi(v, w) := c(v, w) - \pi(v) + \pi(w)$. 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 $c_\pi(v, w) = 0$.

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 [20] 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.

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, 25]. 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.* [15] 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 $|M| \leq k$ 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.

2.1 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 applying the opposite insert/delete operation for each BCP update 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 [25] 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.

2.2 Potential updates

We modify a trick from Vaidya [25] 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 relaxation; 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 $O(k)$.

3 Approximating Min-Cost Partial Matching through Cost-Scaling

In this section we describe an approximation algorithm for computing a min-cost partial matching. We reduce the problem to computing a min-cost circulation in a flow network (Section 3.1). We adapt the cost-scaling algorithm by Goldberg *et al.* [13] for computing

min-cost flow of a unit-capacity network (Section 3.2). Finally, we show how their algorithm can be implemented in $O\left((n + k^{3/2}) \operatorname{polylog}(n) \log(1/\varepsilon)\right)$ time in our setting (Section 3.2).

3.1 From matching to circulation

Given a bipartite graph G with vertex sets A and B, we construct a flow network $N = (V, \vec{E})$ in a standard way [20] so that a min-cost matching in G corresponds to a min-cost integral circulation in N.

Flow network. Each vertex in G becomes a node in N and each edge (a, b) in G becomes an arc $a \to b$ in N; we refer to these nodes (resp. arcs) as *bipartite nodes* (resp. *bipartite arcs*). We also include a *source* node s and *sink* node t in N. For each $a \in A$, we add a *left dummy arc* $s \to a$ and for each $b \in B$ we add a *right dummy arc* $b \to t$. The cost $c(v \to w)$ of each arc $v \to w$ in N is equal to $c(v, w)$ if $v \to w$ is a bipartite arc and 0 if $v \to w$ is a dummy arc. All arcs in N have unit capacity.

Let $\phi : V \to \mathbb{Z}$ be an integral supply/demand function on nodes of N. The positive values of $\phi(v)$ are referred to as *supply*, and the negative values of $\phi(v)$ as *demand*. A *pseudoflow* $f : \vec{E} \to [0, 1]$ is a function on arcs of N. The *support* of f in N, denoted as $\operatorname{supp}(f)$, is the set of arcs with positive flows: $\operatorname{supp}(f) \coloneqq \left\{ v \to w \in \vec{E} \mid f(v \to w) > 0 \right\}$. Given a pseudoflow f, the *imbalance* of a vertex (with respect to f) is

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

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

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

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

If we set $\phi(s) = k$, $\phi(t) = k$, and $\phi(v) = 0$ for all $v \in A \cup B$, then an integral circulation f corresponds to a partial matching M of size k and vice versa. Moreover, $\operatorname{cost}(M) = \operatorname{cost}(f)$. Hence, the problem of computing a min-cost matching of size k in $G(A, B)$ transforms to computing an integral circulation in N. The following lemma will be useful for our algorithm.

▶ **Lemma 3.1.** ⟨⟨*Define N and f properly.*⟩⟩

(i) *For any integral circulation in N, the size of* $\operatorname{supp}(f)$ *is at most* $3k$.

(ii) *For any integral pseudoflow in N with $O(k)$ excess, the size of* $\operatorname{supp}(f)$ *is $O(k)$.*

3.2 A cost-scaling algorithm

Before describing the algorithm, we need to introduce a few more concepts.

Residual network and admissibility. If f is an integral pseudoflow (that is, $f(v \to w) \in \{0, 1\}$ for every arc in $\vec{E}$), then each arc $v \to w$ in N is either *idle* with $f(v \to w) = 0$ or *saturated* with $f(u \to v) = 1$. ⟨⟨define earlier?⟩⟩

Given a pseudoflow f, the *residual network* $N_f = (V, \vec{E}_f)$ is defined as follows. For each idle arc $v \to w$ in $\vec{E}$, we add a *forward* residual arc $v \to w$ in N_f. For each saturated arc $v \to w$ in $\vec{E}$, we add a *backward* residual arc $w \to v$ in N_f. The set of residual arcs in N_f is therefore

$$\vec{E}_f := \{v \to w \mid f(v \to w) = 0\} \cup \{w \to v \mid f(v \to w) = 1\}.$$

The cost of a forward residual arc $v \to w$ is $c(v \to w)$, while the cost of a backward residual arc is $w \to v$ is $-c(v \to w)$. Each arc in N_f also has unit capacity. By Lemma 3.1, N_f has $O(k)$ backward arcs if f has $O(k)$ excess.

⟨⟨**Need to include a description of augmenting f by g; $f + g$ is incorrect due to no antisymmetry**⟩⟩

Using LP duality for min-cost flow, we assign a *potential* $\pi(v)$ to each node v in N. The *reduced cost* of an arc $v \to w$ in N with respect to π is defined as

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

Similarly we define the reduced cost of arcs in N_f: the reduced cost of a forward residual arc $v \to w$ in N_f is $c_\pi(v \to w)$, and the reduced cost of a backward residual arc $w \to v$ in N_f is $-c_\pi(v \to w)$. Abusing the notation, we also use c_π to denote the reduced cost of arcs in N_f.

⟨⟨**Unclear. Doese $c_\pi(w \to v)$ have positive or negative value for a backward arc $w \to v$?**⟩⟩

The *dual feasibility constraint* asks that $c_\pi(v \to w) \geq 0$ holds for every arc $v \to w$ in $\vec{E}$; potentials π that 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, 24] 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 $v \to w$. We will widely use the following well-known property of θ-optimality.

▶ **Lemma 3.2.** *Let f be an θ-optimal pseudoflow in N and let g be an admissible pseudoflow in N_f. Then f augmented by g is also θ-optimal in N.*

Using Lemma 3.1, the following lemma can be proved about θ-optimality:

▶ **Lemma 3.3.** *Let f be a θ-optimal integer circulation in N, and f^* be an optimal integer circulation for N. Then,* $\text{cost}(f) \leq \text{cost}(f^*) + 6k\theta$.

Estimating the value of $\text{cost}(f^*)$**.** We now describe a procedure for estimating $\text{cost}(f^*)$ within a polynomial factor, which will be useful in setting the scaling parameters of the cost-scaling algorithm.

Let T be a minimum spanning tree of $A \cup B$ under the L_p^q-metric. Let $e_1, e_2, \ldots, e_{n-1}$ be the edges of T sorted in nondecreasing order of length; in other words, $c(e_i) \leq c(e_{i+1})$ where $c(e) = \|e\|_p^q$. Let T_i be the forest consisting of the vertices of $A \cup B$ and $e_1, \ldots, e_i$. We call a matching M of $G(A, B)$ *intra-cluster* if both endpoints of every edge in M lie in the same connected component of T_i. We define i^* to be the smallest index i such that there exists an intra-cluster matching of size k in T_i. Set

▶ Lemma 3.3. The value of i^* can be computed in $O(n \log n)$ time.

$c(e_{i^*}) \leq \text{cost}(f^*) \leq \overline{\theta}$.

There is a $\overline{\theta}$-optimal circulation in the network N with respect to the 0 potential $\pi = 0$, assuming $\phi(s) = k$, $\phi(t) = -k$, and $\phi(v) = 0$ for all $v \in A \cup B$.

Set

We now describe a fast implementation of the refinement stage. The Hungarian search and augmentation steps are similar: each traversing through the residual network using admissible arcs starting from the excess vertices. Due to lack of space, we only describe the Hungarian search process.

At a high level, let X be the subset of nodes visited by the Hungarian search. Initially X is the set of excess nodes. At each step, the algorithm finds a minimum reduced cost arc $v \to w$ in N_f from X to $V \setminus X$. If $v \to w$ is not admissible, the potential of all nodes in X is increased by $\lceil c_\pi(v \to w)/\theta \rceil$ to make $v \to w$ admissible. If w is a deficit node, the search terminates. Othwerise, w is added to X and the search continues.

⟨⟨**does the following paragraph give good intuition for why we need active/inactive?**⟩⟩

Implementing the Hungarian search efficiently is more difficult than in Section 2 because (a) excess nodes may show up in A as well as in B, (b) a balanced node may become imbalanced later in the rounds, and (c) the potential of excess nodes may be non-uniform. We therefore need a more complex data structure.

We call a node v of N *inactive* if $\phi_f(v) = 0$ and no arc of $\mathrm{supp}(f)$ is incident to it; otherwise v is *active*. We note that s and t are always active. Let $\overline{A}$ (resp. $\overline{B}$, $\overline{X}$) denote the set of active nodes of A (resp. B, X). We treat active and inactive nodes separately to implement the Hungarian search efficiently. By definition, inactive vertices only adjoin forward arcs in N_f. Thus, the in-degree (resp. out-degree) of $\overline{A}$ (resp. $\overline{B}$) is 1, and any path passing through an inactive vertex has a subpath of the form $s \to v \to b$ for some $b \in B$ or $a \to v \to t$ for some $a \in A$. Consequently, a path in N_f may have at most two consecutive inactive nodes, and in the case of two consecutive inactive nodes there is a subpath of the form $s \to v \to w \to t$ where $v \in \overline{A}$ and $w \in \overline{B}$.

Using this observation, we adapt the Hungarian search to "skip over" inactive nodes as follows. At each step, we find a minimum reduced cost path of length at most 3 from an active node of X to an active node of $V \setminus X$, and add the nodes of this path into X in a single step. The end of this path is the next active node that the original Hungarian search would have explored. We update potentials in X according to the total reduced cost of this path. There are $O(k)$ active nodes, so the number of minimization queries per Hungarian search is $O(k)$.

We find the minimum reduced cost path of length 1, 2, and 3, and then choose the cheapest among them. We now describe a data structure for each path length. For each data structure, our "time budget" per Hungarian search is $O(k \operatorname{polylog} n)$.

Finding length-1 paths. This data structure finds a minimum reduced cost arc from an active node of X to an active node of $V \setminus X$. There are $O(k)$ backward arcs, so the minimum among backward arcs can be maintained explicitly in a priority queue on c_π and retrieved in $O(1)$ time.

There are three types of forward arcs: $s \to a$ for some $a \in \overline{A}$, $b \to t$ for some $b \in \overline{B}$, and bipartite arc $a \to b$ with two active endpoints. Edges of the first (resp. second) type can be found by maintaining $\overline{A} \setminus X$ (resp. $\overline{B} \cap X$) in a priority queues on π, but should only be queried if $s \in X$ (resp. $t \notin X$). The cheapest arc of the third type is an (additively weighted) bichromatic closest pair (BCP) between $\overline{A} \cap X$ and $\overline{B} \setminus X$, with reduced cost as the pair distance ⟨⟨**potentials?**⟩⟩. We thus maintain $\overline{A} \cap X$, $\overline{B} \setminus X$ in a dynamic BCP data structure [15] on which insertions/deletions can be performed in $O(\operatorname{polylog} k)$ time.

⟨⟨**for now: write this to reference rewinding; and "additionally" wrt active-switch**⟩⟩

Finding length-2 paths. We describe how to find the cheapeast path of the form $s{\to}v{\to}b$ where v is inactive and $b \in \overline{B}$. A cheapest path of the form $a{\to}v{\to}t$ can be found similarly. As for length-1 paths, we only query paths originating from s if $s \in X$, and only query paths ending at t if $t \notin X$.

Note that $c_\pi(s{\to}v{\to}b) = c(v,b) + \pi(b) - \pi(s)$. Since $\pi(s)$ is common in all such paths, it suffices to find the pair minimizing

$$\min_{v \in A \setminus \overline{A}, w \in \overline{B} \setminus X} c(v,w) + \pi(w)$$

This is done by maintaining a dynamic BCP data structure between $A \setminus \overline{A}$ and $\overline{B} \setminus X$ with the cost of a pair (v,w) being $c(v,w) + \pi(w)$. We may require an update operation for each active node added to X during the Hungarian search, of which there are $O(k)$, so the time spent during a search is $O(k \operatorname{polylog} n)$.

Since the size of $A \setminus \overline{A}$ is at least $r - k$, we cannot construct this BCP from scratch at the beginning of each iteration of Hungarian search. To resolve this, we use the idea of rewinding from Section 2.1, with a slight twist. There are now *two* ways that the initial BCP may change across consecutive Hungarian searches: (1) the initial set X may change as vertices lose excess through augmentation, and (2) the set of inactive A vertices may change; that is, when flow is augmented across a vertex of $A \setminus \overline{A}$. The first is identical to the situation in Section 2.1; the number of excess depletions is $O(k)$ over the course of Refine. For the second, the active/inactive status of a node can change only if the blocking flow found in the augmentation process passes through it. By Lemma 3.5 below, there are $O(k)$ such changes. Thus, updating $A \setminus \overline{A}$ for the BCP (after augmentation) can be done in $O(k \operatorname{polylog} n)$ time for each Hungarian search.

Finding length-3 paths. We now describe how to find the cheapest path of the form $s{\to}v{\to}w{\to}t$ where $v \in A \setminus \overline{A}$ and $w \in B \setminus \overline{B}$. Note that $c_\pi(s{\to}v{\to}w{\to}t) = c(v{\to}w) - \pi(s) + \pi(t)$. Hence, we need to find a BCP between $A \setminus \overline{A}$ and $B \setminus \overline{B}$, where the cost of a pair (v,w) is simply $c(v,w)$. This can be done by maintaining a dynamic BCP data structure similar to the case of length-2 paths.

The BCP sets have no dependence on X — the only updates required correspond to changes to $\overline{A}$ or $\overline{B}$, after an augmentation. Applying Lemma 3.5 again, there are $O(k)$ active/inactive updates caused by an augmentation, so the time for these updates per Hungarian search is $O(k \operatorname{polylog} n)$.

Updating potentials. The Hungarian search periodically raises the potentials for all nodes in X, and we need to implement this efficiently for the data structures above. Note that the data structures above do not utilize potentials of inactive nodes. Potentials for active nodes can be updated in a batched fashion using the method in Section 2.2.

For inactive nodes, we ignore their potentials entirely and instead recover valid potentials for them once they switch from inactive to active (and additionally, at the end of a round). Specifically, for a newly active $a \in A$ we set $c_\pi(a) \leftarrow c_\pi(s)$ and for newly-active $b \in B$ we set $c_\pi(b) \leftarrow c_\pi(t)$. It is straightforward to verify that this choice preserves θ-optimality throughout f as well the admissibility of any length-2/length-3 paths that involved them.

⟨⟨do we need to extend the definition of admissibility to paths?⟩⟩

The following lemmas are crucial to analyzing the running time of the Hungarian search.

▶ **Lemma 3.4.** *The Hungarian search explores $O(k)$ nodes before a deficit vertex is reached, and each Augmentation ⟨⟨?⟩⟩ explores $O(k)$ nodes before finding a blocking flow.*

Proof. To sketch the proof, both procedures perform a search over the set of active nodes, of which there are $O(k)$. Nodes are not explored more than once (added to X, or a stack) by either search. ◀

▶ **Lemma 3.5.** *The blocking flow found by Augmentation is incident to $O(k)$ vertices.*

Proof. Each excess-deficit path that composes the blocking flow is a sequence of active nodes connected by a subpath across 0, 1, or 2 inactive nodes. Each such subpath corresponds to the depth-first search of Augmentation exploring a new active node. By Lemma 3.4, the total number of nodes crossed is therefore $O(k)$. ◀

Augmentation can also be implemented in $O(k \operatorname{polylog} n)$ time, after $O(n \operatorname{polylog} n)$ preprocessing, using a similar set of data structures but for nearest neighbor instead of BCP. Putting everything together, we obtain the following:

▶ **Lemma 3.6.** *After $O(n \operatorname{polylog} n)$ preprocessing, each iteration of* Refine *can be implemented in $O(k \operatorname{polylog} n)$ time.*

⟨⟨**still need to show $\sqrt{k}$ iterations per refine**⟩⟩

4 Unbalanced Transportation

In this section, we give an exact algorithm which solves the planar transportation problem in $O((r^2\sqrt{n} + rn^{3/2}) \operatorname{polylog} n)$ 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 [19], 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 [19]. See Appendix A.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 4.1 (Orlin [19, 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 A.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 A.3 and A.4.

4.1 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 $b \in \Sigma_a$.

4.2 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 A.5. For a proof of the following lemma, see Appendix A.6.

▶ **Lemma 4.2.** *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 $|\Sigma_a| \leq \sqrt{n}$. 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 4.2 we bound the time for each Hungarian search.

▶ **Lemma 4.3.** *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 A.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 4.1, the total number of membership changes is $O(rn \log n)$. The total time spent on big-to-small updates is $O(rn\,\text{polylog}\, n)$, and the total time spent on small-to-big updates is $O(r^2\sqrt{n}\,\text{polylog}\, n)$. Membership changes themselves can be performed in $O(\text{polylog}\, n)$ time each.

Preprocessing time. It takes $O(rn\,\text{polylog}\, n)$ 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 $O(rn\,\text{polylog}\, n)$.

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 $O(r\sqrt{n}\,\text{polylog}\, n)$ by Lemma 4.3. The most recent augmentation may have deactivated $O(1)$ active excess and deficit vertices, which takes $O(\sqrt{n}\,\text{polylog}\, n)$ 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 4.1, 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 4.3, 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*, 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 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⟩.

15 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)*, 2495–2504, 2017. ⟨https://doi.org/10.1137/1.9781611974782.165⟩.

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

17 Yin Tat Lee and Aaron Sidford. Path finding II : An $\tilde{O}(m\sqrt{n})$ algorithm for the minimum cost flow problem. *CoRR* abs/1312.6713, 2015. ⟨http://arxiv.org/abs/1312.6713⟩.

18 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)*, 253–262, 2013. ⟨https://doi.org/10.1109/FOCS.2013.35⟩.

19 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⟩.

20 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)*, 581–590, 2012. ⟨https://doi.org/10.1109/FOCS.2012.9⟩.

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

22 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)*, 306–317, 2012. ⟨https://dl.acm.org/citation.cfm?id=2095116.2095145⟩.

23 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)*, 385–394, 2012. ⟨https://doi.org/10.1145/2213977.2214014⟩.

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

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

26 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)*, 320–331, 1998. ⟨https://doi.org/10.1109/SFCS.1998.743466⟩.

A Missing Details and Proofs from Section 4

A.1 Uncapacitated MCF by excess scaling

We give an outline of the strongly polynomial-time algorithm for uncapacitated min-cost flow problem from Orlin [19]. 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 3.2). 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 [19] 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^*.

A.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 $\mathrm{supp}(f)$ 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 A.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 $\hat{w}$.

A.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 A.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 [19], 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 A.2 (Orlin [19, 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 A.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 $\mathrm{supp}(f)$ 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 A.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 $\mathrm{supp}(f)$ 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 $E(\mathrm{supp}(f))$; see Lemma A.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 $O(\mathrm{polylog}\, n)$ 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 $O(n\, \mathrm{polylog}\, n)$. Combining with Lemma A.1, we obtain the following.

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

A.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 A.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 $q(t) = (1 - t)b_1 + t a_1$ for $t \in [0, 1]$. Consider the single-variable functions $g_{a_1}(q(t))$ and $g_{a_2}(q(t))$.

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

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, $g_{a_1}(q(t)) \leq g_{a_2}(q(t))$ 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 $t \geq 0$, 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 $\frac{d}{dt'} g_{a_1}(q(t')) \leq \frac{d}{dt'} g_{a_2}(q(t'))$, 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 A.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 A.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.*

A.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 $\operatorname{supp}(f)$ 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 $O(n \log n)$.

A.6 Number of relaxations

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

▶ **Lemma A.6 (Agarwal *et al.* [2]).** *If arcs of $\operatorname{supp}(f)$ are relaxed first as they arrive on the frontier, then $E(\operatorname{supp}(f))$ 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 $F \coloneqq E(\text{supp}(f)) \setminus \bigcup_{a \in A} E(\Sigma_a)$. Using Lemma A.6, we can show that the number of support arcs outside support stars ($|F|$) is small.

▶ **Lemma A.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 A.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 4.2.** *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 A.7. The lemma follows. ◀

▶ **Lemma 4.3.** *Hungarian search takes $O(r\sqrt{n}\,\mathrm{polylog}\,n)$ time.*

Proof. The number of relaxation steps outside of support stars is $O(r)$ by Lemma 4.2. The time per relaxation outside of support stars is $O(\sqrt{n}\,\mathrm{polylog}\,n)$. The time spent processing relaxations within a support star is $O(\sqrt{n}\,\mathrm{polylog}\,n)$, and at most r are relaxed during the search. The total time is therefore $O(r\sqrt{n}\,\mathrm{polylog}\,n)$. ◀

A.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 $|\Sigma_a| \le \sqrt{n}/2$, and star that is currently small is turned into a big star once $|\Sigma_a| \ge 2\sqrt{n}$. 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 $|\Sigma_a| \le \sqrt{n}/2$. The state change deletes $\mathcal{D}_{\mathrm{big}}(a)$ and inserts Σ_a into $\mathcal{D}_{\mathrm{small}}$. Thus, the time spent for a big-to-small state change is $O(\sqrt{n}\,\mathrm{polylog}\,n)$, 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 $O(\mathrm{polylog}\,n)$.

A star Σ_a undergoing a small-to-big state change has size $|\Sigma_a| \ge 2\sqrt{n}$. We can write its size as $|\Sigma_a| = \sqrt{n} + x$ for some integer $x \ge \sqrt{n}$, so we also have $|\Sigma_a| \le 2x$. When switching, we delete all $|\Sigma_a|$ points from $\mathcal{D}_{\mathrm{small}}$ and construct a new $\mathcal{D}_{\mathrm{big}}(a)$. Constructing $\mathcal{D}_{\mathrm{big}}(a)$ 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 $(r + 2x) \cdot O(\mathrm{polylog}\,n)$, 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 $O((r/x)\,\mathrm{polylog}\,n)$. Since $x \ge \sqrt{n}$, this is at most $O((r/\sqrt{n})\,\mathrm{polylog}\,n)$.

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 4.1 and 4.2. Thus, the total time spent on state changes is $O((r^2\sqrt{n} + rn)\,\mathrm{polylog}\,n)$.