Geometric Partial Matching and Unbalanced Transportation

Pankaj K. Agarwal Hsien-Chih Chang Allen Xiao

November 27, 2018

1 Introduction

Consider the problem of finding a minimum-cost bichromatic matching between a set of red points A and a set of blue points B lying in the plane, where the cost of a matching edge (a, b) is the Euclidean distance $\|a - b\|$; in other words, the minimum-cost bipartite matching problem on the Euclidean complete graph $G = (A \cup B, A \times B)$. Let $r := |A|$ and $n := |B|$. Without loss of generality, assume that $r \leq n$. We consider the problem of *partial matching*, where the task is to find a minimum-cost matching of size $k \leq r$. When $k = r = n$, we say the matching instance is **balanced**. When $k = r < n$ (A and B have different sizes, but the matching is maximal), we say the matching instance is **unblanced**. We call the geometric problem of finding a size k matching of point sets A and B the **geometric partial matching problem**.

1.1 Contributions

In this paper, we present two algorithms for geometric partial matching that are based on fitting nearest-neighbor (NN) and geometric closest pair (BCP) oracles into primal-dual algorithms for non-geometric bipartite matching and minimum-cost flow. This pattern is not new, see for example (...)⟨⟨**TODO cite**⟩⟩. Unlike these previous works, we focus on obtaining running time dependencies on k or r instead of n, that is, faster for inputs with small r or k. We begin in Section 2 by introducing notation for matching and minimum-cost flow.

First in Section 3, we show that the Hungarian algorithm [6] combined with a BCP oracle solves geometric partial matching exactly in time $O((n + k^2) \operatorname{polylog} n)$. Mainly, we show that we can separate the $O(n \operatorname{polylog} n)$ preprocessing time for building the BCP data structure from the augmenting paths' search time, and update duals in a lazy fashion such that the number of dual updates per augmenting path is $O(k)$.

Theorem 1.1. *Let A and B be two point sets in the plane with $|A| = r$ and $|B| = n$ satisfying $r \leq n$, and let k be a parameter. A minimum-cost geometric partial matching of size k can be computed between A and B in $O((n + k^2) \operatorname{polylog} n)$ time.*

⟨⟨**State the settings separately so no need to repeat in the theorem statement.**⟩⟩

Next in Section 5, we apply a similar technique to the unit-capacity min-cost circulation algorithm of Goldberg, Hed, Kaplan, and Tarjan [3]. The resulting algorithm finds a $(1 + \varepsilon)$-approximation to the optimal geometric partial matching in $O((n + k\sqrt{k}) \operatorname{polylog} n \log(n/\varepsilon))$ time.

Theorem 1.2. *Let A and B be two point sets in the plane with $|A| = r$ and $|B| = n$ satisfying $r \leq n$, and let k be a parameter. A $(1 + \varepsilon)$ geometric partial matching of size k can be computed between A and B in $O((n + k\sqrt{k}) \operatorname{polylog} n \log(n/\varepsilon))$ time.*

Our third algorithm solves the transportation problem in the unbalanced setting. The transportation problem is a weighted generalization of the matching problem. Each point of A is weighted with an integer *supply* and each point of B is weighted with integer *demand* such that the sum of supply and demand are equal. The goal of the transportation problem is to find a minimum-cost mapping of all supplies to demands, where the cost of moving a unit of supply at $a \in A$ to satsify a unit of demand at $b \in B$ is $\|a - b\|$. For this, we use the strongly polynomial uncapacitated min-cost flow algorithm by Orlin [7]. The result is an $O(n^{3/2} r \operatorname{polylog} n)$ time algorithm for unbalanced transportation. This improves over the $O(n^2 \operatorname{polylog} n)$ time algorithm of Agarwal *et al.* [1] when $r = o(\sqrt{n})$.

Theorem 1.3. *Let A and B be two point sets in the plane with $|A| = r$ and $|B| = n$ satisfying $r \leq n$, with supplies and demands given by the function $\lambda : (A \cup B) \to \mathbb{Z}$ such that $\sum_{a \in A} \lambda(a) = \sum_{b \in B} \lambda(b)$. An optimal transportation map can be computed in $O(rn^{3/2} \operatorname{polylog} n)$ time.*

By nature of the BCP/NN oracles we use, these results generalize to when $\|a - b\|$ is any L_p distance, and not just the Euclidean distance between a and b.

2 Preliminaries

2.1 Matching

Let G be a bipartite graph between vertex sets A and B and edge set E, with costs $c(v, w)$ for each edge e in E. ⟨⟨**Should we define the problem on point sets instead, and construct the graph afterwards?**⟩⟩ We use $C := \max_{e \in E} c(e)$, and assume that the problem is scaled such that $\min_{e \in E} c(e) = 1$. ⟨⟨**where do we use this assumption?**⟩⟩ A **matching** $M \subseteq E$ is a set of edges where no two edges share an endpoint. We use $V(M)$ to denote the vertices matched by M. The **size** of a matching is the number of edges in the set, and the **cost** of a matching is the sum of costs of its edges. The **minimum-cost partial matching problem (MPM)** asks to find a size-k matching M^* of minimum cost.

⟨⟨**Define LP-duality and admissibility for matchings**⟩⟩

3 Computing Min-cost Partial Matching using Hungarian algorithm

The Hungarian algorithm maintains a 0-optimal (initially empty) matching M, and repeatedly augments by alternating augmenting paths of admissible edges until $|M| = k$. To this end, the algorithm maintains a set of feasible potentials π and updates them to find augmenting paths of admissible edges. ⟨⟨**admissible augmenting path?**⟩⟩ It maintains the invariant that matching edges are admissible. Since there are k augmentations and each alternating path has length at most $2k - 1$, the total time spent on bookkeeping the matching is $O(k^2)$. This leaves the analysis of the subroutine that updates the potentials and finds an admissible augmenting path; we call this subroutine the **Hungarian search**.

Theorem 3.1 (Time for Hungarian algorithm)**.** *Let $G = (A \cup B, A \times B)$ be an instance of geometric partial matching with $r := |A|$, $n := |B|$, $r \leq n$, and parameter $k \leq r$. Suppose the Hungarian search finds each augmenting path in $T(n, k)$ time after a one-time $P(n, k)$ preprocessing time. Then, the Hungarian algorithm finds the optimal size k matching in time $O(P(n, k) + kT(n, k) + k^2)$.*

Algorithm 1 Hungarian algorithm

```
1: function MATCH(G = (A ∪ B, E), k)
2:     M ← ∅
3:     π(v) ← 0 for all v ∈ A ∪ B
4:     while |M| < k do
5:         Π ← HUNGARIAN-SEARCH(G, M, π)
6:         M ← M ⊕ Π
7:     return M
```

Algorithm 2 Hungarian Search (matching)

Requirement: $c_\pi(a,b) = 0$ for all $(a,b) \in M$

```
1: function HUNGARIAN-SEARCH(G = (A ∪ B, E), M, π)
2:     S ← a ∈ (A \ V(M))  ⟨⟨arbitrary a?⟩⟩
3:     repeat
4:         (a', b') ← arg min{c_π(a,b) | (a,b) ∈ (S ∩ A) × (B \ S)}
5:         γ ← c_π(a', b')
6:         π(v) ← π(v) + γ, ∀v ∈ S                ▷ make (a', b') admissible
7:         S ← S ∪ {b'}
8:         if b' ∉ V(M) then                       ▷ b' unmatched
9:             Π ← alternating augmenting path from a to b'
10:            return Π
11:        else                                    ▷ b' is matched to some a'' ∈ A ∩ V(M)
12:            S ← S ∪ {a''}
13:    until S = A ∪ B
14:    return failure
```

3.1 Hungarian search

Let S be the set of vertices that can be reached from an unmatched $a \in A$ by admissible residual edges, initially the unmatched vertices of A. The Hungarian search updates potentials in a Dijkstra's algorithm-like manner, expanding S until it includes an unmatched $b \in B$ (and thus an admissible alternating augmenting path). The "search frontier" of the Hungarian search is $(S \cap A) \times (B \setminus S)$. We *relax* the minimum-reduced cost edge in the frontier, changing the potentials of vertices S such that the edge becomes admissible, and adding the head of the edge into S. ⟨⟨**Well, either you add both endpoints into S, or an admissible augmenting path is found.**⟩⟩

The potential update uniformly decreases the reduced costs of the frontier edges. Since (a', b') is the minimum reduced cost frontier edge, the potential update in line 6 does not make any reduced cost negative, and thus preserves the dual feasibility constraint for all edges. The algorithm is shown below as Algorithm 2.

By tracking the forest of relaxed edges (e.g. back pointers), it is straightforward to recover the alternating augmenting path Π once we reach an unmatched $b' \in B$. We make the following observation about the Hungarian search:

Lemma 3.2. *There are at most k edge relaxations before the Hungarian search finds an alternating augmenting path.*

Proof. Each edge relaxation either leads to a matched vertex in B (there are at most $k-1$ such vertices), or finds an unmatched vertex and ends the search. □

In general graphs, the minimum edge is typically found by pushing all encountered $(S \cap A) \times (B \setminus S)$ edges into a priority queue. However, in the bipartite complete graph, this may take $\Theta(rn \operatorname{polylog} n)$ time for each Hungarian search — edges are being pushed into the queue even when they are not relaxed. We avoid this problem by finding an edge with minimum cost using **bichromatic closest pair** (BCP) queries on an additively weighted Euclidean distances, for which there exist fast dynamic data structures. Given two point sets P and Q in the plane, the BCP is the pair of points $p \in P$ and $q \in Q$ minimizing the (adjusted) distance $\|p-q\| - \omega(p) + \omega(q)$, for some real-valued vertex weights $\omega(p)$. In our setting, the vertex weights will mostly be set as the potentials; the adjusted distance is equal to the reduced cost.

⟨⟨**Short history on BCP?**⟩⟩ The state of the art dynamic BCP data structure from Kaplan, Mulzer, Roditty, Seiferth, and Sharir [5] supports point insertions and deletions in $O(\operatorname{polylog} n)$ time, and answers queries in $O(\log^2 n)$ time. The following lemma, combined with Theorem 3.1, completes the proof of Theorem 1.1.

Lemma 3.3. *Using the dynamic BCP data structure from Kaplan et al., we can implement Hungarian search with $T(n,k) = O(k \operatorname{polylog} n)$ and $P(n,k) = O(n \operatorname{polylog} n)$.*

Proof. Recall that we maintain a BCP data structure between $P = (S \cap A)$ and $Q = (B \setminus S)$. Changes to the P and Q are entirely driven by changing S; that is, updates to S incur BCP insertions/deletions. We first analyze the bookkeeping besides the potential updates, and then show how potential updates can be implemented efficiently.

⟨⟨**Untangles the algorithm with the proof; move the algorithm out of the proof of the lemma.**⟩⟩

1. Let S_0^t ⟨⟨**Is there a reason why you want the subscript? Do you ever define S^t?**⟩⟩ denote the initial set S at the beginning of the t-th Hungarian search, that is, the set of unmatched points in A after t augmentations. At the very beginning of the Hungarian algorithm, we initialize $S_0^0 \leftarrow A$ (meaning that $P = A$ and $Q = B$), which is a one-time insertion of $O(n)$ points into BCP, attributed to $P(n,k)$. On each successive Hungarian search, S_0^t shrinks as more and more points in A are matched. Assume for now that, at the beginning of the $(t+1)$-th Hungarian search, we are able to construct S_0^t from the previous iteration. To construct S_0^{t+1}, we simply remove the point in A that was matched by the t-th augmenting path. Thus, with that assumption, we are able to initialize S using one BCP deletion operation per augmentation.

2. During each Hungarian search, points are added to P (that is, some points in A are added to S) and removed from Q (points in B added to S), which will happen at most once per edge relaxation. By Lemma 3.2 the number of relaxed edges is at most k, so the number of such BCP operations is also at most k.

3. To obtain S_0^t, we keep track ⟨⟨**give a name to such points**⟩⟩ of the points added since S_0^t in the last Hungarian search (i.e. those of (2)). ⟨⟨**Unclear**⟩⟩ After the augmentation, we use this log ⟨⟨**use the name**⟩⟩ to delete the added vertices from S and recover S_0^t. By the argument in (2) there are $O(k)$ of such points to delete, so reconstructing S_0^t takes $O(k)$ BCP operations. ⟨⟨**TODO change to full persistence: loglogm overhead with $m = r$ modifications**⟩⟩

We spend $P(n,k) = O(n \operatorname{polylog} n)$ time to build the initial BCP. The number of BCP operations associated with each Hungarian search is $O(k)$, so the time spent on BCP operations in each Hungarian search is $O(k \operatorname{polylog} n)$.

As for the potential updates, we modify a trick from Vaidya [9] to batch potential updates. Potentials have a **stored value**, i.e. the current value of $\pi(v)$, and a **true value**, which may have changed from $\pi(v)$. The algorithm uses the true value when dealing with reduced costs and updates the stored value rarely; we explain the mechanism shortly.

Throughout the course of the algorithm, we maintain a nonnegative value δ (initially 0) which aggregates potential changes. Vertices that are added to S are immediately added to a BCP data structure with weight $\omega(p) \leftarrow \pi(p) - \delta$, for whatever value δ is at the time of insertion. When the points of S have potentials increased by γ in (2), we instead raise $\delta \leftarrow \delta + \gamma$. Thus, true value for any potential of a point in S is $\omega(p) + \delta$. For points of $(A \cup B) \setminus S$, the true potential is equal to the stored potential.

Since potentials for S points are uniformly offsetted by δ, the minimum edge returned by the BCP oracle does not change. Once a point is removed from S, we update its stored potential to be $\pi(p) \leftarrow \omega(p) + \delta$, for the current value of δ. Importantly, δ is not reset at the end of a Hungarian search, and persists throughout the entire algorithm. This way, the unmatched points in each S_0^t have their true potentials accurately represented by δ and $\omega(p)$.

The number of updates to δ is equal to the number of edge relaxations, which is $O(k)$ per Hungarian search. We update stored potentials when removing a point from S (by the rewind mechanism, or due to an augmentation) which occurs $O(k)$ times per Hungarian search. The time spent on potential updates per Hungarian search is therefore $O(k)$. Overall, the time spent per Hungarian search is $T(n,k) = O(k \operatorname{polylog} n)$.

⟨⟨**The proof gets more handwavy as the paragraph progresses. Consider a revision after this round.**⟩⟩ □

4 Approximating Min-Cost Partial Matching through Cost-Scaling

The goal of section is to prove Theorem 1.2; that is, to compute a geometric partial matching of size k between two point sets A and B in the plane, with cost at most $(1+\varepsilon)$ times the optimal matching, in time $O((n + k\sqrt{k}) \operatorname{polylog} n \log(n/\varepsilon))$.

⟨⟨**Insert outline of the section.**⟩⟩

4.1 Preliminaries on Network Flows

Network. Let $G = (V, E)$ be a directed graph, augemtned by edge costs c and capacities u, and a supply-demand function ϕ defined on the vertices. One can turn the graph G into a **network** $\boldsymbol{N = (V, A)}$: For each directed edge (v, w) in E, insert two **arcs** $v{\to}w$ and $w{\to}v$ into the arc set A ⟨⟨**better notation?**⟩⟩; the **forward arc** $v{\to}w$ inherits the capacity and cost from the directed graph G (that is, $u(v{\to}w) = u(v,w)$ and $c(v{\to}w) = c(v,w)$), 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, A) 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; in other words,

$$\sum_{v \in V(G)} \phi(v) = 0.$$

A **unit-capacity** network has all its edge capacities equal to 1. In this section assume all networks are of unit-capacity. ⟨⟨**correct?**⟩⟩

Pseudoflows. Given a network $N := (V, A, c, u, \phi)$, a **pseudoflow** (or **flow** to be short) $f : A \to \mathbb{Z}$ on N is an antisymmetric function on the arcs of N satisfying $f(v{\to}w) \leq u(v{\to}w)$ for every arc $v{\to}w$.[1] We sometimes abuse the terminology by allowing pseudoflow to be defined on a directed graph, in which case we are actually refering to the pseudoflow on the corresponding network by extending the flow values antisymmetrically to the arcs. 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 $\mathbf{supp}(f)$, is the set of arcs with positive flows:

$$\mathrm{supp}(f) := \{v{\to}w \in A \mid f(v{\to}w) > 0\}.$$

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

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

We call positive imbalance **excess** and negative imbalance **deficit**; and vertices with positive and negative imbalance **excess vertices** and **deficit vertices**, respectively. 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

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

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

Residual network. Given a pseudoflow f, one can defined the *residual network* as follows. Recall that the set of *residual arcs* A_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 $N = (V, A, c, u, \phi)$ be a network with a pseudoflow f. The **residual graph** has V as its vertex set and A_f as its arc set. The **residual capacity** $\boldsymbol{u_f}$ 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:

$$A_f = \{v{\to}w \mid u_f(v{\to}w) > 0\}.$$

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

LP-duality and admissibility. To solve the minimum-cost flow problem, we focus on the primal-dual algorithms using linear programming. Let $G = (V, E)$ be a given directed graph with the corresponding network $N = (V, A, c, u, \phi)$. Formally, the **potentials** $\boldsymbol{\pi(v)}$ are the variables of the linear program dual to the standard linear program for the minimum-cost flow problem, with

[1] In general the pseudoflows are allowed to take real-values. Here under the unit-capacity assumption any optimal flows are integer-valued. ⟨⟨**cite integrality theorem?**⟩⟩

variables $f(v, w)$ for each directed edge in E. Assignments to the primal variables satisfying the capacity constraints extend natually into a pseudoflow on the network N. Let (V, A_f) be the residual graph under pseudoflow f. The **reduced cost** of an arc $v \to w$ in A_f with respect to π is defined as

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

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 pseudoflow f is **ε-optimal [?, ?]**, with respect to π if $c_\pi(v \to w) \geq -\varepsilon$ for every residual arc $v \to w$ in A_f; pseudoflow f is **ε-optimal** if it is ε-optimal with respect to some potentials π. Given a pseudoflow f and potentials π, a residual arc $v \to w$ in A_f is **admissible** if $c_\pi(v \to w) \leq 0$. We say that a pseudoflow f' in G_f is **admissible** if all support arcs of f' on G_f are admissible; in other words, $f'(v \to w) > 0$ holds only on admissible arcs $v \to w$.

Lemma 4.1. *Let f be an ε-optimal pseudoflow in G and let f' be an admissible flow in G_f. Then $f + f'$ is also ε-optimal.* ⟨⟨***Lemma 5.3 in [4]?***⟩⟩

Proof. Augmentation by f' will not change the potentials, so any previously ε-optimal arcs remain ε-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 ε-optimality condition.

If an arc $v \to w$ is newly introduced this way, then by defintion 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 $v \to w$ ($f'(w \to v) > 0$). By assumption, the arcs of $\operatorname{supp}(f')$ are admissible, so $w \to v$ was an admissible arc ($c_\pi(w \to v) \leq 0$). By antisymmetry of reduced costs, this implies $c_\pi(v \to w) \geq 0 \geq -\varepsilon$. Therefore, all arcs with $u_{f+f'}(v, w) > 0$ respect the ε-optimality condition, and thus $f + f'$ is ε-optimal. □

4.2 Reduction to Unit-Capacity Min-Cost Flow Problem

The goal of the subsection is to reduce the minimun-cost partial matching problem to the unit-capacity minimum-cost flow problem with a polynomial bound on diameter. To this end we first provide an upper bound on the size of support of an integral pseudoflow on the standard reduction network between the two problems. This upper bound in turns provides an additive approximation on the cost of an ε-optimal circulation. Next we employ a technique by Sharathkumar and Agarwal [8] to transform an additive ε-approximate solution into a multiplicative $(1 + \varepsilon)$-approximation for the geometric partial matching problem. The reduction does not work out of the box, as Sharathkumar and Agarwal were tackling a similar but different problem on geometric transportations.

Lemma 4.2. *Computing a $(1 + \varepsilon)$-approximate geometric partial matching can be reduced to the following problem in $O(n \operatorname{polylog} n)$ time: Given a reduction network N over a point set with diameter at most $K \cdot kn^3$ for some constant K, compute an $(K \cdot \varepsilon/6k)$-optimal circulation on N.*

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 network N_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 this way has cost 0 and capacity 1. Denote the new graph with vertex set $V = A \cup B \cup \{s, t\}$ and edge set E as the **reduction graph** $\boldsymbol{H}$. 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 naturally corresponds to H as the **reduction network**, denoted by $\boldsymbol{N_H}$.

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 $\boldsymbol{M_f}$. Notice that the cost of the circulation f is equal to the cost of the corresponding matching M_f. In other words, a $(1+\varepsilon)$-approximation to the MCF problem on the reduction graph H translates to a $(1+\varepsilon)$-approximation to the geometric matching problem on the input graph G.

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

Lemma 4.3. *Let f be an integer circulation in the reduction network N_H. Then, the size of the support of f is at most $3k$. 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 $3k$. □

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

Lemma 4.4. *Let f be an ε-optimal integer circulation in N_H, and f^* be an optimal integer circulation for N_H. Then, $\text{cost}(f) \le \text{cost}(f^*) + 6k\varepsilon$.*

Proof. By Lemma 4.3, 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 $\sum_i |\Gamma_i| \le 6k$.

Let π be some potential certifying that f is ε-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

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

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

Multiplicative approximation. Now we empoly a technique from Sharathkumar and Agarwal [8] to convert the additive approximation into a multiplicative one.

Let T be the minimum spanning tree on input graph G and order its edges by increasing length as $e_1, \ldots, e_{r+n-1}$. Let T_ℓ denote the subgraph of T obtained by removing the heaviest ℓ edges in T. Let i be the largest index so that the optimal solution to the MPM problem has edges between components of T_i. Choose j to be the smallest index larger than i satisfying $c(e_j) \ge kn \cdot c(e_i)$. For each component K of T_j, let G_K be the subgraph of G induced on vertices of K; let $\boldsymbol{A_K} := K \cap A$ and $\boldsymbol{B_K} := K \cap B$, respectively. We partition A and B into the collection of sets A_K and B_K according to the components K of T_j. Since $j < i$, the optimal partial matching in G can be partitioned into edges between A_K and B_K within G_K; no optimal matching edges lie between components.

Lemma 4.5 (Sharathkumar and Agarwal [8, §3.5]). *Let $G = (A, B, E_0)$ be the input to MPM problem, and consider the partitions A_K and B_K defined as above. Let M^* be the optimal partial matching in G. Then,*

(i) $c(e_i) \leq \text{cost}(M^*) \leq kn \cdot c(e_i)$, *and*

(ii) the diameter of G_K is at most $kn^2 \cdot c(e_i)$ for every $K \in T_j$,

To prove Lemma 4.2, we need to further modify the point set so that the cost of the optimal solution does not change, while the diameter of the *whole* point set is bounded. Move the points within each component in *translation* so that the minimum distances between points across components are at least $kn \cdot c(e_i)$ but at most $O(n \cdot kn^2 \cdot c(e_i))$. This will guarantee that the optimal solution still uses edges within the components by Lemma 4.5. The simpliest way of achieveing this is by aligning the components one by one into a "straight line", so that the distance between the two farthest components is at most $O(n)$ times the maximum diameter of the cluster.

Now one can prove Lemma 4.2 by computing an $(\varepsilon c(e_i)/6k)$-optimal circulation f on the point set after translations using additive approximation from Lemma 4.4, together with the bound $c(e_i) \leq \text{cost}(M^*)$ from Lemma 4.5.

One small problem remains: We need to show that such reduction can be performed in $O(n \operatorname{polylog} n)$ time. Sharathkumar and Agarwal [8] have shown that the partition of A and B into A_Ks and B_Ks can be computed in $O(n \operatorname{polylog} n)$ time, assuming that the indices i and j can be determined in such time as well. However in our application the choice of index i depends on the optimal solution of MPM problem which we do not know.

To solve this issue we perform a binary search on the edges $e_1, \ldots, e_{r+n-1}$. ⟨⟨**Hmm, we have no way to check Lemma 4.5(i); but in fact a polynomial bound is good enough.**⟩⟩ ⟨⟨**UNRESOLVED ISSUE**⟩⟩

4.3 High-Level Description of Cost-Scaling Algorithm

Our main algorithm for the unit-capacity minimum-cost flow problem is based on the **cost-scaling** technique, originally due to Goldberg and Tarjan [4]; Goldberg, Hed, Kaplan, and Tarjan [3] applied the technique on unit-capacity networks. The algorithm finds ε-optimal circulations for geometrically shrinking values of ε. Each fixed value of ε is called a **cost scale**. Once ε is sufficiently small, the ε-optimal flow is a suitable approximation according to Lemma 4.2[2]

The cost-scaling algorithm initializes the flow f and the potential π to be zero. Note that the zero flow is trivially a kC-optimal flow. At the beginning of each scale starting at $\varepsilon = kC$,

- SCALE-INIT takes the previous circulation (now 2ε-optimal) and transforms it into an ε-optimal pseudoflow with $O(k)$ excess.
- REFINE then reduces the excess in the newly constructed pseudoflow to zero, making it an ε-optimal circulation.

Thus, the algorithm produces an ε^*-optimal circulation after $O(\log(kC/\varepsilon^*))$ scales. Using the reduction in Lemma 4.2, we have the diameter of the point set, thus maximum cost C, bounded by $O(K \cdot kn^3)$ for some value K. By setting ε^* to be $K \cdot \varepsilon/6k$, the number of cost scales is bounded above by $O(\log(n/\varepsilon))$.

[2] When the costs are integers, an ε-optimal circulation for a sufficiently small ε (say less than $1/n$) is itself an optimal solution [4, 3]. We present this algorithm without the integral-cost assumption because in the geometric partial matching setting (with respect to Euclidean distances) the costs are generally not integers.

Scale initialization. Recall that H is the *reduction graph* and N_H is the *reduction network*, both constructed in Section 4.2. The vertex set of H consists of the 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 the **forward arcs**, and those arcs that points in the opposite direction as **backward arcs**.

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

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

Because the arcs are of unit-capacity in N_H, each desaturation creates one unit of excess. By Lemma 4.3 the number of backward arcs is at most $3k$. Thus the total amount of excess created is also $O(k)$.

In total, potential updates and backward arc desaturations, and thus the whole precedure SCALE-INIT, take $O(n)$ time.

Refinement. The procedure REFINE is implemented using a primal-dual augmentation algorithm, which sends flows on admissible arcs that reduces the total excess, like the Hungarian algorithm. Unlike the Hungarian algorithm, it uses *blocking flows* instead of augmenting paths. An **augmenting path** is a path in the residual network from an excess vertex to a deficit vertex. We call a pseudoflow f on residual network N_g a **blocking flow** if ⟨⟨**f is admissible?**⟩⟩ f saturates at least one residual arc in every augmenting path in N_g. ⟨⟨**Is N_g an admissible network?**⟩⟩ 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 ε, 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 edges to construct an admissible blocking flow. It suffices to repeatedly extract admissible augmenting paths until no more admissible excess-deficit paths remain. By definition, the union of such paths is a blocking flow. ⟨⟨**Move to where the blocking flow is introduced?**⟩⟩

The algorithm continues until the total excess becomes zero and the ε-optimal flow is now a circulation.

First we analysis the number of iterations executed by REFINE. The proof follows the strategy in Goldberg *et al.* [3, Section 3.2]. ⟨⟨**and maybe §5 of Goldberg-Tarjan?**⟩⟩ To this end we need a bound on the size of the support of f right before and throughout the execution of REFINE.

Lemma 4.6. *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 $\operatorname{supp}(f)$ 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 4.7. *The size of* $\mathrm{supp}(f)$ *is at most* $O(k)$ *for pseudoflow* f *right before or during the execution of* REFINE.

Lemma 4.8. *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 flow and potential at the start of the procedure REFINE. Let f and π be the current flow and the potential. Let $\boldsymbol{d(v)}$ defined to be the amount of potential increase at v, measured in units of ε; in other words, $d(v) := (\pi(v) - \pi_0(v))/\varepsilon$. Consider the set of arcs $\boldsymbol{E^+} := \{v{\to}w \mid f(v{\to}w) < f_0(v{\to}w)\}$. Goldberg *et al.* [3, Lemma 3.5] showed that every vertex v has $d(v) \le 3n - 3$, which we can improve to $O(k)$ on our reduction network, by the fact that the size of E^+ is bounded by the sum of support sizes of f and f_0, which by Corollary 4.7 is at most $O(k)$.

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 $d(v) \ge \sqrt{k}$. At most $\sqrt{k}$ iterations belong to the first phase as each Hungarian search increases the potential π by at least ε 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 $O(\sqrt{k})$. The total amount of excess is upper bounded by the number of arcs in E^+ that crosses an arbitrarily given cut X (when the network has unit-capacity) [3, Lemma 3.6]. Consider the set of cuts $X_i := \{v \mid d(v) > i\}$ for $0 \le i < \sqrt{k}$. Each arc in E^+ crosses at most 3 cuts of type X_i [3, Lemma 3.1]. So there is one X_i crossed by at most $3|E^+|/\sqrt{k}$ arcs in E^+. Again by Corollary 4.7 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. □

The goal of the rest 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 the $O(\sqrt{k})$ bound on the number of iterations we just proved, the procedure REFINE can be implemented in $O((n + k\sqrt{k}) \operatorname{polylog} n)$ time. Together with our analysis on scale initialation and the bound on number of cost scales, this concludes the proof to Theorem 1.2.

4.4 Fast Implementation

5 Approximating Min-cost Partial Matchings — OLD

⟨⟨**THIS IS THE OLD SECTION**⟩⟩

5.0.1 Empty vertices and the shortcut graph

⟨⟨**A figure might be helpful for this section.**⟩⟩

As it turns out, there are some vertices whose relaxation events we cannot charge to the support size. However, we can replace H_f with an equivalent graph that excludes them, and run HUNGARIAN-SEARCH2 and DFS on the resulting graph.

We say $v \in A \cup B$ is an **empty vertex** if $\phi_f(v) = 0$ and no edges of $\text{supp}(f)$ adjoin v. ⟨⟨**Hmm. How do you feel about calling them "irrelavant vertices" or "null vertices"?**⟩⟩ We are unable to charge relaxation steps involving empty vertices to $|\text{supp}(f)|$, so the algorithm must deal with them separately. Namely, there is no edge with $f(e) > 0$ adjacent to an empty vertex, reaching an empty vertex does not terminate the search, and there may be $\Omega(n)$ empty vertices at once (consider $H_{f=0}$ ⟨⟨**notation overload**⟩⟩, the residual graph of the empty flow). We use $A_\emptyset$ and $B_\emptyset$ to denote the empty vertices of A and B respectively. Vertices that are not empty are called **non-empty vertices**.

For an empty vertex v, either residual in-degree ($v \in A_\emptyset$) or residual out-degree ($v \in B_\emptyset$) is 1. Call a length 2 paths through v to/from non-empty vertices an **empty 2-path**. For example, if $v \in A_\emptyset$ (resp. $v \in B_\emptyset$), then its empty 2-paths have the form (s, v, b) (resp. (a, v, t)) for each $b \in B \setminus B_\emptyset$ (resp. $a \in A \setminus A_\emptyset$). We say that (s, v, b) is an empty 2-path **surrounding** empty vertex v. Separately, we define the length 3 s-t paths that pass through two empty vertices to be **empty 3-paths**. As with 2-paths, we say an empty 3-path (s, v_1, v_2, t) surrounds $v_1 \in A_\emptyset$ and $v_2 \in B_\emptyset$.

As for the costs of empty paths, consider an empty 2-path (s, v, b) that surrounds $v \in A_\emptyset$. Because reduced costs telescope for residual paths, the reduced cost of (s, v, b) does not depend on the potential of v.

$$c_\pi((s, v, b)) = c_\pi(s, v) + c_\pi(v, b) = c(v, b) - \pi(s) + \pi(b)$$

Something similar holds for empty 2-paths surrounding $B_\emptyset$ vertices, and empty 3-paths.

We construct the **shortcut graph** $\tilde{H}_f$ from H_f by removing all empty vertices and their adjacent edges, and then inserting a direct arc between the end points of each empty path Π of equal cost. We call this direct edge the **shortcut** $\text{short}(\Pi)$ of empty path Π. For example, the empty 2-path (s, v, b) for $v \in A_\emptyset$ is replaced with a shortcut (s, b) of cost $c(\text{short}(s, v, b)) := c(v, b)$. Similarly, the empty 3-path (s, v_1, v_2, t) would be replaced with a shortcut (s, t) of cost $c(\text{short}((s, v_1, v_2 t))) := c(v_1, v_2)$.

The resulting multigraph $\tilde{H}_f$ contains only the non-empty vertices of V, and has the same connectivity between non-empty vertices as H_f. Consider a path Π from non-empty v to non-empty w in H_f. Any empty vertex in Π is surrounded by an empty 2- or 3-path contained in Π, since the only nontrivial residual paths through an empty vertex are its surrounding empty paths. Thus, there is a corresponding v-to-w path $\tilde{\Pi}$ in $\tilde{H}_f$ by replacing each empty path contained in Π with its shortcut. Furthermore, we have $c(\Pi) = c(\tilde{\Pi})$. We argue now that $\tilde{H}_f$ is fine as a surrogate for H_f, by showing that we can recover ε-optimal potentials for the non-empty vertices.

Lemma 5.1. *Let $\tilde{\pi}$ be a ε-optimal set of potentials for non-empty vertices of H_f. Construct potentials π, extending $\tilde{\pi}$ to empty vertices, by setting $\pi(a) \leftarrow \tilde{\pi}(s)$ for $a \in A_\emptyset$ and $\pi(b) \leftarrow \tilde{\pi}(t)$ for $b \in B_\emptyset$. Then,*

1. *π is a set of ε-optimal potentials for H_f, and*
2. *if a shortcut $\text{short}(\Pi)$ is admissible under $\tilde{\pi}$, then every arc of Π is admissible under π.*

Proof. Reduced costs for non-empty to non-empty arcs are unchanged between $\tilde{\pi}$ and π, so ε-optimality are preserved for these. Recall that an empty path is comprised of one A-to-B arc, and 1 or 2 zero-cost arcs (connecting the empty vertex/vertices to s and t). With our choice of empty vertex potentials, we observe that the zero-cost arcs have reduced cost 0: for an empty $a \in A_\emptyset$, $c_\pi(s, a) = 0$, for an empty $b \in B_\emptyset$, $c_\pi(b, t) = 0$. These arcs are both ε-optimal ($\geq -\varepsilon$) and admissible

(≤ 0), so it remains to prove ε-optimality and admissibility for arcs (a, b) where either a or b is an empty vertex.

Let $(a, b) \in A \times B$ such that at least one of a or b is empty. There exists an empty path Π that contains (a, b). Observe that $c_\pi(a, b) = c_\pi(\Pi)$, which we can prove for all varieties of empty paths.

- If $\Pi = (s, a, b)$ for $a \in A_\emptyset$:

$$c_\pi(a, b) = c(a, b) - \pi(a) + \pi(b) = c(a, b) - \pi(s) + \pi(b) = c_\pi(\Pi)$$

- If $\Pi = (a, b, t)$ for $b \in B_\emptyset$:

$$c_\pi(a, b) = c(a, b) - \pi(a) + \pi(b) = c(a, b) - \pi(a) + \pi(t) = c_\pi(\Pi)$$

- If $\Pi = (s, a, b, t)$ for $a \in A_\emptyset$ and $b \in B_\emptyset$:

$$c_\pi(a, b) = c(a, b) - \pi(a) + \pi(b) = c(a, b) - \pi(s) + \pi(t) = c_\pi(\Pi)$$

By construction, $c_\pi(\Pi) = c_{\tilde{\pi}}(\text{short}(\Pi))$, so we have $c_\pi(a, b) = c_{\tilde{\pi}}(\text{short}(\Pi)) \geq -\varepsilon$ and (a, b) is ε-optimal. Additionally, if $\text{short}(\Pi)$ is admissible under $\tilde{\pi}$, then so is (a, b) under π. Empty paths cover all arcs adjoining empty vertices, so we have proved both parts of the lemma for all arcs in H_f. □

In REFINE, we do not explicitly construct $\tilde{H}_f$ for running HUNGARIAN-SEARCH2 or DFS, but query its edges using BCP/NN oracles and min/max heaps on elements of H_f. Potentials for empty vertices are only required at the end of REFINE (for the next scale), and right before an augmentation sends flow through an empty path, making its surrounded vertices non-empty. During these occasions, we use the procedure in Lemma 5.1 to find feasible, ε-optimal potentials for empty vertices which also preserve the structure of admissibility.

Lemma 5.2. *The number of end-of-*REFINE *empty vertex potential updates is $O(n)$. The number of augmentation-induced empty vertex potential updates in each invocation of* REFINE *is $O(\sum_i N_i)$ where N_i is the number of positive flow arcs in the i-th blocking flow.*

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 empty path, causing a potential update on the surrounded empty vertex. We charge this potential update to the edges of that empty 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 $O(N_i)$ for the i-th blocking flow, and $O(\sum_i N_i)$ over the course of REFINE. □

Ultimately, we prove that $\sum_i N_i = O(k\sqrt{k})$, but this requires that we explain the process creating each blocking flow. We revisit this lemma after analyzing DFS.

5.0.2 Hungarian search

Logically, we are executing the Hungarian search ("raise prices") from [3, Section 3.2] on the shortcut graph $\tilde{H}_f$. We describe how we can query the minimum-reduced cost arc leaving S in $O(\text{polylog}\, n)$ time, for the shortcut graph, without constructing $\tilde{H}_f$ explicitly. For this purpose, let S' be a set of "reached" vertices maintained alongside S, identical except whenever a shortcut is relaxed, we add its surrounded empty vertices to S' in addition to its (non-empty) endpoints. Observe that the arcs of $\tilde{H}_f$ leaving S fall into $O(1)$ categories.

Algorithm 3 Hungarian Search (cost-scaling)

function HUNGARIAN-SEARCH2($H = (V, E)$, f, π)
  $\tilde{H}_f \leftarrow$ the shortcut graph of H_f
  $S \leftarrow \{v \in V \mid \phi_f(v) > 0\}$
  repeat
    $(v', w') \leftarrow \arg\min\{c_\pi(v', w') \mid v' \in S, w' \notin S, (v', w') \in \tilde{H}_f)\}$
    $\gamma \leftarrow c_\pi(v', w')$
    if $\gamma > 0$ **then** ▷ make (v', w') admissible if it isn't
      $\pi(v) \leftarrow \pi(v) + \lceil \frac{\gamma}{\varepsilon} \rceil \cdot \varepsilon, \forall v \in S$
    $S \leftarrow S \cup \{w'\}$
    if $\phi_f(w') < 0$ **then** ▷ reached a deficit
      return π
  until $S = (A \setminus A_\emptyset) \cup (B \setminus B_\emptyset)$
  return failure

1. Non-shortcut backward arcs (v, w) with $(w, v) \in \text{supp}(f)$. For these, we can maintain a min-heap on $\text{supp}(f)$ arcs as each v arrives in S.

2. Non-shortcut A-to-B forward arcs. For these, we can use a BCP data structure between $(A \setminus A_\emptyset) \cap S$ and $(B \setminus B_\emptyset) \setminus S$, 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 $B \setminus S$, queried while $s \in S$. For t, we can maintain a max-heap on the potentials of $A \cap S$, queried while $t \notin S$.

4. Shortcut arcs (s, b) corresponding to empty 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 empty 2-path (s, a, b). This is only queried while $s \in S'$.

5. Shortcut arcs (a, t) corresponding to empty 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 empty 2-path (a, b, t). This is only queried while $t \notin S'$.

6. Shortcut arcs (s, t) corresponding to empty 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 empty 3-path (s, a, b, t). This is only queried while $s \in S'$ and $t \notin S'$.

By construction, the BCP distance of each datastructure in (4-6) is equal to the reduced cost of the shortcut, which is equal to the reduced cost of the corresponding empty path. Each of the above data structures requires one query per relaxation, and an insertion/deletion operation whenever a new vertex moves into S. The data structures above can perform both in $O(\text{polylog}\, n)$ time each, so the running time of HUNGARIAN-SEARCH2 outside of potential updates can be bounded in the number of relaxation steps.

Lemma 5.3. *There are $O(k)$ non-shortcut relaxations in* HUNGARIAN-SEARCH2 *before a deficit vertex is reached.*

Proof. Each edge relaxation adds a new vertex to S, and non-shortcut relaxations only add non-empty 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 $\operatorname{supp}(f)$ 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 non-empty, this must be 2. We charge 0.5 to each of the two $\operatorname{supp}(f)$ arcs; the arcs of $\operatorname{supp}(f)$ have no more than 1 charge each. Thus, the number of type (ii) vertex relaxations is $O(|\operatorname{supp}(f)|)$. By Corollary 4.7, $O(|\operatorname{supp}(f)|) = O(k)$. □

Lemma 5.4. *There are $O(k)$ shortcut relaxations in* HUNGARIAN-SEARCH2 *before a deficit vertex is reached.*

Proof. Recall the categories of shortcuts from the list of datastructures above. We have shortcuts corresponding to (i) empty 2-paths surrounding $a \in A_\emptyset$, (ii) empty 2-paths surrounding $b \in B_\emptyset$, and (iii) empty 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 non-empty $b \in B \setminus B_\emptyset$ into S. Since b is non-empty, it must either have deficit or an adjacent arc of $\operatorname{supp}(f)$. We charge this relaxation to b if it is deficit, or the adjacent arc of $\operatorname{supp}(f)$ otherwise. No vertex is charged more than once, and no $\operatorname{supp}(f)$ edge is charged more than twice, therefore the total number of type (i) relaxations is $O(|\operatorname{supp}(f)|)$. By Corollary 4.7, $O(|\operatorname{supp}(f)|) = O(k)$. □

Corollary 5.5. *There are $O(k)$ relaxations in* HUNGARIAN-SEARCH2 *before a deficit vertex is reached.*

In the following lemma, we complete the time analysis of HUNGARIAN-SEARCH2 by proving that potentials can be maintained in $O(k)$ time over the course of the search.

Lemma 5.6. *Using a dynamic BCP, we can implement* HUNGARIAN-SEARCH2 *with $T_1(n,k) = O(k \operatorname{polylog} n)$ and $P_1(n,k) = O(n \operatorname{polylog} n)$.*

Proof. The initial sets for each data structure can be constructed in $O(n \operatorname{polylog} n)$ time. For each of the $O(1)$ data structures that are queried during a relaxation, the new vertex moved into S as a result of the relaxation causes $O(1)$ insertion/deletion operations. For each of the data structures mentioned above, insertions and deletions can be performed in $O(\operatorname{polylog} n)$ time. Using Lemma 3.3 as a basis, we first analyze the number of BCP operations over the course of HUNGARIAN-SEARCH2.

1. Let S_0^t denote the initial set S at the beginning of the t-th Hungarian search, i.e. the set of $v \in V$ with $\phi_f(v) > 0$ after t blocking flows. Assume for now that, at the beginning of the $(t+1)$-th Hungarian search, we have on hand the S_0^t from the previous iteration. To construct S_0^{t+1}, we remove the vertices that had excess decreased to 0 by the t-th blocking flow. Thus, with that assumption, we are able to initialize S at the cost of one BCP deletion per excess vertex, which sums to $O(k)$ over the entire course of REFINE.

2. During each Hungarian search, a vertex entering S may cause P or Q to update and incur one BCP insertion/deletion. Like before, we can charge these to the number of edge relaxations over the course of HUNGARIAN-SEARCH2. The number of these is $O(k)$ by Corollary 5.5.

3. Like before, we can meet the assumption in (1) by rewinding a log of point additions to S, and recover S_0^t.

For potential updates, we use the same trick as in Lemma 3.3 to lazily update potentials after vertices leave S, but only for non-empty vertices. Non-empty vertices are stored in each data structure with weight $\omega(v) = \pi(v) - \delta$, and δ is increased in lieu of increasing the potential of all S vertices. When vertices leave S (through the rewind mechansim above), we restore their potentials as $\pi(v) \leftarrow \omega(v) + \delta$. With lazy updates, the number of potential updates on non-empty vertices is bounded by the number of relaxations in the Hungarian search, which is $O(k)$ by Corollary 5.5. Note that empty vertex potentials are not handled in HUNGARIAN-SEARCH2. □

5.0.3 Depth-first search

Algorithm 4 Depth-first search

function DFS($H = (V, E)$, f, π)
  $\tilde{H}_f \leftarrow$ the shortcut graph of H_f
  $f' \leftarrow 0$.
  $S \leftarrow \{v \in V \mid \phi_f(v) > 0\}$
  $S_0 \leftarrow \{v \in V \mid \phi_f(v) > 0\}$ ▷ stack of excess vertices
  $P \leftarrow$ POP(S_0) ▷ current path; stack
  repeat
    $v' \leftarrow$ PEEK(P)
    if $\phi_f(v') < 0$ **then** ▷ if we reached a deficit, save the path to f'
      add to f' a unit flow on the path P
      $P \leftarrow$ POP(S_0)
    else
      $w' \leftarrow \arg\min\{c_\pi(v', w') \mid w' \notin S, (v', w') \in \tilde{H}_f\}$
      $\gamma \leftarrow c_\pi(v', w')$
      if $\gamma \leq 0$ **then** ▷ if (v', w') is admissible, extend the current path
        $S \leftarrow S \cup \{w'\}$
        $P \leftarrow$ PUSH(P, w')
      else ▷ No admissible arcs leaving v', remove from P
        POP(P)
  until $S_0 = \emptyset$
  return f'

The depth-first search is similar to HUNGARIAN-SEARCH2 in that it uses the relaxation of minimum-reduced cost arcs/empty paths, this time to identify admissible arcs/empty paths in a depth-first manner. This requires some adjustments to the data structures for finding the minimum-reduced cost arc leaving $v' \in S$. Given $v' \in S$, we would like to query:

1. Non-shortcut backward arcs (v', w') with $(w', v') \in \text{supp}(f)$. For these, we can maintain a min-heap on $(w', v') \in \text{supp}(f)$ arcs for each non-empty $v' \in V$.

2. Non-shortcut A-to-B forward arcs. For these, we maintain a NN data structure over $P = (B \setminus B_\emptyset) \setminus S$, 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 $B \setminus S$, 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 empty 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 empty 2-path (s, a, b). This is only queried if $v' = s$.

5. Shortcut arcs (a, t) corresponding to empty 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 empty 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 $t \in S$.

6. Shortcut arcs (s, t) corresponding to empty 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 empty 3-path (s, a, b, t). This is only queried while $v' = s$ and $t \notin S'$.

Each data structure above performs $O(\text{polylog}\, n)$ time worth of query and insertion/deletion per relaxation, so the running time is again bounded by $O(\text{polylog}\, n)$ times the number of relaxations.

Lemma 5.7. *There are $O(k)$ non-shortcut relaxations in* DFS.

Lemma 5.8. *There are $O(k)$ shortcut relaxations in* DFS.

Corollary 5.9. *There are $O(k)$ relaxations in* DFS *before a deficit vertex is reached.*

There are no potentials to update within DFS, so the running time of DFS boils down to the time spent to querying and updating the data structures.

Lemma 5.10. *Using a dynamic NN, we can implement* DFS *with $T_2(n, k) = O(k \,\text{polylog}\, n)$ and $P_2(n, k) = O(n \,\text{polylog}\, n)$.*

Proof. At the beginning of REFINE, we can initialize the $O(1)$ data structures used in DFS in $P_2(n, k) = O(n \,\text{polylog}\, n)$ time. We use the same rewinding mechanism as HUNGARIAN-SEARCH2 (Lemma 5.6) to avoid reconstructing the data structures across iterations of REFINE, so the total time spent is bounded by the $O(\text{polylog}\, n)$ times the number of relaxations. By Corollary 5.9, we obtain $T_2(n, k) = O(k \,\text{polylog}\, n)$. □

5.0.4 Size of the blocking flow and completing time analysis

With Lemmas 5.6 and 5.10, we can complete the proof of Lemma **??** (time per REFINE) by bounding the total number of arcs whose flow is updated by a blocking flow during REFINE. This bounds both the time spent updating the flow value of these arcs, and also the time spent on empty vertex potential updates (Lemma 5.2).

Lemma 5.11. *Let N_i be the number of positive flow arcs in the i-ith blocking flow of* REFINE. *Then, $\sum_i N_i = O(k\sqrt{k})$.*

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, which appear as path P in Algorithm 4. 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 5.9, we have $N_i = O(k)$.

By Lemma 4.8, there are $O(\sqrt{k})$ iterations of REFINE before it terminates. Summing, we see that $\sum_i N_i = O(k\sqrt{k})$. □

We now complete the proof of Lemma **??**. There $O(\sqrt{k})$ iterations of REFINE, each of which executes HUNGARIAN-SEARCH2 and DFS. By Lemmas 5.6 and 5.10, these calls take $O(T_1(n,k) + T_2(n,k)) = O(k \operatorname{polylog} n)$ time per iteration. HUNGARIAN-SEARCH2 and DFS require some once-per-REFINE preprocessing to initialize data structures in $P_1(n,k) + P_2(n,k) = O(n \operatorname{polylog} n)$ time. Outside of these, we need to account for the time spent on flow value updates and augmentation-induced empty vertex potential updates. By Lemma 5.11, the former is $O(k\sqrt{k})$ over the course of REFINE. Combining Lemmas 5.11 and 5.2, the time for the latter is also $O(k\sqrt{k})$.

Filling in the values of $P_1(n,k)$, $P_2(n,k)$, $T_1(n,k)$, and $T_2(n,k)$, the total time for REFINE is $O((n + k\sqrt{k}) \operatorname{polylog} n)$. Together with Lemmas **??** and **??**, this completes the proof of Theorem 1.2.

6 Unbalanced transportation

In this section, we give an exact algorithm which solves the transportation problem in $O(rn^{3/2} \operatorname{polylog} n)$ time, proving Theorem 1.3. This algorithm is a geometric implementation of the uncapacitated min-cost flow algorithm due to Orlin [7], combined with some of the tools developed in Sections 3 and 5 (e.g. rewinding relaxation updates). ⟨⟨**Be specific, since the tools have been introduced.**⟩⟩

Let A and B be points in the plane with $r := |A|$ and $n := |B|$. Let $\lambda : A \cup B \to \mathbb{Z}$ be a **supply-demand function** with positive value on points of A, negative value on points of B, and $\sum_{a \in A} \lambda(a) = -\sum_{b \in B} \lambda(b)$. We use $U := max_{p \in A \cup B} |\lambda(p)|$. A **transportation map** is a function $\tau : A \times B \to \mathbb{R}_{\geq 0}$. A transportation map τ is **feasible** if $\sum_{b \in B} \tau(a,b) = \lambda(a)$ for all $a \in A$, and $\sum_{a \in A} \tau(a,b) = -\lambda(b)$ for all $b \in B$. In other words, the value $\tau(a,b)$ describes how much supply at a should be sent to meet demands at b, and we require that all supplies are sent and all demands are met. We define the cost of τ to be

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

Given A, B, and λ, the **transportation problem** asks to find a feasible transportation map of minimum cost. Using terminology from matchings, we focus on analyzing the **unbalanced** setting where $r \leq n$.

There is a simple reduction from the transportation problem to uncapacitated min-cost flow. Consider the complete bipartite graph G between A and B (with all edges directed A-to-B), set the costs $c(a,b) = \|a - b\|$, all capacities to infinity, and use $\phi = \lambda$. Any circulation f in the network $N = (G, c, u, \phi)$ can be converted into a feasible transportation map τ_f by taking $\tau_f(a,b) \leftarrow f(a,b)$. Furthermore, $\operatorname{cost}(f) = \operatorname{cost}(\tau_f)$.

6.1 Uncapacitated MCF by excess scaling

We give an outline of the strongly polynomial algorithm for uncapacited MCF from Orlin [7]. Orlin's algorithm follows an **excess-scaling** paradigm originally due to Edmonds and Karp [2]. Recall that we can compute an optimal flow by beginning with $f = 0$, $\pi = 0$ and repeatedly

augmenting f with improving flows on 0-admissible arcs. The excess-scaling algorithm maintain a parameter Δ, and uses for its improving flow an augmenting path between an excess vertex with $\phi_f(v) \geq \Delta$, and a deficit vertex with $\phi_f(v) \leq -\Delta$. Vertices with $|\phi_f(v)| \geq \Delta$ are called **active**. Once there are either no more active excess or no more active deficit vertices, Δ is halved. Each sequence of augmentations where Δ holds a constant value is called an **excess scale**. The algorithm initializes $\Delta = U$, so 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 [7] obtains an algorithm with a strongly polynomial bound on the number of augmentations and excess scales. First, an **active** vertex is redefined to be one where $|\phi_f(v)| \geq \alpha\Delta$, for a parameter $\alpha \in (1/2, 1)$. Second, arcs which have $f(v, w) \geq 3n\Delta$ are **contracted**, creating a new vertex $\hat{v}$ which inherits all the arcs of v and w, and has $\phi(\hat{v}) \leftarrow \phi(\hat{v}) + \phi(\hat{w})$. 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. Third, Δ is aggressively lowered to $\max_{v \in V} \phi_f(v)$ if there are no active excess vertices, and $f(v, w) = 0$ on all non-contracted arcs $(v, w) \in \hat{E}$. 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 contractions. The algorithm 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^*.

Theorem 6.1 (Orlin [7], Theorems 2 and 3)**.** *Orlin's algorithm finds optimal potentials after $O(n \log n)$ scaling phases, and $O(n \log n)$ total augmentations.*

Geometric implementation of contractions. Following Agarwal *et al.* [1], our geometric datastructures must utilize actual points of A and B, rather than the contracted entities in $\hat{G}$. We will track the contracted components described in $\hat{G}$ (e.g. with a disjoint-set data structure) and label the arcs of $\text{supp}(f)$ which were contracted. We maintain potentials on the points of $A \cup B$ instead of the contracted entities.

When conducting the Hungarian search, we initialize S with all vertices from **active contracted components** who (in sum) meet the respective imbalance criteria. Upon relaxing any $v \in \hat{v}$, we immediately relax all the contracted $\text{supp}(f)$ arcs which span it. Since the input network is uncapacitated, each contracted component is strongly connected in the residual network by the admissible forward/reverse arcs corresponding to each $\text{supp}(f)$ arc. Thus, these relaxations can be performed without further potential changes.

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

Geometric implementation of recovery. Instead of running a generic max-flow algorithm after finding the optimal potentials, we use the following observation.

⟨⟨**TODO is the "admissible graph is planar" thing true for other L_p distances?**⟩⟩

6.2 Dead vertices and support stars

Given Theorem 6.1, our goal is to implement each augmentation in $O(rn^{1/2} \operatorname{polylog} n)$ time. To find an augmenting path, we again use some form of Hungarian search with geometric data structures to perform relaxations quickly.

Like in Section 5, there are vertices which cannot be charged to the flow support. Even worse, the size of the flow support have size $\Omega(n)$ for the transportation map (consider an instance with $r = 1$, and the demand uniformly distributed among vertices of B). Still, by classifying vertices carefully, we can do better than an $O(n)$ bound on the number of relaxations.

Let $E(\text{supp}(f)) \coloneqq \{(v, w) \mid v \to w \in \text{supp}(f)\}$ be the set of undirected edges corresponding to the arcs in $\text{supp}(f)$. Clearly, $|\text{supp}(f)| = |E(\text{supp}(f))|$. Let the **support degree** of a vertex be its degree in $E(\text{supp}(f))$.

Dead vertices. We call a vertex $b \in B$ **dead** if it has support degree 0 and is not an active excess or deficit. These are essentially equivalent to the *empty vertices* of Section 5. 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. They would not be relaxed by the Hungarian search even if they were included, so their potentials will not change as long as they are dead. We use $\boldsymbol{A}_\ell$ and $\boldsymbol{B}_\ell$ to denote the living vertices of points in A and B, respectively.

We say a dead vertex is **revived** when it stops meeting either condition of the definition. Dead vertices are only revived after Δ decreases (i.e. in 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 a number of data structures. 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 adjacent $\text{supp}(f)$ arcs 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 $O(n \log n)$.

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 around $a \in A$.

Roughly speaking, we would like to handle each support star as a single unit. When the Hungarian search reaches a or any $b \in \Sigma_a$, then the entirety of Σ_a (as well as a) is also admissible-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, so aside from a there are no vertices of A which can be reached by traveling through Σ_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 $b \in \Sigma_a$.

6.3 Implementation details

Before describing our workaround for support stars, we analyze the number of relaxation steps for arcs outside of support stars. We can modify the Hungarian search slightly to ensure that $\text{supp}(f)$ is always acyclic in the undirected sense, i.e. where $E(\text{supp}(f))$ is acylic. Specifically, we relax support arcs before non-support arcs as they enter the frontier.

⟨⟨this proof isn't in the conference version, and our full version isn't published anywhere⟩⟩ ⟨⟨HC: include here, at least in the full version⟩⟩

Lemma 6.2 (Agarwal *et al.* [1])**.** *If arcs of* $\text{supp}(f)$ *are relaxed first as they arrive on the frontier, then* $E(\text{supp}(f))$ *is acyclic.*

Let $E(\Sigma_a)$ be the underlying edges of the support star centered at a. ⟨⟨**Do you really need this?**⟩⟩ Using Lemma 6.2, we can show that the number of support arcs outside support stars is small.

Lemma 6.3. $|B_\ell \setminus \bigcup_{a \in A} \Sigma_a| \leq r$.

Proof. By Lemma 6.2, $E(\text{supp}(f))$ is acylic and therefore forms a spanning forest over $A \cup B_\ell$. By eliminating the edges of support stars and dead vertices, the remaining edges of $E(\text{supp}(f))$ must adjoin vertices in B of support degree at least 2. Thus, all leaves of the bipartite forest $F \coloneqq E(\text{supp}(f)) \setminus \bigcup_{a \in A} E(\Sigma_a)$ are vertices of A. ⟨⟨**Define F outside the lemma because it is being reused.**⟩⟩

Pick an arbitrary root for each connected component of the bipartite forest 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 because it has at most one parent in F. Thus, the number of B_ℓ vertices of support degree at least 2 is no more than r. □

Lemma 6.4. *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 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 6.3. The lemma follows. □

The running time of a Hungarian search will be $O(r)$ times the time it takes us to implement each relaxation.

Relaxations outside support stars. For relaxations that don't involve support star vertices, we can once again maintain a BCP 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. This can be queried in $O(\log n)$ time and updated in $O(\text{polylog}\, n)$ time per point. Since it does not involve support stars, there is at most one insertion/deletion per relaxation step.

For B_ℓ-to-A_ℓ, support backward arcs are kept admissible by invariant so we relax them immediately when they arrive on the frontier.

Relaxing a support star. We classify support stars into two categories: **big stars** are those with $|\Sigma_a| > \sqrt{n}$, and **small stars** are those with $|\Sigma_a| \leq \sqrt{n}$. Let $A_{\text{big}} \subseteq A$ denote the centers of big stars and and $A_{\text{small}} \subseteq A$ denote the centers of small stars. We keep the following data structures to manage support stars.

1. For each big star Σ_a, we use a data structure $\mathcal{D}_{\text{big}}(a)$ to maintain the BCP between $P = A_\ell \cap S$ and $Q = \Sigma_a$, weighted by potentials. We query this until $a \in S$ or any vertex of Σ_a is added to S.

2. All small stars are added to a single BCP data structure $\mathcal{D}_{\text{small}}$ between $P = A_\ell \cap S$ and $Q = (\bigcup_{a \in A_{\text{small}}} \Sigma_a) \setminus S$, weighted by potentials. When $a \in A_s$ or any vertex of Σ_a 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. Relaxations of non-support star $b \in B_\ell$ will not affect the support star data structures.

Suppose a relaxation step adds some $a \in A_\ell$ to S. For the support star data structures, we must (i) remove a from every $\mathcal{D}_{\text{big}}(\cdot)$, (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), (ii), and (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, as there are $O(n/\sqrt{n})$ data structures $\mathcal{D}_{\text{big}}(\cdot)$. Thus, the query time within each relaxation step is $O(\sqrt{n}\log n)$.

Updating support stars. As the flow support changes, the membership of support stars may shift and a big star may eventually become small (or vice versa). To efficiently support this, introduce some leniency for when a star should be represented as a big star versus a small star and use *fully persistent* versions of the $\mathcal{D}_{\text{big}}(\cdot)$ data structures.

Initially, we label stars big or small according to the $\sqrt{n}$ threshold. Instead of changing a star from big-to-small (or vice versa) immediately when crossing the $\sqrt{n}$ threshold, we keep a big star in $\mathcal{D}_{\text{big}}(a)$ so long as $|\Sigma_a| \geq \sqrt{n}/2$ and keep a small star in $\mathcal{D}_{\text{small}}$ so long as $|\Sigma_a| \leq 2\sqrt{n}$. Intuitively, switching data structures will incur some "rebuilding" expense which we can charge to the $O(\sqrt{n})$ insertions/deletions that were needed for the star to switch types.

At the beginning of the algorithm, we build a single "empty" $\mathcal{D}_{\text{big}}(\cdot)$ containing the set P (vertices of A) but omitting Q, which we call $\mathcal{D}_{\text{root}}$. We maintain changes in the set of active excesses (the intial set of S) by updating a single persistent branch of $\mathcal{D}_{\text{root}}$, which we call the **root branch**. Within a single iteration, we create a new branch of $\mathcal{D}_{\text{root}}$ which tracks $P = A_\ell \cap S$ and $Q = \emptyset$.

Any "newly-big" star Σ_a will make a persistent copy of the latest $\mathcal{D}_{\text{root}}$ and insert into Q the points of the star, creating a new branch which maintains $\mathcal{D}_{\text{big}}(a)$. Note that $|\Sigma_a| = \sqrt{n} + x > 2\sqrt{n}$, so we have $|\Sigma_a| < 2x$. There were at least x insertions into Σ_a since Σ_a was first small, so the $|\Sigma_a|$ insertions to construct $\mathcal{D}_{\text{big}}(a)$ from $\mathcal{D}_{\text{root}}$ can be charged to previous changes to Σ_a. Σ_a must also be deleted from $\mathcal{D}_{\text{small}}$; these deletions can be charged in the same way.

When a big star crosses $|\Sigma_a| < \sqrt{n}/2$, we delete the branch of $\mathcal{D}_{\text{root}}$ corresponding to $\mathcal{D}_{\text{big}}(a)$ and insert its points into $\mathcal{D}_{\text{small}}$. There were at least $\sqrt{n}/2$ points removed from Σ_a since it was first big, so we can charge the $|\Sigma_a|$ insertions to the previous changes to Σ_a.

In summary, switching stars from big-to-small or small-to-big can be charged to the star membership changes since the star last changed types. Membership changes can themselves be bounded above the length of augmenting paths (flow support must change for stars to change).

Preprocessing time. To build the very first set of data structures, we take $O(rn\,\text{polylog}\, n)$ time. There are $r|\Sigma_a|$ points in each $\mathcal{D}_{\text{big}}(a)$, but the Σ_a are disjoint so the sum of points is $O(rn)$. $\mathcal{D}_{\text{small}}$ also has at most $O(rn)$ points. Each BCP data structure can be constructed in $O(\text{polylog}\, n)$ times its size, so the total preprocessing time is $O(rn\,\text{polylog}\, n)$.

⟨⟨**TODO just invoke persistence here**⟩⟩

Between searches. After an augmentation, we must reset the above data structures to their initial state plus the change from the augmentation. Using the rewinding mechanism from the previous sections, this can be done in time proportional to $O(\sqrt{n}\,\text{polylog}\, n)$ times the number of relaxations

(to rewind), and then plus $O(\sqrt{n}\operatorname{polylog} n)$ time to change the initial set of active excesses in S. Additionally, augmentation may change the membership of some support stars. The number of these support star changes is bounded above by the length of the path, which is bounded above the number of relaxations. Each support star membership update updates a single data structure, so takes $O(\operatorname{polylog} n)$ time. There are $O(r)$ relaxations by the bound in Lemma 6.4 plus at most one for each support star (to reach the center). Thus, the running time per augmentation is $O(r\sqrt{n}\operatorname{polylog} n)$.

Between excess scales. When the excess scale changes, the sets of active and living vertices may change. Notably, the sets of active/living vertices may only *grow* when Δ decreases. If we have the data structures built on the active excesses at the end of the previous scale, then we can add in each newly active $a \in A$ and charge this insertion to the (future) augmenting path or contraction which eventually makes the vertex inactive, or absorbs it into another component. By Theorem 6.1, there are $O(n \log n)$ such newly active vertices. The time to perform data structure updates for each of them is $O(\sqrt{n}\operatorname{polylog} n)$, so the total time spent bookkeeping newly active vertices is $O(n^{3/2}\operatorname{polylog} n)$.

References

[1] P. K. Agarwal, K. Fox, D. Panigrahi, K. R. Varadarajan, and A. Xiao. Faster algorithms for the geometric transportation problem. In *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, pages 7:1–7:16, 2017.

[2] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[3] A. V. Goldberg, S. Hed, H. Kaplan, and R. E. Tarjan. Minimum-cost flows in unit-capacity networks. *Theory Comput. Syst.*, 61(4):987–1010, 2017.

[4] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15(3):430–466, 1990.

[5] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir. Dynamic planar voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2495–2504, 2017.

[6] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.

[7] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993.

[8] R. Sharathkumar and P. K. Agarwal. Algorithms for the transportation problem in geometric settings. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 306–317, 2012.

[9] P. M. Vaidya. Geometry helps in matching. *SIAM J. Comput.*, 18(6):1201–1225, 1989.