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 of pairs in $A \times B$ so that any point of $A \cup B$ appears in at most one pair. Given two positive integers p and q, we define the cost of matching M to be $c(M) = \sum_{(a,b)\in M} \|a - b\|_p^q$ 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 problem that work for any powers of L_p-norm matching objective: An exact algorithm that runs in $O((n + k^2) \operatorname{polylog} n)$ time, and a $(1 + \varepsilon)$-approximation algorithm that runs in $O((n + k\sqrt{k}) \operatorname{polylog} n \cdot \log \varepsilon^{-1})$ 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 $O(\min\{n^2, rn^{3/2}\} \operatorname{polylog} n)$ time.

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.* [8] algorithm. We thank Debmalya Panigrahi for helpful discussions.

Lines 533

1 Introduction

Given two point sets A and B in the plane, 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 $c(a, b) = \|a - b\|_p^q$, for some positive integers p and q. A *matching* M in $G(A, B)$ is a set of edges sharing no endpoints. The *size* of M is the number of edges in M. The cost of matching M, denoted $c(M)$, is defined to be the sum of costs of edges in M. For a parameter k, 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 $\phi : A \cup B \to \mathbb{Z}$ be an integral *supply-demand function* with positive value on points of A and negative value on points of B, satisfying $\sum_{a \in A} \phi(a) = -\sum_{b \in B} \phi(b)$. A *transportation map* is a function $\tau : A \times B \to \mathbb{R}_{\geq 0}$ such that $\sum_{b \in B} \tau(a, b) = \phi(a)$ for all $a \in A$ and $\sum_{a \in A} \tau(a, b) = -\phi(b)$ for all $b \in B$. We define the cost of τ to be

$$c(\tau) \coloneqq \sum_{(a,b) \in A \times B} c(a, b) \cdot \tau(a, b).$$

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 [9] and the $O(m \min\{\sqrt{m}, n^{2/3}\})$ time algorithm by Even and Tarjan [6], 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 [12], which uses an interior-point algorithm, runs in $O(m^{10/7} \operatorname{polylog} n)$ time.

The Hungarian algorithm [11] 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 [7] and the improved $O(m\sqrt{n} \log C)$ time algorithm by Duan *et al.* [5] assuming the edge costs are integral; here C is the maximum cost of an edge. Ramshaw and Tarjan [14] showed that the Hungarian algorithm can be extended to compute a minimum-cost partial matching of size k in $O(km + k^2 \log r)$ time, where r is the size of the smaller side of the bipartite graph. They also proposed a cost-scaling algorithm for partial matching that runs in time $O(m\sqrt{k} \log(kC))$, again assuming that costs are integral. By reduction to unit-capacity min-cost flow, Goldberg *et al.* [8] developed a cost-scaling algorithm for partial matching with an identical 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 $O(n^2 \operatorname{polylog} n)$ [10] (see also [1, 18]). This algorithm computes an optimal size-k matching in time $O(kn \operatorname{polylog} n)$. Faster approximation algorithms have been developed for computing perfect matchings in geometric settings [3, 15, 18, 19]. Recall that the cost of the edges are the qth power of their L_p-distances. When $q = 1$, the best algorithm to date by Sharathkumar and Agarwal [16] computes a $(1+\varepsilon)$-approximation to the optimal perfect matching in $O(n \operatorname{polylog} n \cdot \operatorname{poly} \varepsilon^{-1})$ 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 $O(n^{3/2} \operatorname{polylog} n \log(1/\varepsilon))$ time [15]; it is not obvious how to extend this algorithm to the partial matching setting.

The transportation problem can also be formulated as an instance of the minimum-cost flow problem. The strongly polynomial uncapacitated min-cost flow algorithm by Orlin [13] solves the transportation problem in $O((m + n \log n)n \log n)$ time. Lee and Sidford [**?**] give a weakly polynomial algorithm that runs in $O(m\sqrt{n} \operatorname{polylog}(n, U))$ 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 $O(n^2 \operatorname{polylog} n)$. By adapting the

[1] Partial matching is also called *imperfect matching* or *imperfect assignment* [8, 14].

Lee-Sidford algorithm, they developed a $(1+\varepsilon)$-approximation algorithm that runs in $O(n^{3/2}\varepsilon^{-2}\operatorname{polylog}(n,U))$ 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. **⟨⟨cite the preconditioning paper?⟩⟩**

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 and an integer $k \le n$, a minimum-cost matching of size k between A and B can be computed in $O((n+k^2)\operatorname{polylog} n)$ 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, 10]. But unlike their algorithms which take $\Omega(n)$ time to find an augmenting path, we show that after $O(n \operatorname{polylog} n)$ preprocessing, an augmenting path can be found in $O(k \operatorname{polylog} n)$ 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, 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.* [8].

▶ **Theorem 1.2.** *Given two point sets A and B in $\mathbb{R}^2$ each of size at most n, an integer $k \le n$, and a parameter $\varepsilon > 0$, a $(1+\varepsilon)$-approximate min-cost matching of size k between A and B can be computed in $O((n+k\sqrt{k})\operatorname{polylog} n \cdot \log \varepsilon^{-1})$ time.*

The main challenge here is how to deal with the set of *dead nodes*, which neither have excess/deficit nor have flow passing through them, but still contribute to the size of the graph. We show that the number of *alive* nodes is only $O(k)$, and then represent the dead nodes implicitly so that the Hungarian search and computation of a blocking flow can be implemented in $O(k \operatorname{polylog} n)$ time.

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 with $r \le n$, along with supply-demand function $\phi : A \cup B \to \mathbb{Z}$, an optimal transportation map between A and B can be computed in $O(\min\{n^2, rn^{3/2}\}\operatorname{polylog} n)$ time.*

Our result improves over the $O(n^2 \operatorname{polylog} n)$ time algorithm in [2] for $r = o(\sqrt{n})$. As in [2], we also use the strongly polynomial uncapacitated minimum-cost flow algorithm by Orlin [13], but we additional ideas are needed to implement it faster for $r \le \sqrt{n}$. Unlike in the case of matchings, the support of the transportation problem may have size $\Omega(n)$ even when r is a constant; so naïvely we can no longer spend time proportional to the size of the support of the transportation map. However, we ensure that the support is acyclic and use a data structure that can find an augmenting path in $O(r\sqrt{n}\operatorname{polylog} n)$ time, assuming $r \le \sqrt{n}$.

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 by 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 [14] showed that M is guaranteed to be an optimal partial matching.

The Hungarian search grows a set of *reachable vertices* X *from* all unmatched $v \in A$ *using* augmenting paths of admissible edges. Initially, X is the set of unmatched vertices in A. Let the *frontier* of X be the edges in $(A \cap X) \times (B \setminus X)$. X is grown by *relaxing* an edge (a, b) in the frontier: adding b into X, and also modifying potentials to make (a, b) admissible, preserve c_π on other edges within X, and keep π feasible on edges outside of X. Specifically, the algorithm relaxes the minimum-reduced-cost frontier edge (a, b), and then raises $\pi(v)$ by $c_\pi(a, b)$ for all $v \in X$. If b is already matched, then we also relax the matching edge (a', b) and add a' into X. 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, 18]. 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 $P = A \cap X$ and $Q = B \setminus X$, with $\omega = \pi$. Note that the "state" of this BCP is parameterized by X and π.

The dynamic BCP data structure by Kaplan *et al.* [10] supports point insertions and deletions in $O(\operatorname{polylog} n)$ time and answers queries in $O(\log^2 n)$ time for our setting. Outside of potential updates, each relaxation in the Hungarian search requires one query, one deletion, and at most one insertion. As $|M| \le k$ throughout, there are at most $2k$ relaxations in each Hungarian search, and the BCP can be used to implement each Hungarian search in $O(k \operatorname{polylog} n)$ time. Explained shortly, there is an existing technique to handle potential updates without performing BCP updates for each one.

Rewinding mechanism. We 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.

If we remember the sequence of points added to X in the i-th Hungarian search, then at the start of the $(i+1)$-th Hungarian search we can *rewind* this sequence by applying the opposite insert/delete operation for each BCP update in reverse order 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 insertions/deletions is $O(k)$, bounded by the number of relaxations per Hungarian search, therefore we can recover, in $O(k \operatorname{polylog} n)$ time, the initial

BCP data structure for each Hungarian search beyond the first. We refer to this procedure as the *rewinding mechanism*.

⟪I tried to condense this; please check⟫

Potential updates. We modify a trick from Vaidya [18] to batch potential updates. Potentials are tracked with a *stored value* $\gamma(v)$, while the *true value* of $\pi(v)$ may have changed since $\gamma(v)$ was last recorded. This is done by aggregating potential changes into a variable δ, which is initially 0 at the very beginning of the algorithm. Whenever we would raise the potentials of all vertices in X, we raise δ by that amount instead. We maintain the following invariant: $\pi(v) = \gamma(v)$ for $v \notin X$, and $\pi(v) = \gamma(v) + \delta$ for $v \in X$.

At the beginning of the algorithm, X is empty and stored values are equal to true values. When $a \in A$ is added to X, we update its stored value to $\pi(a) - \delta$ for the current value of δ, and use that stored value as its BCP weight. Since the BCP weights are uniformly offset from $\pi(v)$ by δ, the pair reported by the BCP is still minimum. When $b \in B$ is added to X, we update its stored value to $\pi(b) - \delta$ (although it won't be added to a BCP set). When a vertex is removed from X (e.g. by augmentation or rewinding), we update the stored potential $\gamma(v) \leftarrow \pi(v) + \delta$, again for the current value of δ. Unlike [18], we do not reset δ across Hungarian searches.

There are $O(k)$ relaxations and thus $O(k)$ updates to δ per Hungarian search. $O(k)$ stored values are updated per rewinding, so the time spent on potential updates per Hungarian search is $O(k)$. Putting everything together, our implementation of the Hungarian algorithm runs in $O((n + k^2) \operatorname{polylog} n)$ time. This proves Theorem 1.1.

3 Approximating Min-Cost Partial Matching through Cost-Scaling

In this section we 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.* [8] 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 3.3).

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 [14] 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

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

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

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

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

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, $c(M) = c(f)$. 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.** *Let N be the network constructed from $G(A, B)$ above.*
(i) *For any integral circulation g in N, the size of* $\mathrm{supp}(g)$ *is at most* $3k$.
(ii) *For any integral pseudoflow f in N with $O(k)$ excess, the size of* $\mathrm{supp}(f)$ *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.

A *residual pseudoflow* g in N_f can be used to change f into a different pseudoflow on N by a process called *augmentation*. For simplicity, we only describe augmentation for the case where f, g are integer. Specifically, augmenting f by g produces a pseudoflow f' in N where

$$f'(v \to w) = \begin{cases} 0 & w \to v \in \vec{E}_f \text{ and } g(w \to v) = 1 \\ 1 & v \to w \in \vec{E}_f \text{ and } g(v \to w) = 1 \\ f(v \to w) & \text{otherwise.} \end{cases}$$

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

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

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. Does $c_\pi(w{\to}v)$ 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* [4,17] 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 $g(v{\to}w) > 0$ only on admissible arcs $v{\to}w$, and $g(v{\to}w) = 0$ otherwise. [2] We will 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, $c(f) \leq c(f^*) + 6k\theta$.*

Estimating the value of $c(f^*)$. We now describe a procedure for estimating $c(f^*)$ 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 cost function $c(\cdot)$. Let $e_1, e_2, \ldots, e_{n-1}$ be the edges of T sorted in nondecreasing order of length; in other words, $c(e_i) \leq c(e_{i+1})$. 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 $\overline{\theta} := n^q \cdot c(e_{i^*})$. The following lemma will be used by our cost-scaling algorithm:

▶ **Lemma 3.4.** **(i)** *The value of i^* can be computed in $O(n \log n)$ time.*
(ii) $c(e_{i^*}) \leq c(f^*) \leq \overline{\theta}$.
(iii) *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 $\underline{\theta} := \frac{\varepsilon}{6k} \cdot c(e_{i^*})$. As a consequence of Lemmas 3.4ii and 3.3, we have:

▶ **Corollary 3.5.** *The cost of a $\underline{\theta}$-optimal integral circulation in N is at most $(1+\varepsilon)c(f^*)$.*

We are now ready to describe our algorithm.

Overview of the algorithm. We closely follow the algorithm of Goldberg *et al.* [8]. The algorithm works in rounds. In the beginning of each round, we fix a *cost scaling parameter θ* and maintain potentials π with the following property:

⟨⟨would like to use a better/neater/cleaner environment to represent invariant⟩⟩

---

[2] The same admissibility/feasibility definitions will be used later in Section 4. However, the algorithm in Section 4 maintains a 0-optimal f and therefore admissible residual arcs always have $c_\pi(v{\to}w) = 0$.

(∗) There exists a 2θ-optimal integral circulation in N with respect to π.

For the initial round, we set $\theta \leftarrow \overline{\theta}$ and $\pi \leftarrow 0$. By Lemma 3.4(iii), (∗) is satisfied initially. Each round of the algorithm consists of two stages. In the first stage, called *scale initialization* (Scale-Init) computes a θ-optimal pseudoflow f. The second stage, called *refinement* (Refine) converts f into a θ-optimal (integral) circulation g. In both stages, π is updated as necessary. If $\theta \le \underline{\theta}$, we return g. Otherwise, we set $\theta \leftarrow \theta/2$ and start the next round. Note that (∗) is satisfied in the beginning of each round.

By Corollary 3.5, when the algorithm terminates, it returns an integral circulation $\tilde{f}$ in N of cost at most $(1+\varepsilon)c(f^*)$, which corresponds to a $(1+\varepsilon)$-approximate min-cost matching of size k in G. The algorithm terminates in $\log_2(\overline{\theta}/\underline{\theta}) = O(\log(n/\varepsilon))$ rounds.

Next, we describe the two stages in detail.

Scale initialization. In the first round, we compute a $\overline{\theta}$-optimal pseudoflow by simply setting $f(v\to w) = 0$ for all arcs in $\vec{E}$. For subsequent rounds, we adjust the potential and flow in N as follows: we raise the potential of all nodes in A by θ, those in B by 2θ, and of t by 3θ. The potential of s remains unchange. Since the reduced cost of every forward arc in N_f after the previous round is at least -2θ, the above step increases the reduced cost of all forward arcs by θ, and the reduced cost of all forward arcs is at least $-\theta$.

Next, for each backward arc $w\to v$ in N_f with $c_\pi(w\to v) < -\theta$, we set $f(v\to w) = 0$ (that is, make arc $v\to w$ idle), which replaces the backward arc $w\to v$ in N_f with a forward arc of postive reduced cost. After this step, the resulting pseudoflow must be θ-optimal as all arcs of N_f have reduced cost at least $-\theta$.

The desaturation of each backward arc creates one unit of excess. Since there are at most $3k$ backward arcs, the total excess in the resulting pseudoflow is at most $3k$. There are $O(n)$ potential updates and $O(k)$ arcs on which flow might change, therefore the time needed for Scale-Init is $O(n)$.

Refinement. The procedure Refine converts a θ-optimal pseudoflow with $O(k)$ excess into a θ-optimal circulation, using a primal-dual augmentation algorithm. A path in N_f is an *augmenting path* if it begins at an excess vertex and ends at a deficit vertex. We call an admissible pseudoflow g in the residual network N_f an *admissible blocking flow* if g saturates at least one arc in every admissible augmenting path in N_g. In other words, there is no admissible excess-deficit path in the residual network after augmentation by g. Each iteration of Refine finds an admissible blocking flow to be added to the current pseudoflow in two steps:

1. *Hungarian search*: a Dijkstra-like search that begins at the set of excess vertices and raises potentials until there is an excess-deficit path of admissible arcs in N_f.
2. *Augmentation*: using depth-first search through the set of admissible arcs of N_f, construct an admissible blocking flow. It suffices to repeatedly extract admissible augmenting paths until no more admissible excess-deficit paths remain.

The algorithm repeats these steps until the total excess becomes zero. The following lemma bounds the number of iterations in the Refine procedure at each scale.

▶ **Lemma 3.6.** *Let θ be the scaling parameter and π_0 the potential function at the beginning of a round, such that there exists an integral 2θ-optimal circulation with respect to π_0. Let f be a θ-optimal pseudoflow with excess $O(k)$. Then* Refine *terminates within $O(\sqrt{k})$ iterations.*

Proof. We sketch the proof, which is adapted from [8]. Let f_0 be the assumed 2θ-optimal integral circulation with respect to π_0, and let π be the potentials maintained and changing during REFINE. Let $d(v) = (\pi(v) - \pi_0(v))/\theta$, i.e. the increase in potential at v in units of θ. We divide the iterations of REFINE into two phases: before and after every (remaining) excess vertex has $d(v) \geq \sqrt{k}$. Each Hungarian search raises excess potentials by at least θ, since we use blocking flows. Thus, the first phase lasts at most $\sqrt{k}$ iterations.

At the start of the second phase, consider the set of arcs $E^+ = \{v{\to}w \in \vec{E} \mid f(v{\to}w) < f_0(v{\to}w)\}$. One can argue that the remaining excess with respect to f is bounded above by the size of any cut separating the excess and deficit vertices. The proof examines cuts $Y_i = \{v \mid d(v) > i\}$ for $0 \leq i \leq \sqrt{k}$. By θ-optimality of f and 2θ-optimality of f_0, one can show that each arc in E^+ crosses at most 3 cuts. Furthermore, the size of E^+ is $O(k)$, bounded by the support size of f and f_0. Averaging, there is a cut among the Y_i of size at most $3k/\sqrt{k}$, so the total excess remaining is $O(\sqrt{k})$. Each iteration of REFINE eliminates at least one unit of excess, so the number of second phase iterations is also at most $O(\sqrt{k})$. ◀

In the next subsection we show that after $O(n \operatorname{polylog} n)$ preprocessing, an iteration of REFINE can be performed in $O(k \operatorname{polylog} n)$ time (cf. Lemma 3.8). By Lemma 3.6 and the fact the algorithm terminates in $O(\log(n/\varepsilon))$ rounds, the overall running time of the algorithm is $O((n + k^{3/2}) \operatorname{polylog} n \log(1/\varepsilon))$, as claimed in Theorem 1.2.

3.3 Fast implementation of refinement

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 so far. 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.

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 *dead* if $\phi_f(v) = 0$ and no arc of $\operatorname{supp}(f)$ is incident to it; otherwise v is *alive*. We note that s and t are always alive. Let $A^{❀}$ (resp. $B^{❀}$, $X^{❀}$) denote the set of alive nodes of A (resp. B, X). There are only $O(k)$ alive nodes, as each can be charged to its adjoining $\operatorname{supp}(f)$ arcs or its excess/deficit. We treat alive and dead nodes separately to implement the Hungarian search efficiently. By definition, dead vertices only adjoin forward arcs in N_f. Thus, the in-degree (resp. out-degree) of a node in $(A \setminus A^{❀})$ (resp. $(B \setminus B^{❀})$) is 1, and any path passing through an dead 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 dead nodes, and in the case of two consecutive dead nodes there is a subpath of the form $s{\to}v{\to}w{\to}t$ where $v \in (A \setminus A^{❀})$ and $w \in (B \setminus B^{❀})$. We will call such paths, from an alive node to an alive node with only dead interior nodes, *alive-alive paths*. We extend the notions of reduced cost and admissibility to alive-alive paths, where the reduced cost of the path is the sum of its edges. Since reduced costs telescope, the reduced cost of an alive-alive path depends only on the potentials at its (alive) endpoints.

Using this observation, we implement the Hungarian search to "skip over" dead nodes, while logically exploring the same alive nodes in the same order. Alive-alive paths may have length 1 (no dead interior nodes), 2, or 3. At each step, we find a minimum reduced cost alive-alive path Π from an alive node of X to an alive node of $V \setminus X$, and add the nodes of this path into X in a single step. We update potentials in X according to the reduced cost of the path. There are $O(k)$ alive nodes, so the number of minimization queries per Hungarian search is $O(k)$.

We find the minimum reduced cost alive-alive 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 alive node of X to an alive 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 $a \in A^{❀}$, $b{\to}t$ for some $b \in B^{❀}$, and bipartite arc $a{\to}b$ with two alive endpoints. Edges of the first (resp. second) type can be found by maintaining $A^{❀} \setminus X$ (resp. $B^{❀} \cap X$) 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 $A^{❀} \cap X$ and $B^{❀} \setminus X$, with reduced cost as the pair distance **⟨⟨potentials?⟩⟩**. We thus maintain $A^{❀} \cap X$, $B^{❀} \setminus X$ in a dynamic BCP data structure [10] on which insertions/deletions can be performed in $O(\operatorname{polylog} k)$ time.

Finding length-2 paths. We describe how to find the cheapeast path of the form $s{\to}v{\to}b$ where v is dead and $b \in B^{❀}$. 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

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

This is done by maintaining a dynamic BCP data structure between $(A \setminus A^{❀})$ and $B^{❀} \setminus X$ with the cost of a pair (v,w) being $c(v,w) + \pi(w)$. We may require an update operation for each alive 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 $(A \setminus A^{❀})$ 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, 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 dead A vertices may change; that is, when flow is augmented across a vertex of $(A \setminus A^{❀})$. The first is identical to the situation in Section 2; the number of excess depletions is $O(k)$ over the course of Refine. For the second, the alive/dead status of a node can change only if the blocking flow found in the augmentation process passes through it. By Lemma 3.7(2) below, there are $O(k)$ such changes. Thus, updating $(A \setminus A^{❀})$ 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 $v \in (A \setminus A^{❀})$ and $w \in (B \setminus B^{❀})$. 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 $(A \setminus A^{\circledast})$ and $(B \setminus B^{\circledast})$, 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 come from membership changes to $A^{\circledast}$ or $B^{\circledast}$, after an augmentation. Applying Lemma 3.7(2) again, there are $O(k)$ alive/dead 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 dead nodes. Potentials for alive nodes can be updated in a batched fashion using the method in Section 2.

For dead nodes, we ignore their potentials entirely and instead recover "valid" potentials for them once they switch from dead to alive (and additionally, at the end of a round). Specifically, for a newly alive $a \in A$ we set $c_\pi(a) \leftarrow c_\pi(s)$ and for newly-alive $b \in B$ we set $c_\pi(b) \leftarrow c_\pi(t)$. Formally, we will say an alive-alive path is *strongly admissible* under π if all of its arcs are admissible under π. For correctness, we need to show that our choice of recovered potentials (1) preserves θ-optimality and (2) makes any admissible alive-alive path from before augmentation strongly admissible. It is a straightforward calculation to verify that both properties hold.

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

▶ **Lemma 3.7.** **1.** *Both the Hungarian search and augmentation step explore $O(k)$ nodes.*
2. *The blocking flow found by augmentation is incident to $O(k)$ vertices.*

Augmentation can also be implemented in $O(k \operatorname{polylog} n)$ time, after $O(n \operatorname{polylog} n)$ preprocessing, using similar data structures. We thus obtain the following:

▶ **Lemma 3.8.** *After $O(n \operatorname{polylog} n)$ preprocessing, each iteration of* Refine *can be implemented in $O(k \operatorname{polylog} n)$ time.*

4 Transportation Algorithm

Given two point sets A and B in $\mathbb{R}^2$ of sizes r and n respectively (with $r \leq \sqrt{n}$) and a supply-demand function $\phi : A \cup B \to \mathbb{Z}$ as defined in the introduction, we present an $O(rn^{3/2} \operatorname{polylog} n)$ time algorithm for computing an optimal transport map between A and B. By applying this algorithm in the case of $r \leq \sqrt{n}$ and the one in [2] when $r > \sqrt{n}$, we prove Theorem 1.3. We use a standard reduction to the uncapacitated min-csot flow problem (see e.g. [13]) and use Orlin's algorithm [13] as well as some of the ideas from [2] **⟨⟨The full version of the transportation paper is still not available anywhere.⟩⟩** for implementing it faster in geometric settings. We first present an overview of the algorithm and then describe its fast implementation to achieve the desired running time.

4.1 Overview of the algorithm

Orlin's algorithm follows an excess-scaling paradigm and the primal-dual framework. It maintains a *scale parameter* Δ, a flow function f, and potentials π on the nodes. Initially $\Delta = \phi(A)$, $f = 0$, and $\pi = 0$. We fix a constant parameter $\alpha \in (0.5, 1)$. A node v is called *active* if $|\phi_f(v)| \geq \alpha\Delta$. At each step, using the Hungarian search, the algorithm finds an admissible active-excess-to-acitve-deficit path in the residual network and pushes a flow of Δ

along this path. [3] It repeats this step as long as there are both active excess and active deficit nodes. When one of these sets becomes empty, Δ is halved. The sequence of augmentations with a fixed value of Δ is called an *excess scale*.

The algorithm also performs two preprocessing steps at the beginning of each excess scale. If $f(v\to w) \geq 3n\Delta$, $v\to w$ is contracted to a single node z with $\phi(z) = \phi(v) + \phi(w)$. [4] If there are no active excess nodes and $f(v\to w) = 0$ for all arcs, Δ is lowered to $\max_v \phi(v)$.

When the algorithm terminates, it has found an optimal circulation in the contracted network. We use the algorithm described in [2] to recover an optimal circulation in the original network. Orlin showed that the algorithm terminates within $O(n \log n)$ scales and performs a total of $O(n \log n)$ augmentations. In the next subsection, we describe an algorithm that after $O(n \operatorname{polylog} n)$ preprocessing can perform a Hungarian search (i.e. find an excess-deficit admissible path) in $O(r\sqrt{n} \operatorname{polylog} n)$ amortized time. Summing this cost over all augmentations, we obtain the desired running time.

4.2 An efficient implementation

Recall in the previous sections that we could bound the running time of the Hungarian search by the size of $\operatorname{supp}(f)$. Here, the number of active excess/deficit nodes at any scale is $O(r)$, and the length of an augmenting path is also $O(r)$. Therefore one might hope to find an augmenting path in $O(r \operatorname{polylog} n)$ time, by adapting the algorithms described in Sections 2 and 3. The challenge to this approach is that $\operatorname{supp}(f)$ may have $\Omega(n)$ size, therefore an algorithm which runs in $O(|\operatorname{supp}(f)|)$ time is no longer sufficient. Still, we manage to implement a Hungarian search in time roughly $r\sqrt{n}$, by exploiting a few properties of $\operatorname{supp}(f)$ as described below.

We note that each arc of $\operatorname{supp}(f)$ is admissible, i.e., its reduced cost is 0, so we add an arc of $\operatorname{supp}(f)$ as soon as possible when it arrives in $X \times (V \setminus X)$. This strategy ensures the following crucial property.

▶ **Lemma 4.1.** *If the arcs of* $\operatorname{supp}(f)$ *are added as soon as possible,* $\operatorname{supp}(f)$ *is acyclic.*

Next, similar to Section 3, we call a node $u \in V$ *alive* if u is an active excess/deficit node or if u is incident on an arc of $\operatorname{supp}(f)$. Otherwise, u is called *dead*. Unlike Section 3, once a node becomes alive it cannot become dead. Furthermore, a dead node may become alive only in the beginning of a scale (after we have reduced the value of Δ). Unlike Section 3, an augmenting path cannot pass through a dead node. Therefore, we can ignore all dead nodes during Hungarian search, work with only alive nodes, and update the set of alive nodes in the beginning of a scale. For a subset $S \subseteq V$ of nodes, we use $S^{❀}$ to denote the set of alive nodes in S.

Let $B^{\star} \subseteq B^{❀}$ be the set of nodes that are either active excess/deficit nodes or that are incident on at least two arcs of $\operatorname{supp}(f)$. Lemma 4.1 implies the following:

▶ **Lemma 4.2.** $|B^{\star}| = O(r)$.

We can therefore find the min reduced-cost arc $(X \cap A^{❀}) \times (B^{\star} \setminus X)$ using a BCP data structure as in Section 2, along with lazy potential updates and the rewinding mechanism. The total time spent by Hungarian search on the nodes of $B^{\star}$ will be $O(r \operatorname{polylog} n)$. We subsequently focus on handling $B^{❀} \setminus B^{\star}$.

---

[3] Note that this augmentation may convert an excess node into a deficit node.

[4] Intuitively, $f(v\to w)$ is so high that future scales cannot deplete the flow on $v\to w$.

Handling $B^{\circledast} \setminus B^{\star}$. We now describe how we query a min reduced-cost arc between $(X \cap A^{\circledast})$ and $B^{\circledast} \setminus (B^{\star} \cup X)$. Each node $b \in B^{\circledast} \setminus B^{\star}$ is incident on exactly one arc of $\mathrm{supp}(f)$ (i.e., there is only one outgoing arc from b). We partition these nodes into clusters depending on their neighbor in N_f. That is, for a node $a \in A^{\circledast}$, let $B_a = \{b \in B^{\circledast} \setminus B^{\star} \mid a{\to}b \in \mathrm{supp}(f)\}$. We refer to B_a as the *star* of a.

The crucial observation is that a is the only node in N_f reachable from each $b \in B_a$, so once the Hungarian search reaches a node of B_a and thus a (recall we prioritize adding $\mathrm{supp}(f)$ arcs), the Hungarian search need not visit any other nodes of B_a, as they will only lead to a. Hence, as soon as one node of B_a is reached, all other nodes of B_a are discarded from further consideration. Using this observation, we handle $B^{\circledast} \setminus B^{\star}$ as follows.

We classify each $a \in A^{\circledast}$ as *light* or *heavy*. If a is classified as heavy then $|B_a| \geq \sqrt{n}$, and if a is classified as light then $|B_a| \leq 2\sqrt{n}$. Note that if $|B_a| \in [\sqrt{n}, 2\sqrt{n}]$, then a may be classified as light or heavy. We allow this flexibility to allow re-classification in a lazy manner. Namely, a light node is re-classified heavy once $|B_a| > 2\sqrt{n}$, and a heavy node is re-classified light once $|B_a| < \sqrt{n}$. This scheme ensures that the star of a has gone through at least $\sqrt{n}$ updates (insertion/deletion of nodes) between two successive re-classifications, and these updates will pay for the time spent in updating the data structure when a is re-classified — see below.

For each heavy node $a \in A^{\circledast} \setminus X$, we maintain a BCP data structure between B_a and $X \cap A^{\circledast}$. Next, for all light $a \in A^{\circledast} \setminus X$, we collect their stars into a single set $B^{<} = \bigcup_{a \text{ light}} B_a$. We maintain a BCP data structure between $B^{<}$ and $A^{\circledast} \cap X$. Thus, we maintain at most r different BCP data structures for stars.

Using these data structures, we can compute a min reduced-cost arc between $A^{\circledast} \cap X$ and $B^{\circledast} \setminus (B^{\star} \cup X)$. Once an arc $v{\to}w \in (A^{\circledast} \cap X) \times (B^{\circledast} \setminus (B^{\star} \cup X))$ is added such that $w \in B_a$ for some star B_a, then we also add a to X. If a is light, then we delete B_a from $B^{<}$ and update the BCP data structure for $B^{<}$. If a is heavy, then we stop querying the BCP data structure on B_a for the remainder of the search. Finally, since a becomes a part of X, a is added to all $O(r)$ BCP data structures.

Recall that $r \leq \sqrt{n}$ by assumption. Adding the arc $v{\to}w$ thus involves performing $O(\sqrt{n})$ insertion/deletion operations in various BCP data structures, thereby taking $O(\sqrt{n}\,\mathrm{polylog}\, n)$ time.

Putting it together. While proof is omitted, the following lemma bounds the running time of the Hungarian search.

▶ **Lemma 4.3.** *Assuming all BCP data structures are initialized correctly, the Hungarian search terminates within $O(r)$ steps, and takes $O(r\sqrt{n}\,\mathrm{polylog}\, n)$ time.*

Once an augmenting path is found and the augmentation is performed, the set of excess/deficit nodes and $\mathrm{supp}(f)$ arcs change. We thus need to update the set $B^{\star}$, B_a, and $B^{<}$. This can be accomplished in $O(r\,\mathrm{polylog}\, n)$ amortized time. When we begin a new Hungarian search, we use the rewinding mechanism to set various BCP data structures in the right initial state. Finally, when we move from one scale to another, we also update the sets $A^{\circledast}$ and $B^{\circledast}$. Omitting all the details, we conclude the following.

▶ **Lemma 4.4.** *Each Hungarian search can be performed in $O(r\sqrt{n}\,\mathrm{polylog}\, n)$ time.*

Since there are $O(n \log n)$ augmentations and the flow in the original network can be recovered from that in the contracted network in $O(n \log n)$ time [2], the total running time of the algorithm is $O(rn^{3/2}\,\mathrm{polylog}\, n)$, as claimed in 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. *Proc. 33rd Int. Sympos. Comput. Geom. (SoCG)*, 7:1–7:16, 2017. ⟨https://doi.org/10.4230/LIPIcs.SoCG.2017.7⟩.
3. Pankaj K. Agarwal and Kasturi R. Varadarajan. A near-linear constant-factor approximation for Euclidean bipartite matching? *Proc. 20th Annu. Sympos. Comput. Geom. (SoCG)*, 247–252, 2004. ⟨https://doi.org/10.1145/997817.997856⟩.
4. D. Bertsekas and D. El Baz. Distributed asynchronous relaxation methods for convex network flow problems. *SIAM J. Control and Opt.* 25(1):74–85, 1987. ⟨https://doi.org/10.1137/0325006⟩.
5. Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for weighted matching in general graphs. *ACM Trans. Algorithms* 14(1):8:1–8:35, 2018. ⟨https://doi.org/10.1145/3155301⟩.
6. 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⟩.
7. 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⟩.
8. Andrew V. Goldberg, Sagi Hed, Haim Kaplan, and Robert E. Tarjan. Minimum-cost flows in unit-capacity networks. *Theoret. Comput. Sci.* 61(4):987–1010, 2017. ⟨https://doi.org/10.1007/s00224-017-9776-7⟩.
9. 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⟩.
10. Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. *Proc. 28th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, 2495–2504, 2017. ⟨https://doi.org/10.1137/1.9781611974782.165⟩.
11. Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 2(1-2):83–97. John Wiley & Sons, 1955.
12. Aleksander Mądry. Navigating central path with electrical flows: From flows to matchings, and back. *54th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, 253–262, 2013. ⟨https://doi.org/10.1109/FOCS.2013.35⟩.
13. 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⟩.
14. Lyle Ramshaw and Robert E. Tarjan. A weight-scaling algorithm for min-cost imperfect matchings in bipartite graphs. *Proc. 53rd Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, 581–590, 2012. ⟨https://doi.org/10.1109/FOCS.2012.9⟩.
15. R. Sharathkumar and Pankaj K. Agarwal. Algorithms for the transportation problem in geometric settings. *Proc. 23rd Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, 306–317, 2012. ⟨https://dl.acm.org/citation.cfm?id=2095116.2095145⟩.
16. R. Sharathkumar and Pankaj K. Agarwal. A near-linear time ϵ-approximation algorithm for geometric bipartite matching. *Proc. 44th Annu. ACM Sympos. Theory Comput. (STOC)*, 385–394, 2012. ⟨https://doi.org/10.1145/2213977.2214014⟩.
17. Éva Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica* 5(3):247–256, 1985. ⟨https://doi.org/10.1007/BF02579369⟩.

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

19 Kasturi R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, 320–331, 1998. ⟨https://doi.org/10.1109/SFCS.1998.743466⟩.