

Geometric Partial Matchings and Unbalanced Transportation Problem

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

1 Abstract

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

We present the first set of algorithms for geometric partial matching that work for any powers of L_p -norm matching objective and where the running time is near-linear in n : An exact algorithm which runs in $O((n + k^2) \text{polylog } n)$ time, and a $(1 + \varepsilon)$ -approximation algorithm which runs in $O((n + k\sqrt{k}) \text{polylog } n \cdot \log \varepsilon^{-1})$ time. Both algorithms are based on primal-dual flow augmentation scheme; the main improvements are obtained by using dynamic data structures to achieve efficient flow augmentations. Using similar techniques, we give an exact algorithm for the planar transportation problem, that runs in $O((r^2\sqrt{n} + rn^{3/2}) \text{polylog } n)$ time. For $r = o(\sqrt{n})$, this algorithm is faster than the state-of-art quadratic time algorithm by Agarwal *et al.* [SOCG 2016].

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms

Keywords and phrases partial matching, transportation, minimum-cost flow, rms-distance, bi-chromatic closest pair, cost scaling, excess scaling, primal-dual

Lines 505

1 Introduction

Given two point sets A, B in \mathbb{R}^2 , we consider the problem of finding the minimum-cost partial matching between A and B . Formally, suppose $|A| = r$ and $|B| = n$, with $r \leq n$. Let $G(A, B)$ be the undirected complete bipartite graph between A and B , and let the cost of $(a, b) \in A \times B$ be $c(a, b) = \|a - b\|_p$, for some $1 \leq p < \infty$. Define $C := \max_{(a,b) \in A \times B} c(a, b)$. 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 . For a given $q \geq 1$, the cost of M is defined to be

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

For a parameter $k \leq r$, we call the problem of finding the minimum-cost size- k matching in $G(A, B)$ the *geometric partial matching problem*. We call the respective problem in general



© Pankaj K. Agarwal, Hsien-Chih Chang, Allen Xiao;
licensed under Creative Commons License CC-BY

The 35th International Symposium on Computational Geometry (SOCG 2019).



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



28 bipartite graphs (with arbitrary edge costs) the *partial matching* problem.¹

29 We also consider the *transportation problem*, which is a generalization of the bipartite
 30 matching problem. Let $\lambda : A \cup B \rightarrow \mathbb{Z}$ be a *supply-demand function* with positive value on
 31 points of A , negative value on points of B , satisfying $\sum_{a \in A} \lambda(a) = -\sum_{b \in B} \lambda(b)$. Define
 32 $U := \max_{p \in A \cup B} |\lambda(p)|$. A *transportation map* is a function $\tau : A \times B \rightarrow \mathbb{R}_{\geq 0}$, such that
 33 $\sum_{b \in B} \tau(a, b) = \lambda(a)$ for all $a \in A$ and $\sum_{a \in A} \tau(a, b) = -\lambda(b)$ for all $b \in B$. We define the
 34 cost of τ to be

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

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

37 The transportation problem is essentially the discrete form of the *optimal transport*
 38 problem, which has been studied extensively in mathematics (see the books by Villani [40, 41]).
 39 Both discrete and continuous optimal transport have been used in a broad range of applications
 40 in e.g. computer vision and graphics [28, 32, 37], machine learning [7, 9], economics [18],
 41 engineering [26, 33], and medical imaging [21]. More recently, computational/numerical
 42 interest has expanded greatly due to the development of fast approximate solvers [5, 6, 11, 13];
 43 see the book by Cuturi and Peyré [29] and the survey by Solomon [36].

44 1.1 Related work

45 Both minimum-cost bipartite matching and maximum-cardinality bipartite matching are
 46 classical problems in the study of algorithms. Upper bounds for the latter include the
 47 $O(m\sqrt{n})$ time algorithm by Hopcroft and Karp [22] and the $O(m \min\{\sqrt{m}, n^{2/3}\})$ time
 48 algorithm by Even and Tarjan [16]. The first improvement over [16, 22] in over thirty years
 49 was the interior point algorithm by Madry [25], which runs in $O(m^{10/7} \text{polylog } n)$ time.

50 The Hungarian algorithm [24] computes a minimum-weight maximum matching in a
 51 bipartite graph in roughly mn time, where n is the number of vertices and m is the
 52 number of edges. Faster algorithms have been developed since for minimum-weight bipartite
 53 matching, such as the $O(m\sqrt{n} \log(nC))$ time algorithms by Gabow and Tarjan [17] and the
 54 $O(m\sqrt{n} \log C)$ time algorithm by Duan *et al.* [12]. The Hungarian algorithm can be extended
 55 to compute a minimum-cost partial matching of size k in time $O(km + k^2 \log r)$ time, see
 56 e.g. [30]. Ramshaw and Tarjan [30] proposed a cost-scaling algorithm for partial matching that
 57 runs in time $O(m\sqrt{n} \log(nC))$, assuming that the costs are integral; here C is the maximum
 58 cost of an edge. By reduction to unit-capacity min-cost flow, Goldberg *et al.* [19] developed
 59 a cost-scaling algorithm for partial matching with $O(m\sqrt{k} \log(kC)) = O(nr\sqrt{k} \log(kC))$
 60 running time, again only for integral edge costs.

61 In geometric settings, the Hungarian algorithm can be implemented to compute an optimal
 62 perfect matching between A and B (assuming $|A| = |B| = n$) in time $O(n^2 \text{polylog } n)$ [23] (see
 63 also [1, 38]). This algorithm can compute an optimal size- k matching in time $O(kn \text{polylog } n)$.
 64 Faster approximation algorithms have been developed for computing a perfect matching
 65 in geometric settings [4, 34, 38, 39]. The best known algorithm, by Sharathkumar and
 66 Agarwal [35], can compute a $(1 + \varepsilon)$ -approximation of the optimal perfect matching in
 67 $O(n\varepsilon^{-O(1)} \text{polylog } n)$ expected time with high probability, assuming $q = 1$, i.e. the cost of a
 68 matching is the sum of edge costs and the cost of an edge (a, b) is $\|a - b\|_p$ for some $p \geq 1$.
 69 Their algorithm can also compute a $(1 + \varepsilon)$ -approximate optimal partial matching within

24 ¹ Sometimes the partial matching problem is also called the *imperfect matching* or *imperfect assignment*
 25 [19, 30].

the same time bound. For $q > 1$, the best known approximation algorithm for computing a perfect matching runs in $O(n^{3/2} \log(1/\varepsilon) \text{polylog } n)$ time [34]. It is not obvious how to extend this algorithm to compute an approximate partial matching.

There is also some work on computing an optimal or near-optimal partial matching when B is fixed but A is allowed to translate and/or rotate [3, 8, 10, 31]. Here, the goal is to (i) compute a near-optimal matching over all possible transformations of B , or (ii) to compute a set \mathcal{M} of matchings so that for any translation/rotation \mathbb{T} of B , a (near) optimal matching of A and $\mathbb{T}(B)$ is in the set \mathcal{M} . Many of these algorithms use a partial matching algorithm on static points as a black box, so our results improve the final time bounds in e.g. [3, 8, 31].

The transportation problem can be formulated as a *minimum-cost flow* problem in a graph. Thus, the strongly polynomial uncapacitated min-cost flow algorithm by Orlin [27] solves transportation in $O((m + n \log n)n \log n)$ time. Lee and Sidford give a weakly polynomial algorithm that runs in $O(m\sqrt{n} \text{polylog}(n, U))$ time.

Agarwal *et al.* have shown that Orlin's algorithm can be implemented in the geometric setting in time $O(n^2 \text{polylog } n)$. It is an open question whether this algorithm can be implemented to run in $O(nr \text{polylog } n)$ time (recall that $|A| = n$, $|B| = r$). By adapting the Lee-Sidford algorithm, they developed an $(1 + \varepsilon)$ -approximation algorithm with $O(n^{3/2} \varepsilon^{-2} \text{polylog } n)$ running time. They also gave a Monte-Carlo algorithm that computes a $O(\log^2(1/\varepsilon))$ -approximate solution in $O(n^{1+\varepsilon})$ time with high probability.

1.2 Our results

There are three main results in this paper. First (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, B in \mathbb{R}^2 , with $|A|, |B| \leq n$, a minimum-cost matching of size k between A and B can be computed in $O((n + k^2) \text{polylog } n)$ time.*

As in [1, 23], we use *bichromatic closest pair* (BCP) data structures to implement the Hungarian algorithm efficiently. But unlike [1, 23], which take $\Omega(n)$ time to find an augmenting path, we show that after $O(n \text{polylog } n)$ preprocessing an augmenting path can be found in $O(k \text{polylog } n)$ time, without starting a completely new *Hungarian search*. We refer to this idea as a *rewinding* mechanism, and it will be used in our other algorithms as well.

Next (Sections 3, 4), by adapting the unit-capacity min-cost flow algorithm by Goldberg *et al.* [19] and implementing it efficiently we obtain a $(1 + \varepsilon)$ -approximation algorithm for the 2D geometric partial matching problem:

► **Theorem 1.2.** *Given two point sets A, B in \mathbb{R}^2 , with $|A|, |B| \leq n$, a $(1 + \varepsilon)$ -approximate minimum-cost matching of size k between A and B can be computed in $O((n + k\sqrt{k}) \log(1/\varepsilon) \text{polylog } n)$ time.*

The main challenge for this algorithm is what we call *null vertices*, which do not contribute to an augmentation, but for which the algorithm may spend $\Omega(n)$ time every augmentation. Instead, we run the unit-capacity min-cost flow algorithm on a *shortcut network* which has paths circumventing all null vertices. The shortcut graph itself may have $\Omega(n^2)$ edges, but we can query its minimum arcs efficiently using a BCP oracle without explicit construction. Once we have done this, we are able to charge the execution time of the iteration to the size of the *flow support*, the set of arcs with positive flow, which turns out to be size $O(k)$.

Finally (Section 5) we present a faster algorithm for the transportation problem when the two point sets are unbalanced i.e. $|A| = r$, $|B| = n$ and $r \ll n$:

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

117 Our third algorithm uses the strongly polynomial uncapacitated minimum-cost flow
 118 algorithm by Orlin [27], adapted for geometric costs as in Agarwal *et al.* [2]. This improves
 119 over their $O(n^2 \text{polylog } n)$ time algorithm when $r = o(\sqrt{n})$. Unlike matchings, the flow
 120 support in transportation may have size $\Omega(n)$ even if $r = O(1)$, so it seems difficult to
 121 charge execution time to flow support as before. However, we show that most of these
 122 support arcs are degree one and partition into *stars* centered around vertices of A , and
 123 the remainder is size $O(r)$. We describe a data structure that can process these stars in
 124 amortized $O((r^2/\sqrt{n} + r\sqrt{n}) \text{polylog } n)$ time per augmentation.

125 2 Minimum-Cost Partial Matchings using Hungarian Algorithm

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

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

133 A vertex v is *matched* by M if v is the endpoint of some matching edge in M ; otherwise
 134 v is *unmatched*. Given a matching M , an *augmenting path* $\Pi = (a_1, b_1, \dots, a_\ell, b_\ell)$ is an
 135 odd-length path with unmatched endpoints $(a_1$ and $b_\ell)$; Π alternates between edges outside
 136 and inside of matching M . The symmetric difference $M \oplus \Pi$ creates a new matching of size
 137 $|M| + 1$, called the *augmentation* of M by Π .

138 2.1 The Hungarian Algorithm

139 The Hungarian algorithm is initialized with $M = \emptyset$ and $\pi = 0$. Each iteration of the
 140 Hungarian algorithm augments M with an admissible augmenting path Π , discovered using
 141 a procedure called the *Hungarian search*. The algorithm terminates once M has size k ;
 142 Ramshaw and Tarjan [30] showed that M is guaranteed to be an optimal partial matching.

143 The Hungarian search tries grow a set of *reachable vertices* S by augmenting paths
 144 consisting of admissible edges. Initially, S is the set of unmatched vertices in A . Let the
 145 *frontier* of S be the edges in $(A \cap S) \times (B \setminus S)$. In each iteration, the Hungarian search first
 146 *relaxes* the minimum-reduced-cost edge (a, b) in the frontier, raising $\pi(a)$ by $c_\pi(a, b)$ for all
 147 $a \in S$ to make (a, b) admissible, and adding b into S . If b is already matched, then we also
 148 relax the matching edge (a', b) and add a' into S . The search finishes when b is unmatched,
 149 and an admissible augmenting path now can be recovered.

150 2.2 Fast implementation of Hungarian search

151 In the geometric setting, we find the min-reduced-cost frontier edge using a dynamic *bichro-*
 152 *matic closest pair* (BCP) data structure, as observed in []. Given two point sets P and Q
 153 in the plane, the bichromatic closest pair are two points $a \in P$ and $b \in Q$ minimizing the
 154 additively weighted distance $\|a - b\| - \omega(a) + \omega(b)$ for some real-valued vertex weights ω .
 155 $\langle\langle d(a, b) \rangle\rangle$ Thus, the minimum reduced-cost among the frontier edges is precisely the cost
 156 of the BCP of point sets $P = A \cap S$ and $Q = B \setminus S$, with $\omega(p) = \pi(p)$.

The dynamic BCP data structure by Kaplan *et al.* [23] supports point insertions and deletions in $O(\text{polylog } n)$ time and answers queries in $O(\log^2 n)$ time for our setting. During each relaxation, we perform at most one query and add a vertex to S incurring one BCP insertion or deletion.

Rewinding mechanism. We cannot afford to take $O(n \text{ polylog } n)$ time to initialize the BCP data structure at the beginning of every Hungarian search beyond the first. To resolve the issue, observe that exactly one vertex is removed from the set of unmatched vertices in A after each Hungarian search. Thus we can recover the initial state of the data structure by keeping track of a list of the points added to S over the course of the Hungarian search. At the start of the next Hungarian search, *rewind* the data structure by tracing the list in reverse order, and perform a single deletion to remove the newly matched vertex in A .

The number of points in the list is at most $O(k)$ as it is bounded by the number of relaxations per Hungarian search. Thus, in $O(k \text{ polylog } n)$ time, we can recover the BCP data structure for each Hungarian search beyond the first. We refer to this procedure as the *rewinding mechanism*.

We adapt a trick from Vaidya [38] to batch potential updates under the rewinding mechanism, so that the time spent on potential updates per Hungarian search is $O(k)$. See Appendix A.1 for details. Putting everything together we obtain the following:

► **Lemma 2.1.** *Each Hungarian search can be implemented in $O(k \text{ polylog } n)$ time after a one-time $O(n \text{ polylog } n)$ preprocessing.*

Our implementation of the Hungarian algorithm therefore runs in $O((n + k^2) \text{ polylog } n)$ time. This proves Theorem 1.1.

3 Approximating Min-Cost Partial Matching through Cost-Scaling

The goal of this section and the next is to prove Theorem 1.2; that is, to compute a size- k geometric partial matching between two point sets A and B in the plane, with cost at most $(1 + \varepsilon)$ times the optimal matching, in time $O((n + k\sqrt{k}) \text{ polylog } n \log(1/\varepsilon))$. The improvements come from **«TO BE FINISHED»**

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

3.1 Preliminaries on Network Flows

Due to the space restriction, we omit the definitions of standard network flow theory terminologies from the main text. For a reference see Appendix B.1, or any texts on network flows [?] **«cite»**. We emphasize that a directed graph $G = (V, E)$ is augmented by edge costs c and capacities u , and a supply-demand function ϕ defined on the vertices. A *network* $N = (V, \vec{E})$ turns each edge in E into a pair of *arcs* $v \rightarrow w$ and $w \rightarrow v$ in arc set \vec{E} . With the unit-capacity assumption on the network, all the pseudoflows in this section take integer values. The *support* of a pseudoflow f in N , is the set of arcs with positive flows: $\text{supp}(f) := \{v \rightarrow w \in \vec{E} \mid f(v \rightarrow w) > 0\}$. If all vertices are *balanced*, the pseudoflow is a

198 *circulation*. The *cost* of a pseudoflow is defined to be

$$199 \quad \text{cost}(f) := \sum_{v \rightarrow w \in \text{supp}(f)} c(v \rightarrow w) \cdot f(v \rightarrow w).$$

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

202 **LP-duality and admissibility.** To solve the minimum-cost flow problem, we focus on the
203 primal-dual algorithms using linear programming. Let $G = (V, E)$ be a given directed graph
204 with the corresponding network $N = (V, \vec{E}, c, u, \phi)$. Formally, the *potentials* $\pi(v)$ are the
205 variables of the linear program dual to the standard linear program for the minimum-cost
206 flow problem with variables $f(v, w)$ for each directed edge in E . Assignments to the primal
207 variables satisfying the capacity constraints extend naturally into a pseudoflow on the network
208 N . Let $G_f = (V, \vec{E}_f)$ be the residual graph under pseudoflow f . The *reduced cost* of an arc
209 $v \rightarrow w$ in \vec{E}_f with respect to π is defined as

$$210 \quad c_\pi(v \rightarrow w) := c(v \rightarrow w) - \pi(v) + \pi(w).$$

211 Notice that the cost function c_π is also antisymmetric.

212 The *dual feasibility constraint* says that $c_\pi(v \rightarrow w) \geq 0$ holds for every directed edge (v, w)
213 in E ; potentials π which satisfy this constraint are said to be *feasible*. Suppose we relax the
214 dual feasibility constraint to allow some small violation in the value of $c_\pi(v \rightarrow w)$. We say that
215 a pair of pseudoflow f and potential π is *ε -optimal* [?, ?] if $c_\pi(v \rightarrow w) \geq -\varepsilon$ for every residual
216 arc $v \rightarrow w$ in \vec{E}_f . Pseudoflow f is *ε -optimal* if it is ε -optimal with respect to some potentials
217 π ; potential π is *ε -optimal* if it is ε -optimal with respect to some pseudoflow f . Given a
218 pseudoflow f and potentials π , a residual arc $v \rightarrow w$ in \vec{E}_f is *admissible* if $c_\pi(v \rightarrow w) \leq 0$. We
219 say that a pseudoflow g in G_f is *admissible* if all support arcs of g on G_f are admissible; in
220 other words, $g(v \rightarrow w) > 0$ holds only on admissible arcs $v \rightarrow w$. Throughout the rest of the
221 paper we make use of the property that perform an admissible flow augmentation does not
222 change the ε -optimality; see Lemma B.1 for a proof.

223 3.2 Reduction to Unit-Capacity Min-Cost Flow Problem

224 The goal of the subsection is to reduce the minimum-cost partial matching problem to
225 the unit-capacity minimum-cost flow problem with a polynomial bound on diameter of the
226 underlying point set.

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

237 First we show that the number of arcs used by any integer pseudoflow in N_H is asymp-
238 totically bounded by the excess of the pseudoflow.

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

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

243 ► **Lemma 3.2.** *Let f be an ε -optimal integer circulation in N_H , and f^* be an optimal integer*
 244 *circulation for N_H . Then, $\text{cost}(f) \leq \text{cost}(f^*) + 6k\varepsilon$.*

245 **Multiplicative approximation.** Now we employ a technique from Sharathkumar and Agarwal [34] to convert the additive approximation into a multiplicative one. The reduction does
 246 not work out of the box, as they were tackling a similar but different problem on geometric
 247 transportations. See Appendix B.2 for details.

249 ► **Lemma 3.3.** *Computing $(1 + \varepsilon)$ -approximate geometric partial matching reduces to the*
 250 *following problem in $O(n \text{ polylog } n)$ time: Given reduction network N over a point set*
 251 *with diameter at most $K \cdot kn^3$ **«really? $K \cdot kn^{q+2}$?»** for some constant K , compute a*
 252 *$(K \cdot \varepsilon/6k)$ -optimal circulation on N .*

253 3.3 High-Level Description of Cost-Scaling Algorithm

257 Our main algorithm for the unit-capacity minimum-cost flow problem is based on the *cost-*
 258 *scaling* technique, originally due to Goldberg and Tarjan [20]; Goldberg *et al.* [19] applied
 259 the technique on unit-capacity networks. The algorithm finds ε -optimal circulations for
 260 geometrically shrinking values of ε . Each fixed value of ε is called a *cost scale*. Once ε is
 261 sufficiently small, the ε -optimal flow is a suitable approximation according to Lemma 3.3.²

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

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

269 Thus for any $\varepsilon^* > 0$, the algorithm produces an ε^* -optimal circulation after $O(\log(kC/\varepsilon^*))$
 270 scales. Using the reduction in Lemma 3.3, we have the diameter of the point set, thus
 271 maximum cost C , bounded by $O(K \cdot kn^3)$ for some value K . By setting ε^* to be $K \cdot \varepsilon/6k$,
 272 the number of cost scales is bounded above by $O(\log(n/\varepsilon))$.

273 **Scale initialization.** The vertex set of H consists of two point sets A and B , as well as
 274 two dummy vertices s and t . The directed edges in H are pointed from s to A , from A
 275 to B , and from B to t . We call those arcs in N_H whose direction is consistent with their
 276 corresponding directed edges as *forward arcs*, and those arcs that points in the opposite
 277 direction as *backward arcs*.

278 **«Describe how it's different from original.»** The procedure SCALE-INIT transforms a
 279 2ε -optimal circulation from the previous cost scale into an ε -optimal flow with $O(k)$ excess,
 280 by raising the potentials π of all vertices in A by ε , those in B by 2ε , and the potential of t

254 ² When the costs are integers, an ε -optimal circulation for a sufficiently small ε (say less than $1/n$) is itself
 255 an optimal solution [19, 20]. We present this algorithm without the integral-cost assumption because in
 256 the geometric partial matching setting (with respect to L_p norms) the costs are generally not integers.

281 by 3ε . The potential of s remains unchanged. Now the reduced cost of every forward arc is
 282 dropped by ε , and thus all the forward arcs have reduced cost at least $-\varepsilon$.

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

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

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

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

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

304 The algorithm continues until the total excess becomes zero.

305 First we analyze the number of iterations executed by REFINE. The proof follows the
 306 strategy in Goldberg *et al.* [19, Section 3.2]. Due to space constraint we omit all the proofs
 307 here; see Appendix B.3 for complete proofs. **«Explain what is new here.»**

308 ► **Lemma 3.4.** *Let f be a pseudoflow in N_H with $O(k)$ excess. The procedure REFINE runs
 309 for $O(\sqrt{k})$ iterations before the excess of f becomes zero.*

310 4 Fast Implementation of Refinement

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

316 Both Hungarian search and depth-first search are implemented in a Dijkstra-like fashion,
 317 traversing through the residual graph using admissible arcs starting from the excess vertices.
 318 Each step of the search procedures *relaxes* a minimum-reduced-cost arc from the set of visited
 319 vertices to an unvisited vertex, until a deficit vertex is reached. At a high level, our analysis
 320 strategy is to charge the relaxation events to the support arcs of f , which has size at most
 321 $O(k)$ by Corollary B.4.

4.1 Null vertices and shortcut graph

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

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

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

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

«Talk about the size of shortcut graph briefly.»

4.2 Dynamic data structures for search procedures

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

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

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

Depth-first search. Depth-first search is similar to Hungarian search in that it uses the relaxation of minimum-reduced-cost arcs/null paths, this time to identify admissible arcs/null paths in a depth-first manner. Similar to the Hungarian search, for each category of arcs in \tilde{H}_f leaving \tilde{S} , we maintain a proper data structure to answer the minimum-reduced-cost arc leaving a *fixed* vertex in \tilde{S} given by the query. Thus unlike Hungarian search which uses BCP data structures, we use dynamic nearest-neighbor data structures instead \square .

Each of the above data structures requires $O(1)$ queries and updates per relaxation. So in collaboration each relaxation can be implemented in $O(\text{polylog } n)$ time $[?, ?]$.

Time analysis. The complete time analysis can be found in Appendix C.2, C.3, and C.4; here we sketch the ideas. First we show (in Appendix C.2) that both Hungarian search and depth-first search performs $O(k)$ relaxations before a deficit vertex is reached, by looking at shortcut and non-shortcut relaxations separately. Both types of relaxations are eventually charged to the support size of f . As for the time analysis (see Appendix C.3), using the same rewinding mechanism as in Section 2.2, the running time of the Hungarian search and depth-first search, other than the potential updates, can be charged to the number of relaxations. Again using the trick by Vaidya [38] we can charge the potential updates of normal vertices to the number of relaxations in the Hungarian search. We never explicitly maintain the potentials on the null vertices; instead they are reconstructed whenever needed, either at the end of each iteration of refinement or when an augmentation sends flow through a null vertex. We show that such updates does not happen often in Appendix C.4. This completes the time analysis, which we summarize as follows:

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

5 Unbalanced Transportation

In this section, we give an exact algorithm which solves the planar transportation problem in $O((n + k\sqrt{k}) \log(1/\varepsilon) \text{ polylog } n)$ time, proving Theorem 1.3. Our strategy is to use the standard reduction to the uncapacitated min-cost flow problem, and provide a fast implementation under the geometric setting for the uncapacitated min-cost flow algorithm by Orlin [27], combined with some of the tools developed in Sections 2 and 3.

We assume the readers are familiar with Orlin’s strongly polynomial-time algorithm for uncapacitated min-cost flow problem [27]. For a brief introduction to Orlin’s algorithm see Appendix D.1, as well as the original well-written paper. In short, Orlin’s algorithm follows the *excess-scaling* paradigm under the primal-dual framework: Maintain a *scale parameter* Δ , initially set to U . A vertex v is *active* if $|\phi_f(v)| \geq \alpha\Delta$ for a fixed parameter $\alpha \in (0.5, 1)$. Repeatedly runs a *Hungarian search* that raises potentials (while maintaining dual feasibility) to create an admissible augmenting excess-deficit path between active vertices, on which we perform flow augmentations. Once there are no more active excess or deficit vertices, Δ is halved. Each sequence of augmentations where Δ holds a constant value is called an *excess scale*. On top of that, the algorithm performs contraction on arcs with flow value at least $3n\Delta$ at the beginning of a scale, in which case the flow and potentials are no longer tracked, as well as aggressive Δ -lowering under circumstances.

► **Theorem 5.1 (Orlin [27, Theorems 2 and 3]).** *Orlin’s algorithm finds a set of optimal potentials after $O(n \log n)$ scaling phases and $O(n \log n)$ total augmentations.*

408 The remainder of the section focuses on showing that each augmentation can be im-
 409 plemented in $O((r^2/\sqrt{n} + r\sqrt{n}) \text{polylog } n)$ time (after preprocessing). A subtle issue is
 410 that our geometric data structures must deal with real points in the plane instead of the
 411 contracted components. A solution is provided by Agarwal *et al.* [2]; we supply the details
 412 for maintaining contractions under dynamic data structures in Appendix D.2.

413 **Recovering optimal flow.** Use a strategy from Agarwal *et al.* [2], we can recover the optimal
 414 flow in time $O(n \text{polylog } n)$. If furthermore the cost function is just the p -norm (without the
 415 q th-power), an even stronger result stands: In this case, the set of admissible arcs under an
 416 optimal potential forms a planar graph, and thus we can apply the planar maximum-flow
 417 algorithm [?, 15] which runs in $O(n \log n)$ time. For details see Appendix D.3 and D.4.

418 5.1 Support stars

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

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

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

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

430 Roughly speaking, we would like to handle each support star as a single unit. When the
 431 Hungarian search reaches a or any $b \in \Sigma_a$, the entirety of Σ_a (as well as a) is also admissibly-
 432 reachable and can be included into S without further potential updates. Additionally, the
 433 only outgoing residual arcs of every $b \in \Sigma_a$ lead to a , thus the only way to leave $\Sigma_a \cup \{a\}$
 434 is through an arc leaving a . Once a relaxation step reaches some $b \in \Sigma_a$ or a itself, we
 435 would like to quickly update the state such that the rest of $b \in \Sigma_a$ is also reached without
 436 performing relaxation steps to each individual $b \in \Sigma_a$.

437 5.2 Implementation details

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

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

446 **Relaxations outside support stars.** For relaxations that don't involve support star vertices,
 447 we can once again maintain a BCP data structure to query the minimum A_ℓ -to- B_ℓ arc. To
 448 elaborate, this is the BCP between $P = A_\ell \cap S$ and $Q = (B_\ell \setminus (\bigcup_{a \in A_\ell} \Sigma_a)) \setminus S$, weighted

by potentials. Since the query is outside the support stars, there is at most one update per relaxation. Backward (support) arcs are kept admissible by the invariant, so we relax them immediately when they arrive at the frontier.

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

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

- All small stars are added to a single BCP data structure \mathcal{D}_{small} between $P = A_\ell \cap S$ and $Q = (\bigcup_{a \in A_{small}} \Sigma_a) \setminus S$. When an $a \in A_{small}$ or any vertex of its support star is added to S , we remove the points of Σ_a from \mathcal{D}_{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 $b \rightarrow a$, as all support arcs are admissible.

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

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

Updating support stars. As the flow support changes, the membership of support stars may shift and a big star may eventually become small (or vice versa). To efficiently support this, we introduce a soft boundary in determining whether a support star is big or small. Standard charging argument shows that the amortized update time is $O(r\sqrt{n}(r + \sqrt{n}) \text{polylog } n)$; for a complete argument see Appendix D.7. The number of star membership changes by adding a single augmenting path is bounded above by twice of its length. There are $O(rn \log n)$ augmentations in total. Each membership change can be performed in $O(\text{polylog } n)$ time. The total time spent on big-to-small updates is $O(rn \text{polylog } n)$, and the total time spent on small-to-big updates is $O(r^2 \sqrt{n} \text{polylog } n)$.

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

Between searches. After each augmentation, we reset the data structures to their initial state plus the change from augmentation using rewinding mechanism (see Section 2.2). This takes time proportional to the time for Hungarian search, which is $O(r\sqrt{n} \text{polylog } n)$ by Lemma 5.3. The most recent augmentation may have deactivated $O(1)$ active excess and deficit vertices, which takes $O(\sqrt{n} \text{polylog } n)$ time to update. Finally, we note that an augmenting path cannot reduce the support degree of a vertex to zero, and therefore no new dead vertices are created by augmentation.

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

Putting it together. After $O(rn \text{polylog } n)$ preprocessing, we spend $O(r\sqrt{n} \text{polylog } n)$ time on each Hungarian search by Lemma 5.3. After each augmentation, we spend the same amount of time to initialize data structures for the next Hungarian search. We spend up to $O((r^2\sqrt{n} + rn) \text{polylog } n)$ time making the switch between big/small stars. We spend $O(n^{3/2} \text{polylog } n)$ time activating and reviving vertices. Thus, the algorithm takes $O(rn(r/\sqrt{n} + \sqrt{n}) \text{polylog } n)$ time to produce optimal potentials π^* , from which we can recover f^* in $O(r\sqrt{n} \text{polylog } n)$ additional time. This completes the proof of Theorem 1.3.

Acknowledgment. We thank Haim Kaplan for useful discussion and suggesting to use Goldberg *et al.* [19] for our approximation algorithm.

References

- 1 Pankaj K. Agarwal, Alon Efrat, and Micha Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM J. Comput.* 29(3):912–953, 1999. <https://doi.org/10.1137/S0097539795295936>.
- 2 Pankaj K. Agarwal, Kyle Fox, Debmalya Panigrahi, Kasturi R. Varadarajan, and Allen Xiao. Faster algorithms for the geometric transportation problem. *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, 7:1–7:16, 2017. <https://doi.org/10.4230/LIPIcs.SocG.2017.7>.
- 3 Pankaj K. Agarwal, Haim Kaplan, Geva Kipper, Wolfgang Mulzer, Günter Rote, Micha Sharir, and Allen Xiao. Approximate minimum-weight matching with outliers under translation. *CoRR* abs/1810.10466, 2018. <http://arxiv.org/abs/1810.10466>.
- 4 Pankaj K. Agarwal and Kasturi R. Varadarajan. A near-linear constant-factor approximation for euclidean bipartite matching? *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, 247–252, 2004. <https://doi.org/10.1145/997817.997856>.
- 5 Jason Altschuler, Francis Bach, Alessandro Rudi, and Jonathan Weed. Approximating the quadratic transportation metric in near-linear time. *CoRR* abs/1810.10046, 2018. <http://arxiv.org/abs/1810.10046>.
- 6 Jason Altschuler, Jonathan Weed, and Philippe Rigollet. Near-linear time approximation algorithms for optimal transport via sinkhorn iteration. *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA, 1961–1971*, 2017. <http://papers.nips.cc/paper/6792-near-linear-time-approximation-algorithms-for-optimal-transport-via-sinkhorn-iteration>.
- 7 Martín Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 214–223, 2017. <http://proceedings.mlr.press/v70/arjovsky17a.html>.

- 534 **8** Rinat Ben Avraham, Matthias Henze, Rafel Jaume, Balázs Keszegh, Orit E. Raz, Micha
535 Sharir, and Igor Tubis. Partial-matching RMS distance under translation: Combinator-
536 ics and algorithms. *Algorithmica* 80(8):2400–2421, 2018. ([https://doi.org/10.1007/
537 s00453-017-0326-0](https://doi.org/10.1007/s00453-017-0326-0)).
- 538 **9** Federico Bassetti, Antonella Bodini, and Eugenio Regazzini. On minimum Kantorovich
539 distance estimators. *Statistics & Probability Letters* 76(12):1298 – 1302, 2006. ([http:
540 //www.sciencedirect.com/science/article/pii/S0167715206000381](http://www.sciencedirect.com/science/article/pii/S0167715206000381)).
- 541 **10** Sergio Cabello, Panos Giannopoulos, Christian Knauer, and Günter Rote. Matching point
542 sets with respect to the earth mover’s distance. *Comput. Geom.* 39(2):118–133, 2008.
543 (<https://doi.org/10.1016/j.comgeo.2006.10.001>).
- 544 **11** Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. *Ad-
545 vances in Neural Information Processing Systems 26: 27th Annual Conference on Neural
546 Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013,
547 Lake Tahoe, Nevada, United States.*, 2292–2300, 2013. ([http://papers.nips.cc/paper/
548 4927-sinkhorn-distances-lightspeed-computation-of-optimal-transport](http://papers.nips.cc/paper/4927-sinkhorn-distances-lightspeed-computation-of-optimal-transport)).
- 549 **12** Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for approximate and exact
550 maximum weight matching. *CoRR* abs/1112.0790, 2011. ([http://arxiv.org/abs/1112.
551 0790](http://arxiv.org/abs/1112.0790)).
- 552 **13** Pavel Dvurechensky, Alexander Gasnikov, and Alexey Kroshnin. Computational op-
553 timal transport: Complexity by accelerated gradient descent is better than by sink-
554 horn’s algorithm. *Proceedings of the 35th International Conference on Machine Learning,
555 ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 1366–1375, 2018.
556 (<http://proceedings.mlr.press/v80/dvurechensky18a.html>).
- 557 **14** Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency
558 for network flow problems. *J. ACM* 19(2):248–264, 1972. ([https://doi.org/10.1145/
559 321694.321699](https://doi.org/10.1145/321694.321699)).
- 560 **15** Jeff Erickson. Maximum flows and parametric shortest paths in planar graphs. *Proceedings
561 of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010,
562 Austin, Texas, USA, January 17-19, 2010*, 794–804, 2010. ([https://doi.org/10.1137/1.
563 9781611973075.65](https://doi.org/10.1137/1.9781611973075.65)).
- 564 **16** Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity.
565 *SIAM J. Comput.* 4(4):507–518, 1975. (<https://doi.org/10.1137/0204043>).
- 566 **17** Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for network problems.
567 *SIAM J. Comput.* 18(5):1013–1036, 1989. (<https://doi.org/10.1137/0218069>).
- 568 **18** Alfred Galichon. *Optimal transport methods in economics*. Princeton University Press,
569 2016.
- 570 **19** Andrew V. Goldberg, Sagi Hed, Haim Kaplan, and Robert E. Tarjan. Minimum-cost
571 flows in unit-capacity networks. *Theory Comput. Syst.* 61(4):987–1010, 2017. ([https:
572 //doi.org/10.1007/s00224-017-9776-7](https://doi.org/10.1007/s00224-017-9776-7)).
- 573 **20** Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by success-
574 ive approximation. *Math. Oper. Res.* 15(3):430–466, 1990. ([https://doi.org/10.1287/
575 moor.15.3.430](https://doi.org/10.1287/moor.15.3.430)).
- 576 **21** Alexandre Gramfort, Gabriel Peyré, and Marco Cuturi. Fast optimal transport averaging of
577 neuroimaging data. *CoRR* abs/1503.08596, 2015. (<http://arxiv.org/abs/1503.08596>).
- 578 **22** John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings
579 in bipartite graphs. *SIAM J. Comput.* 2(4):225–231, 1973. ([https://doi.org/10.1137/
580 0202019](https://doi.org/10.1137/0202019)).
- 581 **23** Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic
582 planar voronoi diagrams for general distance functions and their algorithmic applications.
583 *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*,

- 584 *SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, 2495–2504, 2017. (<https://doi.org/10.1137/1.9781611974782.165>).
- 585
- 586 24 Harold W Kuhn. The Hungarian method for the assignment problem. *Naval Research*
- 587 *Logistics (NRL)* 2(1-2):83–97. Wiley Online Library, 1955.
- 588 25 Aleksander Madry. Navigating central path with electrical flows: From flows to matchings,
- 589 and back. *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS*
- 590 *2013, 26-29 October, 2013, Berkeley, CA, USA*, 253–262, 2013. (<https://doi.org/10.1109/FOCS.2013.35>).
- 591
- 592 26 VI Olikier. Near radially symmetric solutions of an inverse problem in geometric optics.
- 593 *Inverse Problems* 3(4):743. IOP Publishing, 1987.
- 594 27 James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations*
- 595 *Research* 41(2):338–350, 1993. (<https://doi.org/10.1287/opre.41.2.338>).
- 596 28 Nicolas Papadakis. *Optimal Transport for Image Processing. (Transport Optimal pour le*
- 597 *Traitement d’Images)*. 2015. (<https://tel.archives-ouvertes.fr/tel-01246096>).
- 598 29 Gabriel Peyré and Marco Cuturi. Computational Optimal Transport. *ArXiv e-prints* p.
- 599 arXiv:1803.00567, 2018.
- 600 30 Lyle Ramshaw and Robert Endre Tarjan. A weight-scaling algorithm for min-cost im-
- 601 perfect matchings in bipartite graphs. *53rd Annual IEEE Symposium on Foundations of*
- 602 *Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, 581–590,
- 603 2012. (<https://doi.org/10.1109/FOCS.2012.9>).
- 604 31 Günter Rote. Partial least-squares point matching under translations. *Proc. 26th European*
- 605 *Workshop Comput. Geom. (EuroCG 2010)* 249–251, 2010.
- 606 32 Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A metric for distributions with
- 607 applications to image databases. *6th Internat. Conf. on Comp. Vision*, 59–66, 1998. (<http://dx.doi.org/10.1109/ICCV.1998.710701>).
- 608
- 609 33 Yuliy Schwartzburg, Romain Testuz, Andrea Tagliasacchi, and Mark Pauly. High-contrast
- 610 computational caustic design. *ACM Trans. Graph.* 33(4):74:1–74:11, 2014. (<https://doi.org/10.1145/2601097.2601200>).
- 611
- 612 34 R. Sharathkumar and Pankaj K. Agarwal. Algorithms for the transportation problem in geo-
- 613 metric settings. *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Dis-*
- 614 *crete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, 306–317, 2012. (<http://portal.acm.org/citation.cfm?id=2095145&CFID=63838676&CFTOKEN=79617016>).
- 615
- 616 35 R. Sharathkumar and Pankaj K. Agarwal. A near-linear time ϵ -approximation algorithm for
- 617 geometric bipartite matching. *Proceedings of the 44th Symposium on Theory of Computing*
- 618 *Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, 385–394, 2012. (<https://doi.org/10.1145/2213977.2214014>).
- 619
- 620 36 Justin Solomon. Optimal transport on discrete domains. *CoRR* abs/1801.07745, 2018.
- 621 (<http://arxiv.org/abs/1801.07745>).
- 622 37 Justin Solomon, Fernando de Goes, Gabriel Peyré, Marco Cuturi, Adrian Butscher, Andy
- 623 Nguyen, Tao Du, and Leonidas J. Guibas. Convolutional wasserstein distances: efficient
- 624 optimal transportation on geometric domains. *ACM Trans. Graph.* 34(4):66:1–66:11, 2015.
- 625 (<http://doi.acm.org/10.1145/2766963>).
- 626 38 Pravin M. Vaidya. Geometry helps in matching. *SIAM J. Comput.* 18(6):1201–1225, 1989.
- 627 (<https://doi.org/10.1137/0218080>).
- 628 39 Kasturi R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching
- 629 in the plane. *39th Annual Symposium on Foundations of Computer Science, FOCS ’98,*
- 630 *November 8-11, 1998, Palo Alto, California, USA*, 320–331, 1998. (<https://doi.org/10.1109/SFCS.1998.743466>).
- 631
- 632 40 Cédric Villani. *Topics in optimal transportation*. no. 58. American Mathematical Soc.,
- 633 2003.

1:16 **Geometric Partial Matchings and Unbalanced Transportation Problem**

- 634 **41** Cédric Villani. *Optimal transport: old and new*. vol. 338. Springer Science & Business
635 Media, 2008.

A Missing Details and Proofs from Section 2

A.1 Potential updates

We modify a trick from Vaidya [38] to batch potential updates. Potentials have a *stored value*, i.e. the currently recorded value of $\pi(v)$, and a *true value*, which may have changed from $\pi(v)$. The resulting algorithm queries the minimum-reduced-cost under the true values of π and updates the stored value occasionally.

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

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

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

B Missing Details and Proofs from Section 3

B.1 Preliminaries on Network Flows

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

Pseudoflows. Given a network $N := (V, \vec{E}, c, u, \phi)$, a *pseudoflow* (or *flow* to be short) $f: \vec{E} \rightarrow \mathbb{Z}^3$ on N is an antisymmetric function on the arcs of N satisfying $f(v \rightarrow w) \leq u(v \rightarrow w)$ for every arc $v \rightarrow w$. We sometimes abuse the terminology by allowing pseudoflow to be defined on a directed graph, in which case we are actually referring to the pseudoflow on the corresponding network by extending the flow values anti-symmetrically to the arcs.

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

We say that f *saturates* an arc $v \rightarrow w$ if $f(v \rightarrow w) = u(v \rightarrow w)$; an arc $v \rightarrow w$ is *residual* if $f(v \rightarrow w) < u(v \rightarrow w)$. The *support* of f in N , denoted as $\text{supp}(f)$, is the set of arcs with positive flows:

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

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

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

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

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

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

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

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

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

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

Admissible flow augmentation.

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

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

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

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

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

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

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

Let π be some potential certifying that f is 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

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

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

B.2 Multiplicative approximation

Let T be the minimum spanning tree on input graph G and order its edges by increasing length as e_1, \dots, 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 geometric partial matching problem has edges between components of T_i . Choose j to be the smallest index larger than i satisfying $c(e_j) \geq kn \cdot c(e_i)$. For each component K of T_j , let G_K be the subgraph of G induced on vertices of K ; let $A_K := K \cap A$ and $B_K := K \cap B$, 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 B.2 (Sharathkumar and Agarwal [34, §3.5]).** *Let $G = (A, B, E_0)$ be the input to geometric partial matching 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 3.3, 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 B.2. The simplest way of achieving 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 3.3 by computing an $(\varepsilon c(e_i)/6k)$ -optimal circulation f on the point set after translations using additive approximation from Lemma 3.2, together with the bound $c(e_i) \leq \text{cost}(M^*)$ from Lemma B.2.

One small problem remains: We need to show that such reduction can be performed in $O(n \text{ polylog } n)$ time. Sharathkumar and Agarwal [34] have shown that the partition of A and B into A_K s and B_K s can be computed in $O(n \text{ polylog } n)$ 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 geometric partial matching problem which we do not know.

To solve this issue we perform a binary search on the edges e_1, \dots, 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»

B.3 Number of iterations during refinement.

To this end we need a bound on the size of the support of f right before and throughout the execution of REFINE. This bound will also be used in the analysis for the running time of REFINE.

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

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

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

► **Lemma 3.4.** *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 $d(v)$ defined to be the amount of potential increase at v , measured in units of ε ; in other words, $d(v) := (\pi(v) - \pi_0(v))/\varepsilon$.

Now divide the iterations executed by the procedure REFINE into two phases: The transition from the first phase to the second happens when every excess vertex v has $d(v) \geq \sqrt{k}$. At most \sqrt{k} iterations belong to the first phase as each Hungarian search increases the potential π by at least 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 $O(\sqrt{k})$. Consider the set of arcs $E^+ := \{v \rightarrow w \mid f(v \rightarrow w) < f_0(v \rightarrow w)\}$. The total amount of excess is upper bounded by the number of arcs in E^+ that crosses an arbitrarily given cut X that separates the excess vertices from the deficit vertices, when the network has unit-capacity [19, Lemma 3.6]. Consider the set of cuts $X_i := \{v \mid d(v) > i\}$ for $0 \leq i < \sqrt{k}$; every such cut separates the excess vertices from the deficit vertices at the end of first phase. Each arc in E^+ crosses at most 3 cuts of type X_i [19, Lemma 3.1]. So there is one X_i crossed

by at most $3|E^+|/\sqrt{k}$ arcs in E^+ . The size of E^+ is bounded by the sum of support sizes of f and f_0 ; by Corollary B.4 the size of E^+ is $O(k)$. This implies an $O(\sqrt{k})$ bound on the total excess after the first phase, which in turn bounds the number of iterations in the second phase. \blacktriangleleft

C Missing Details and Proofs from Section 4

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

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

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

C.1 Dynamic data structures for search procedures

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

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

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

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

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

C.2 Number of relaxations

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

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

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

An A or B vertex with 0 imbalance must have an even number of $\text{supp}(f)$ edges. There is either only one positive-capacity incoming arc (for A) or outgoing arc (for B), so this quantity is either 0 or 2. Since the vertex is normal, this must be 2. We charge 0.5 to each of the two $\text{supp}(f)$ arcs; the arcs of $\text{supp}(f)$ have no more than 1 charge each. Thus, the number of type (ii) vertex relaxations is $O(|\text{supp}(f)|)$. By Corollary B.4, $O(|\text{supp}(f)|) = O(k)$.

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

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

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

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

C.3 Time analysis

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

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

Proof. Each of the constant number of data structures used by the Hungarian search can be constructed in $O(n \text{ polylog } n)$ time. For each data structure queried during a relaxation, the new vertex moved into S causes a constant number of updates, each of which can be implemented in $O(\text{polylog } n)$ time. We first prove that the number of BCP operations during the Hungarian search over is bounded by $O(k)$.

1. Let S^t denote the initial set S at the beginning of the t -th Hungarian search. Assume for now that, at the beginning of the $(t + 1)$ -th Hungarian search, we have the set S^t from the previous iteration. To construct S^{t+1} , we remove the vertices that had excess decreased to zero by the t -th blocking flow. Thus, 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. **«Too strong as a bound? Is it enough to look at one Hungarian search?»**
2. During each Hungarian search, a vertex entering S may incur one BCP insertion/deletion. We can charge the updates to the number of relaxations over the course of Hungarian search. The number of relaxations in a Hungarian search is $O(k)$ by Lemma C.1.
3. To obtain S^t , we keep track of the points added to S^t since the last Hungarian search. After the augmentation, we remove those points added to S^t . By (2) there are $O(k)$ such points to be deleted, so reconstructing S^t takes $O(k)$ BCP operations.

For potential updates, we use the same trick by Vaidya [38] to lazily update potentials after vertices leave S (similar to Lemma ??), but this time only for normal vertices. Normal 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 vertices in S . When a vertex leave S (through the rewind mechanism above), we restore its potential as $\pi(v) \leftarrow \omega(v) + \delta$. With lazy updates, the number of potential updates on normal vertices is bounded by the number of relaxations in the Hungarian search, which is $O(k)$ by Lemma C.1. Note that null vertex potentials are not handled in the Hungarian search. **«then where? Lemma C.6»** ◀

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

Proof. At the beginning of REFINE, we can initialize the $O(1)$ data structures used in DFS in $O(n \text{ polylog } n)$ time. We use the same rewinding mechanism as in Hungarian search (Lemma C.3) 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 C.2, the running time for depth-first search is $O(k \text{ polylog } n)$. ◀

C.4 Number of potential updates on null vertices

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

Size of blocking flows. Now we bound the total number arcs whose flow is updated by a blocking flow during the course of REFINE. This bounds both the time spent updating the flow on these arcs and also the time spent on null vertex potential updates (Lemma C.6).

► **Lemma C.5.** *The support of each blocking flow found in REFINE is of size $O(k)$.*

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

► **Lemma C.6.** *The number of end-of-REFINE null vertex potential updates is $O(n)$. The number of augmentation-induced null vertex potential updates in each invocation of REFINE is $O(k \log k)$.*

Proof. The number of end-of-REFINE potential updates is $O(n)$. Each update due to flow augmentation involves a blocking flow sending positive flow through a null path, causing a potential update on the passed null vertex. We charge this potential update to the edges of that null path, which are in turn arcs with positive flow in the blocking flow. For each blocking flow, no positive arc is charged more than twice. It follows that the number of augmentation-induced updates is at most the size of support of the blocking flow, which is $O(k)$ by Lemma C.5. According to Lemma 3.4 there are $O(\sqrt{k})$ iterations of REFINE before it terminates. Summing up we have an $O(k\sqrt{k})$ bound over the course of REFINE. ◀

Now combining Lemma C.3, Lemma C.4, and Lemma C.6 completes the proof of Theorem 1.2.

D Missing Details and Proofs from Section 5

D.1 Uncapacitated MCF by excess scaling

We give an outline of the strongly polynomial-time algorithm for uncapacitated min-cost flow problem from Orlin [27]. Orlin's algorithm follows an *excess-scaling* paradigm originally due to Edmonds and Karp [14]. Consider the basic primal-dual framework used in the previous sections: The algorithm begins with both flow f and potentials π set to zero. Repeatedly runs a *Hungarian search* that raises potentials (while maintaining dual feasibility) to create an admissible augmenting excess-deficit path, on which we perform flow augmentations. In terms of cost, f is maintained to be 0-optimal with respect to π and each augmentation over admissible edges preserves 0-optimality (by Lemma B.1). Thus, the final circulation must be optimal. The excess-scaling paradigm builds on top of this skeleton by specifying (i) between which excess and deficit vertices we send flows, and (ii) how much flow is sent by the augmentation.

The excess-scaling algorithm maintains a *scale parameter* Δ , initially set to U . A vertex v with $|\phi_f(v)| \geq \Delta$ is called *active*. Each augmenting path is chosen between an active

excess vertex and an active deficit vertex. Once there are no more active excess or deficit vertices, Δ is halved. Each sequence of augmentations where Δ holds a constant value is called an *excess scale*. There are $O(\log U)$ excess scales before $\Delta < 1$ and, by integrality of supplies/demands, f is a circulation.

With some modifications to the excess-scaling algorithm, Orlin [27] obtains a strongly polynomial bound on the number of augmentations and excess scales. First, an *active* vertex is redefined to be one satisfying $|\phi_f(v)| \geq \alpha\Delta$, for a fixed parameter $\alpha \in (0.5, 1)$. Second, arcs with flow value at least $3n\Delta$ at the beginning of a scale are *contracted* to create a new vertex, whose supply-demand is the sum of those on the two endpoints of the contracted arc. We use $\hat{G} = (\hat{V}, \hat{E})$ to denote the resulting *contracted graph*, where each $\hat{v} \in \hat{V}$ is a contracted component of vertices from V . Intuitively, the flow is so high on contracted arcs that no set of future augmentations can remove the arc from $\text{supp}(f)$. Third, in addition to halving, Δ is aggressively lowered to $\max_{v \in V} \phi_f(v)$ if there are no active excess vertices and $f(v \rightarrow w) = 0$ holds for every arc $v \rightarrow w \in \hat{E}$. Finally, flow values are not tracked within contracted components, but once an optimal circulation is found on \hat{G} , optimal potentials π^* can be *recovered* for G by sequentially undoing the contractions. The algorithm then performs a post-processing step which finds the optimal circulation f^* on G by solving a max-flow problem on the set of admissible arcs under π^* .

D.2 Implementing contractions

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

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

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

D.3 Recovering the optimal flow

We use the recovery strategy from Agarwal *et al.* [2], which runs in $O(n \text{ polylog } n)$ time. The main idea is that, if \mathcal{T} is an undirected *spanning tree of admissible edges* under optimal potentials π^* , then there exists an optimal flow f^* with support only on arcs corresponding to edges of \mathcal{T} . Intuitively, \mathcal{T} is a maximal set of linearly independent dual LP constraints for the optimal dual (π^*), so there exists an optimal primal solution (f^*) with support only on these arcs. To see this, we can use a perturbation argument: raising the cost of each non-tree edge by $\varepsilon > 0$ does not change $\text{cost}(\pi^*)$ or the feasibility of π^* , but does raise the cost of

any circulation f using non-tree edges. Strong duality suggests that $\text{cost}(f^*) = \text{cost}(\pi^*)$ is unchanged, therefore f^* must have support only on the tree edges.

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

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

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

It remains to show how we construct \mathcal{T} . We begin with a (spanning) *shortest path tree* (SPT) T in the residual network of f , under reduced costs and rooted at an arbitrary vertex r . For the SPT to span, we need the additional assumption that G is strongly connected. We can make G strongly connected by adding a 0-supply vertex s with arcs $s \rightarrow a$ for all $a \in A$ and $b \rightarrow a$ for all $b \in B$, with some high cost M . Following Orlin [27], these arcs cannot appear in an optimal flow if M is sufficiently high, and we can extend π^* to include s using $\pi^*(s) = 0$ if $M > \max_{b \in B} \pi^*(b)$. This extension to π^* preserves feasibility.

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

► **Lemma D.2 Orlin [27] Lemma 3.** *Let f be a flow satisfying the optimality conditions with respect to π^* . Then, (i) f satisfies the optimality conditions with respect to $\pi^\#$, and (ii) all SPT arcs are admissible under $\pi^\#$.*

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

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

Computing the SPT. We conclude by describing the procedure for building the SPT, i.e. by running Dijkstra's algorithm in the residual network. We use a geometric implementation

that is very similar to Hungarian search. We begin with $S = \{r\}$ and $d_r(r) = 0$, where r is our arbitrary root. For all other vertices, $d_r(v)$ is initially unknown. In each iteration, we relax the minimum-reduced cost arc $v \rightarrow w$ in the frontier $S \times (A \cup B) \setminus S$, adding w to S , and setting $d_r(w) = d_r(v) + c_{\pi^*}(v, w)$. Once $S = A \cup B$, the SPT T is the set of relaxed arcs.

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

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

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

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

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

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

When the matching objective uses the just the p -norms (that is, when $q = 1$), we can prove that the subgraph formed by admissible arcs is in fact *planar*. Planarity gives us two things towards a simple f^* recovery: there are only a linear number of admissible arcs, and the max-flow on them can be solved in near-linear time with planar graph max-flow algorithms.

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

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

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

An arc $a \rightarrow b$ is admissible when

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

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

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

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

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

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

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

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

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

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

We can construct the entire set of admissible arcs by repeatedly querying the minimum-reduced-cost outgoing arc for each $a \in A$ until the result is not admissible. By Lemma D.4 the resulting arc set forms a planar graph, so by Euler's formula the number of arcs to query is $O(n)$. We can then find the maximum flow in time $O(n \log n)$ time, using for example the planar maximum-flow algorithm by Erickson [15]. **◀cite others like Klein▶**

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

D.5 Dead vertices

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

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

D.6 Number of relaxations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D.7 Updating support stars

Initially, we label stars big or small according to the \sqrt{n} threshold. A star that is currently big is turned into a small star once $|\Sigma_a| \leq \sqrt{n}/2$. A star that is currently small is turned into a big star once $|\Sigma_a| \geq 2\sqrt{n}$. This way, the time spent rebuilding/updating the respective data structures can be amortized to the insertions/deletions that preceded the switch, plus some $O(r)$ extra work if the the update is small-to-big.

1243 A star Σ_a that is switching from big-to-small has size $|\Sigma_a| \leq \sqrt{n}/2$. When switching, we
 1244 delete $\mathcal{D}_{\text{big}}(a)$ and insert Σ_a into $\mathcal{D}_{\text{small}}$. Thus, the time spent for big-to-small update is
 1245 $O(\sqrt{n} \text{polylog } n)$, and there were at least $\sqrt{n}/2$ points removed from Σ_a since it was last big.
 1246 **«A MESS; NEED TO BE REWRITTEN»** A star Σ_a that is switching from small-
 1247 to-big has size $|\Sigma_a| = \sqrt{n} + x \geq 2\sqrt{n}$, for some integer $x \geq \sqrt{n}$. Rearranging, we have
 1248 $|\Sigma_a| \leq 2x$. When switching, we delete all $|\Sigma_a|$ points from $\mathcal{D}_{\text{small}}$ and construct a new $\mathcal{D}_{\text{big}}(a)$.
 1249 Constructing $\mathcal{D}_{\text{big}}(a)$ requires inserting $O(r)$ points of A (into P) and the $|\Sigma_a|$ points of the
 1250 star (into Q). Thus, the time spent for a small-to-big update is $O((r+x) \text{polylog } n)$, and
 1251 there were at least $x \geq \sqrt{n}$ points added to Σ_a since it was last small.