A Faster Maximum Cardinality Matching Algorithm with Applications in Machine Learning

Nathaniel Lahn*
School of Computing and Information Sciences
Radford University
Radford, VA 24142
nlahn@radford.edu

Sharath Raghvendra
Department of Computer Science
Virginia Tech Blacksburg, VA 24061
sharathr@vt.edu

Jiacheng Ye
Department of Computer Science
Virginia Tech Blacksburg, VA 24061
yjc0513@vt.edu

Abstract

Maximum cardinality bipartite matching is an important graph optimization problem with several applications. For instance, maximum cardinality matching in a δ-disc graph can be used in the computation of the bottleneck matching as well as the ∞-Wasserstein and the Lévy-Prokhorov distances between probability distributions. For any point sets $A, B \subset \mathbb{R}^2$, the δ-disc graph is a bipartite graph formed by connecting every pair of points $(a, b) \in A \times B$ by an edge if the Euclidean distance between them is at most δ. Using the classical Hopcroft-Karp algorithm, a maximum-cardinality matching on any δ-disc graph can be found in $\tilde{O}(n^{3/2})$ time. [2] In this paper, we present a simplification of a recent algorithm (Lahn and Raghvendra, JoCG 2021) for the maximum cardinality matching problem and describe how a maximum cardinality matching in a δ-disc graph can be computed asymptotically faster than $O(n^{3/2})$ time for any moderately dense point set. As applications, we show that if A and B are point sets drawn uniformly at random from a unit square, an exact bottleneck matching can be computed in $\tilde{O}(n^{4/3})$ time. On the other hand, experiments suggest that the Hopcroft-Karp algorithm seems to take roughly $\Theta(n^{3/2})$ time for this case. This translates to substantial improvements in execution time for larger inputs.

1 Introduction

Computing a maximum cardinality matching is a fundamental graph optimization problem. With origins in economics and logistics, matchings have found numerous applications. Computing popular distances between distributions, such as the Wasserstein distance as well as the Lévy-Prokhorov distance, can be reduced to a bipartite matching problem. In this paper, we consider the *δ-disc graph matching* which is the following:

Given two sets A and B of n two dimensional points and a parameter $\delta > 0$, a δ-disc graph G_δ is a bipartite graph obtained by connecting any pair of vertices $(a, b) \in A \times B$ with an edge provided that the Euclidean distance between a and b is at most δ, i.e., $\|a - b\| \leq \delta$. Let m be the number of edges

*Authors are ordered by last name. All authors contributed equally to this work.

[2] We use $\tilde{O}(\cdot)$ to suppress poly-logarithmic terms in the complexity.

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

in the graph G_δ. A *matching* M is a set of vertex-disjoint edges in G_δ. In the δ-disc graph matching problem, we wish to compute a *maximum cardinality matching* in G_δ, i.e., a matching that has the largest number of edges.

Any algorithm for computing a δ-disc graph matching can also be used to compute a bottleneck matching as well as the Lévy-Prokhorov distance, both of which are defined next. Let $M \subseteq A \times B$ be any *perfect* matching of A and B, which is a matching where every vertex of A is matched, i.e., $|M| = n$. The edge of M with the largest Euclidean length is its bottleneck edge. The *bottleneck matching* is a perfect matching M^* whose bottleneck edge length is minimized. The Euclidean length of the bottleneck edge of M^* is the *bottleneck distance* between A and B.

Distances between distributions: Next, consider the case where A and B define discrete distributions $\mathcal{P}_A$ and $\mathcal{P}_B$ and let every point $a \in A$ (resp. $b \in B$) carry a probability of $1/n$, i.e., $\mathcal{P}_A(a) = 1/n$ (resp. $\mathcal{P}_B(b) = 1/n$). The ∞-Wasserstein distance between $\mathcal{P}_A$ and $\mathcal{P}_B$, denoted by $W_\infty(\mathcal{P}_A, \mathcal{P}_B)$, is simply the bottleneck distance between the point sets A and B. The *Lévy-Prokhorov* distance between $\mathcal{P}_A$ and $\mathcal{P}_B$ is defined as follows: For $\varepsilon > 0$ and any subset $X \subseteq A$ (resp. $Y \subseteq B$), let $X^\varepsilon = \{b \in B \mid \exists a \in X \text{ such that } \|a - b\| \leq \varepsilon\}$ (resp. $Y^\varepsilon = \{a \in A \mid \exists b \in Y \text{ such that } \|a - b\| \leq \varepsilon\}$). Note that $X^\varepsilon \subseteq B$ and $Y^\varepsilon \subseteq A$. We say that the *Lévy-Prokhorov* [34] distance $\pi(\mathcal{P}_A, \mathcal{P}_B)$ is equal to the smallest value of ε for which the following property is true:

$$\forall X \subseteq A, \mathcal{P}_A(X) \leq \mathcal{P}_B(X^\varepsilon) + \varepsilon \text{ and, } \forall Y \subseteq B, \mathcal{P}_B(Y) \leq \mathcal{P}_A(Y^\varepsilon) + \varepsilon;$$

here $\mathcal{P}_A(X) = \sum_{a \in X} \mathcal{P}_A(a) = |X|/n$ and $\mathcal{P}_B(Y) = \sum_{b \in Y} \mathcal{P}_B(b) = |Y|/n$. We can, therefore, write this condition as

$$\forall X \subseteq A, |X| \leq |X^\varepsilon| + \varepsilon n \text{ and } \forall Y \subseteq B, |Y| \leq |Y^\varepsilon| + \varepsilon n. \tag{1}$$

Determining if the exact bottleneck distance (equivalently $W_\infty(\mathcal{P}_A, \mathcal{P}_B)$) is $\leq \delta$ can be done by simply finding the maximum cardinality matching M in G_δ. It is easy to see that M is perfect if and only if the bottleneck distance is at most δ. Similarly, by applying Hall's theorem, one can show that $\pi(\mathcal{P}_A, \mathcal{P}_B) \leq \varepsilon$ if and only if the maximum cardinality matching M in G_ε has a size of at least $(1 - \varepsilon)n$. Thus, the δ-disc graph matching directly relates to computing ∞-Wasserstein distance as well as the Lévy-Prokhorov distance between probability distributions.

Computing δ-disc graph matching: One can use any of-the-shelf matching algorithm [18, 31, 33, 39] to compute a maximum cardinality matching in a δ-disc graph. For instance, using the well-known Hopcroft-Karp algorithm will lead to an execution time of $O(m\sqrt{n})$ on any δ-disc graph. The HK-Algorithm executes in phases. Each phase takes $O(m)$ time, and, in the worst-case, the algorithm converges to a maximum matching in $O(\sqrt{n})$ phases. The best-known exact algorithm for computing the exact bottleneck matching combines geometric data structures with the HK-algorithm in order to reduce the execution time of each phase from $O(m)$ to $\tilde{O}(n)$. As a result, they obtain an exact bottleneck matching in $\tilde{O}(n^{3/2})$ time.

Inspired by a series of algorithms for weighted matching in graphs with small separators [4, 27], Lahn and Raghvendra [28] presented a weighted approach to the maximum cardinality matching problem. This LR algorithm identifies a set of "edge separators" incident on ω "boundary vertices". These edges are assigned a weight of 1 and all other edges receive a weight of 0. A property satisfied by these separator vertices is that after their removal, every connected component in the graph has no more than r vertices. Then, they present an algorithm to compute a perfect matching in $O(m\sqrt{r} + m\sqrt{\omega} + mr\omega/n \log n)$ time. As an application of their result, they show how to compute the bottleneck distance of any point sets A and B within a multiplicative factor of $(1 + \varepsilon)$ in $\tilde{O}(n^{4/3}\text{poly}(1/\varepsilon))$ time. The LR algorithm assigns dual weights to vertices and is similar in style to the Kuhn-Munkres [24] and Gabow-Tarjan [15] algorithms. The dual weights on vertices play a vital role in the proofs of correctness and efficiency of the LR algorithm.

In this paper, we make the following contributions:

- We remove the need to maintain dual weights in the LR algorithm, resulting in a significantly simpler algorithm.

- Using this algorithm, we show how to find a maximum cardinality matching in a unit-disc graph G_δ in time $\tilde{O}(n^{4/3}k^{1/3})$ where k is the maximum number of points of $A \cup B$ contained in any disc of radius δ. Note that our algorithm is asymptotically faster than the classical Hopcroft-Karp based algorithm when $k = o(\sqrt{n})$.
- Using our algorithm for δ-disc graph matching, we show how to compute the exact bottleneck distance between point sets A and B. When $\mathcal{P}_A$ and $\mathcal{P}_B$ are discrete distributions with each point having probability $1/n$, the bottleneck distance can be used to compute the distances $W_\infty(\mathcal{P}_A, \mathcal{P}_B))$ and $\pi(\mathcal{P}_A, \mathcal{P}_B)$. We are not aware of any previous polynomial time algorithms to compute the Lévy-Prokhorov distance. When A, B are chosen uniformly at random from a unit square, our algorithm for the exact bottleneck distance runs in $\tilde{O}(n^{4/3})$ time. All previous algorithms take $\Omega(n^{3/2})$ time.
- We run experiments for the case where A and B are chosen uniformly at random from a unit square. Our experiments suggest that the Hopcroft-Karp algorithm on G_δ takes $\Theta(n^{3/2})$ time. In contrast, our algorithm runs substantially faster and executes in $\tilde{O}(n^{4/3})$ time.

Note that, for the HK-algorithm, the upper bound of $O(\sqrt{n})$ phases is only in the worst-case. As noted by Motwani [32], the Hopcroft-Karp algorithm converges, with high probability, to a maximum matching in $O(\log n)$ phases for expander graphs in general and Erdős-Rényi random graphs in particular. Similarly, does the HK algorithm execute asymptotically fewer than $\sqrt{n}$ iterations when used to compute bottleneck matchings? Interestingly, our experiments suggest that the answer to this question may be in the negative. Based on our experimental results, when A and B are drawn uniformly at random from a unit square and when δ is set to the bottleneck distance, the number of phases seem to grow at the rate of $\Omega(\sqrt{n})$. Therefore, bottleneck matching on random point sets may represent a natural hard instance for the HK-Algorithm. In this paper, we show how to overcome the $\Omega(n^{3/2})$ barrier for uniformly distributed point sets by developing an $\tilde{O}(n^{4/3})$ time algorithm.

For any $\varepsilon > 0$ and any point set $A \cup B$, the LR algorithm can also be used to compute a multiplicative $(1+\varepsilon)$-approximation of the bottleneck matching in $\tilde{O}(n^{4/3}\text{poly}(1/\varepsilon))$ time. This is done by using a grid where the side-length of each cell is a function of ε. The algorithm rounds every point to the closest cell center and finds a δ-disc graph matching using the LR algorithm. See Section 6 of [28] for details. One can also use a similar approach to compute a multiplicative $(1+\varepsilon)$ approximation of the Lévy-Prokhorov distance. Replacing the original LR algorithm with our dual-free implementation leads to simpler approximation algorithms.

Applications: Wasserstein distance has found numerous applications in machine learning and computer vision [3, 5, 8, 10, 14, 36]. Due to these applications, computing approximations of Wasserstein distances has received substantial attention [2, 9, 12, 26, 30, 35]. However, exact algorithms (even for discrete distributions) have a relatively high execution time [15, 24, 39]. The Lévy-Prokhorov distance have been extensively studied for its theoretical properties [11, 38]. For instance, Lévy-Prokhorov distance metrizes weak convergence on any separable metric space [19]. However, a brute-force algorithm based on the definition of this metric will require a search on exponentially many possible subsets causing it to seldom be used in practice [16]. However, we use Hall's theorem to show that the computation of the Lévy-Prokhorov metric reduces to the δ-disc graph matching problem and so, it is only as hard as computing the ∞-Wasserstein distance, at least for discrete distributions of the type $\mathcal{P}_A$ and $\mathcal{P}_B$ described above.

In this paper, we provide exact algorithms for computing the ∞-Wasserstein and the Lévy-Prokhorov distances for 2-dimensional discrete distributions. Our algorithms can be useful in several scenarios. For instance, one can estimate Lévy-Prokhorov distances between any two fixed-dimensional continuous distributions by simply computing the distances between samples drawn from these distributions. From the fact that the Lévy-Prokhorov distance metrizes weak convergence, for large enough samples we can get accurate distance estimates. Faster high-precision algorithms are critical in obtaining such estimates; see [7, 6]. For high dimensional discrete distributions, Wasserstein distance is sometimes estimated by embedding them into a lower dimensional space and computing high precision solution in this space. For example, see the sliced Wasserstein distance [23].

In the emerging area of topological data analysis, high dimensional point clouds are characterized by two-dimensional point sets called persistence diagrams where each point represents the so-called birth and death times of a topological feature. Different high-dimensional point clouds can be compared by computing the bottleneck distance between the corresponding diagrams [17, 22, 1]. This has

led to development of practical implementations of bottleneck matching algorithms [17, 22]. More recently, other Wasserstein distances between persistence diagrams have also been considered. See for instance [25, 40].

The special case of computing bottleneck matching for point sets A and B that are drawn uniformly at random from a unit square has also received considerable attention. For instance, it has been used in the context of testing pseudo-random generators, average case analysis of bin packing algorithms [29], and also in statistics for analyzing the Glivenko-Cantelli convergence of empirical measures [37]. δ-disc graphs have other applications as well, including in the modeling of the topology of ad-hoc wireless networks [20].

2 Matching algorithms

In this section, we present and compare two algorithms for solving the maximum cardinality matching problem on an arbitrary graph: The Hopcroft-Karp (HK) algorithm [18], and our simplification of the LR algorithm [28]. In section 2.1, we introduce the basic definitions used by most combinatorial matching algorithms and give an overview of the HK algorithm. In section 2.2, we present our simplification of the LR algorithm, highlighting the differences it has from the HK algorithm.

2.1 Preliminaries

Given any matching M, let A_F and B_F denote the vertices of A and B respectively that are not matched in M. We refer to these vertices as *free* vertices. An *alternating path* P is a path that alternates between edges that are in the matching and those that are not in the matching. An *augmenting path* is an alternating path that starts and ends at a free vertex. We define the *length* of P as the number of edges in P.

We can augment a matching M along an augmenting path P by updating the matching to $M \leftarrow M \oplus P$; where $\oplus$ denotes the symmetric difference operator. It is easy to see that augmenting a matching along an augmenting path P increases the size of the matching M by 1. Furthermore, it can be shown that G has no augmenting paths with respect to a matching M if and only if M has maximum cardinality. These observations are the basis of the following commonly-used approach for computing a maximum-cardinality matching: repeatedly compute an augmenting path P with respect to M and augment M along P until M has maximum cardinality. Since the largest possible matching has size at most n, any such algorithm will arrive at a maximum-cardinality matching after n augmentations. This is the approach used by the classical Ford-Fulkerson and HK algorithms as well as the recent LR algorithm. However, the algorithms differ in the details of *how* these augmenting paths are found.

Residual graph: Given a matching M, the *residual graph* G_M is a directed graph that assists in finding augmenting paths. The graph G_M contains the same set of vertices V as G. For any edge (a, b) in G, if $(a, b) \in M$ then we add an edge directed from a to b to G_M. Otherwise, we add an edge directed from b to a to G_M. Furthermore, we create a source vertex s and a sink vertex t with the following additional edges. For each free vertex $b \in B_F$, we add an edge (s, b) directed from the source s to b in G_M and for each free vertex $a \in A_F$ we add an edge from a to the sink t in G_M. Note that, for the residual graph, we use (u, v) to denote an edge directed from u to v. On the other hand, for the undirected graph G, we may use (u, v) and (v, u) interchangeably to represent the same edge between vertices u and v. Consider any directed path P from s to t in the residual graph. Note that removing s and t from P will result in an augmenting path. In the Ford-Fulkerson algorithm, a single augmenting path can be found in $O(m)$ time using any common graph search algorithm such as breadth-first search (BFS) or depth-first search (DFS), leading to an $O(mn)$ time algorithm. The HK and LR algorithms both improve upon this running time by finding potentially many augmenting paths in each iteration.

HK algorithm: Initially, let $M = \emptyset$. The HK algorithm executes in *phases*. A phase is divided up into two *stages*. The first stage executes a BFS starting from s and identifies the length of the shortest path (path with the fewest edges) in G_M from s to every other vertex in G_M. Let ℓ_u be the length of the shortest path from s to u and let $\ell = \ell_t$. The algorithm then computes an *admissible graph* $\mathcal{A}$ consisting of all edges (u, v) in G_M such that (a) ℓ_u and ℓ_v are at most ℓ and (b) $\ell_v = \ell_u + 1$. Note that these edges capture the set of all minimum-length augmenting paths in G_M. The second

stage iteratively conducts multiple *partial DFSs* that start from s and terminate early if a path to t is found. Following the termination of a partial-DFS, all edges visited by it are removed from the residual graph. The algorithm proceeds to the next phase if a partial-DFS terminates without finding a path from s to t. It can be shown that, in each phase, the HK algorithm finds a maximal set of vertex-disjoint shortest augmenting paths in G_M. Each phase involves execution of a single BFS and multiple partial DFSs. Since no two executions of DFS visit the same edge, the combined execution time of the multiple partial DFSs is bounded by $O(m)$.

Hopcroft and Karp showed that the length of the shortest augmenting path increases by at least 1 after each phase. Therefore, after $\sqrt{n}$ phases, the shortest augmenting path has length at least $\sqrt{n}$. Using this, they showed that there are no more than $\sqrt{n}$ free vertices remaining, all of which can be matched by augmenting along an additional $O(\sqrt{n})$ augmenting paths. Thus, the total number of phases executed by the algorithm is $O(\sqrt{n})$. Since each phase takes $O(m)$ time, the total time taken by the algorithm is $O(m\sqrt{n})$. Somewhat surprisingly, Hopcroft and Karp [18] also showed that the total length of all n augmenting paths, across all phases, is only $O(n \log n)$.

2.2 A simplified implementation of the LR algorithm

Recently, Lahn and Raghvendra presented an algorithm [28] to compute a maximum cardinality matching. Their algorithm resembles the Kuhn-Munkres algorithm for weighted matching. In this section, we present a cleaner implementation of the LR algorithm. These simplifications result in a closer resemblance to the HK algorithm. Unlike the LR algorithm, our algorithm does not maintain any dual weights. In the following, we describe and contrast our algorithm with the HK-algorithm.

Apart from a bipartite graph $G(V, E)$, we are also given a subset $E_S \subseteq E$ of "separator edges" as input. For any separator edge $(u, v) \in E_S$, we denote the vertices u and v as *boundary vertices*. Let B be the set of all boundary vertices. The analysis of the algorithm depends on $\omega = |B|$ and another parameter r that is defined next. Consider the graph $G'(V, E \setminus E_S)$. Let $\mathbb{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_t\}$ be the set of connected components of G' and let $\mathcal{V}_i$ and $\mathcal{E}_i$ be the set of vertices and edges of $\mathcal{P}_i$ for all $1 \leq i \leq t$. We refer to each $\mathcal{P}_i$ in G' as a *piece* of the original graph G. Let $r = \max_{\mathcal{P}_i \in \mathbb{P}} |\mathcal{V}_i|$, i.e., the size of the piece of G with the largest number of vertices.

Setting weights on the edges of the graph G and its residual graph G_M: For any edge $(u, v) \in E$, we assign it a *weight* $w(u, v)$. For any separator edge $(u, v) \in E_S$, we set $w(u, v)$ to 1. For any other edge $(u', v') \in E \setminus E_S$, we set $w(u', v')$ to 0. Every edge (u, v) in the residual graph inherits the weight of the corresponding edge (u, v) in $G(V, E)$. All edges incident on the source s and the sink t in G_M receive a weight of 0. For any path P, its weight is simply the sum of the weights of its edges.

Preprocessing: In the preprocessing step, our algorithm finds a maximum cardinality matching for each piece by applying the HK-Algorithm. Let M be the union of these matchings computed across all pieces. At the end of this step, the difference $|M^*| - |M|$ is $O(\omega)$, where M^* is the maximum cardinality matching in G. So, our algorithm has to find an additional $O(\omega)$ augmenting paths in order to compute a maximum cardinality matching.

The remaining $O(\omega)$ unmatched vertices are subsequently matched in *phases*. Like the HK algorithm, each phase of our algorithm consists of two *stages*. These stages somewhat resemble the stages of the HK algorithm. We highlight the differences in the description below.

Stage 1: In the first stage, our algorithm finds, for any vertex $v \in V$, the minimum weight path from s to v in the residual graph G_M using the weights $w(\cdot, \cdot)$. Note that, since every edge weight is either 0 or 1, a standard BFS implementation can be modified to support such a minimum-weight search algorithm in $O(m)$ time by simply prioritizing edges of weight 0 over edges of weight 1. We call this modified version of BFS, *0/1 BFS*. For any vertex $v \in V$, let ℓ_v be the weight from s to v in G_M as computed by the $0/1$ BFS and let $\ell = \ell_t$. Any edge (u, v) of G_M is *admissible* if $\ell_u, \ell_v \leq \ell$ and $\ell_v = \ell_u + w(u, v)$. The *admissible* graph $\mathcal{A}$ is identical to G_M, except it contains only admissible edges. Similar to the HK algorithm, it can be shown that the admissible graph $\mathcal{A}$ captures every minimum-weight augmenting path.

Stage 2: The second stage of our algorithm finds a set of shortest augmenting paths (by weight). It does so by iteratively conducting partial-DFSs from s until no augmenting path is found. Each partial-DFS immediately terminates if an augmenting path P is found. Let $\mathcal{K}$ be the set of *affected pieces*, which are pieces that contain at least one edge of P. Unlike in the HK-Algorithm, the

matching M is immediately augmented along P and every edge visited by this partial-DFS that does not belong to an affected piece is deleted. Note that any edge from an affected piece that was visited by this partial-DFS does not get deleted and could be revisited by a later partial-DFS. In other words, edges in an affected piece can be visited multiple times within the same phase.

Differences with HK algorithm: The main differences between our algorithm and the HK algorithm are:

(1) Our algorithm assigns weights of 0 and 1 to the edges. No weights are assigned in the HK algorithm.

(2) Our algorithm has a preprocessing step that computes a maximum cardinality matching within each piece.

(3) In Stage 1, our algorithm executes a $0/1$-BFS instead of the BFS executed by HK algorithm.

(4) In Stage 2 of our algorithm, the partial-DFS reuses edges from affected pieces. As a result, in each phase, our algorithm may find augmenting paths that are not vertex-disjoint.

Differences (1) – (3) between HK algorithm and our algorithm do not impact the execution time of the algorithm by any more than a small constant factor. See Section F of the supplement for a discussion on this. The critical difference between the two algorithms is (4). Unlike the HK algorithm, Stage 2 of our algorithm reuses edges from affected pieces and computes a set of augmenting paths that are not necessarily vertex-disjoint. This allows for computing many more augmenting paths within each phase.

As we show later, the total number of phases executed as well as the augmenting paths computed by our algorithm are identical to those computed by the LR algorithm. Therefore, the analysis of Lahn and Raghvendra can be directly applied to our algorithm. They show that after each phase, the weight of the shortest augmenting path increases by at least one. After $\sqrt{\omega}$ phases, they show that there are $O(\sqrt{\omega})$ free vertices which can be matched using an additional $O(\sqrt{\omega})$ phases. Thus the total number of phases can be bounded by $O(\sqrt{\omega})$.

Edge revisits cause the execution time of a phase to increase. Note that an edge can be revisited only if it was inside an affected piece when it was most recently visited. Lahn and Raghvendra show that the total number of affected pieces is $O(\omega \log \omega)$ (a piece that is affected k times is counted k times in this sum). For graphs that admit recursive separators (such as planar and graphs with excluded minors), they show that the number of edges for any piece can be bounded by $O(mr/n)$ leading to an $O(\frac{mr\omega}{n} \log \omega)$ bound on the total number of revisits.

Theorem 1. *Consider a graph G and a set of separator edges. Suppose each piece has at most $O(mr/n)$ edges. Our algorithm computes a maximum cardinality matching in $O(m\sqrt{r} + m\sqrt{\omega} + \frac{mr\omega}{n} \log n)$ time.*

For our algorithm, the assumption on an upper bound on the number of edges within each piece can be eliminated when graphs, such as those considered in this paper, support a dynamic data structure $\mathcal{D}$ of the following form: $\mathcal{D}$ can store any subset $A' \subseteq A$ of vertices and, given any query vertex $b \in B$, it can return a vertex $a \in A'$ that minimizes the weight of the edge (b, a). Note that the weight of (b, a) will be 1 only if every edge from b to any vertex $a' \in A'$ has a weight of 1. If no edge exists between b and any vertex of A', then the data structure returns NULL. Suppose that $\mathcal{D}$ supports arbitrary insertions and deletions from A', as well as queries, each in $\Phi(n)$ time. Then, one can use this data structure to dynamically maintain the set of unvisited nodes of A during a 0/1 BFS or DFS. Consequently, one can execute 0/1 BFS and DFS in time $O(n\Phi(n))$. As a result, the execution time of our algorithm can be improved to $O(n\Phi(n)\sqrt{r} + n\Phi(n)\sqrt{\omega} + r\omega\Phi(n) \log n)$. In contrast, using $\mathcal{D}$ to execute a Hungarian Search inside the LR algorithm seems challenging.

Theorem 2. *Given a graph G that supports a dynamic nearest neighbor data structure with query and update time of $\Phi(n)$, a maximum cardinality matching can be computed by our algorithm in $O(\Phi(n)(n\sqrt{r} + n\sqrt{\omega} + r\omega \log n))$. The HK algorithm computes a maximum cardinality matching in $O(n^{3/2}\Phi(n))$ time.*

Equivalency to original LR algorithm: The original algorithm maintains a dual weight $y(v)$ for every vertex $v \in A \cup B$ at any point during the algorithm. For any edge $(a, b) \in (A \times B) \cap E$, the dual weights satisfy the following:

$$y(b) - y(a) \leq w(a,b) \qquad \text{if } (a,b) \notin M, \tag{2}$$
$$y(a) - y(b) = w(a,b) \qquad \text{if } (a,b) \in M. \tag{3}$$

Additionally, their algorithm maintains the invariants that all free vertices of B_F have the same dual weight of $y_{\max} = \max_{v \in A \cup B} y(v)$ and all free vertices of A have the same dual weight of 0. The *slack* $s(a,b)$ of any edge $(a,b) \in (A \times B) \cap E$ is defined as follows: if $(a,b) \notin M$, then $s(a,b) = w(a,b) - y(b) + y(a)$; otherwise, $(a,b) \in M$ and $s(a,b) = 0$.

In the original LR algorithm, the first stage adjusts the dual weights so that there is at least one zero-slack augmenting path in G_M. The second stage takes the subgraph consisting of zero slack edges and repeatedly executes a DFS from s. This DFS stops early if a path to t, i.e., an augmenting path, is found. After augmenting along a path, all edges visited by the DFS are deleted, unless they were in an affected piece.

Note that the only fundamental difference between the original version of the LR algorithm and our simplified version is the fact that we compute minimum-weight augmenting paths while they compute zero-slack augmenting paths. The following lemma, whose proof appears in Section A of the supplement, shows that the a zero slack path computed in the LR algorithm is also a minimum weight path. It follows that the two versions of the algorithm are equivalent.

Lemma 1. *During Stage 2 of the* LR *algorithm, an augmenting path has zero slack if and only if it has minimum weight.*

3 Applications

In this section, we show how our algorithm can be applied to efficiently compute a maximum cardinality matching on a δ-disc graph, an optimal bottleneck matching, as well as the Lévy-Prokhorov distance between distributions. All applications considered are for point sets $A, B \subset \mathbb{R}^2$. For all applications, one can build a data structure $\mathcal{D}$ from Theorem 2 with $\Phi(n) = \log^{O(1)} n$ by using a dynamic Euclidean nearest neighbor data structure; see Section H.3 of the supplement for details

3.1 δ-disc graph matching

Let $P = A \cup B$. Let $\mathbb{B}(p)$ be a ball centered at p with radius δ. Consider $k = \max_{p \in \mathbb{R}^2} |\mathbb{B}(p) \cap P|$, i.e., k is the largest number of points of $A \cup B$ inside any ball of radius δ. We refer to k as the *δ-density* of the point set P. We show that a maximum cardinality matching in a δ-disc graph can be computed using our algorithm in $\tilde{O}(n^{4/3}k^{1/3})$ time. Thus, when the δ-density $k = o(\sqrt{n})$, our algorithm outperforms the HK algorithm.

Theorem 3. *For any point set $P = A \cup B$ and a parameter $\delta > 0$, a maximum cardinality matching in the δ-disc graph defined on P can be computed in $\tilde{O}(n^{4/3}k^{1/3})$ time, where k is the δ-density of P.*

This result also extends to the case where the points of A and B are independently and identically distributed random variables drawn from distributions $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively. We say that a distribution $\mathcal{P}$ has a δ-density of k if, for any ball $\mathbb{B}(p)$ of radius δ, the probability that a point drawn from $\mathcal{P}$ lies inside the ball is at most k/n.

Theorem 4. *Let A, B be drawn iid from distributions $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively. For a parameter $\delta > 0$, a maximum cardinality matching in the δ-disc graph defined on $A \cup B$ can be computed, with high probability, in $\tilde{O}(n^{4/3}k^{1/3})$ time, where k is the maximum of the δ-density of $\mathcal{P}_A$ and $\mathcal{P}_B$.*

Proof of Theorem 3: We show how a set of separator edges can be generated so that $\omega = O(n^{2/3}k^{2/3})$ and $r = O(n^{2/3}/k^{1/3})$. From Theorem 2 and since $n\sqrt{r} = O(n^{4/3}/k^{1/6})$, $n\sqrt{\omega} = O(n^{4/3}k^{1/3})$, and, $r\omega = O(n^{4/3}k^{1/3})$, the execution time of the LR algorithm can be bounded by $\tilde{O}(n^{4/3}k^{1/3})$. Next, we describe how to generate the separator edges E_S.

We use a *grid* $\mathbb{G}$ to generate the separator edges. Any grid consists of a set of equispaced horizontal and vertical lines that partition $\mathbb{R}^2$ into *cells*. Each cell is a square and any grid can be seen as a

set of these squares. Let $\mathcal{L}(\mathbb{G})$ denote the side-length of any cell $C \in \mathbb{G}$. We say that a cell C is *non-empty* if $C \cap P \neq \emptyset$. Let $\theta = \lceil n^{1/3}/k^{2/3} \rceil$. To generate our pieces, we choose a grid $\mathbb{G}$ where the side-length of each cell is set to $\mathcal{L}(\mathbb{G}) = \theta\delta$. The separator edge set E_S consists of all edges of the δ-disc graph that have their endpoints in different cells. All such edges are assigned a weight of 1. Any edges whose endpoints are contained within the same cell of $\mathbb{G}$ are in $E \setminus E_S$ and are assigned a weight of 0. Any point that has at least one separator edge incident on it becomes a boundary vertex. To generate $\mathbb{G}$, we check $O(\theta)$ possible vertical and horizontal shifts and pick the one that minimizes the number of boundary vertices. We provide the details of generating $\mathbb{G}$ in Section B.1 of the supplement. Our choice of $\mathbb{G}$ guarantees that $\omega = O(n^{2/3}k^{2/3})$ for any point set, independent of its δ-density. We show this in Section B.2 of the supplement.

Bounding r: By this definition, the number of vertices of any piece is bounded by the maximum number of points that can lie inside any cell of $\mathbb{G}$, i.e., $\max_{C \in \mathbb{G}} |C \cap P|$. Note that we can cover any cell of $\mathbb{G}$ with $\Theta(\theta^2)$ balls of radius δ each. Due to the δ-density of P being k, each of these balls can contain at most k points and the total number of points inside any cell can be bounded by $O(\theta^2 k) = O(n^{2/3}/k^{1/3})$ as desired. In other words, $r = O(n^{2/3}/k^{1/3})$.

Proof of Theorem 4: The construction of the grid here will be identical to the one in the proof of Theorem 3. Note also that the bound on ω provided in that proof depends only on the construction of $\mathbb{G}$ and not on the δ-density of the point set. Therefore, the same bound continues to hold here as well. In Section C of the supplement, we use the δ-density of $\mathcal{P}_A$ and $\mathcal{P}_B$ along with Chernoff's bound to prove that $r = O(n^{2/3}/k^{1/3})$ points with high probability.

3.2 Bottleneck distance

In this section, we assume that A and B are points drawn uniformly at random from a unit square. From the work of Leighton and Shor [29], we know that, for appropriate constants $c_{\min}$ and $c_{\max}$, the optimal bottleneck distance is at least $\delta_{\min} = \frac{c_{\min} \log^{3/4}(n)}{\sqrt{n}}$ and at most $\delta_{\max} = \frac{c_{\max} \log^{3/4}(n)}{\sqrt{n}}$ with very high probability (probability exceeding $1 - 1/n^{\alpha}$ for some $\alpha = \Omega(\sqrt{\log n})$). Observe that the optimal bottleneck distance will be equal to the length of some edge of $A \times B$. As in the work of Efrat *et al.* [13], our algorithm will use a selection algorithm of Katz and Sharir [21] to find the jth smallest edge, $1 \leq j \leq n^2$ in $O(n^{4/3} \log^2 n)$ time. Let $d(j)$ be the length of the jth smallest edge returned by their algorithm. This allows us to execute a binary search over the edges of $A \times B$, ordered by their length. Let $g_{\min} = 1$ and $g_{\max} = n^2$. We repeat the following process until $g_{\max} = g_{\min} + 1$: We choose $j = \lfloor (g_{\max} + g_{\min})/2 \rfloor$ and find the jth smallest edge whose length is denoted by $d(j)$. If $d(j) \geq \delta_{\max}$, we set $g_{\max} \leftarrow j$. If $d(j) \leq \delta_{\min}$, we set $g_{\min} \leftarrow j$. Otherwise, $\delta_{\min} \leq d(j) \leq \delta_{\max}$, and we find the maximum cardinality matching in a δ-disc graph where δ is set to $d(j)$. If we obtain a perfect matching, we set $g_{\max} = j$. Otherwise, the maximum matching is not perfect, and we set $g_{\min} = j$. When the algorithm terminates, $d(g_{\max})$ is the optimal bottleneck distance.

Analysis: The algorithm makes $O(\log n)$ many guesses. These guesses are found by a selection algorithm that runs in $O(n^{4/3} \log^2 n)$ time. For each guess j where $\delta_{\min} \leq d(j) \leq \delta_{\max}$, we must compute a maximum-cardinality matching on a δ-disc graph, which takes $O(n^{4/3}k^{1/3})$ time using the LR algorithm. Since, $\mathcal{P}(A)$ and $\mathcal{P}(B)$ are the uniform distribution, their δ-density increases as δ increases. Therefore, the $\delta_{\max}$-density of $\mathcal{P}(A)$ and $\mathcal{P}(B)$ will be an upperbound on the δ-density for any execution of the LR algorithm. The probability that any random point lies within any ball of radius $\delta_{\max}$ is at most $(2\delta_{\max})^2$, which is $\Theta(\log^{3/2}(n)/n)$. Therefore, the $\delta_{\max}$-density of $\mathcal{P}(A)$ and $\mathcal{P}(B)$ is at most $k = O(\log^{3/2}(n))$. Applying Theorem 4 gives the following:

Theorem 5. *Let A, B be drawn uniformly at random from a unit square. An optimal bottleneck matching can be computed between A and B, with high probability, in $\tilde{O}(n^{4/3})$ time.*

Practical considerations: The algorithm described uses two black-boxes that are impractical and have hidden high constants in the Big-O notation. (a) the algorithm relies on a dynamic nearest neighbor data structure, and, (b) the algorithm uses the selection algorithm of Katz and Sharir [21]. In Section D of the supplement, we address both (a) and (b) by presenting more practical alternatives.

3.3 Lévy-Prokhorov distance

In this section, we show that we can use our algorithm for the δ-disc graph matching to also compute the Lévy-Prokhorov distance.

We describe a simple algorithm to decide if $\pi(\mathcal{P}_A, \mathcal{P}_B)$ greater than or at most ε. We compute a maximum cardinality matching M in an ε-disc graph. Let A_F and B_F be the free vertices with respect to M. Then, we say that the distance is greater than ε if $|A_F| > \varepsilon n$. Otherwise, we say that the distance is at most ε. Using a binary search similar to the one described in Section 3.2, we can determine the distance in $\tilde{O}(n^{4/3}k^{1/3})$ time.

Proof via Hall's theorem: Given any bipartite graph $G(A \cup B, E)$, for any set $X \subseteq A$, the neighborhood $\mathcal{N}(X)$ is the set of all vertices of B that share an edge with at least one vertex of X. Thus, X^ε is the neighborhood of X in an ε-disc graph. The deficiency of a graph with respect to A is $\mu(A) = \max_{X \subseteq A} |X| - \mathcal{N}(X)$. Hall's theorem says that a bipartite graph has a perfect matching if and only if the deficiency of the graph with respect to A is non-positive. Hall's theorem can be generalized to the following.

Lemma 2. *For any bipartite graph $G(A \cup B, E)$, where $|A| = |B| = n$, and for any integer $k > 0$, the deficiency with respect to A or B is k if and only if the maximum cardinality matching is of size $n - k$.*

The proof of this generalization follows in a straight-forward way from the Hall's theorem. For the sake of completion, we provide this proof in Section E of the supplement. Next, we show that the algorithm described here correctly computes the Lévy-Prokhorov distance.

Recollect that, our algorithm returns the distance to be greater than ε if $|A_F| > \varepsilon n$. By Lemma 2, we conclude that the deficiency of the graph is greater than εn, i.e., there is a set $X \subseteq A$ such that $|X| - |X^\varepsilon| > \varepsilon n$. Thus, Equation 1 does not hold and the distance is greater than ε.

Our algorithm returns a distance at most ε if $|A_F| \leq \varepsilon n$. In this case, from Lemma 2, the deficiency of the graph with respect to A is less than εn, i.e., for every subset $X \subseteq A$, $|X| - |X^\varepsilon| \leq \varepsilon n$. Note that $|A_F| = |B_F| \leq \varepsilon n$ and so an identical argument applies for B as well. Thus, Equation 1 holds and the distance is at most ε. We conclude that the algorithm terminates with the correct ε.

Theorem 6. *Let the point sets A and B, $|A| = |B| = n$ describe two distributions $\mathcal{P}_A$ and $\mathcal{P}_B$ where each point has a probability of $1/n$ associated with it. The Lévy-Prokhorov distance $\pi(\mathcal{P}_A, \mathcal{P}_B)$ can be computed in $\tilde{O}(n^{4/3}k^{1/3})$ time where k is the δ-density of $A \cup B$.*

4 Experimental results

In this section, we compare the performance of the HK algorithm and our algorithm when applied to computing an exact bottleneck matching between equal-sized point sets $A, B \subset \mathbb{R}^2$ drawn uniformly at random from a unit square, where $n = |A| + |B|$.

Experimental setup: For each value of n in $\{100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 1500000\}$, we execute 10 runs. For each run, we uniformly sample points from a unit square to obtain the point sets A and B. Next, we compute a bottleneck matching between A and B separately, using both the HK algorithm and our algorithm, and record performance metrics for both algorithms. We execute our experiments on a server running CentOS Linux 7, with 12 Intel E5-2683v4 cores and 128GB of RAM.[3]

When guessing the bottleneck distance for each run, instead of enforcing that the number of guesses is $O(\log n)$ it is sufficient in practice to continue the binary search on δ until the relative error becomes less than a sufficiently small value ε (see Section D of the supplement). Both the HK-based algorithm and our algorithm use the same strategy for guessing the bottleneck distance in the experiments.

Experimental results: For each figure, the data presented for each value of n is averaged over all 10 runs. Error bars represent a single standard deviation. Figure 1 presents the actual running time of both algorithms, summed over all guesses of the bottleneck distance. For datasets with more than 10^6 points, our algorithm takes roughly half as much time as the HK algorithm and the gap seems to grow as the input size increases. However, the actual running times can be affected by several

[3] Our implementations of our algorithm and HK can be found at https://github.com/nathaniellahn/JOCGV3

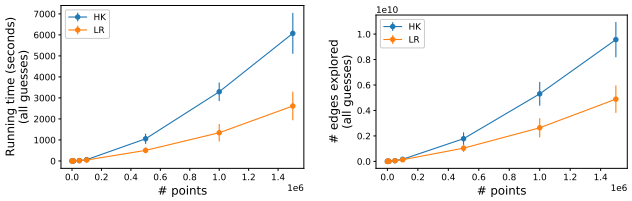

Figure 1: A running time comparison between the HK algorithm and our algorithm. Left: Comparison of actual running time. Right: Comparison of total number of edge visits.

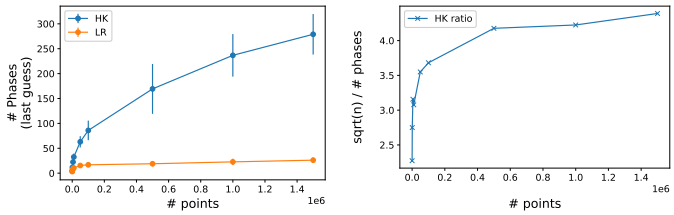

Figure 2: Data for number of phases for the final bottleneck guess. Left: A comparison of the number of phases for the HK algorithm and our algorithm. Right: $\sqrt{n}$ divided by the number of phases executed by the HK algorithm.

factors including the exact implementation details and execution environment. Therefore, we focus on comparing metrics that are accurate independent of the exact implementation details. Recall that both algorithms combine variants of BFS and DFS to compute augmenting paths. As a result, the total number times edges are visited during each algorithm acts as an implementation-independent proxy of the running time. Figure 1 shows the total number of edge visits for both algorithms. Note that this data seems to follow a similar trend to the actual running times of the algorithms.

Next, we summarize our observations that help account for this difference in performance of the two algorithms. For more details, see Section G of the supplement. Recall that there are four main differences (1) – (4) between the HK algorithm and our algorithm. As discussed in Section F of the supplement, differences (1) – (3) do not have any direct significant impact on the relative running times of the two algorithms; the most significant difference is (4) – Stage 2 of the our algorithm reuses edges from affected pieces. This reuse of edges has two main effects on the efficiency of the our algorithm. First, we find that our algorithm executes significantly fewer phases than the HK algorithm. Specifically, as the guess of the bottleneck distance approaches the actual bottleneck distance, our results suggest that the number of phases executed by the HK algorithm seems to grow at a rate of $\Theta(\sqrt{n})$ – exhibiting its worst-case analysis. In contrast, the number of phases executed by our algorithm grows at a much slower rate (see Figure 2). This explains why the our algorithm runs faster than the HK algorithm. The second impact of allowing for edge revisits is that a single edge can be revisited, perhaps many times, during a single phase. Despite this, the total number of edges visited by our algorithm is still significantly less than the total number of edges visited by the HK algorithm (see Figure 1).

5 Conclusion

We consider the maximum cardinality matching problem and present a simplification of a recent algorithm by Lahn and Raghvendra [28]. In particular, we eliminate the need to maintain dual weights in their algorithm. This not only leads to a simpler algorithm but also results in new and improved exact algorithms for computing the δ-disc graph matching, bottleneck matching, as well as the ∞-Wasserstein and the Lévy-Prokhorov distances, in low-density settings. We would like to conclude by stating the following open question: Can we design a parallel combinatorial algorithm to compute a δ-disc graph matching?

Acknowledgements We would like to acknowledge, Advanced Research Computing (ARC) at Virginia Tech, which provided us with the computational resources used to run the experiments. Research presented in this paper was funded by NSF CCF-1909171. We would like to thank the anonymous reviewers for their useful feedback.

References

[1] A. Adcock, D. Rubin, and G. Carlsson, Classification of hepatic lesions using the matching metric, *Computer vision and image understanding*, 121 (2014), 36–42.

[2] J. Altschuler, J. Weed, and P. Rigollet, Near-linear time approximation algorithms for optimal transport via sinkhorn iteration, *Neural Information Processing Systems*, 2017, pp. 1961–1971.

[3] M. Arjovsky, S. Chintala, and L. Bottou, Wasserstein GAN, *arXiv:1701.07875v3 [stat.ML]*, (2017).

[4] M. K. Asathulla, S. Khanna, N. Lahn, and S. Raghvendra, A faster algorithm for minimum-cost bipartite perfect matching in planar graphs, *ACM Trans. Algorithms*, 16 (2020), 2:1–2:30.

[5] J.-D. Benamou, G. Carlier, M. Cuturi, L. Nenna, and G. Peyré, Iterative bregman projections for regularized transportation problems, *SIAM Journal on Scientific Computing*, 37 (2015), A1111–A1138.

[6] E. Bernton, P. E. Jacob, M. Gerber, and C. P. Robert, On parameter estimation with the Wasserstein distance, *Information and Inference: A Journal of the IMA*, 8 (2019), 657–676.

[7] G. Beugnot, A. Genevay, K. Greenewald, and J. Solomon, Improving approximate optimal transport distances using quantization, *arXiv preprint arXiv:2102.12731*, (2021).

[8] J. Bigot, R. Gouet, T. Klein, A. López, et al., Geodesic PCA in the wasserstein space by convex PCA, *Annales de l'Institut Henri Poincaré, Probabilités et Statistiques*, Vol. 53, Institut Henri Poincaré, 2017, pp. 1–26.

[9] J. Blanchet, A. Jambulapati, C. Kent, and A. Sidford, Towards optimal running times for optimal transport, *arXiv:1810.07717 [cs.DS]*, (2018).

[10] M. Cuturi and A. Doucet, Fast computation of wasserstein barycenters, *International Conference on Machine Learning*, 2014, pp. 685–693.

[11] R. M. Dudley, Distances of probability measures and random variables, *The Annals of Mathematical Statistics*, 39 (1968), 1563–1572.

[12] P. Dvurechensky, A. Gasnikov, and A. Kroshnin, Computational optimal transport: Complexity by accelerated gradient descent is better than by sinkhorn's algorithm, *International Conference on Machine Learning*, 2018, pp. 1366–1375.

[13] A. Efrat, A. Itai, and M. J. Katz, Geometry helps in bottleneck matching and related problems, *Algorithmica*, 31 (2001), 1–28.

[14] R. Flamary, M. Cuturi, N. Courty, and A. Rakotomamonjy, Wasserstein discriminant analysis, *Machine Learning*, 107 (2018), 1923–1945.

[15] H. N. Gabow and R. E. Tarjan, Faster scaling algorithms for network problems, *SIAM Journal on Computing*, 18 (1989), 1013–1036.

[16] A. L. Gibbs and F. E. Su, On choosing and bounding probability metrics, *International statistical review*, 70 (2002), 419–435.

[17] F. Godi, Bottleneck distance, in: *GUDHI User and Reference Manual*, GUDHI Editorial Board, 3.4.1 edition, 2021.

[18] J. E. Hopcroft and R. M. Karp., An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs., *SIAM Journal on Computing*, 2 (1905), 1973–231.

[19] P. J. Huber, *Robust Statistics*, New York: John Wiley and Sons, 1981.

[20] M. L. Huson and A. Sen, Broadcast scheduling algorithms for radio networks, *Proceedings of MILCOM'95*, Vol. 2, IEEE, 1995, pp. 647–651.

[21] M. J. Katz and M. Sharir, An expander-based approach to geometric optimization, *SIAM Journal on Computing*, 26 (1997), 1384–1408.

[22] M. Kerber, D. Morozov, and A. Nigmetov, Geometry helps to compare persistence diagrams, *Journal of Experimental Algorithmics (JEA)*, 22 (2017), 1–20.

[23] S. Kolouri, K. Nadjahi, U. Simsekli, R. Badeau, and G. Rohde, Generalized sliced wasserstein distances, *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), Vol. 32, Curran Associates, Inc., 2019.

[24] H. W. Kuhn, The hungarian method for the assignment problem, *Naval Research Logistics Quarterly*, 2 (1955), 83–97.

[25] T. Lacombe, M. Cuturi, and S. Oudut, Large scale computation of means and clusters for persistence diagrams using optimal transport, *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), Vol. 31, Curran Associates, Inc., 2018.

[26] N. Lahn, D. Mulchandani, and S. Raghvendra, A graph theoretic additive approximation of optimal transport, *Advances in Neural Information Processing Systems 32*, 2019, pp. 13813–13823.

[27] N. Lahn and S. Raghvendra, A faster algorithm for minimum-cost bipartite matching in minor-free graphs, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2019, pp. 569–588.

[28] N. Lahn and S. Raghvendra, A weighted approach to the maximum cardinality bipartite matching problem with applications in geometric settings, *Journal of Computational Geometry*, 11 (2021). Special Issue of Selected Papers from SoCG 2019.

[29] F. T. Leighton and P. W. Shor, Tight bounds for minimax grid matching, with applications to the average case analysis of algorithms, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, ACM, 1986, pp. 91–103.

[30] T. Lin, N. Ho, and M. I. Jordan, On efficient optimal transport: An analysis of greedy and accelerated mirror descent algorithms, *arXiv:1901.06482 [cs.DS]*, (2019).

[31] A. Madry, Navigating central path with electrical flows: From flows to matchings, and back, *54th Annual IEEE Symposium on Foundations of Computer Science*, 2013, pp. 253–262.

[32] R. Motwani, Average-case analysis of algorithms for matchings and related problems, *J. ACM*, 41 (1994), 1329–1356.

[33] M. Mucha and P. Sankowski, Maximum matchings via gaussian elimination, *45th Annual IEEE Symposium on Foundations of Computer Science*, 2004, pp. 248–255.

[34] Y. V. Prokhorov, Convergence of random processes and limit theorems in probability theory, *Teor. Veroyatnost. i Primenen.*, 1 (1956), 177–238. [English translation: Theory Probab. Appl. 1 (1956), 157-214].

[35] K. Quanrud, Approximating optimal transport with linear programs, *Symposium on Simplicity of Algorithms*, Vol. 69, 2019, pp. 6:1–6:9.

[36] R. Sandler and M. Lindenbaum, Nonnegative matrix factorization with earth mover's distance metric for image analysis, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33 (2011), 1590–1602.

[37] P. W. Shor and J. E. Yukich, Minimax Grid Matching and Empirical Measures, *The Annals of Probability*, 19 (1991), 1338 – 1348.

[38] V. Strassen, The existence of probability measures with given marginals, *The Annals of Mathematical Statistics*, 36 (1965), 423–439.

[39] J. van den Brand, Y. T. Lee, Y. P. Liu, T. Saranurak, A. Sidford, Z. Song, and D. Wang, Minimum cost flows, mdps, and ℓ_1-regression in nearly linear time for dense instances, 2021.

[40] S. Vishwanath, K. Fukumizu, S. Kuriki, and B. K. Sriperumbudur, Robust persistence diagrams using reproducing kernels, *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds.), Vol. 33, Curran Associates, Inc., 2020, pp. 21900–21911.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Any limitations of the work should be apparent from the descriptions of the relevant problem statements that have been included.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] We are not aware of any potential negative societal impacts of this work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] Our paper does not involve human subjects or sensitive data, and we are not aware of any relevant ethical concerns.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] Some proofs are included in the supplemental materials.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A] Our experiments are focused on analyzing running times of algorithms, and are not concerned with training on data sets.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] For applicable charts, we included error bars that represent a single standard deviation.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A] We do not use existing assets.
   (b) Did you mention the license of the assets? [N/A] We do not use existing assets.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We include code in the supplemental materials.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We did not use existing data.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Our only input data consists of randomly generated geometric points.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Proof of Lemma 1

During Stage 2 of LR algorithm, an augmenting path has zero slack if and only if it has minimum weight.

Proof. Let $M, y(\cdot)$ be matching and set of dual weights that are feasible. At any point during the original version of the LR algorithm, consider any augmenting path P in G_M whose first vertex is b' and whose last vertex is a'. Then, it follows that,

$$\begin{aligned}\sum_{(u,v)\in P} w(u,v) =& \sum_{(b,a)\in P\setminus M} (y(b) - y(a) + s(b,a)) + \sum_{(a,b)\in P\cap M} (y(a) - y(b)) \\ =& y(b') - y(a') + \sum_{(u,v)\in P} s(u,v) \\ =& y_{\max} + \sum_{(u,v)\in P} s(u,v).\end{aligned}$$

The last equality follows from the invariant that all free vertices of B_F have the same dual weight $y_{\max}$ and all vertices of A_F have a dual weight of 0. Since all slacks are non-negative, an augmenting path has minimum weight if and only if it has zero slack. □

B Omitted details for proof of Theorem 3

B.1 Grid generation

Next, we describe how to generate the grid $\mathbb{G}$. Without loss of generality, we assume that the points are contained inside a bounding box whose bottom left corner has coordinates $(\theta\delta, \theta\delta)$ and whose top right corner has coordinates (Δ, Δ), where Δ is of the form $t\theta\delta$ for some integer value t, i.e., when we choose $\mathbb{G}$, the number of vertical and horizontal lines will be t each.

Consider another grid $\mathbb{G}_\delta$ where each cell $C \in \mathbb{G}_\delta$ has a side-length of δ. Let $\mathbb{Y} = \{y_1, \ldots, y_t\}$ be the set of vertical lines that define $\mathbb{G}_\delta$. For $1 < i < t$, let μ_i be the number of points of P that lie between the vertical lines y_{i-1} and y_{i+1}; μ_1 and μ_t are defined to be 0. For any subset $Y \subset \mathbb{Y}$, let $\mu(Y)$ denote $\sum_{y_i \in Y} \mu_i$. We generate θ different candidates for the vertical lines of $\mathbb{G}$ as follows: For $0 \le i < \theta$, let $\mathbb{Y}_i$ be a set of vertical lines that are spaced $\theta\delta$ apart. The left-most line is given by $x = i\delta$ and $\mathbb{Y}_i = \{x = i\delta + j\theta\delta \mid 0 \le j < t - 1\}$. In other words, the set $\mathbb{Y}$ is partitioned into θ groups. We pick the set $\mathbb{Y}_i$ that minimizes $\mu(\mathbb{Y}_i)$. This can be done in $O(n \log n)$ time. A symmetric construction also applies for choosing the set of horizontal lines. $\mathbb{G}$ is the grid resulting from the selected horizontal and vertical lines.

B.2 Bound on ω

Bounding ω: We will show that $\omega = O(n^{2/3}k^{2/3})$. For any vertical line $y_i \in \mathbb{Y}$, any edge of length at most δ that crosses y_i will have its end points between y_{i-1} and y_{i+1}. Therefore, if y_i is chosen as a vertical line for $\mathbb{G}$, then μ_i will be an upper bound on the number of boundary vertices whose edges cross y_i. Thus, if $\mathbb{Y}_i$ is chosen as the set of vertical lines of $\mathbb{G}$, then $\mu(\mathbb{Y}_i)$ will be an upper bound on the number of boundary vertices created in $\mathbb{G}$ due to the vertical lines. For any point p, let y_i and y_{i+1} be the vertical lines between which p lies. Then p contributes to μ_i and μ_{i+1}. Thus, $\sum_{y_i \in \mathbb{Y}} \mu_i \le 2n$. Since each $y_j \in \mathbb{Y}$ participates in exactly one subset $\mathbb{Y}_i$, $\min_{0\le i<\theta} \mu(\mathbb{Y}_i) \le (1/\theta) \sum_{0\le i<\theta} \mu(\mathbb{Y}_i) = (1/\theta) \sum_{y_j \in \mathbb{Y}} \mu_j \le 2n/\theta$. Since $\theta = \lceil n^{1/3}/k^{2/3} \rceil$, we get a bound on the total number of boundary vertices formed due to the vertical lines to be $O(n^{2/3}k^{2/3})$. An identical argument bounds the number of boundary vertices created due to the horizontal lines to be $O(n^{2/3}k^{2/3})$ leading to $\omega = O(n^{2/3}k^{2/3})$ as desired.

C Bound on r in proof of Theorem 4

We bound the number of points of A in any piece by $O(n^{2/3}/k^{1/3})$. An identical argument also applies for B. Let $\mathbb{C} \subseteq \mathbb{G}$ be the subset of cells of $\mathbb{G}$ that contain a non-zero probability with respect

to distribution $\mathcal{P}_A$. Since the δ-density of $\mathcal{P}_A$ is k, and any cell of $\mathbb{G}$ can be covered by $O(\theta^2)$ balls of radius δ, the total probability within any cell is $O(1/(k^{1/3}n^{1/3}))$. We partition the cells in $\mathbb{C}$ into $\Theta(n^{1/3}k^{1/3})$ groups where each group has a total probability density of $\Theta(1/(n^{1/3}k^{1/3}))$. For each group of cells, we show that with high probability that it contains at most $\Theta(n^{2/3}/k^{1/3})$ points. Then, we apply union bound on the $\Theta(n^{1/3}k^{1/3})$ groups to show that no group contains more than $cn^{2/3}/k^{1/3}$ points, for some constant c.

Fix a group of cells C. Recollect that $C \subset \mathbb{C}$ and the total probability density for all cells of C combined in $\Theta(1/(n^{1/3}k^{1/3})) = c'/(n^{1/3}k^{1/3})$ for some constant $c' > 0$. Now define a random variable X to be 1 if a point p chosen from the distribution $\mathcal{P}_A$ is inside one of the cells in the group C. Otherwise, if p is not contained in any of the cells from the group C, then X is 0. Note that X is a Bernoulli random variable with $p = c'/n^{1/3}k^{1/3}$. Now, consider the n points chosen independently from $\mathcal{P}_A$. Let X_i be the outcome of the ith point. Then, $Y = \sum_{i=1}^{n} X_i$ would be the number of points of A inside any of the cells of the group C.

Since $\mathbb{E}[X_i] = c'/(k^{1/3}n^{1/3}))$ and from linearity of expectation, we conclude that $\mathbb{E}[Y] = \mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = c'n^{2/3}/k^{1/3}$. Applying Chernoff's bound,

$$\Pr[Y > 2c'n^{2/3}/k^{1/3}] \leq e^{-c'n^{2/3}/k^{1/3}}.$$

Thus, by applying union bounds on the $\Theta(n^{1/3}k^{1/3})$ groups, we get that with probability at least $1 - (\frac{n^{1/3}k^{1/3}}{e^{n^{2/3}/k^{1/3}}})$ no cell of the grid $\mathbb{G}$ contains more than $2c'n^{2/3}/k^{1/3}$ many points.

D Practical considerations

Recall that the algorithm of Section 3.2 uses two black-boxes that are impractical and have hidden high constants in the Big-O notation. (a) the algorithm relies on a dynamic nearest neighbor data structure, and, (b) the algorithm uses the selection algorithm of Katz and Sharir [21]. In order to address (a), we observe that each vertex has an expected degree of $O(k) = O(\log^{3/2}(n))$. Therefore, it is acceptable to explicitly compute the roughly $m = O(n \log^{3/2}(n))$ edges of the δ-disc graph and apply standard $O(m)$ time graph search algorithms.

We use the following approach to explicitly construct the edges of the δ-disc graph: Let $\hat{\mathbb{G}}$ be a arbitrarily-placed grid with each cell having side-length δ. For any point $a \in A$, let $N(a)$ be the set of all cells of $\hat{\mathbb{G}}$ whose boundary is within a distance of δ from a. Note that $N(a)$ contains exactly 9 cells – the cell containing a itself along with its 8 adjacent neighbors. Any point b within a distance δ of a must lie within one of the cells of $N(a)$. Therefore, it is sufficient to enumerate, for every cell $\square \in N(a)$, every point $b \in B \cap \square$, and add (a, b) to the δ-disc graph if and only if the distance between a and b is at most δ.

To implement this algorithm for edge-generation, we must be able to compute a bidirectional mapping between an input point and the grid cell that contains the point. To facilitate this mapping, we use the following approach: First, sort the points of $A \cup B$ by their x-coordinates. Each non-empty column in the grid corresponds to a contiguous interval of points in this x-sorted list. For each such column, sort the points within the corresponding x-sorted interval by their y-coordinates. Next further divide each column interval of the x-sorted list into non-empty rows by further splitting the x-sorted intervals into y-sorted intervals (within each column). Given any cell $\square$, the points contained within $\square$ can be found by first binary searching through the x-sorted intervals in order to find the list of points within the column of $\square$, and then binary searching through the y-sorted intervals of this column in order to find the corresponding interval for the row of $\square$. Thus, given the bounds of a cell $\square \in \hat{\mathbb{G}}$, the points of $(A \cup B) \cap \square$ can be found in $O(\log n)$ time. Furthermore, as these intervals are being constructed, the algorithm stores, along with each point, the boundary region of grid cell that contains that point. Thus, a point can be mapped to its containing grid cell in $O(1)$ time. Note that the set of intervals can be computed using $O(n)$ space (since empty rows and columns need not have a corresponding interval) and $O(n \log n)$ time. As a result, the total time spent constructing the δ-disc graph is $O(n \log n + \sum_{a \in A} \sum_{\square \in N(a)} |B \cap \square|)$. Since the δ-disc graph has a δ-density of $O(\log^{3/2})$ with high probability, and the cells of $N(a)$ can be covered by $O(1)$ discs of radius δ, the δ-disc graph creation algorithm runs in $\tilde{O}(n)$ time.

In order to address (b), instead of executing an integer binary search over the n^2 edges of $A \times B$, we simply execute a binary search over the interval $[\delta_{\min}, \delta_{\max}]$. Initially, we set $g_{\min} \leftarrow \delta_{\min}$ and $g_{\max} \leftarrow \delta_{\max}$. We repeat the following process until $\delta_{\max} \leq \delta_{\min} + \varepsilon$ for a sufficiently small error parameter ε. For a guess $\delta = (g_{\max} + g_{\min})/2$, we compute a maximum-cardinality matching on the δ-disc graph. If the result is a perfect matching, we set $g_{\max} \leftarrow \delta$. Otherwise, we set $g_{\min} \leftarrow \delta$. This algorithm terminates after $O(\log \frac{1}{\varepsilon})$ iterations, where each iteration executes our algorithm. Under the reasonable assumption that $\varepsilon = 1/n^{O(1)}$, the number of iterations remains only $O(\log n)$.

E Proof of Lemma 2

Recollect that for any bipartite graph $G(A \cup B, E)$ and for any set $X \subseteq A$, the neighborhood $\mathcal{N}(X)$ is the set of all vertices of B that share an edge with at least one vertex of X. The deficiency of a graph with respect to A is $\mu(A) = \max_{X \subseteq A} |X| - \mathcal{N}(X)$. Hall's theorem says that a maximum cardinality matching in a bipartite graph matches every vertex of A if and only if the deficiency of the graph with respect to A is non-positive. Using this, we will show the following lemma.

Lemma: For any bipartite graph $G(A \cup B, E)$, where $|A| = |B| = n$, and for any integer $k > 0$, the deficiency with respect to A or B is k if and only if the maximum cardinality matching is of size $n - k$.

Proof: We begin by proving (i) and (ii) and then use it to prove the lemma.

(i) Suppose the deficiency of the graph with respect to A is k, then we will show that any matching has a cardinality at most $n - k$. From the definition of deficiency, there is a subset $X \subseteq A$ such that $|X| - |\mathcal{N}(X)| = k$. In any maximum cardinality matching M, the vertices of X can only match to a vertex in the neighborhood $\mathcal{N}(X)$. Since, there are k fewer neighbors, at least k vertices in X will remain unmatched in M, i.e., $|M| \leq n - k$.

(ii) If the maximum cardinality matching M is of size $n - k$, then we will show that the deficiency of the graph is at most k. We add k dummy vertices to the set B and connect them to every vertex of A. As a result, the new graph will now match every vertex of A: every vertex of A that was unmatched with respect to M will now match to a dummy vertex of B. By Hall's theorem, the deficiency of this graph with respect to A is 0. Note, however, that the dummy vertices raised the deficiency of every subset $X \subset A$ by exactly k. Therefore, the deficiency of the graph prior to adding the dummy vertices is at most k.

Suppose the deficiency of a graph is k. From (i), the maximum cardinality matching is of size $\leq n - k$, say $n - k'$ for some $k' \geq k$. Since the maximum cardinality matching is of size $n - k'$, from (ii), we conclude that the deficiency of the graph is at most k', i.e., $k \leq k'$. Since both $k \geq k'$ and $k \leq k'$, we get $k = k'$ and the maximum cardinality matching is of size exactly $n - k$.

F Details of differences between the HK algorithm and our algorithm

Recall that the HK algorithm and our algorithm differ in the following three ways: (1) The LR algorithm assigns weights of 0 and 1 to the edges. No weights are assigned in the HK algorithm. (2) The LR has a preprocessing step that computes the maximum cardinality matching for each piece. (3) In Stage 1, the LR algorithm executes a $0/1$-BFS instead of the BFS executed by HK algorithm. Next, we explain why these three differences do not have any significant effect on the comparative running times of the two algorithms. In (1), for any edge (u, v), one can determine whether (u, v) is in E_S in $O(1)$ time. In (2), the LR algorithm executes the HK algorithm for each piece each of which contains at most r vertices. The time taken across all pieces is $\sum_{i=1}^{t} O(|\mathcal{E}_i|\sqrt{|\mathcal{V}_i|}) = O(m\sqrt{r})$. In (3), the 0/1 BFS executed in Stage 1 of the algorithm is almost identical to the BFS executed by the HK algorithm, except that it prioritizes weight 0 edges before the weight 1 edges. It can be implemented to take $O(m)$ time.

G Additional experimental results

Recall that there are four main differences between the HK algorithm and our algorithm. Our algorithm must incorporate weights of 0 and 1 into the edges. For arbitrary inputs, the δ-disc graph

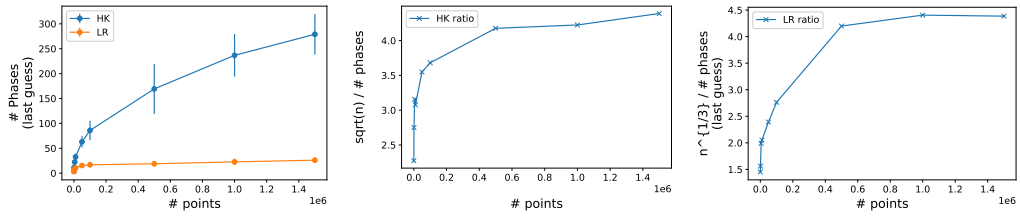

Figure 3: Comparisons of number of phases for the final bottleneck guess. Left: A comparison of the number of phases for the HK algorithm and our algorithm. Middle: $\sqrt{n}$ divided by the number of phases executed by the HK algorithm. Right: $n^{1/3}$ divided by the number of phases of our algorithm. Note that the first two plots also appear in Figure 2.

matching algorithm tries several shifts of a grid in order to minimize the value of ω. However, in practice, taking a single random shift of the grid is sufficient for finding a small number of boundary vertices in expectation. Recall that, given this shift, the corresponding δ-disc graph can be created in $\tilde{O}(n)$ time, which is asymptotically far less than the overall complexity of our algorithm. An additional difference between the HK algorithm and our algorithm is that the our algorithm begins with a preprocessing step, which executes the HK algorithm within each piece. However, this step takes asymptotically less time than the subsequent phases of the algorithm (see Figure 4). Additionally, our experiments found that the total time taken by the edge weight assignment and preprocessing step was insignificant in comparison to the total time taken by our algorithm. The third difference between the HK algorithm and our algorithm is the fact that the our algorithm executes a $0/1$ BFS, which is slightly different from the version of BFS used by the HK algorithm. Within each phase, the $0/1$ BFS can be implemented to run in $O(m)$ time without any significant difference in performance in comparison to the HK algorithm's BFS.

The fourth and final difference between the two algorithms is that our algorithm finds augmenting paths during Stage 2 that are not necessary vertex-disjoint. While the first three differences do not have any significant direct impact on the running time, this final difference is of primary importance. There are two main effects of finding non-disjoint augmenting paths. First, more augmenting paths can be found during each phase, which decreases the number of phases that need to be run, decreasing the relative running time of our algorithm. However, a second effect is that edges may be revisited multiple times during Stage 2 of the same phase of our algorithm, increasing the relative running time of our algorithm.

Next, we present results that illustrate this dynamic in practice. First, we find that our algorithm executes significantly fewer phases than the HK algorithm. Also, as the guess of the bottleneck distance approaches the actual bottleneck distance, both algorithms seems to exhibit close to their worst case theoretical upper bounds of $O(\sqrt{n})$ and $O(n^{1/3})$ respectively for their number of phases (see Figure 3).

Finally, to better understand the running time of our algorithm, we divide the edge visits into three groups - the edges visits that occur during the preprocessing step, the edge visits where an edge is visited for the first time during a phase, and the times an edge is revisited during Stage 2 of a particular phase. These results are given in Figure 4. Note that, from these results, we can conclude that Stage 2 accounts for the majority of the running time, and most of the time taken during Stage 2 is due to revisits of edges.

H Details of dynamic nearest neighbor data structures

In this section, we provide additional details as to how the HK algorithm and our algorithm can be implemented using dynamic nearest neighbor (DNN) data structures. Let $G(V = A \cup B, E)$ be an arbitrary undirected graph where every edge $(u, v) \in E$ has a weight $w(u, v)$ that is either 0 or 1. A DNN structure $\mathcal{D}$ on the graph G stores a subset $A' \subseteq A$ of vertices. Given an arbitrary query vertex $b \in B$, the data structure $\mathcal{D}$ returns a vertex $a \in A'$ that minimizes the weight of the edge (b, a) (i.e., returning weight 0 edges before weight 1 edges). If there is no edge directed from b to any $a \in A'$, then the query returns NULL. The DNN $\mathcal{D}$ supports queries in $\Phi(n)$ time. Furthermore, $\mathcal{D}$ allows an

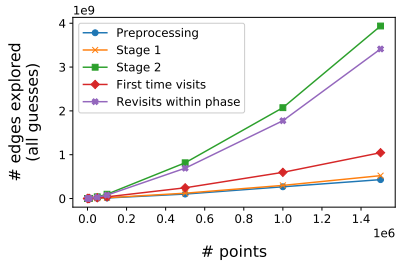

Figure 4: A breakdown of the edge visit counts of our algorithm into categories. First time visits occur whenever an edge is visited for the first time during a particular stage (either Stage 1 or Stage 2) of a phase; all subsequent visits to that edge during the same stage are revisits. Revisits only occur during Stage 2.

arbitrary vertex $a \in A \setminus A'$ to be inserted into A' in $\Phi(n)$ time, and an arbitrary vertex $a \in A'$ to be removed from A' in $\Phi(n)$ time.

In section H.1, we explain how the HK algorithm can be implemented, using a DNN $\mathcal{D}$, to run in $O(n^{3/2}\Phi(n))$ time. In section H.2, we explain how our algorithm can be implemented to run in $O(\Phi(n)(n\sqrt{r} + n\sqrt{\omega} + r\omega \log n))$ time using a DNN $\mathcal{D}$. In section H.3, we explain how such a DNN can be implemented for δ-disc graphs.

H.1 HK algorithm using DNN

In this section, we describe how the HK algorithm can be implemented to run in $O(n^{3/2}\Phi(n))$ time using a DNN data structure $\mathcal{D}$. Note that the HK algorithm does not use the edge weights $w(\cdot, \cdot)$; instead, queries to $\mathcal{D}$ are only necessary in order to identify *an arbitrary* edge of E from a query vertex $b \in B$ to a vertex $a \in A'$.

Stage 1: Next, we describe how the HK algorithm can use $\mathcal{D}$ to support BFS on the residual graph G_M in $O(n\Phi(n))$ time. The purpose of this BFS is to identify, for any vertex $v \in A \cup B$, the minimum distance ℓ_v (in terms of total number of edges) from s to v in G_M. During the BFS, let S be the set of vertices in G_M that have been explored by the BFS, and let $T = V \setminus S$ be the set of vertices that have not yet been reached by the BFS. Throughout the algorithm, the set A' maintained by the DNN data structure $\mathcal{D}$ will be equal to $A \cap T$. Initially, $S = \{s\}$, $T = A \cup B$, and $A' = A$.

Recall that a standard BFS can be implemented using a queue Q that contains vertices of V. In $O(1)$ time, a single vertex can be added to the tail of Q or removed from the head of Q. The queue Q contains vertices that are *partially explored* and any vertex that is removed from Q is *fully explored*. During each iteration of the BFS, let v be the vertex at the head of Q. If $v \in A$, then there is at most one matching edge (v, b) directed from v to some $b \in B$. If $b \in T$, then the algorithm sets $\ell_b \leftarrow \ell_v + 1$, removes b from T, adds b to S, and adds b to Q. Otherwise, if $v \in B$, then there could be many edges outgoing from v in G_M. We are only interested in edges directed from v to some $a \in A \cap T$. Conveniently, such a vertex a can be identified in $O(\Phi(n))$ time by querying $\mathcal{D}$. Given such a vertex a, the algorithm sets $\ell_a \leftarrow \ell_v + 1$, removes a from T, removes a from A', adds a to S, and adds a to Q. If the query to $\mathcal{D}$ on v returns NULL, then all applicable neighbors of v have been explored, and the algorithm removes v from Q.

Stage 2: Recall that Stage 2 of the HK algorithm conducts multiple DFS searches in order to find augmenting paths in the admissible graph $\mathcal{A}$, which contains all edges (u, v) in G_M with $\ell_v = \ell_u + 1$. It may be prohibitively expensive to construct $\mathcal{A}$ explicitly, as it could have $\Theta(m)$ edges. Instead, the algorithm will make use of multiple DNN data structures in order to efficiently identify the next edge to explore during a DFS. Recall that $\ell = \min_{a \in A_F}$ is the minimum distance from s to any free vertex of A_F. For each even-valued $i \in [1, \ell]$ the algorithm maintains a DNN data structure $\mathcal{D}_i$. Each such data structure $\mathcal{D}_i$ contains a set $A'_i \subseteq A$, which initially contains all vertices $a \in A$ with $\ell_a = i$. During Stage 2, if a vertex $a \in A$ is reached during the DFS, the algorithm removes a from A'_{ℓ_a}. When the DFS is exploring neighbors of some $b \in B$, it will query $\mathcal{D}_{\ell_b+1}$ on b. If the query returns a

vertex $a \in A'_i$, then (b, a) is an admissible edge, and a has not previously been reached by a DFS. The algorithm removes a from A'_{ℓ_a} and continues the DFS from a. If the query to $\mathcal{D}_{\ell_b+1}$ returns NULL, then no unexplored neighbors of b remain, and the DFS backtracks from b. Note that, like during the BFS, any vertex $a \in A$ has at most one outgoing edge, which is a matching edge. As a result, the neighbors of any vertex $a \in A$ can be considered explicitly, without need of a DNN data structure.

Efficiency: To bound the running time of BFS using $\mathcal{D}$, observe that each vertex of A is removed from A' at most once. Furthermore, whenever a query is made to $\mathcal{D}$, either it returns NULL (which only occurs at most once per vertex of B) or else a vertex is removed from A'. Therefore, just $O(n)$ operations are required on $\mathcal{D}$. Similarly, during Stage 2, each vertex of A exists in at most one set A'_i, and is removed from this set at most once. Therefore, Stage 2 also requires only $O(n)$ operations on DNN data structures, each of which take $\Phi(n)$ time. As a result, a single phase of the HK algorithm can be implemented to take $O(n\Phi(n))$ time. Over all $O(\sqrt{n})$ phases of the algorithm, the total time taken is $O(n^{3/2}\Phi(n))$.

H.2 Our algorithm using DNN

In this section, given a DNN data structure on G, we explain how our algorithm can be made to run in $O(\Phi(n)(n\sqrt{r} + n\sqrt{\omega} + r\omega \log n))$ time. Most of the details are similar to the description of the HK algorithm using a DNN data structure, so we will focus on highlighting the key differences.

Stage 1: For Stage 1, there is one key difference between the HK algorithm and our algorithm. When selecting the next edge directed from S to T to explore, the HK algorithm ignores weights while our algorithm must select an edge with weight 0 if one is present. In order to support this in our algorithm, one option is to maintain two queues Q_0 and Q_1, each of which contain vertices of $A \cup B$. During each iteration of the BFS, the algorithm identifies an edge from S to T by checking Q_0 first, and only checking Q_1 if Q_0 is empty. If Q_0 is not empty, let $u \in S$ be the head of Q_0. If $u \in A$, then the neighbors of u can be processed without using the DNN structure, so we consider when $u \in B$. Then an edge of weight 0 directed from $u \in S \cap B$ to $v \in T \cap A$ can be identified, if such an edge exists, by querying $\mathcal{D}$. If the result is an edge with weight 0, then the algorithm sets $\ell_v \leftarrow \ell_u$, adds v to Q_0, adds v to S, and removes v from T. Otherwise, u is removed from Q_0 and added to Q_1, and the algorithm continues to the next iteration. If Q_0 is empty, then the vertex at the head of Q_1 is processed instead. In a similar fashion, an edge of weight 1 directed from $u \in S \cap B$ to some $v \in T \cap A$ can be identified, if such an edge exists, by querying $\mathcal{D}$. If the result is an edge with weight 1, then the algorithm sets $\ell_v \leftarrow \ell_u + 1$, adds v to S, and removes v from T. Otherwise, the algorithm removes u from Q_1 and continues to the next iteration.

Stage 2: The second stage of our algorithm has two key differences from the second stage of the HK algorithm that affect the use of the DNN structures: (i) Our algorithm can contain edges of weight 0 in the admissible graph $\mathcal{A}$. (ii) Unlike the HK algorithm, which deletes all visited vertices at the end of a single DFS, our algorithm only deletes the edges that did not participate in an affected piece.

To address (i), instead of making a data structure $\mathcal{D}_i$ for only the even values of i, our algorithm creates a DNN data structure $\mathcal{D}_i$ for every $i \in [1, \ell]$. When exploring a vertex $b \in B$ during the DFS, in order to identify an admissible edge outgoing from b, it is sufficient to query $\mathcal{D}_{\ell_b}$ as well as $\mathcal{D}_{\ell_b+1}$. If the query to $\mathcal{D}_{\ell_b}$ returns either NULL or a vertex a with $w(b, a) = 1$ then there are no admissible edges from b to a vertex of $\mathcal{D}_{\ell_b}$. Furthermore, if the query to $\mathcal{D}_{\ell_b+1}$ returns a vertex a, then $w(b, a)$ must be 1; otherwise, if $w(b, a) = 0$, then a could have been reached via (b, a) during Stage 1, and should have been included in $\mathcal{D}_{\ell_b}$ instead of $\mathcal{D}_{\ell_b+1}$.

For addressing (ii), first note that, if an edge directed from u to v is deleted during our algorithm (considering momentarily the version of our algorithm that does not use DNN structures), then u has been visited by the previous DFS, but the previous DFS backtracked from u; otherwise, u would participate in the augmenting path found by the DFS, contradicting the assumption that (u, v) was deleted. Since the previous DFS backtracked from u, all remaining admissible outgoing edges from u to a vertex of T have been explored. Furthermore, since u is not in an affected piece, no edge incident on u is in an affected piece. Therefore, all admissible edges outgoing from u are deleted, implying that u can be deleted as well. We conclude that, instead of deleting all the visited edges that do not participate in affected pieces after a DFS, it is permissible to delete all of the visited *vertices*

that do not participate in affected pieces. This observation is applicable to an implementation of the our algorithm with or without the use of DNN structures. Note, however, that there could be some edges with weight 1 that were explored during the DFS for which both endpoints were in affected pieces. Since edges with weight 1 are not inside affected pieces, such edges should be deleted. In order to avoid reexploring these edges, whenever the algorithm backtracks from a vertex $b \in B$, that vertex should, for the remainder of Stage 2, only query $\mathcal{D}_{\ell_b}$; it should no longer use $\mathcal{D}_{\ell_b+1}$.

Given this information, we can now address (ii). Recall that, whenever the DFS reaches a vertex $a \in A$, that vertex is removed from A'_{ℓ_a}. For the purposes of querying the DNN, this corresponds to deleting the vertex a. By the end of the DFS, any vertex of A that was explored has been removed from its corresponding DNN structure. Thus, in order allow vertices in affected pieces to be explored again, it suffices to re-insert the deleted vertices from affected pieces back into their corresponding DNN structures. Specifically, as the DFS progresses, the algorithm maintains a set $\mathcal{X}$ of vertices of A that have been reached. Whenever the DFS finishes, the algorithm will, for every $a \in \mathcal{X}$, insert a back into A'_{ℓ_a} if a belongs to an affected piece. All other vertices of $\mathcal{X}$ are deleted for the remainder of the phase.

Efficiency: For Stage 1, each vertex of A is removed from A' at most once, and each query to $\mathcal{D}$ results in either (i) the deletion of a vertex from A', (ii) the removal of a vertex from Q_0, or (iii) the removal of a vertex from Q_1. Sine each vertex is inserted into Q_0 and Q_1 at most once each, the number of operations on $\mathcal{D}$ is $O(n)$.

Next, we bound the time taken by Stage 2. First note that, when identifying an admissible outgoing edge from a vertex $b \in B$, our algorithm makes two queries, one to $\mathcal{D}_{\ell_b}$ and one to $\mathcal{D}_{\ell_b+1}$, as opposed to the HK algorithm, which makes a single query. This difference does not asymptotically affect the running time. Each such pair of queries causes at least one of the following to happen: (i) A vertex is removed from $\mathcal{D}_{\ell_b}$, (ii) a vertex is removed from $\mathcal{D}_{\ell_b+1}$, or (iii) the DFS backtracks from the vertex b. Also observe that the number of deletions from each of the DNN structures is upper bounded by its total number of insertions.

During the second stage of a single phase, each operation on the DNN structures can be attributed to a single vertex's visitation, where each vertex visitation is responsible for $O(1)$ operations. The total number of vertex visits during Stage 2 of a single phase is $O(n)$ plus the total number of revisits. Recall that Lahn and Raghvendra bound the total number of affected pieces as $\omega \log \omega$ [28]. Whenever a piece is an affected piece with respect to an augmenting path, at most $O(r)$ vertices are revisited during that phase. Therefore, the total number of revisits during the entire algorithm is $O(\omega r \log \omega)$. With each vertex visit taking $\Phi(n)$ time, the total time taken by all $O(\sqrt{\omega})$ phases of our algorithm is $O(\Phi(n)(n\sqrt{\omega} + r\omega \log n))$. However, we must also consider the time taken by the preprocessing step. The preprocessing step can apply the HK algorithm, using the DNN structures as described in Section H.1. Since each piece has $O(r)$ vertices, the time taken by each piece is $O(r^{3/2}\Phi(r))$. Over all pieces, the total time taken for preprocessing is $O(n\sqrt{r}\Phi(n))$. Combining the time taken for the preprocessing and the phases gives a total running time of $O(\Phi(n)(n\sqrt{r} + n\sqrt{\omega} + r\omega \log n))$.

H.3 DNN for δ-disc graphs

The algorithms of sections H.1 and H.2 assume the existence of a dynamic nearest neighbor data structure for graphs with edge weights of either 0 or 1 that maintains a set $A' \subseteq A$ and, for any query vertex $b \in B$, returns a vertex $a \in A'$ that minimizes $w(b, a)$, or else NULL if no edge exists between b and any vertex of A'. In this section, we describe how to support such a data structure in the context of δ-disc matching, considering point sets $A, B \subset \mathbb{R}^2$. It is known how to support a similar dynamic nearest neighbor structure when every edge $(a, b) \in A \times B$ exists and has a weight equal to the Euclidean distance between its endpoints. This *Euclidean* DNN supports insertions, deletions, and queries in $\Phi'(n) = \log^{O(1)}(n)$ time. We describe how this Euclidean DNN can be used to support a $0/1$ edge-weighted DNN in the context of the δ-disc matching algorithm described in Section 3.1.

Recall that, in the δ-disc graph matching algorithm, any edge whose endpoints are in the same piece are assigned a weight of 0 and all other edges, whose endpoints lie in different pieces, are assigned a weight of 1. To implement the operations of $\mathcal{D}$, we construct a *global* Euclidean DNN structure $\mathcal{D}'$ on the vertices of the entire graph, as well as a *local* Euclidean DNN structure $\mathcal{D}'_j$ on the vertices within each piece $\mathcal{P}_j, 1 \leq j \leq t$. The global data structure $\mathcal{D}'$ maintains the same subset A' as $\mathcal{D}$, and

the data structure $\mathcal{D}'_j$ for each piece $\mathcal{P}_j$ maintains the subset $A' \cap \mathcal{V}_j$, supporting queries on vertices $B \cap \mathcal{V}_j$ within the piece.

Given this setup, $\mathcal{D}$ supports queries on a vertex $b \in B$ with respect to the set A' as follows: Let $\mathcal{P}_j$ be the piece that contains b. First, the algorithm will query $\mathcal{D}'_j$ in order to obtain the point $a \in A' \cap \mathcal{V}_j$ that is closest to b within the piece. If the resulting Euclidean distance $d(a, b)$ between a and b is at most δ, then (a, b) exists in the δ-disc graph, and has a weight $w(a, b) = 0$. The data structure returns a. Otherwise, there is no point $a \in A' \cap \mathcal{V}_j$ that is within a distance of δ from b, and we can conclude that there are no weight 0 edges incident on b with respect to A'. In this case, the algorithm proceeds to query the global data structure $\mathcal{D}'$ in order to obtain a weight 1 edge incident on b. If $\mathcal{D}'$ returns a point $a \in A'$ that is within a distance δ from the query b, then $w(a, b) = 1$, and $\mathcal{D}$ returns a. Otherwise, there are no edges incident on B with respect to A', and $\mathcal{D}$ returns NULL. To insert a point $a \in \mathcal{V}_j$ into the set A', the data structure $\mathcal{D}$ can simply insert a into both $\mathcal{D}'$ and $\mathcal{D}'_j$. Similarly, a can be removed from A' by removing a from both $\mathcal{D}'$ and $\mathcal{D}'_j$. Since each query, insertion, and deletion operation on $\mathcal{D}$ requires only $O(1)$ operations on corresponding Euclidean DNN structures, each operation on $\mathcal{D}$ can be supported in $\Phi(n) = O(\Phi'(n)) = \log^{O(1)}(n)$ time.