An $\tilde{O}(n^{5/4})$ Time ε-Approximation Algorithm for RMS Matching in a Plane

Nathaniel Lahn* Sharath Raghvendra†

Abstract

The 2-Wasserstein distance (or RMS distance) is a useful measure of similarity between probability distributions with exciting applications in machine learning. For discrete distributions, the problem of computing this distance can be expressed in terms of finding a minimum-cost perfect matching on a complete bipartite graph given by two multisets of points $A, B \subset \mathbb{R}^2$, with $|A| = |B| = n$, where the ground distance between any two points is the squared Euclidean distance between them. Although there is a near-linear time relative ε-approximation algorithm for the case where the ground distance is Euclidean (Sharathkumar and Agarwal, JACM 2020), all existing relative ε-approximation algorithms for the RMS distance take $\Omega(n^{3/2})$ time. This is primarily because, unlike Euclidean distance, squared Euclidean distance is not a metric. In this paper, for the RMS distance, we present a new ε-approximation algorithm that runs in $\mathcal{O}(n^{5/4}\text{poly}\{\log n, 1/\varepsilon\})$ time. Our algorithm is inspired by a recent approach for finding a minimum-cost perfect matching in bipartite planar graphs (Asathulla et al., TALG 2020). Their algorithm depends heavily on the existence of sublinear sized vertex separators as well as shortest path data structures that require planarity. Surprisingly, we are able to design a similar algorithm for a complete geometric graph that is far from planar and does not have any vertex separators. Central components of our algorithm include a quadtree-based distance that approximates the squared Euclidean distance and a data structure that supports both Hungarian search and augmentation in sublinear time.

1 Introduction

Given two sets A and B of n points in $\mathbb{R}^2$, let $\mathcal{G}(A \cup B, A \times B)$ be the complete bipartite graph on A, B. A matching M is a set of vertex-disjoint edges of $\mathcal{G}$. The matching M is *perfect* if it has cardinality n. For any $p \geq 1$, the cost of an edge (a, b) is simply $\|a - b\|^p$; here, $\|a - b\|$ is the Euclidean distance between a and b. Consider the problem of computing a perfect matching M that minimizes the sum of all its edges' costs, i.e., the matching with smallest $w_p(M) = \sum_{(a,b)\in M} \|a - b\|^p$. When $p = 1$, this problem is the well-known *Euclidean bipartite matching problem*. When $p = 2$, the matching computed minimizes the sum of the squared Euclidean distances of its edges and is referred to as the *RMS matching*. For $p = \infty$, the matching computed will minimize the largest cost edge and is referred to as the *Euclidean bottleneck matching*. For a parameter $\varepsilon > 0$ and $p \geq 1$, we say that the matching M is an ε-approximate matching if $w_p(M) \leq (1 + \varepsilon)w_p(M_{OPT})$ where M_{OPT} is a perfect matching with the smallest cost. In this paper, we consider the problem of computing an ε-approximate RMS matching in the plane and present a randomized $\tilde{\mathcal{O}}(n^{5/4})$ time[1] algorithm. For the remainder of the paper, we assume that $w(M) = w_2(M)$.

When A and B are multi-sets, the cost of the RMS matching is also known as the 2-Wasserstein distance – a popular measure of similarity between two discrete distributions. Wasserstein distances are very popular in machine learning applications. For instance, 2-Wasserstein distance has been used as a similarity metric for images using color distributions [25]. A 2-dimensional grayscale image can be represented as a discrete distribution on 2-dimensional points, and Wasserstein distance can be used to compare the similarity between such distributions in a fashion similar to [3, 6, 8]. The 2-Wasserstein distance has also been used for 2-dimensional shape reconstruction [7].

Wasserstein distance is also used as an objective function for generative adversarial neural networks (GANs). GANs are used to generate fake objects, such as images, that look realistic [4, 13, 22]. Here, we have a 'real' distribution $\mathcal{R}$ and a 'fake' distribution $\mathcal{F}$. Sampling m images from both $\mathcal{F}$ and $\mathcal{R}$ and computing the Wasserstein distance between the two samples gives a measure of how good the fake image generator imitates real data. The matchings (or maps) corresponding to

*School of Computing and Information Sciences, Radford University. Email: nlahn@radford.edu. This research was done when the author was a student at Virginia Tech.

†Department of Computer Science, Virginia Tech. Email: sharathr@vt.edu This research is supported by NSF Grant CCF-1909171

[1] We use $\tilde{\mathcal{O}}(\cdot)$ to hide poly$\{\log n, 1/\varepsilon\}$ factors in the complexity.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

the 2-Wasserstein distance are also attractive because they permit a unique interpolation between the distributions; see for instance [30].

Previous Results: For any weighted bipartite graph with m edges and n vertices, the fundamental Hungarian algorithm can be used to find a minimum-cost maximum-cardinality matching in $\mathcal{O}(mn+n^2 \log n)$ time [17].[2] When edge costs are positive integers upper-bounded by a value C, the algorithm given by Gabow and Tarjan computes a minimum-cost maximum cardinality matching in $\mathcal{O}(m\sqrt{n}\log(nC))$ time [12]. The Gabow-Tarjan algorithm executes $\mathcal{O}(\sqrt{n})$ phases where each phase executes an $\mathcal{O}(m)$ time search on a graph to compute a set of augmenting paths.

In geometric settings, one can use a dynamic weighted nearest neighbor data structure to efficiently execute the search for an augmenting path in $\tilde{\mathcal{O}}(n)$ time. Consequently, there are many $\tilde{\mathcal{O}}(n^{3/2})$ time exact and approximation algorithms for computing matchings in geometric settings [9, 23, 27, 31], including the RMS matching. Improving upon the $\Omega(n^{3/2})$ bound for exact and approximation algorithms remains a major open question in computational geometry. There are no known exact geometric matching algorithms for 2-dimensions or beyond that break the $\Omega(n^{3/2})$ barrier. However, there has been some progress for approximation algorithms for $p = 1$, which we summarize next.

For the Euclidean bipartite matching problem, Agarwal and Varadarajan [1] gave an $\mathcal{O}(\log(1/\varepsilon))$ approximation algorithm that executes in $\mathcal{O}(n^{1+\varepsilon})$ time. Indyk [15] extended this approach to obtain a constant approximation algorithm that runs in near-linear time. Sharathkumar and Agarwal [26] presented a near-linear time ε-approximation algorithm for the Euclidean bipartite matching problem. Each of these algorithms rely on approximating the Euclidean distance by using a "randomly shifted" quadtree. Extending this to $p > 1$ seems very challenging since the expected error introduced by the randomness grows very rapidly when $p = 2$ and beyond.

When the costs satisfy metric properties, the uncapacitated minimum-cost flow between multiple sources and sinks is the same as minimum-cost matching problem. Using a generalized preconditioning framework, Sherman [29] provided an $\mathcal{O}(m^{1+o(1)})$ time approximation algorithm to compute the uncapacitated minimum-cost flow in any weighted graph G with m edges and n vertices, where the cost between any two vertices is the shortest path cost between them in G. Using this algorithm, one can use Euclidean spanners of small size to obtain an $\mathcal{O}(n^{1+o(1)})$ time algorithm that ε-approximates the Euclidean bipartite matching cost. Khesin *et al.* [16] provided a more problem-specific preconditioning algorithm that returns an ε-approximate Euclidean bipartite matching with an improved execution time of $\tilde{\mathcal{O}}(n)$; see also [11]. Unlike with Euclidean costs, squared Euclidean costs do not satisfy triangle inequality, and the reduction to uncapacitated minimum-cost flow as well as the use of spanners does not apply. Therefore, these previous techniques seem to have limited applicability in the context of RMS matching.

Recently, Asathulla *et al.* [5] as well as Lahn and Raghvendra [19, 20] presented algorithms that exploit sublinear sized graph separators to obtain faster algorithms for minimum-cost matching as well as maximum cardinality matching on bipartite graphs. For instance, for any bipartite graph with m edges and n vertices and with a balanced vertex separator of size n^δ, for $1/2 \le \delta < 1$, Lahn and Raghvendra [20] presented a $\tilde{\mathcal{O}}(mn^{\delta/(1+\delta)})$ time algorithm to compute a maximum cardinality matching. The ε-approximate bottleneck matching problem can be reduced to finding a maximum cardinality matching in a grid-based graph. Using the fact that a d-dimensional grid has a balanced, efficiently computable vertex separator of size $\mathcal{O}(n^{1-1/d})$, they obtain an $\tilde{\mathcal{O}}(n^{1+\frac{d-1}{2d-1}})$ time algorithm to compute an ε-approximate bottleneck matching of two sets of d dimensional points.

Given the wide applicability of Wasserstein distances, machine learning researchers have designed algorithms that compute an approximate matching within an additive error of εn. Some of these algorithms run in $\tilde{\mathcal{O}}(n^2 C/\varepsilon)$ for arbitrary costs [18, 24]; recollect that C is the diameter of the input point set. For 2-Wasserstein distance, such a matching can be computed in time that is near-linear in n and C/ε [2]. Some of the exact and relative approximation algorithms [28] have informed the design of fast methods for machine learning applications [18].

Our Results: Our main result is the following.

THEOREM 1.1. *For any point sets $A, B \subset \mathbb{R}^2$, with $|A| = |B| = n$, and for any parameter $0 < \varepsilon \le 1$, an ε-approximate RMS matching can be computed in $\mathcal{O}(n^{5/4}\text{poly}\{\log n, 1/\varepsilon\})$ time with high probability.*

All previous algorithms that compute an ε-approximate RMS matching take $\Omega(n^{3/2})$ time [23, 28]. We would like to note that the algorithm in [28] computes an ε-approximate RMS matching for two dimensions in $\mathcal{O}(n^{3/2}\text{poly}\{\log n, \log(1/\varepsilon)\})$ time. Our algorithm outperforms the algorithm in [28] except when ε is extremely small, i.e., $\varepsilon = 1/n^{\Omega(1)}$.

Basics of Matching: Given a matching M, an *alternating path* is a path whose edges alternate between

[2]Note that $m = \mathcal{O}(n^2)$ in our setting.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

edges of M and edges not in M. A vertex is *free* if it is not matched in M. An *augmenting path* is an alternating path that begins and ends at a free vertex. Given an augmenting path P, it is possible to obtain a new matching $M' \leftarrow M \oplus P$ of one higher cardinality by *augmenting* along P. A standard technique for finding augmenting paths is to form a so-called *residual graph* by directing the edges of the original graph based on whether they are in the matching or not. Any path in this directed residual graph that begins and ends at a free vertex forms an augmenting path.

Standard algorithms for minimum-cost bipartite matching use a *primal-dual* approach where in addition to a matching M, the algorithm also maintains a set of *dual weights* $y(\cdot)$ on the vertices. A matching M along with a set of dual weights $y(\cdot)$ is *feasible* if, for every edge (a, b), in the input graph:

$$y(a) + y(b) \leq c(a, b).$$
$$y(a) + y(b) = c(a, b) \quad \text{if } (a, b) \in M.$$

Here, $c(a, b)$ is the cost of the edge (a, b). It can be shown that any feasible perfect matching is also a minimum-cost perfect matching.

The *slack* of any edge with respect to these feasibility conditions is given by $s(a, b) = c(a, b) - y(a) - y(b)$. A set of edges is *admissible* if it has a slack of zero. The fundamental Hungarian algorithm [17] computes a minimum-cost matching by iteratively adjusting the dual weights and finding an augmenting path P containing zero-slack edges. Augmenting along this admissible path does not violate feasibility. As a result, the Hungarian algorithm executes n iterations.

2 Overview of our Approach

Our algorithm draws insight from a recent $\tilde{\mathcal{O}}(n^{4/3})$ time algorithm for computing a minimum-cost perfect matching in bipartite planar graphs [5]. The algorithm of [5] relies on the existence of a planar vertex separator of size $\mathcal{O}(\sqrt{n})$. A complete bipartite graph is far from planar and does not have any vertex separators. Despite this, we are able to adapt the approach of [5] to our setting. We begin with a summary of their algorithm.

Planar Bipartite Matching Algorithm: The algorithm of [5] is a primal-dual algorithm that iteratively adjusts the dual weights of the vertices to find an augmenting path containing zero 'slack' edges and then augments the matching along this path. For a parameter $r > 0$, their algorithm conducts an $\mathcal{O}(n\sqrt{r})$ time pre-processing step and computes a matching of size $n - \mathcal{O}(n/\sqrt{r})$. After this, their algorithm finds the remaining augmenting paths in sublinear time by the use of an r-division: An r-division divides any planar graph into $\mathcal{O}(n/r)$ edge-disjoint pieces, each of size $\mathcal{O}(r)$, with only $\mathcal{O}(n/\sqrt{r})$ many *boundary vertices* that are shared between pieces. The algorithm then conducts a search for each augmenting path as follows:

- Using an r-division of a planar bipartite graph $G(A \cup B, E)$, the algorithm constructs a compact residual graph $\tilde{G}$ with a set $\tilde{V}$ of $\mathcal{O}(n/\sqrt{r})$ vertices – each boundary vertex of the r-division is explicitly added to this vertex set. In addition, the compact graph has $\mathcal{O}(r)$ edges per piece and $\mathcal{O}(n)$ edges in total. The algorithm assigns a dual weight for every vertex of $\tilde{V}$ that satisfies a set of dual feasibility constraints on the edges of $\tilde{G}$. Interestingly, given dual weights on $\tilde{V}$ that satisfy the *compressed feasibility* conditions, one can derive dual weights for $A \cup B$ satisfying the classical dual feasibility conditions, and vice versa. Therefore, instead of conducting a search on G, their algorithm searches for an augmenting path in the compact residual graph $\tilde{G}$.

- Their algorithm builds, for each piece of G, a data structure in $\tilde{\mathcal{O}}(r)$ time (see [10]). This data structure stores the $\mathcal{O}(r)$ edges of $\tilde{G}$ belonging to the piece and using this data structure, the algorithm conducts a primal-dual search for an augmenting path in $\tilde{\mathcal{O}}(|\tilde{V}|) = \tilde{\mathcal{O}}(n/\sqrt{r})$ time. Over $\mathcal{O}(n/\sqrt{r})$ augmenting path searches, the total time taken is bounded by $\tilde{\mathcal{O}}(n^2/r)$.

Augmenting along a path reverses the direction of its edges in the residual graph. Therefore, their algorithm has to re-build the shortest path data structure for every *affected piece*, a piece containing at least one edge of the augmenting path. This can be done in $\tilde{\mathcal{O}}(r)$ time per piece. In order to reduce the number of affected pieces, an additive cost of $\sqrt{r}$ is introduced to every edge incident on the boundary vertices. It is then shown that the total additive cost across all augmenting paths found by the algorithm cannot exceed $\mathcal{O}(n \log n)$, implying that the number of affected pieces is at most $\mathcal{O}((n/\sqrt{r}) \log n)$. The time taken to re-build the data structure for the affected pieces is $\tilde{\mathcal{O}}((n/\sqrt{r}) \log n) \times \tilde{\mathcal{O}}(r) = \tilde{\mathcal{O}}(n\sqrt{r})$. By choosing $r = n^{2/3}$, they balance the search time with the re-build time, leading to an $\tilde{\mathcal{O}}(n^{4/3})$ time algorithm.

The successful application of a compact residual network as well as the additive cost of $\sqrt{r}$ on the edges relies on the existence of an r-division in planar graphs. In order to extend these techniques to the geometric setting, we build upon ideas from another matching algorithm, which produces an ε-approximation for the Euclidean bipartite matching problem [26]. We give a brief overview of this algorithm next.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Approximate Euclidean Matching: The algorithm of [26] introduces an ε-approximation of the Euclidean distance based on a quad-tree Q. The input is transformed so that the optimal matching cost is $\mathcal{O}(n/\varepsilon)$ and the height of the quad-tree Q is $\mathcal{O}(\log n)$. Any edge of the complete bipartite graph *appears* at the least common ancestor of its endpoints in Q. The set of edges appearing within each quadtree square is then partitioned into poly$\{\log n, 1/\varepsilon\}$ many *bundles* and all edges within the same bundle are assigned the same cost. This assigned cost is an upper bound on the actual Euclidean cost. Furthermore, the authors show that, if the quad-tree is randomly shifted, the expected cost assigned to any edge is at most $(1+\varepsilon)$ times the Euclidean distance. Using this, the authors switch to computing a matching with respect to this new quad-tree distance.

Using the edge bundles and certain carefully precomputed shortest paths in the residual graph, the algorithm of [26] stores a poly$\{\log n, 1/\varepsilon\}$ size *associated graph* at each square of the quad-tree. Their algorithm iteratively finds a minimum-cost augmenting path P. Note that this is not done by using a primal-dual method, but by executing a Bellman-Ford search on the associated graph of each square that contains at least one point on the path P. Since each point of P has at most $\mathcal{O}(\log n)$ ancestors and the size of the associated graph is poly$\{\log n, 1/\varepsilon\}$ within each square, the total time taken to find an augmenting path can be bounded by $\tilde{\mathcal{O}}(|P|)$. Augmenting the matching along P requires the associated graph to be reconstructed for the $\mathcal{O}(\log n)$ ancestors of each of the points of P. This again can be done using the Bellman-Ford algorithm, resulting in a total update time of $\tilde{\mathcal{O}}(|P|)$. The total length of all the augmenting paths computed by the algorithm can be shown to be $\tilde{\mathcal{O}}(n \log n)$, and so the total time taken by the algorithm is near-linear in n.

Our Algorithm: Similar to the Euclidean case, we can transform our input so that our optimal matching cost is $\mathcal{O}(n/\varepsilon^2)$ (see Section 3.1) and store the input in a quadtree Q of height $\mathcal{O}(\log n)$. For the squared Euclidean distance, we combine the ideas from the two algorithms of [5] and [26] in a non-trivial fashion. First, we note that using $\mathcal{O}(\text{poly}\{\log n, 1/\varepsilon\})$ edge bundles leads to an explosion in the expected distortion. In order to keep the expected distortion small, we create approximately $\tilde{\mathcal{O}}(2^{i/2})$ edge bundles for a square of side-length 2^i[3]. This causes larger squares to have many more bundles of edges (See Section 3). For instance, a square of side-length n can have roughly $\sqrt{n}$ edge bundles. A useful property of this distance approximation is that any edge appearing in a square of side-length 2^i has a quad-tree distance value roughly between $\Omega(2^i)$ and $\mathcal{O}(2^{2i})$. This implies that all edges with a small quad-tree distance appear within edge bundles of the smaller squares. Like in the Euclidean case, we can show that our distance is an upper bound on the squared Euclidean distance. Furthermore, if Q is a randomly shifted quad-tree, we can show that the expected cost of our distance is at most $(1+\varepsilon)$ times the squared Euclidean distance.

In the squared Euclidean quad-tree distance, the number of edge bundles at each square of the quad tree is a polynomial in n. Using these bundles, we define a sublinear sized associated graph. However, unlike the algorithm of [26], using the Bellman-Ford search procedure to find an augmenting path in the associated graph will lead to an $\Omega(n^{3/2})$ time algorithm. Instead, we employ a primal-dual approach.

Prior to describing our algorithm and the data structure, we note that primal-dual search procedures, such as Hungarian search and our algorithm, find augmenting paths in increasing order of their "costs". As a result, such a search on quad-tree distances will initially only involve edges with small quadtree distance and, as the algorithm progresses, larger quad-tree distances get involved. Therefore, searches can initially be localized to smaller squares of the quad-tree and our algorithm only needs to build the associated graphs in the smaller squares. As the algorithm progresses, however, longer edges participate in the augmenting paths, which forces our algorithm to build associated graph data structures in larger squares, increasing the time taken to conduct a Hungarian search. We refer to these squares where the data structure is maintained as *active* squares.

Now we present an overview of our algorithm and the data structure within an active square $\square^*$ of width 2^i. We partition $\square^*$ into $\mathcal{O}(2^{2i/3})$ *pieces* using a grid of side-length $2^j = 2^{\lfloor 2i/3 \rfloor}$. Each piece is further recursively divided into four squares. The entire hierarchical structure is stored within a carefully defined *active tree*. We build an associated graph $\tilde{G}$ at each node of the active tree. For the first level of the active tree, we build the associated graph as follows: The vertex set $\tilde{V}$ contains $\tilde{\mathcal{O}}(2^{i/3})$ vertices per piece and $\tilde{\mathcal{O}}(2^i)$ vertices in total. For pairs of vertices u, v that belong to the same piece, we explicitly store an edge; we refer to these edges as *internal* edges. There are $\tilde{\mathcal{O}}(2^{2i/3})$ internal edges per piece and the internal edges in each piece can be constructed in $\tilde{\mathcal{O}}(2^i)$ time (see Sections 6.2–6.4). Similar associated graphs are also constructed for every subsequent levels of the active tree. Similar to the approximate Euclidean matching algorithm, these internal edges of the associated graph represent certain short-

[3]Throughout this paper, we set the side-length of the square to be the difference in the x-coordinate values of the the vertical boundaries, i.e., the Euclidean length of each of its four edges.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

est paths in the residual graph. Additionally, for any pair of vertices $u, v \in \tilde{V}$, we add a *bridge edge* between them with a cost that is approximately the squared Euclidean distance between the end-points. We do not store the bridge edges explicitly. Instead, we build an ε-Well Separated Pair Decomposition (WSPD) of size $\tilde{\mathcal{O}}(2^i)$ to store them. Therefore, the total size of the graph is restricted to $\tilde{\mathcal{O}}(2^i)$ vertices and $\tilde{\mathcal{O}}(2^{4i/3})$ edges.

Next, we define dual weights on every vertex of the associated graph and define compressed feasibility conditions that are satisfied by its edges (see Section 6.5). Recollect that for planar graphs, compressed feasibility conditions are defined only on a single global compressed residual graph. In our case, however, the residual graph is represented in a compressed fashion via a hierarchical set of associated graphs defined on every node of the active tree. It is significantly more challenging to design compressed dual feasibility conditions that allows for a sublinear time Hungarian search procedure on such a hierarchical structure. Interestingly, one can use the feasible dual weights on the associated graph vertices to derive a set of dual weights satisfying the classical matching feasibility conditions (see Section 6.7). Using compressed feasibility, we provide a quick way to conduct primal-dual searches on the associated graph resulting in a running time of $\tilde{\mathcal{O}}(2^{4i/3})$ per search (see Section 6.8.3). We show that the number of primal-dual searches on the associated graph of any active square with side-length 2^i is only $\tilde{\mathcal{O}}(n/2^i)$ (see Section 5.1). Therefore, the total time spent for all searches within active squares of side-length 2^i is $\tilde{\mathcal{O}}(n2^{i/3})$.

Suppose the primal-dual search at $\square^*$ returns an admissible augmenting path. The algorithm then augments the matching along this path. Augmentation forces the algorithm to rebuild the set of internal edges within every *affected piece* of the associated graph at $\square^*$, i.e., pieces that contain at least one edge of P. In order to reduce the number of such updates, similar to [5], we assign an additive cost of roughly $\frac{\varepsilon^2 2^{2i/3}}{\log n}$ to every bridge edge of the associated graph. We argue that this additional error does not increase the optimal matching cost by more than a multiplicative factor of ε.

To bound the time taken to rebuild the internal edges, like [5], we show that the total additive cost of the edges on the augmenting paths, computed over the entire algorithm, is $\tilde{\mathcal{O}}(n)$ (see Section 5.1). Each bridge edge of the associated graph $\tilde{G}$ has an error of at least $\frac{\varepsilon^2 2^{2i/3}}{\log n}$. Therefore, the number of times such edges participate across all augmenting paths is only $\tilde{\mathcal{O}}(\frac{n}{2^{2i/3}})$. As a result, the total number of rebuilds of internal edges, for pieces of all active squares of side-length 2^i, across the entire algorithm, is $\tilde{\mathcal{O}}(n/2^{2i/3})$. Rebuilding the internal edges of one piece takes $\tilde{\mathcal{O}}(2^i)$ time (see Section 6.6). Therefore, the total time spent rebuilding pieces is $\tilde{\mathcal{O}}(n2^{i/3})$, matching the total time taken for all searches on the associated graph for layer i active squares.

As the algorithm progresses, larger squares become active. When the side-length of the active square is approximately $n^{3/4}$, the time taken to execute a single search on the associated graph becomes $\Omega(n)$. At this point, we show that there are only $\tilde{\mathcal{O}}(n^{1/4})$ free vertices remaining. Each remaining free vertex can be matched by conducting an efficient Hungarian search on the original points in $\tilde{\mathcal{O}}(n)$, taking $\tilde{\mathcal{O}}(n^{5/4})$ time in total. The total time spent on searches and rebuilds on active squares with side-length at most $2^{(3/4)\log_2 n} = n^{3/4}$ using our data structure is $\tilde{\mathcal{O}}(n2^{(1/4)\log_2 n}) = \tilde{\mathcal{O}}(n^{5/4})$, giving a total running time of $\tilde{\mathcal{O}}(n^{5/4})$.

Comparison with [19]: Following the work of Asathulla *et al.* [5], using the same framework, Lahn and Raghvendra presented a faster $\tilde{\mathcal{O}}(n^{6/5})$ algorithm to compute a minimum-cost perfect matching in planar graphs. Their main idea was to carefully compute multiple augmenting paths in one scan of the graph, leading to a faster convergence to the optimal matching. We would like to note that any augmenting path found in our algorithm is localized within an active square. Therefore, our algorithm identifies one augmenting path in a single access to an active square and many augmenting paths in a single access to the entire graph (spanning all the active squares). Unlike for planar graphs, employing the approach of Lahn and Raghvendra [19] in our setting does not lead to any additional advantage in terms of the convergence to a perfect matching.

Extensions and Open Problems: Achieving a near-linear execution time in the two-dimensional case and $o(n^{3/2})$ time algorithms for d-dimensions remain important open questions. Our approach can achieve this goal provided we overcome the following difficulty: Currently, Hungarian search runs in time linear in the number of internal edges. In planar graphs, although the compressed residual graph has n edges, one can use a shortest-path data structure by Fakcharoenphol and Rao [10] to execute each Hungarian search in $\tilde{\mathcal{O}}(|\tilde{V}|) = \tilde{\mathcal{O}}(n^{2/3})$ time. Design of a similar data structure that conducts Hungarian search on associated graph in time $\tilde{\mathcal{O}}(|\tilde{V}|)$ will lead to a near-linear time ε-approximation algorithm for RMS matching in two-dimensions and an $o(n^{3/2})$ time algorithm in higher dimensions.

Organization: The remainder of the paper is organized as follows: In Section 3 we describe the details of our distance function while highlighting differences from the distance function of [26]. In Section 4 we introduce a quad-tree based dual-feasibility condition that incorporates an additional additive cost on each edge.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

In Section 5, we give a detailed description of the algorithm, along with its analysis. The algorithm description assumes the existence of a data structure built on active squares. This data structure includes the compressed feasible matching as well as several procedures, such as the sublinear time Hungarian search and augment that operate on a compressed feasible matching, and is described in detail in Section 6. Due to page limitations, many proofs have been omitted from this version. Ommitted proofs can be found in the full version of the paper [21].

3 Our Distance Function

3.1 Initial Input Transformation For the purposes of describing both the distance function and our algorithm, we assume that the point sets A and B satisfy certain properties. We can transform any point sets $A', B' \subset \mathbb{R}^2$ containing n points, to point sets A and B with n points such that each point of A' (resp. B') maps to a unique point of A (resp. B) and: (A1) Every point in $A \cup B$ has non-negative integer coordinates bounded by $\Delta = n^{\mathcal{O}(1)}$, (A2) no pair of points a, b where $a \in A$ and $b \in B$ are co-located, i.e., $\|a - b\| \geq 1$, (A3) the optimal matching of A and B has a cost of at most $\mathcal{O}(n/\varepsilon^2)$, and (A4) any ε-approximate matching of A and B corresponds to an 3ε-approximate matching of A' and B'. The details of this transformation are described in the full version of the paper [21], but we present an overview: First, we obtain an $n^{O(1)}$-approximation of the optimal matching in linear-time [1]. We further refine this estimate by making $\mathcal{O}(\log n)$ guesses of the optimal cost, and at least one guess gives a 2-approximation. By executing our algorithm $\mathcal{O}(\log n)$ times, one for each guess, at least one algorithm will have a 2-approximation of the optimal matching cost. Using this refined estimate, we rescale the points such that the optimal cost becomes $\mathcal{O}(n/\varepsilon^2)$. Finally, rounding the resulting points to integers, such that no point of A is co-located with a point of B, does not contribute too much error. As a result, in the rest of the paper, we assume that properties (A1)–(A4) hold. Next, we define a quad-tree based distance that approximates the squared Euclidean distances between points.

3.2 Randomly Shifted Quadtree Decomposition Similar to [26], we define our distance function based on a randomly-shifted quadtree. Without loss of generality, assume Δ is a power of 2. First, we pick a pair of integers $\langle x, y\rangle$ each independently and uniformly at random from the interval $[0, \Delta]$. We define a square $G = [0, 2\Delta]^2 - \langle x, y\rangle$ that contains all points of $A \cup B$. This square will form the root node of our quadtree, and each internal node of the tree is subdivided into 4 equal-sized squares to form its children in the tree.

Specifically, for $\delta = \log_2(2\Delta)$ and a constant $c_1 > 0$, we construct a quadtree Q of height $\delta + 2\log(\log(\Delta)/\varepsilon) + c_1 = \mathcal{O}(\log n)$ (from (A1)). The layers of Q can be seen as a sequence of grids $\langle G_\delta, \ldots, G_0, \ldots, G_{-2\log(\log(\Delta)/\varepsilon)-c_1}\rangle$. The grid G_i is associated with squares with side-length 2^i and the grid of leaf nodes $G_{-2\log(\log(\Delta)/\varepsilon)-c_1}$ is associated with cells of width $1/2^{2\log(\log(\Delta)/\varepsilon)+c_1}$. Although, cells of grid G_0 contain at most one point (or possibly multiple copies of the same point) and can be considered leaf nodes of the quadtree, it is notationally convenient to allow for us to define grids G_i for all $i \geq -2\log(\log(\Delta)/\varepsilon) - c_1$ and consider their cells to be part of the quadtree. Specifically, the additional levels help facilitate a cleaner definition of subcells (see Section 3.2.1). We say that a square $\square$ has a *level* i if $\square$ is a cell in grid G_i. For any two cells $\square$ and $\square'$, let $\ell_{\min}(\square, \square')$ (resp. $\ell_{\max}(\square, \square')$) be the minimum (resp. maximum) distance between the boundaries of $\square$ and $\square'$, i.e., the minimum distance between any two points u and v where u is on the boundary of $\square$ and v is on the boundary of $\square'$. Next, we describe how any cell of this quadtree that has a level greater than or equal to 0 can be divided into subcells, a concept essential to describe our distance function.

3.2.1 Division of a Cell Into Subcells For any grid G_i with $i \geq 0$, we define the *minimum subcell size* to be $\mu_i = 2^{\lfloor i/2\rfloor - 2\log\frac{\log\Delta}{\varepsilon} - c_1}$, where $c_1 > 0$ is the constant from the construction of the quadtree. Each cell $\square \in G_i$ is subdivided into a set of subcells, with each subcell having width at least μ_i. In [26], the minimum subcell size μ_i was much larger, being roughly $2^{i - \mathcal{O}(\log\frac{\log\Delta}{\varepsilon})}$. For them subcells using a uniform grid of side-length μ_i was sufficient, resulting in $\mathcal{O}((2^i/\mu_i)^2) = \text{poly}\{\log n, 1/\varepsilon\}$ subcells. However, for squared Euclidean distances, much smaller subcells are required, and using a uniform grid would result in $\Omega(2^i)$ subcells, which is too large for our purposes. Instead, we replace the uniform grid of subcells with an *exponential* grid of subcells, reducing the number of subcells to $\tilde{\mathcal{O}}(2^i/\mu_i) = \tilde{\mathcal{O}}(2^{i/2})$. We describe this process of forming the exponential grid next. For a visual example of the exponential grid, see Figure 1.

For any cell $\square$ of Q with a level $i \geq 0$, let $\square_1, \square_2, \square_3$ and $\square_4$ be its four children. We define *subcells* of any cell $\square$ as the leaf nodes of another quadtree $Q_\square$ with $\square$ as its root and its four children recursively sub-divided in $Q_\square$ as follows. Let $u \leftarrow \square_1$, we recursively divide u into four cells until: (a) The side-length of u is the minimum subcell size μ_i, or (b) the side-length of u is at most $(\varepsilon/144)\ell_{\min}(\square_1, u)$. Similarly, we decompose $\square_2, \square_3$ and $\square_4$ into subcells as well. Note that every

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

cell of the quadtree $Q_\square$ is also a cell in the quadtree Q and the leaves of $Q_\square$ (the *subcells* of $\square$) will satisfy (a) or (b). We denote the subcells of $\square$ by $\mathbb{G}[\square]$. Note that, for any subcell $u \in \mathbb{G}[\square]$ where u is a descendant of $\square_1$, the side-length of u is larger than the minimum subcell size if and only if $\ell_{\min}(\square_1, u)$ is sufficiently large. i.e., as we move away from the boundary of $\square_1$, the subcell size becomes larger. Using this, in Lemma 3.1, we show that the total number of subcells for any cell $\square \in G_i$ is $\tilde{\mathcal{O}}(\mu_i)$. The argument can be seen intuitively from the fact that the outermost ring of subcells along the boundary of $\square_1$ has size $\tilde{\mathcal{O}}(\mu_i)$. Furthermore, subcells increase in size as we move towards the center of $\square_1$, implying that their count decreases geometrically.

LEMMA 3.1. *For any cell $\square$ of Q with level i, the total number of subcells is $\tilde{\mathcal{O}}(\mu_i)$.*

For some edge $(a, b) \in A \times B$, let $\square$ be the least common ancestor of a and b in Q. For each $\square \in G_i$, we say that the edge (a, b) *appears at* level i. From (A2), all edges of $A \times B$ appear at or above level 1. The quadtree distance between a and b defined in [26] is given by the distance between the subcells ξ_a and ξ_b of $\mathbb{G}[\square]$ that contain a and b respectively. Therefore, the set of edges that appear at layer i can be represented using pairs of subcells from the set $\bigcup_{\square' \in G_i} \mathbb{G}[\square']$. However, the use of all pairs of subcells is prohibitively expensive. We further reduce the number of pairs of subcells by grouping them into a Well-Separated Pair Decomposition which we describe next.

3.2.2 Well-Separated Pair Decompositions In this section, we extend Well-Separated Pair Decomposition (WSPD) that is commonly defined for points to approximate distances between pairs of subcells. A Well-Separated Pair Decomposition (WSPD) is a commonly used tool that, given a set P of n points, compactly approximate all $\mathcal{O}(n^2)$ distances between points of P by using a sparse set $\mathcal{W}$ of only $\tilde{\mathcal{O}}(n)$ well-separated pairs. Each pair $(S, T) \in \mathcal{W}$ consists of two subsets $S, T \subseteq P$ of points. For any pair of points $(u, v) \in P \times P$, there is a unique pair $(S, T) \in \mathcal{W}$ such that $(u, v) \in S \times T$. For each pair (S, T), an arbitrary pair of representatives $s \in S$ and $t \in T$ can be chosen, and the distance between any pair $(s', t') \in S \times T$ can be approximated using the distance between the representatives s and t. This approximation will be of good quality so long as the pair (S, T) is well-separated, meaning the distance between any pair of points *within* S or *within* T is sufficiently small compared to the distance between any pair of points *between* S and T.

For any parameter $\varepsilon > 0$, using the construction algorithm of [14], it is possible to build in

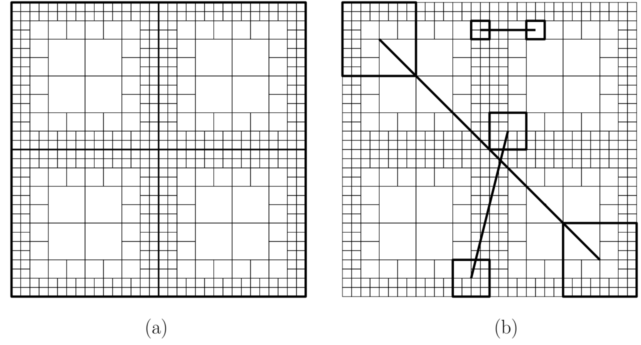

Figure 1: (a) A division of $\square$ into subcells. (b) Examples of a few possible WSPD pairs of $\mathcal{W}_\square$. Every pair of subcells in different children of $\square$ would be represented by some pair.

$\tilde{\mathcal{O}}(n\text{poly}\{\log n, 1/\varepsilon\})$ time a WSPD of the edges of $A \times B$ where the costs of the edges belonging to any pair in the decomposition are within a factor of $(1+\varepsilon)$ of each other. Furthermore, if the ratio of the largest edge to smallest edge cost is bounded by $n^{\mathcal{O}(1)}$, then it can be shown that every point participates in only poly$\{\log n, 1/\varepsilon\}$ pairs. Such a WSPD can be used to execute a single Hungarian search in near-linear time in the number of points. However, in order to execute a Hungarian search in sublinear time, we must build a WSPD on the sublinear number of subcells instead of the original points. Luckily, the algorithm of [14] can be applied in a straightforward fashion to generate a WSPD on subcells. Next, we describe the properties of our WSPD on subcells.

For any level i cell $\square$ of Q, consider two subsets of subcells, $S \subseteq \mathbb{G}[\square]$ and $T \subseteq \mathbb{G}[\square]$. We define $\ell_{\max}(S, T) = \max_{\xi \in S, \xi' \in T} \ell_{\max}(\xi, \xi')$. We say that S and T are ε-well separated if, for every pair of subcells $\xi \in S$ and $\xi' \in T$,

$$\ell_{\max}(S, T) \leq (1 + \varepsilon/12)\ell_{\max}(\xi, \xi'). \tag{3.1}$$

For each cell $\square$ let $\square_1, \square_2, \square_3$ and $\square_4$ be its four children. We precompute a WSPD $\mathcal{W}_\square = \{(S_1, T_1), \ldots, (S_r, T_r)\}$, where $S_i \subseteq \mathbb{G}[\square]$, $T_i \subseteq \mathbb{G}[\square]$ and S_i, T_i are ε-well separated. Furthermore, for every pair of subcells $(\xi_1, \xi_2) \in \mathbb{G}[\square] \times \mathbb{G}[\square]$ (resp.$(\xi_2, \xi_1) \in \mathbb{G}[\square] \times \mathbb{G}[\square]$) where ξ_1 and ξ_2 are in two different children of $\square$, there is a unique *ordered* pair in $(X, Y) \in \mathcal{W}_\square$ (resp. $(Y, X) \in \mathcal{W}_\square$) such that $\xi_1 \in X$ and $\xi_2 \in Y$. We denote the ordered pair $(X, Y) \in \mathcal{W}_\square$ that the pair of sub-cells (ξ_1, ξ_2) maps to as (S_{ξ_1}, T_{ξ_2}). For notational convenience, we prefer that the pairs within the WSPD are ordered. Such an ε-WSPD can be constructed by executing a standard quadtree based construction algorithm presented in [14]. This algorithm uses the subtree of Q rooted at $\square$ to build the WSPD. Since

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

we are interested in ξ_1 and ξ_2 that are contained inside two different children of $\square$, we can trivially modify the algorithm of [14] to guarantee that every pair (S_i, T_i) in the WSPD is such that the subcells of S_i and the subcells of T_i are contained in two different children of $\square$. See Figure 1 for examples of WSPD pairs in $\mathcal{W}_\square$. Finally, the algorithm of [14] naturally generates unordered pairs. To ensure that every pair of subcells is covered by an ordered pair in the WSPD, for every pair $(X, Y) \in \mathcal{W}_\square$ generated by the algorithm, we add (Y, X) to $\mathcal{W}_\square$.

3.2.3 Distance Function Given the definitions of subcells and the WSPDs, we can finally define the distance function. For $p, q \in A \cup B$, let $\square$ be the least common ancestor of p and q in Q and let i be the level of $\square$. We denote the *level* of the edge (p, q) to be the level of the least common ancestor of its end points, i.e., the level of $\square$. For some edge (p, q) with least common ancestor $\square$, let ξ_p and ξ_q be subcells from $\mathbb{G}[\square]$ that contain p and q respectively. Note that ξ_p and ξ_q are contained inside two different children of $\square$. There is a unique ordered *representative pair* $(\Psi_p, \Psi_q) \in \mathcal{W}_\square$ with $\xi_p \in \Psi_p$ and $\xi_q \in \Psi_q$. We set the distance between p and q to be

$$d_Q(p, q) = (\ell_{\max}(\Psi_p, \Psi_q))^2.$$

From the properties of our WSPD, if the unique representative pair of (p, q) is (X, Y), then the representative pair for (q, p) will be (Y, X), implying that our distance $d_Q(\cdot, \cdot)$ is symmetric. For any subset $E \subseteq A \times B$ of edges, we define its cost by $d_Q(E) = \sum_{(a,b)\in E} d_Q(a, b)$. Since $p \in \xi_p, q \in \xi_q$ and $\xi_p \in \Psi_p$, $\xi_q \in \Psi_q$, we have

$$\|p-q\|^2 \leq (\ell_{\max}(\xi_p, \xi_q))^2 \leq (\ell_{\max}(\Psi_p, \Psi_q))^2 = d_Q(p, q). \tag{3.2}$$

Furthermore, it can be shown that if Q is a randomly shifted quad-tree, any optimal matching M_{OPT} with respect to the original squared Euclidean costs satisfies

$$\mathbb{E}[d_Q(M_{\mathrm{OPT}})] \leq (1 + \varepsilon) \cdot \sum_{(a,b)\in M_{\mathrm{OPT}}} \|p - q\|^2. \tag{3.3}$$

Additional Notations: Next, we define additional notations that will be helpful in describing our data structure in Section 6. Any point $p \in A \cup B$ is contained inside one cell of each of the grids G_i in Q. Let $\square = \square_p^i$ be the cell of G_i that contains p. Let $\xi_p^\square \in \mathbb{G}[\square]$ be the subcell that contains p. As a property of the WSPD construction algorithm, the decomposition $\mathcal{W}_\square$ ensures $\xi_p^\square$ participates in $\tilde{\mathcal{O}}(1)$ WSPD pairs of $\mathcal{W}_\square$. Let this set be denoted by $N^i(p)$. Every edge of level i incident on p is represented by exactly one pair in $N^i(p)$. Since there are $\mathcal{O}(\log n)$ levels, every edge incident on p is represented by $\tilde{\mathcal{O}}(1)$ WSPD pairs. We refer to these WSPD pairs as $N^*(p) = \bigcup_i N^i(p)$. We can have a similar set of definitions for a subcell ξ instead of a point p. Consider any cell $\square \in G_i$ and a subcell $\xi \in \mathbb{G}[\square]$. Using a similar argument, we conclude that all edges of level i incident on any vertex of $(A \cup B) \cap \xi$ are uniquely represented by $\tilde{\mathcal{O}}(1)$ WSPD pairs denoted by $N^i(\xi)$. Furthermore, all edges of level $\geq i$ are uniquely represented by $N^*(\xi) = \bigcup_{j\geq i} N^j(\xi)$. Note that $|N^*(\xi)| = \tilde{\mathcal{O}}(1)$.

4 Dual Feasibility Conditions

In this section, we introduce a new set of feasibility conditions based on the randomly shifted quadtree. These feasibility conditions will allow our algorithm to find minimum-cost augmenting paths more efficiently. In order to describe this distance function, we partition the edges into a set of local edges and a set of non-local as described next. A similar definition of local and non-local edges was used in [26].

Local and Non-Local Edges: We say that any two matching edges $(a, b) \in M$ and $(a', b') \in M$ belong to the same equivalence class if and only if they have the same least common ancestor $\square$ and their ordered representative pairs in $\mathcal{W}_\square$ are the same, i.e., $(\Psi_a, \Psi_b) = (\Psi_{a'}, \Psi_{b'})$. Let $\mathcal{K}_M = \{M_1, ..., M_h\}$ be the resulting partition of matching edges into classes. For each M_k for $1 \leq k \leq h$, let $A_k = \bigcup_{(a_j,b_j)\in M_k} a_j$ and $B_k = \bigcup_{(a_j,b_j)\in M_k} b_j$. The set $\{A_1, ..., A_h\}$ partitions the matched vertices of A and $\{B_1, ..., B_h\}$ partitions the matched vertices of B. We say an edge $(a, b) \in A \times B$ is *local* if $(a, b) \in A_k \times B_k$ for some $1 \leq k \leq h$. Otherwise, (a, b) is *non-local*. We refer to the local edges (both non-matching and matching) of $A_k \times B_k$ as *class k*.

Next, we define a set of feasibility conditions based on the randomly-shifted quadtree. For a matching M in the graph, $G(A \cup B, A \times B)$, we assign a *dual weight* $y(v)$ for every $v \in A \cup B$. Recall that μ_i is the minimum subcell size at level i in the quadtree. For any edge (a, b) of level i, let $\mu_{ab} = \mu_i$. A matching M and set of dual weights $y(\cdot)$ are Q-feasible if for every edge (a, b),

$$y(a) + y(b) \leq d_Q(a, b) + \mu_{ab}^2. \tag{4.4}$$

$$y(a) + y(b) = d_Q(a, b) \quad \text{if } (a, b) \text{ is a local edge.} \tag{4.5}$$

A Q-feasible perfect matching is a *Q-optimal matching*. Let M_{OPT} be the optimal RMS matching in $G(A \cup B, A \times B)$. Similar to the Gabow-Tarjan [12] and Asathulla *et al.* [5] algorithms, the addition of an additive error of μ_{ab}^2 for non-local edges distorts the cost of non-local edges of M_{OPT} by μ_{ab}^2. However, it can be shown that this additional error for any non-local edge (a, b) of the optimal matching is, in expectation, less

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

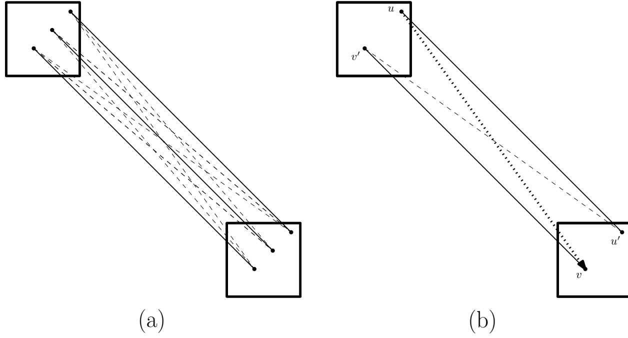

Figure 2: (a) A set of local edges between a WSPD pair of cells. Solid edges are in the matching, and dashed edges are not. (b) A local non-matching edge from $u \in B$ to $v \in A$ implies the existence of a length 3 alternating path $P = \langle u, u', v', v \rangle$ with net-cost $\phi(P) = d_Q(u, v)$.

than $\varepsilon\|a - b\|^2/2$ due to the random shift. This is because short edges of the optimal matching have a small probability of appearing at higher levels of the quadtree. By combining this argument with properties of the distance function, we can show the following lemma:

LEMMA 4.1. *For $A, B \subset \mathbb{R}^2$, let M_{OPT} be the optimal RMS matching. For a parameter $\varepsilon > 0$, given a randomly shifted quadtree Q and the distance $d_Q(\cdot, \cdot)$, let M be any Q-optimal matching. Then,*

$$\mathbb{E}[w(M)] \leq (1 + \varepsilon/2) \sum_{(a,b) \in M_{OPT}} \|a - b\|^2.$$

From Lemma 4.1, it follows that any Q-optimal matching is, in expectation, an ε-approximate RMS matching. Therefore, it suffices to design an efficient algorithm for computing a Q-optimal matching. By executing such an algorithm $\mathcal{O}(\log n)$ times, we can obtain an ε-approximate RMS matching with high probability.

5 Algorithm

Matching Preliminaries: For any matching M, an *alternating path* (resp. *alternating cycle*) with respect to M is one which alternates between edges of M and edges not in M. A vertex is *free* if it is not the endpoint of any edge of M and *matched* otherwise. We use A_F (resp. B_F) to denote the set of free vertices of A (resp. B). An *augmenting path* P is an alternating path between two free vertices. The matching $M' = M \oplus P$ has one higher cardinality than M. An alternating path P is called *compact* if the largest contiguous set of local edges of P has size at most 3 (see Figure 2). Throughout this paper, we use the notation a, a' and a_j for $1 \leq j \leq n$ to denote points in A and b, b' and b_j for $1 \leq j \leq n$ to denote points in B.

For any non-local edge (a, b), we define its *slack* as $s(a, b) = d_Q(a, b) + \mu_{ab}^2 - y(a) - y(b)$, i.e., how far the feasibility constraint (4.4) for (u, v) is from holding with equality. For all local edges the slack $s(a, b)$ is defined to be 0. Note that, for a Q-feasible matching, the slack on any edge is non-negative. We say any edge is *admissible* with respect to a set of dual weights if it has zero slack. The admissible graph is simply the subgraph induced by the set of zero-slack edges. Note that all local edges are also admissible.

As is common, we define the residual graph $\mathcal{G}_M$ of a matching M by assigning directions to edges of the graph $\mathcal{G}$. For any edge $(a, b) \in A \times B$, we direct (a, b) from a to b if $(a, b) \in M$ and from b to a otherwise. For any Q-feasible matching, we construct a weighted residual graph $\mathcal{G}'_M$ where the edges of the graph are identical to $\mathcal{G}_M$ and each edge (a, b) has a weight equal to $s(a, b)$. Any directed path in $\mathcal{G}_M$ is alternating, and any directed path in $\mathcal{G}_M$ that starts with a free vertex of B_F and ends at a free vertex of A_F is an augmenting path. Our algorithm will maintain a Q-feasible matching M and set of dual weights $y(\cdot)$. Initially $M = \emptyset$, and we set $y(v) \leftarrow 0$ for every vertex $v \in A \cup B$; clearly, this initial dual assignment is Q-feasible. Like the Hungarian algorithm, our algorithm will iteratively conduct a Hungarian search to find an augmenting path consisting only of admissible edges. Then the algorithm augments the matching along this path. This process repeats until a Q-optimal matching is found. Conducting a Hungarian search on the entire graph is prohibitively expensive. Therefore, we introduce a data structure that conducts Hungarian search and augment operations by implicitly modifying the dual weights in sublinear time.

First, our algorithm executes in $\lceil 3 \log n/4 \rceil$ phases, starting with phase 0. At the end of the execution of these phases, it produces a matching that has $\tilde{\mathcal{O}}(n^{1/4})$ free vertices. Finally, the algorithm matches the remaining free vertices one at a time by conducting a Hungarian search to find an augmenting path and then augmenting the matching along this path.

At the start of any phase $i \geq 1$, we are given a Q-feasible matching M along with a set of dual weights such that every free vertex $b \in B_F$ has a dual weight of μ_{i-1}^2. At the end of phase i, the dual weight of any free vertex $b \in B_F$ in our Q-feasible matching increases to μ_i^2.

The data structure is used only during the execution of phases. After the $\lceil 3 \log n/4 \rceil$ phases have been executed, the algorithm will conduct explicit Hungarian searches and augmentations. For any phase $i \leq \lceil 3 \log n/4 \rceil$, we describe the data structure $\mathcal{D}_i$. This data structure supports two global operations:

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

- Build : This operation takes as input a Q-feasible matching M and a set of dual weights $y(\cdot)$ such that for every free vertex $v \in B_F$, the dual weight $y(v) = \mu_{i-1}^2$. Given $M, y(\cdot)$, the procedure builds the data structure.

- GenerateDuals : At any time in phase i, the execution of this procedure will return the matching M stored by the data structure along with a set of dual weights $y(\cdot)$ such that $M, y(\cdot)$ is Q-feasible. We denote this matching as the *associated Q-feasible matching.*

The total time taken by both of these operations is bounded by $\tilde{\mathcal{O}}(n\mu_i^{2/3})$.

The data structure does not explicitly maintain a set of Q-feasible dual weights at all times because updating all the dual weights after each Hungarian search could take $\Omega(n)$ time. Instead it maintains a smaller set of 'up-to-date' dual weights, and updates other dual weights in a 'lazy' fashion. While a similar strategy was used in [5], applying the same strategy in our case requires the design of a new set of compressed feasibility conditions that are significantly more complex than the ones used in [5]. An example of just one such complexity is the fact that our compressed feasibility relate vertices, edges, and dual weights defined across all levels of the quadtree, while the compressed feasibility conditions in [5] do not require multiple 'levels'.

A set of up-to-date Q-feasible dual weights for all vertices could be recovered after any Hungarian search or augmentation by simply executing GenerateDuals. However, doing so is too expensive. Instead, the algorithm only executes GenerateDuals once at the end of every phase. Nonetheless, the GenerateDuals procedure guarantees the existence of a Q-feasible dual assignment for the matching M. We use this associated Q-feasible matching to describe the other operations supported by the data structure.

During phase i, we say that a cell $\square \in G_i$ is *active* if $\square \cap B_F \neq \emptyset$. The edges that go between active cells have a cost of at least μ_i^2 and do not become admissible during phase i because the dual weights of all vertices of B are at most μ_i^2 whereas the points of A have a non-positive dual weight. Therefore edges between active cells need not be considered during any Hungarian searches or augmentations of phase i. As a result, each Hungarian search and augmentation can be conducted completely within a single active cell.

During any phase i and for any active cell $\square^* \in G_i$, our data structure supports the following operations:

- HungarianSearch : This procedure conducts a Hungarian search. At the end of the search, either the dual weight of every free vertex $b \in B_F \cap \square^*$ with respect to the associated Q-feasible matching has risen to μ_i^2, or the search returns an augmenting path P inside $\square^*$ such that P is both admissible and compact with respect to the associated Q-feasible matching.

- Augment : This procedure augments the matching along an augmenting path P returned by HungarianSearch and updates the data structure to reflect the new matching.

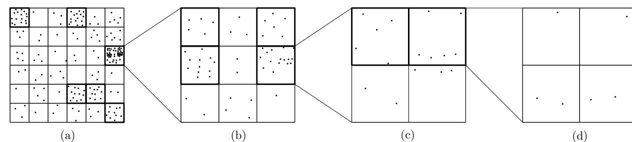

Figure 3: (a) The active cells of the current layer. Each full active cell (bold) has descendants in its active tree. (b) The pieces of a full active cell. (c) Each full piece has 4 descendants in the active tree. (d) Each branch of the active tree terminates with a sparse leaf cell.

We postpone the implementation details of the four operations supported by this data structure until Section 6. In order to describe the execution time of these procedures, we define *active trees* next.

Active Tree: We say a cell $\square$ at level i in the quadtree Q is *sparse* if $|(A \cup B) \cap \square| \leq \mu_i^2$ or if $\square \in G_0$. Otherwise, $\square$ is *full*. For each active cell $\square^*$ during phase i, we maintain an *active tree* denoted by $\mathcal{T}_{\square^*}$. The active tree $\mathcal{T}_{\square^*}$ is rooted at $\square^*$ and contains a subset of the nodes in the subtree of $\square^*$ in Q. If $\square^*$ is sparse, then $\square^*$ is also a leaf node and the active tree contains only one node. Otherwise, if $\square^*$ is full, let all cells of $G_{\lfloor 2i/3 \rfloor}$ that partition $\square^*$ be the children of $\square^*$ in the active tree. We refer to every child of $\square^*$ in the active tree as a *piece* of $\square^*$. For each piece $\square$ of $\square^*$, if $\square$ is sparse, then $\square$ will become a leaf node of the active tree. Otherwise, if $\square$ is full, then $\square$ is an internal node of the active tree, and the four children of $\square$ in Q are also contained in the active tree. We recursively apply this process to construct the active tree for each of the four children; each full child is decomposed into its four children in Q. Every leaf node of $\mathcal{T}_{\square^*}$ is a sparse cell and every internal node is a full cell.

Consider any augmenting path P computed inside an active cell $\square^*$ during phase i. Let $\mathcal{A}(P)$ be the set of all cells of the active tree, excluding $\square^*$, that contain at least one vertex of P. We call such cells the *affected* cells of P. Let $\mathcal{A}_j(P)$ be the set of level j affected cells of P. Then the time taken for a single execution of the HungarianSearch procedure that returns an

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

augmenting path P is $\tilde{\mathcal{O}}(\mu_i^{8/3}+\sum_{j=0}^{\lfloor 2i/3\rfloor}|\mathcal{A}_j(P)|\mu_j^3)$ (see Section 6.8.3), and the time taken for an execution of the AUGMENT procedure on an augmenting path P is $\tilde{\mathcal{O}}(\sum_{j=0}^{\lfloor 2i/3\rfloor}|\mathcal{A}_j(P)|\mu_j^3)$ (see Section 6.8.4).

Using these operations, we now present our algorithm for any phase $0 \le i < \lceil 3\log n/4\rceil$. At the start of phase $i \ge 1$, the dual weight of every vertex in B_F is equal to μ_{i-1}^2. We mark all active cells as unprocessed. The algorithm for phase i conducts the following steps.

- Build the data structure $\mathcal{D}$ using the BUILD procedure.
- While there is an unprocessed active cell $\square^*$,
  - Execute the HUNGARIANSEARCH procedure on $\square^*$.
  - If HUNGARIANSEARCH returned an augmenting path P, then execute AUGMENT on P.
  - If either $B_F \cap \square^* = \emptyset$ (i.e., $\square^*$ is no longer active) or the dual weight of every vertex of $B_F \cap \square^*$ is μ_i^2, then mark $\square^*$ as processed.
- Use GENERATEDUALS to obtain the associated Q-feasible matching M and the dual weights $y(\cdot)$.

After the execution of all $\lceil 3\log n/4\rceil$ phases, we match the remaining free vertices one at a time by iteratively executing Hungarian search to find an augmenting path and augmenting the matching M along the path. Unlike during the phases, each Hungarian search is global and executed on $\mathcal{G}_M$ without use of the data structure. We describe the details of this global Hungarian search next.

Hungarian Search: We add a source vertex s to the graph $\mathcal{G}'_M$ and connect s to each vertex $b \in B_F$ with a cost 0 edge. Then, we execute a Dijkstra search in the resulting graph, starting from s. For any point $u \in A \cup B$, let d_u be the shortest path distance from s to u as computed by Dijkstra's algorithm. We define as value d as,

$$d = \min_{f \in A_F} d_f.$$

Next, for every $u \in A \cup B$ with $d_u \le d$, we perform the following dual adjustment. If $u \in B$, we set $y(u) \leftarrow y(u) + d - d_u$, and if $u \in A$, we set $y(u) \leftarrow y(u) - d + d_u$. Using a straightforward and standard argument, it is easy to show that this dual adjustment maintains Q-feasibility. Furthermore, after the Hungarian search, $\mathcal{G}_M$ contains an augmenting path P consisting solely of admissible edges.

When implemented naively, the Hungarian search could take $\Omega(n^2)$ time. However, we recall that each vertex of $\mathcal{G}_M$ is part of only $\tilde{\mathcal{O}}(1)$ WSPD pairs. All edges (u, v) with the same representative WSPD pair have the same value of $d_Q(u, v)$ as well as the same value of μ_{uv}^2. Using this fact, Dijkstra's algorithm can efficiently find the next edge to add to the shortest path tree in amortized $\tilde{\mathcal{O}}(1)$ time per addition. As a result, a single Hungarian search can be executed in $\tilde{\mathcal{O}}(n)$ time.

Augment: Let P be an admissible augmenting path found by the Hungarian search procedure. We describe how to augment M along P while maintaining Q-feasibility. First, we set $M \leftarrow M \oplus P$. This causes some non-local edges (potentially both matching and non-matching) to become local. We must adjust the dual weights along P to ensure that every newly introduced local edge (u, v) satisfies the Q-feasibility constraint $y(u) + y(v) = d_Q(u, v)$. Let $(a, b) \in A_k \times B_k$ be an edge of P that is local and in class k edge after augmentation, but was non-local prior to augmentation. If there are no other class k local edges after augmentation that were also local prior to augmentation, we set $y(a) \leftarrow y(a) - \mu_{ab}^2$. Otherwise, there must be at least one local edge (a', b') in class k after augmentation that was local prior to augmentation, and we set $y(a) \leftarrow y(a')$ and $y(b) \leftarrow y(b')$.

Invariants: During the execution of the phases, the algorithm guarantees the following invariants:

(I1) The associated matching $M, y(\cdot)$ is Q-feasible.

(I2) The dual weight $y(b)$ of every vertex $b \in B$ is non-negative. Furthermore, in phase $i \ge 1$, the dual weight of every free vertex $b \in B_F$ is $\mu_{i-1}^2 \le y(b) \le \mu_i^2$. The dual weight of every vertex $a \in A$ is non-positive and for every free vertex $a \in A_F$, $y(a) = 0$.

Since each step of the algorithm is a call to the data structure $\mathcal{D}_i$, it suffices to show that $\mathcal{D}_i$ maintains these invariants. We do this in Section 6.

After the execution of the phases, the algorithm switches to conducting explicit Hungarian searches. Using a standard argument, it is easy to show that the dual updates of these Hungarian searches maintain Q-feasibility. However, the dual adjustments during augmentations are non-standard due to a careful handling of local edges. The following lemma establishes that the augmentation process continues to maintain Q-feasibility. Therefore, at the end of the algorithm, we produce a Q-optimal matching as desired.

LEMMA 5.1. *Let P be an admissible path with respect to a Q-feasible matching M and set of dual weights $y(\cdot)$. Let $M', y'(\cdot)$ be the matching and set of dual weights after augmenting along P. Then $M', y'(\cdot)$ are Q-feasible.*

5.1 Analysis of the Algorithm

In this section, we bound the time taken by the algorithm under

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

the assumption that the data structure works as described. We begin by defining notations that will be used throughout the analysis. Let $\mathbb{P} = \langle P_1, \ldots, P_t \rangle$ be the t augmenting paths computed during the $\lceil 3 \log n/4 \rceil$ phases of the algorithm. Let M_0 be the initial empty matching and, for any $k \geq 1$, let M_k be the matching obtained after augmenting the matching M_{k-1} along P_k, i.e., $M_k = M_{k-1} \oplus P_k$. For any augmenting path P with respect to some matching M, let $N(P)$ be the set of non-local edges of P. The following Lemma establishes important properties of the algorithm during the $\lceil 3 \log n/4 \rceil$ phases of the algorithm.

LEMMA 5.2. *The algorithm maintains the following properties during the $\lceil 3 \log n/4 \rceil$ phases:*

(i) The total number of free vertices remaining at the end of phase i is $\tilde{\mathcal{O}}(n/\mu_i^2)$, and,

(ii) $\sum_{k=1}^{t} \sum_{(a,b) \in N(P_k)} \mu_{ab}^2 = \tilde{\mathcal{O}}(n)$.

Using Lemma 5.2, we can bound the efficiency of the algorithm. First, we bound the total time taken after the $\lceil 3 \log n/4 \rceil$ phases have been executed. After the last phase is executed, $\mu_i^2 = \Omega(n^{3/4}/\text{poly}\{\log n, 1/\varepsilon\})$. From Lemma 5.2, there are only $\tilde{\mathcal{O}}(n^{1/4})$ unmatched vertices remaining. Using the WSPD, each of these unmatched vertices are matched in $\tilde{\mathcal{O}}(n)$ time. Therefore, the time taken for these phases is $\tilde{\mathcal{O}}(n^{5/4})$.

Next, we bound the time taken by the $\lceil 3 \log n/4 \rceil$ phases of the algorithm. For any such phase i, we execute the BUILD procedure to create the data structure $\mathcal{D}_i$. At the end of phase i, we execute the GENERATEDUALS procedure to generate a Q-feasible matching. Both of these operations take $\tilde{\mathcal{O}}(n\mu_i^{2/3})$ time during phase i.

Next, we bound the time taken by AUGMENT, which takes $\tilde{\mathcal{O}}(\sum_{j=0}^{\lfloor 2i/3 \rfloor} |\mathcal{A}(P)|\mu_j^3)$ time when executed on an augmenting path P. Therefore, to bound the total time taken by AUGMENT, we will bound the total number of level j edges over all augmenting paths computed during phases of the algorithm. From Lemma 5.2, we have

$$\sum_{1 \leq k \leq t} \sum_{(u,v) \in N(P_k)} \mu_{uv}^2 = \tilde{\mathcal{O}}(n). \tag{5.6}$$

Each non-local edge (u, v) of level j contributes $\Omega(\mu_{uv}^2) = \Omega(\mu_j^2)$ towards the RHS of equation (5.6). As a result, there can be at most $\tilde{\mathcal{O}}(n/\mu_j^2)$ such edges in all augmenting paths computed during the phases of the algorithm.

Recall that two matching edges (a_i, b_i) and (a_k, b_k) are in the same class if they share the least common ancestor $\square$ and their representative pair $(\Psi_{a_i}, \Psi_{b_i}) \in \mathcal{W}_\square$ is the same as (Ψ_{a_k}, Ψ_{b_k}). Consider any augmenting path P, and consider any maximal sub-path S with

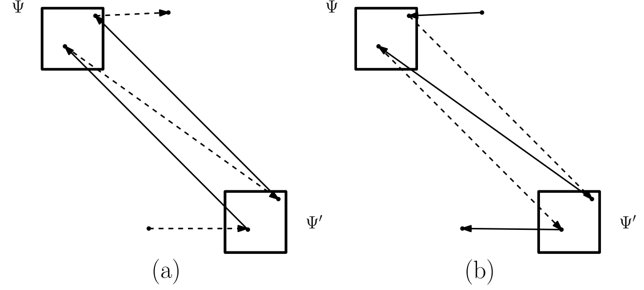

Figure 4: (a) A local path S of length 3 between the cells Ψ and Ψ'. Although S does not contain any non-local edges, augmentation will reduce the number of matching edges between the two cells by 1. (b) The number of matching edges between the cells Ψ and Ψ' can only increase when a local path containing at least one non-local edge between Ψ and Ψ' participates in an augmenting path.

the property that all its matching edges (resp. non-matching edges) belong to the same class with representative pair (Ψ, Ψ') (resp. (Ψ', Ψ)). We will call any such path a *local path*. Intuitively, all matching edges of S will belong to the same class and, upon augmentation, the non-matching edges of S will all enter the matching and belong to the same class. We say that S is a level j local path if all edges of S appear at level j. For any augmenting path P, let $L_j(P)$ be the set of level j local paths of P. It is easy to see that either: (1) S contains at least one non-local edge or (2) the first and the last edge of S are matching edges. In case (2), the number of matching edges that have (Ψ, Ψ') as their representative pair decreases by 1 after augmenting along P. Furthermore, new matching edges with representative (Ψ, Ψ') can only be created through an occurrence of case (1). Therefore, each occurrence of case (2) can be taxed on an occurrence of case (1). Combining this observation with the bound on the number of non-local edges gives:

$$\sum_{1 \leq k \leq t} |L_j(P_k)| = \tilde{\mathcal{O}}(n/\mu_j^2). \tag{5.7}$$

From the fact that each P_k is compact, there are at most $6|L_j(P_k)| + 1$ cells in $\mathcal{A}_j(P_k)$. From Lemma 5.2, there are $\tilde{\mathcal{O}}(n/\mu_i^2)$ augmenting paths found during phase i. Combining these observations with (5.7) gives the following for any level j,

$$\sum_{k=1}^{t} |\mathcal{A}_j(P_k)|\mu_j^3 = \tilde{\mathcal{O}}(\mu_j^3(n/\mu_i^2 + n/\mu_j^2)) = \tilde{\mathcal{O}}(n\mu_j).$$

When summing over all levels that contain affected pieces, the top level $\lfloor 2i/3 \rfloor$ dominates, which bounds

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

the total time for AUGMENT during phase i as,

$$\sum_{k=1}^{t}\sum_{j=0}^{\lfloor 2i/3\rfloor}|\mathcal{A}_j(P_k)|\mu_j^3=\tilde{\mathcal{O}}(n\mu_i^{2/3}). \tag{5.8}$$

Finally, we bound the time taken by the HUNGARIANSEARCH procedure during phase i. From Lemma 5.2, the number of unmatched vertices remaining at the beginning of phase i is $\mathcal{O}(n/\mu_i^2)$. This value also bounds the number of active cells during phase i. Therefore, the number of calls to HUNGARIANSEARCH during phase i is $\mathcal{O}(n/\mu_i^2)$. Each execution of HUNGARIANSEARCH during phase i takes $\tilde{\mathcal{O}}(\mu_i^{8/3}+\sum_{j=0}^{\lfloor 2i/3\rfloor}|\mathcal{A}_j(P)|\mu_j^3)$ time. Summing over all paths computed during phase i and applying (5.8) gives a total time of $\tilde{\mathcal{O}}(n\mu_i^{2/3})$ for the HUNGARIANSEARCH procedure during phase i. Combining the times taken by all data structure procedures during phase i gives a total time of $\tilde{\mathcal{O}}(n\mu_i^{2/3})$. The time taken for the last phase $\lceil 3\log n/4-1\rceil$ dominates, taking a total of $\tilde{\mathcal{O}}(n^{5/4})$ time.

6 Data Structure

6.1 Preliminaries To simplify the presentation of the data structure, we introduce additional notations and also present an auxiliary property that will be useful in proving the correctness of the data structure.

We define the *adjusted* cost of an edge $\Phi(a,b)$ as

$$\Phi(a,b)=d_Q(a,b)+\mu_{ab}^2 \quad \text{if } (a,b) \text{ is non-local.}$$
$$\Phi(a,b)=d_Q(a,b) \quad \text{otherwise.}$$

For any edge (a,b) of $\mathcal{G}$, we define its *net-cost* $\phi(a,b)$ as follows. If (a,b) is non-matching edge, its net-cost is $\phi(a,b)=\Phi(a,b)$. Otherwise, (a,b) if the edge is in the matching, we define $\phi(a,b)=-\Phi(a,b)$. For any set of edges S, we define its *net-cost* as $\phi(S)=\sum_{(a,b)\in M}\phi(a,b)$.

Recall that Q-feasibility is defined with respect to the graph $\mathcal{G}(A\cup B, A\times B)$. Also, the dual updates performed during the algorithm ensure that each point $b\in B$ is assigned a non-negative dual weight $y(b)$ and each point $a\in A$ is assigned a non-positive dual weight $y(a)$. For simplicity in exposition, we use this observation to restate the Q-feasibility constraints with respect to the residual graph $\mathcal{G}_M$. A matching M and a set of dual assignments $y(\cdot)$ is Q-feasible if for any edge (u,v) of the residual graph $\mathcal{G}_M$ directed from u to v,

$$|y(u)|-|y(v)|\le\phi(u,v),$$
$$|y(u)|-|y(v)|=\phi(u,v) \quad \text{if } (u,v) \text{ is local.}$$

For any non-local edge (u,v), we define its *slack* as $s(u,v)=\phi(u,v)-|y(u)|+|y(v)|$, i.e., how far the feasibility constraint for (u,v) is from being violated. Note that the slack on any edge is non-negative with local edges having a slack of zero. We say any edge is *admissible* with respect to a set of dual weights if it has a slack of zero. The admissible graph is simply the subgraph induced by the set of zero-slack edges. The advantage of redefining Q-feasibility conditions in this fashion is that it extends to any directed path in $\mathcal{G}_M$ as presented in the following lemma.

LEMMA 6.1. *Let $M,y(\cdot)$ be a Q-feasible matching and set of dual weights maintained by the algorithm. Let P be any alternating path with respect to M starting at a vertex u and ending at a vertex v. Then,*

$$|y(u)|-|y(v)|+\sum_{(u',v')\in P}s(u',v')=\phi(P).$$

For any phase i, we define our data structure for each active cell $\square^*\in G_i$. The data structure is based on the active tree $\mathcal{T}_{\square^*}$. We define a sublinear in n sized associated graph $\mathcal{AG}_\square$ for each cell $\square$ of $\mathcal{T}_{\square^*}$. This graph will help us compactly store the dual weights and help conduct the Hungarian Search, find augmenting paths and augment the matching along the path.

For any cell $\square$, let $A_\square=A\cap\square$ and $B_\square=B\cap\square$. For any set of cells X, we denote $A_X=\bigcup_{\square\in X}A_\square$ and $B_X=\bigcup_{\square\in X}B_\square$. For any cell $\square$ let $M_\square$ be the set of edges of M that have both endpoints contained in $\square$, and let $\mathcal{G}_{M_\square}$ be the vertex-induced subgraph of $(A_\square\cup B_\square)$ on $\mathcal{G}_M$. For simplicity in notation we use $\mathcal{G}_\square$ to denote $\mathcal{G}_{M_\square}$.

Recall that a cell $\square\in G_j$ is *sparse* if $|A_\square\cup B_\square|\le\mu_j^2$ or $\square\in G_0$ and full otherwise. For any cell $\square$ of $\mathcal{T}_{\square^*}$, our data structure constructs an *associated graph* $\mathcal{AG}_\square$. If $\square$ is sparse, the associated graph $\mathcal{AG}_\square$ is simply given by $\mathcal{G}_\square$. In the following, we define the associated graph for any full cell.

6.2 Vertices of the Associated Graph We define an associated graph for an arbitrary cell in the active tree $\mathcal{T}_{\square^*}$. For any cell $\square$ in the active tree, let $D(\square)$ denote the children of $\square$. We extend our definition to subcells as well. For any subcell $\xi\in\mathbb{G}[\square]$, let $\square'\in D(\square)$ be the cell that contains ξ. Let $D(\xi)=\{\xi'\mid\xi'\in\mathbb{G}[\square'] \text{ and } \xi'\subseteq\xi\}$.

If $\square$ is a full cell, for each of its children $\square'$, suppose $\square'$ is of level j. We cluster $A_{\square'}\cup B_{\square'}$ into $\tilde{\mathcal{O}}(\mu_j)$ clusters. We cluster points in such a way that all edges going between any two clusters X and Y, where X and Y are clusters for two different children $\square'$ and $\square''$ of $\square$, have the same net-cost. We create one vertex in $V_\square$ for every cluster of $\square'$ and repeat this for every child $\square'$ of $\square$. The clusters created here are similar to that in [26]. Recall

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

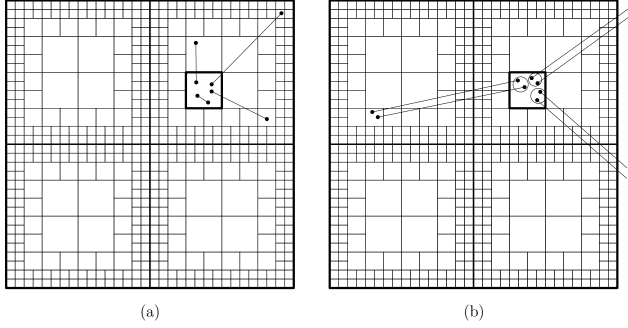

Figure 5: Internal and boundary clusters of a subcell ξ within the child $\square'$ of $\square$. (a) All vertices matched within $\square'$ are part of an internal cluster. (b) All vertices matched outside of $\square'$ are divided into boundary clusters based on WSPD pairs at higher levels.

that two matching edges (a_i, b_i) and (a_k, b_k) are in the same class if they share the least common ancestor $\square$ and their representative pair $(\Psi_{a_i}, \Psi_{b_i}) \in \mathcal{W}_\square$ is the same as (Ψ_{a_k}, Ψ_{b_k}). For any matched point a_i (resp. b_i), we refer to b_i (resp. a_i) as its *partner* point. For any $\xi \in \mathbb{G}[\square']$, we partition A_ξ and B_ξ into three types of clusters.

- *Free clusters*: All free points of A_ξ (resp. B_ξ) belong to a single cluster
$$A^F_\xi = A_F \cap \xi, \quad B^F_\xi = B_F \cap \xi.$$
- *Internal clusters*: All points of A_ξ (resp. B_ξ) whose partner point is also inside $\square'$ belong to a single cluster
$$A^I_\xi = \{a_i \in A_\xi \mid (a_i, b_i) \in M, b_i \in B_{\square'}\},$$
$$B^I_\xi = \{b_i \in B_\xi \mid (a_i, b_i) \in M, a_i \in A_{\square'}\}.$$
- *Boundary clusters*: Recollect that $\square' \in G_j$. All points of A_ξ (resp. B_ξ) whose partner points are outside $\square'$ are partitioned into *boundary clusters*. Two such vertices belong to the same boundary cluster if the matching edges incident on them belong to the same class. Note that all such matching edges have level at least j and are incident on at least one vertex of $A_\xi \cup B_\xi$. Any such matching edges are captured by one of $\tilde{\mathcal{O}}(1)$ many WSPD pairs given by the set $N^*(\xi)$. Since there is at most one class per WSPD pair, there are at most $\tilde{\mathcal{O}}(1)$ many boundary clusters per subcell. More specifically, for every $(\Psi_1, \Psi_2) \in N^*(\xi)$, we create a cluster,
$$A^{(\Psi_1,\Psi_2)}_\xi = \{a_i \in A_\xi \mid (a_i, b_i) \in M, b_i \in B_{\Psi_2}\},$$
$$B^{(\Psi_1,\Psi_2)}_\xi = \{b_i \in B_\xi \mid (a_i, b_i) \in M, a_i \in A_{\Psi_2}\}.$$

For every cell $\square'$ and any subcell $\xi \in \mathbb{G}[\square']$, there are a total of $\tilde{\mathcal{O}}(1)$ clusters. Therefore, the total number of clusters at $\square'$ is $\tilde{\mathcal{O}}(\mu_j)$. Let $\mathbb{X}_{\square'}$ be the set of clusters at $\square$ that are generated from its child $\square'$. The cluster set at $\square$ is simply $V_\square = \bigcup_{\square' \in D(\square)} \mathbb{X}_{\square'}$. We use $\mathcal{A}_\square$ (resp. $\mathcal{B}_\square$) to denote the vertices of type A (resp. type B) in $V_\square$; $V_\square = \mathcal{A}_\square \cup \mathcal{B}_\square$.

Next, we partition the clusters in $\mathbb{X}_{\square'}$ into two subsets called the *entry* and *exit* clusters respectively,

$$\mathbb{X}^{\downarrow}_{\square'} = \{B^F_\xi, A^I_\xi, B^{(\Psi_1,\Psi_2)}_\xi \mid \xi \in \mathbb{G}[\square'], (\Psi_1, \Psi_2) \in N^*(\xi)\},$$
$$\mathbb{X}^{\uparrow}_{\square'} = \{A^F_\xi, B^I_\xi, A^{(\Psi_1,\Psi_2)}_\xi \mid \xi \in \mathbb{G}[\square'], (\Psi_1, \Psi_2) \in N^*(\xi)\}.$$

We also denote all the clusters at $\square$ from $\square' \in D(\square)$ that contains points of A and B as $\mathbb{A}_{\square'}$ and $\mathbb{B}_{\square'}$ respectively; $\mathbb{X}_{\square'} = \mathbb{A}_{\square'} \cup \mathbb{B}_{\square'}$.

We next describe the significance of entry and exit clusters. For any directed path Π in $\mathcal{G}_{\square^*}$, let $\overline{\pi}$ be a maximal connected sub-path of Π that lies inside $\square$. Suppose $\overline{\pi}$ contains at least one edge. For the two endpoints p, q of $\overline{\pi}$, we refer to p as entry and q as exit point if $\overline{\pi}$ is directed from p to q. Then, it was shown in [26] that the entry point lies in an entry cluster and exit point lies in an exit cluster.

6.3 Relating Parent-Child Clusters

Let $\square$ be any node in $\mathcal{T}_{\square^*}$. Note that all internal nodes of the active tree except the root have four children. For any cell $\square$ of an active tree $\mathcal{T}_{\square^*}$, clusters are defined with respect to the subcells of its children. For any cell $\square$ of $\mathcal{T}_{\square^*}$, including the root, let ξ be a subcell of $\square' \in D(\square)$ and let $\square$ be a cell of level i. Then, we get the following relationship between clusters of $\square$ and $\square'$.

$$A^F_\xi = \bigcup_{\xi' \in D(\xi)} A^F_{\xi'}, \quad B^F_\xi = \bigcup_{\xi' \in D(\xi)} B^F_{\xi'},$$
$$A^{(\Psi_1,\Psi_2)}_\xi = \bigcup_{\xi' \in D(\xi)} A^{(\Psi_1,\Psi_2)}_{\xi'}, \forall (\Psi_1, \Psi_2) \in N^*(\xi),$$
$$B^{(\Psi_1,\Psi_2)}_\xi = \bigcup_{\xi' \in D(\xi)} B^{(\Psi_1,\Psi_2)}_{\xi'}, \forall (\Psi_1, \Psi_2) \in N^*(\xi),$$
$$A^I_\xi = \bigcup_{\xi' \in D(\xi)} (A^I_{\xi'} \cup (\bigcup_{(\Psi,\Psi') \in N^i(\xi')} A^{(\Psi,\Psi')}_{\xi'})),$$
$$B^I_\xi = \bigcup_{\xi' \in D(\xi)} (B^I_{\xi'} \cup (\bigcup_{(\Psi,\Psi') \in N^i(\xi')} B^{(\Psi,\Psi')}_{\xi'})).$$

For any cluster X defined at a full cell $\square$, we use the notation $D(X)$ to denote all the clusters at the children that combine to form X. Note that if X is a cluster generated at a subcell ξ of a leaf (i.e., sparse) cell

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

$\square' \in D(\square)$ of $\mathcal{T}_\square$, then we set $D(X)$ to be all the points that are contained in X. The following lemma whose proof is straightforward states the property of the above hierarchical clustering scheme.

LEMMA 6.2. *For any cell $\square \in Q$, let $\square_1, \square_2$ be two of its children. Let $X \in \mathbb{X}_{\square_1}$ and $Y \in \mathbb{X}_{\square_2}$. Then the net-costs of all edges in $X \times Y$ are the same in $\mathcal{G}_M$ and all such edges are oriented in the same direction — either all are oriented from B to A or all of them are oriented from A to B.*

6.4 Edges of the Associated Graph Given a full cell $\square$, we already defined the vertex set $V_\square$ for the associated graph. We have the following types of edges in the edge set $E_\square$ of the associated graph.

- *Internal Edges:* For any child $\square'$ of $\square$, we add edges from X to Y provided $X \in \mathbb{X}^{\downarrow}_{\square'}$ is an entry cluster of $\square'$ and $Y \in \mathbb{X}^{\uparrow}_{\square'}$ is an exit cluster of $\square'$.

- *Bridge Edges:* For any children $\square' \neq \square''$ of $\square$, for any two clusters X and Y where $X \in \mathbb{X}_{\square'}$ and $Y \in \mathbb{X}_{\square''}$, suppose $X \in \mathbb{B}_{\square'}$ and $Y \in \mathbb{A}_{\square''}$. We add an edge directed from Y to X (resp. X to Y) if, for every edge $(x, y) \in X \times Y$, (x, y) is a local (resp. non-local) edge. We continue to refer to such edges of the associated graph as local (resp. non-local) edges.

Bridge Edge Costs: Note that for any local bridge edge from cluster X to Y there is at least one matching edge say $(x, y) \in X \times Y$. We set the cost of (X, Y), denoted by $\phi(X, Y)$ to $\phi(x, y)$. For a non-local bridge edge (X, Y), every edge $(x, y) \in (X \times Y)$ has the same net-cost, which defines the net-cost of (X, Y), i.e., $\phi(X, Y) = \phi(x, y)$. Next, we describe the cost of an internal edge.

Internal Edge Costs: For any child $\square'$ of $\square$, and any internal edge $(X, Y) \in (\mathbb{X}^{\downarrow}_{\square} \times \mathbb{X}^{\uparrow}_{\square})$ in $E_\square$, we define its *projection* $P(X, Y)$. If $\square'$ is sparse, then $P(X, Y)$ is a minimum net-cost path in $\mathcal{G}_{\square'}$ from any $x \in X$ to any $y \in Y$. Otherwise, $\square'$ is full, and the projection $P(X, Y)$ is a minimum net-cost path through $\mathcal{AG}_{\square'}$ from any $X' \in D(X)$ to any $Y' \in D(Y)$. In either case, the net-cost of (X, Y) is equal to the net-cost of its projection; i.e., $\phi(X, Y) = \phi(P(X, Y))$. The following lemma, which follows from a simple induction on the recursive definition of projection, states that any internal edge $(X, Y) \in (\mathbb{X}^{\downarrow}_{\square'} \times \mathbb{X}^{\uparrow}_{\square'})$ corresponds to a minimum net-cost path from X to Y in $\mathcal{G}_{\square'}$.

LEMMA 6.3. *For any $u, v \in \square$ let $\Pi_{u,v,\square}$ be a minimum net-cost alternating path in $\mathcal{G}_\square$ from u to v. For any internal edge $(X, Y) \in (\mathbb{X}^{\downarrow}_{\square}, \mathbb{X}^{\uparrow}_{\square})$, consider $(x, y) = \mathrm{argmin}_{(x', y') \in X \times Y}\, \phi(\Pi_{x', y', \square})$. Then $\phi(X, Y) = \phi(\Pi_{u,v,\square})$.*

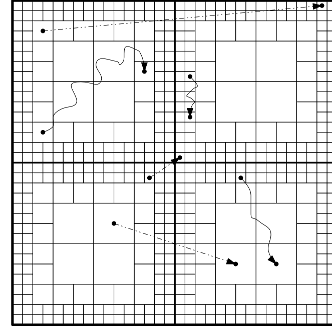

Figure 6: Examples of edges of $E_\square$. Internal edges (solid) represent shortest paths between clusters within a child of $\square$. Bridge edges (dashed) are between clusters in two different children of $\square$.

6.5 Compressed Feasibility Consider any active cell $\square^*$ of the quadtree and the active tree $\mathcal{T}_{\square^*}$ rooted at $\square^*$. Consider an assignment of dual weights $y(\cdot)$ to the vertices of $V_\square$ for all cells $\square \in \mathcal{T}_{\square^*}$. We say that $M_{\square^*}$ along with these dual weights are compressed feasible if for every cell $\square$ in $\mathcal{T}_{\square^*}$.

(C1) For every edge directed from X to Y in $\mathcal{AG}_\square$,

$$|y(X)| - |y(Y)| \leq \phi(X, Y),$$
$$|y(X)| - |y(Y)| = \phi(X, Y) \quad \text{if } (X, Y) \text{ is local with respect to } M_\square.$$

(C2) If $\square$ is full, then for each exit cluster $X \in \mathbb{X}^{\uparrow}_{\square}$, for any $X' \in D(X)$, $|y(X')| \leq |y(X)|$.

If $\square$ is sparse, then we are at a leaf node of the active tree and only condition (C1) applies. Condition (C1) implies that $M_\square$ and $y(\cdot)$ are Q-feasible when $\square$ is sparse.

We define slack of any edge (bridge or internal) directed from X to Y, denoted by $s(X, Y)$, as $\phi(X, Y) - |y(X)| + |y(Y)|$. From (C1), it follows that the slack of any edge is non-negative. We define a slack-weighted associated graph, denoted by $\mathcal{AG}'_\square$, to be identical to the associated graph $\mathcal{AG}_\square$, but where the weight of any edge (X, Y) is its slack $s(X, Y)$.

We introduce two procedures, namely SYNC and CONSTRUCT. Both these procedures will be used to support BUILD, GENERATEDUALS, HUNGARIANSEARCH and AUGMENT operations.

6.6 CONSTRUCT Procedure In this section, we present the CONSTRUCT procedure, which will be used

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

to compute the internal edges of an associated graph. CONSTRUCT accepts a cell $\square'$, such that $\square' \neq \square^*$ and $\mathcal{AG}_{\square'}$ has already been computed, along with dual weights $y(\cdot)$ for all vertices of $V_{\square'}$. It assumes that $M_{\square'}, y(\cdot)$ satisfy the compressed feasibility conditions. Let $\square$ be the parent of $\square'$ in $\mathcal{T}_{\square^*}$. The procedure computes the internal edges of $\mathbb{X}^{\downarrow}_{\square'} \times \mathbb{X}^{\uparrow}_{\square'}$ in $\mathcal{AG}_{\square}$. It also assigns dual weights to the vertices of $V_{\square}$ that correspond to clusters generated for the subcells of $\square'$.

We describe the process for building the internal edges going out of each cluster $X \in \mathbb{X}^{\downarrow}_{\square'}$. We add an additional vertex s to $\mathcal{AG}'_{\square'}$ and add an edge from s to each cluster $X' \in D(X)$ with a cost equal to $|y(X')|$. After creating this *augmented associated graph*, we simply execute Dijkstra's algorithm from s to find the shortest path distance from s to every node in $V_{\square'}$. Let d_v denote the shortest path distance from s to v in the augmented associated graph. For each exit cluster $Y \in \mathbb{X}^{\uparrow}_{\square'}$, we create an internal edge from X to Y in $\mathcal{AG}_{\square}$ and set its cost to be $\min_{Y' \in D(Y)}(d_{Y'} - |y(Y')|)$. We repeat this procedure for each entry cluster.

This completes the description how to construct the internal edges of $\mathcal{AG}_{\square}$ in $\square'$. We next assign dual weights to each cluster $X \in \mathbb{X}_{\square'}$ as follows: If X is an entry cluster, let $X' = \text{argmin}_{Y \in D(X)} |y(Y)|$. Otherwise, X is an exit cluster, and we let $X' = \text{argmax}_{Y \in D(X)} |y(Y)|$. In either case, we set $y(X) \leftarrow y(X')$. Next, we show that the CONSTRUCT procedure correctly assigns the net-cost of edges in $\mathbb{X}^{\downarrow}_{\square'} \times \mathbb{X}^{\uparrow}_{\square'}$.

LEMMA 6.4. *Let $X \in \mathbb{X}^{\downarrow}_{\square'}$ and $Y \in \mathbb{X}^{\uparrow}_{\square'}$ be a pair of clusters of $\square'$ that form an internal edge $(X, Y) \in E_{\square}$. Then the* CONSTRUCT *procedure ensures that $\phi(X, Y) = \phi(P(X, Y)))$.*

In the following Lemma, we argue that the internal edges of $\square'$ in $\mathcal{AG}_{\square}$ are feasible after CONSTRUCT is called on $\square'$.

LEMMA 6.5. *After* CONSTRUCT *is called on a cell $\square'$ with parent $\square$, then, for every internal edge $(X, Y) \in \mathbb{X}^{\downarrow}_{\square'} \times \mathbb{X}^{\uparrow}_{\square'}$ of $E_{\square}$, we have, $|y(X)| - |y(Y)| \leq \phi(X, Y)$.*

Efficiency of CONSTRUCT: Next, we bound the time taken for a single call to CONSTRUCT on a cell $\square \in \mathcal{T}_{\square^*}$. Assume that $\square$ appears at level j. The CONSTRUCT procedure executes a Dijkstra search from each of the $|\mathbb{X}_{\square}| = \tilde{\mathcal{O}}(\mu_j)$ clusters of $\square$. If $\square$ is full, then each Dijkstra search takes $\tilde{\mathcal{O}}(|E_{\square}|) = \tilde{\mathcal{O}}(|V_{\square}|^2) = \tilde{\mathcal{O}}(\mu_j^2)$ time. If $\square$ is sparse, then each Dijkstra search can be executed efficiently in $\tilde{\mathcal{O}}(|\mathbb{X}_{\square}| + |A_{\square} \cup B_{\square}|)$ time using the fact that the edges of $\mathcal{G}_M$ outgoing from any vertex v belong to only $\tilde{\mathcal{O}}(1)$ different WPSD pairs; the same technique was used for the Hungarian search in Section 5. Since $\square$ is sparse, $|A_{\square} \cup B_{\square}| \leq \mu_j^2$, and each Dijkstra search takes $\tilde{\mathcal{O}}(\mu_j^2)$ time. The CONSTRUCT procedure executes $|\mathbb{X}^{\downarrow}_{\square}| = \tilde{\mathcal{O}}(\mu_j)$ Dijkstra searches, and each Dijkstra search takes $\tilde{\mathcal{O}}(\mu_j^2)$ time, so the total time taken by CONSTRUCT is $\tilde{\mathcal{O}}(\mu_j^3)$ for any cell of level j. This gives the following Lemma.

LEMMA 6.6. *Any execution of* CONSTRUCT *on a cell $\square$ of layer j takes $\tilde{\mathcal{O}}(\mu_j^3)$ time. Furthermore, if $\square$ is sparse, the time taken can be bounded by $\tilde{\mathcal{O}}(\mu_j(\mu_j + |A_{\square}| + |B_{\square}|))$.*

6.7 SYNC Procedure

For an active cell $\square^*$ and a compressed feasible matching $M_{\square^*}$ along with a set of dual weights $y(\cdot)$, the SYNC procedure takes the updated dual weights on clusters of $\mathbb{X}_{\square}$ at any non-root cell $\square \in \mathcal{T}_{\square^*}$ and uses them to update the dual weights of $V_{\square}$ such that the matching continues to be compressed feasible, and,

(T1) For any entry cluster $X \in \mathbb{X}^{\downarrow}_{\square}$, and for any $X' \in D(X)$, $|y(X')| \geq |y(X)|$.

(T2) For any free or boundary cluster $X \in \mathbb{X}_{\square}$, and for any $X' \in D(X)$, $y(X') = y(X)$.

The SYNC procedure consists of executing the following algorithm for each entry cluster $X \in \mathbb{X}^{\downarrow}_{\square}$: We create a new vertex s and add an edge from s to each vertex $X' \in D(X)$. We assign a weight $|y(X')|$ to the edge from s to X'. Then, we execute Dijkstra's algorithm starting from s. Let d_v be the shortest path distance from s to v as computed by Dijkstra's algorithm. For any vertex v with $d_v < |y(X)|$, if $v \in \mathcal{B}_{\square}$ we update the dual weight to $y(v) \leftarrow y(v) + |y(X)| - d_v$. Otherwise, $v \in \mathcal{A}_{\square}$, and we update the dual weight $y(v) \leftarrow y(v) - |y(X)| + d_v$. Note that in both cases the magnitude of the dual weight increases by $|y(X)| - d_v$. The dual weight of every other vertex with $d_v \geq |y(X)|$ does not change. This completes the description of the algorithm initiated with respect to X.

Note that for any cluster $X' \in D(X)$ if $|y(X')| \geq |y(X)|$, then the procedure will only further increase the magnitude of $y(X')$ and so, (T1) holds. If, on the other hand, $|y(X')| < |y(X)|$, then the length of the edge from s to X' is $|y(X')|$, and so the shortest path distance $d_{X'} \leq |y(X')| < |y(X)|$. The magnitude of the dual weight of X' increases by $|y(X)| - d_{X'} \geq |y(X)| - |y(X')|$ implying that the new magnitude of $y(X')$ is at least the magnitude of $y(X)$. Therefore (T1) holds for any entry cluster.

After we execute this for all entry clusters, we perform the following dual adjustment: For any $X \in \mathbb{X}_{\square}$, and for any $X' \in D(X)$, we will explicitly update the dual weight $y(X')$ to match $y(X)$. Therefore, (T2) holds after the execution of SYNC.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

To prove the correctness of SYNC one can show that the updated dual weights satisfy the compressed feasibility conditions. Additionally, Lemma 6.7 shows that applying SYNC to compressed edges with zero slack will lead to a path containing admissible edges; see [21] for a proof.

LEMMA 6.7. *Let $\square$ be a level i cell. For any internal edge $(X,Y) \in \mathbb{X}^{\downarrow}_{\square} \times \mathbb{X}^{\uparrow}_{\square}$ in the associated graph $\mathcal{AG}_{\hat{\square}}$ of the parent $\hat{\square}$ of $\square$, suppose $s(X,Y)$ is 0. We can recursively apply* SYNC *on all internal edges of $P(X,Y)$ to obtain its projection $\Pi_{u,v,\square}$ with $u \in X$ and $v \in Y$. This projection will be an admissible path. For every vertex p in $\Pi_{u,v,\square}$, let $\mathbb{P}(p)$ be all the clusters for cells of level i or lower that contain the point p. Then, for every $v' \in \mathbb{P}(p)$, $y(v') = y(p)$.*

Efficiency Analysis of SYNC: Next, we bound the time taken for a single call to SYNC executed on a cell $\square$ that updates the dual weights of $V_\square$. The argument is nearly identical to that used for CONSTRUCT. Assume that $\square$ appears at level j. The SYNC procedure executes a Dijkstra search once from each of the $\tilde{\mathcal{O}}(\mu_j)$ entry clusters of $\mathbb{X}_\square$. If $\square$ is full, then each Dijkstra search takes time $\tilde{\mathcal{O}}(|E_\square|) = \tilde{\mathcal{O}}(|V_\square|^2) = \tilde{\mathcal{O}}(\mu_j^2)$ time. If $\square$ is sparse, then each Dijkstra search can be executed efficiently in $\tilde{\mathcal{O}}(|\mathbb{X}_\square| + |A_\square \cup B_\square|)$ time. Since $\square$ is sparse, $|A_\square \cup B_\square| \le \mu_j^2$, and each Dijkstra search takes $\tilde{\mathcal{O}}(\mu_j^2)$ time. This gives the following:

LEMMA 6.8. *Any execution of* SYNC *on a cell $\square$ of layer j takes $\tilde{\mathcal{O}}(\mu_j^3)$ time. Furthermore, if $\square$ is sparse, the time taken can be bounded by $\tilde{\mathcal{O}}(\mu_j(\mu_j + |A_\square| + |B_\square|))$.*

6.8 Data Structure Operations

For any phase i, we present the implementation of the four operations supported by the data structure using the SYNC and CONSTRUCT procedures. Before we describe the operations, we will state an additional property that the compressed feasible matching maintained by the data structure satisfies. In any phase $i \ge 1$, suppose that $M_{\square^*}$, $y(\cdot)$ is a compressed feasible matching with the additional condition being satisfied:

(J) For each vertex $b \in \mathcal{B}_{\square^*}$, $y(b) \ge 0$, and for each $a \in \mathcal{A}_{\square^*}$, $y(a) \le 0$. Furthermore, let $y_{\max} = \max_{v \in \mathcal{B}_{\square^*}} y(v)$. For every free vertex $b \in \mathcal{B}_{\square^*}$, $y(b) = y_{\max}$ and $\mu_{i-1}^2 \le y_{\max} \le \mu_i^2$. For every free cluster $a \in \mathcal{A}_{\square^*}$, $y(a) = 0$.

As we show in Section 6.8.2, a compressed feasible matching that satisfies (J) can be converted to an associated Q-feasible matching that satisfies (I1) and (I2). Therefore, it suffices to maintain (J) during the execution of our algorithm.

6.8.1 BUILD Operation

As input, the BUILD operation takes a Q-feasible matching $M_{\square^*}$ and set of dual weights $y(\cdot)$ on the vertices of $A_{\square^*} \cup B_{\square^*}$. We execute the CONSTRUCT procedure on every non-root cell $\square$ of $\mathcal{T}_{\square^*}$ in the order of their level in Q, processing lower layers first. This ensures that, when CONSTRUCT is called on $\square$, the associated graph $\mathcal{AG}_\square$ has already been computed, along with the dual weights for vertices of $V_\square$. After CONSTRUCT is called on all pieces of $\square^*$, the result is an associated graph $\mathcal{AG}_\square$ for every full cell $\square \in \mathcal{T}_{\square^*}$ and dual weights $y(\cdot)$ for all vertices of $\bigcup_{\square \in \mathcal{T}_{\square^*}} V_\square$. It can be argued that this set of dual weights is compressed feasible with respect to $M_{\square^*}$. Furthermore, upon applying the BUILD procedure on any Q-feasible matching that satisfies (I1) and (I2), the resulting compressed feasible matching will satisfy (J); see [21] for a proof.

Execution Time for BUILD: We show that the time taken by BUILD during phase i is $\tilde{\mathcal{O}}(n\mu_i^{2/3})$. During BUILD, CONSTRUCT is called on all non-root cells of $\mathcal{T}_{\square^*}$ for each full active cell $\square^*$. We assign each non-root cell $\square \in \mathcal{T}_{\square^*}$ to one of four categories: (a) $\square$ is full. (b) $\square$ is sparse and the parent of $\square$ in $\mathcal{T}_{\square^*}$ is a full cell that is not the root $\square^*$. (c) $\square$ is sparse, its parent in $\mathcal{T}_{\square^*}$ is the root $\square^*$, and $|A_\square \cup B_\square| \le \mu_i^{2/3}$. (d) $\square$ is sparse, its parent in $\mathcal{T}_{\square^*}$ is the root $\square^*$, and $|A_\square \cup B_\square| > \mu_i^{2/3}$. We separately bound the total time taken for a single CONSTRUCT call on every cell in each of the four categories, over all active cells for phase i, showing that the time taken is $\tilde{\mathcal{O}}(n\mu_i^{2/3})$.

First, we bound the time taken by cells of category (a). We bound the the time for a CONSTRUCT call on all full cells in some grid G_j. Since these full cells together contain at most n points, the total number of full cells in G_j is bounded by $\tilde{\mathcal{O}}(n/\mu_j^2)$. A CONSTRUCT call on a full cell $\square \in G_j$ takes $\tilde{\mathcal{O}}(\mu_j^3)$ time. Therefore, the total time taken for all full cells of G_j is $\tilde{\mathcal{O}}(n\mu_j)$. During phase i, CONSTRUCT is only called on cells of G_j where $j \le \lfloor 2i/3 \rfloor$. The time taken by $G_{\lfloor 2i/3 \rfloor}$ dominates, taking $\tilde{\mathcal{O}}(n\mu_i^{2/3})$ time. This completes the bound on cells in category (a).

Next, we bound the time taken for category (b). If a sparse cell $\square$ of level j in an active tree has a non-root parent $\square'$ in level $j+1$, then its parent $\square'$ must fall into category (a). The time taken for a call to CONSTRUCT on $\square$ is $\tilde{\mathcal{O}}(\mu_j^3) = \tilde{\mathcal{O}}(\mu_{j+1}^3)$, which can be taxed on the time taken to execute CONSTRUCT on the parent $\square'$. Specifically, since each non-root cell $\square'$ in the active tree has at most 4 children in the active tree, the time taken for a CONSTRUCT call on all sparse children of $\square'$ is $\tilde{\mathcal{O}}(\mu_{j+1}^3)$, which is also the bound on the time taken for CONSTRUCT on $\square'$ itself. Therefore, the total time taken by category (b) is bounded by the time taken by

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

(a), and is $\tilde{\mathcal{O}}(n\mu_i^{2/3})$.

Now we bound the time for category (c). All cells of category (c) are pieces of some active tree, so we begin by bounding the total number of pieces over all active cells. Since these cells together contain at most n points, the number of full active cells during phase i is $\tilde{\mathcal{O}}(n/\mu_i^2)$. Each such active cell $\square^*$ has its pieces in grid $G_{\lfloor 2i/3 \rfloor}$. Since $\square^*$ has diameter $\tilde{\mathcal{O}}(\mu_i^2)$ and each piece of $\square^*$ has diameter $\Omega(\mu_{\lfloor 2i/3 \rfloor}^2/\text{poly}\{\log n, 1/\varepsilon\}) = \Omega(\mu_i^{4/3}/\text{poly}\{\log n, 1/\varepsilon\})$, $\square^*$ has $\tilde{\mathcal{O}}((\mu_i^{2/3})^2) = \tilde{\mathcal{O}}(\mu_i^{4/3})$ pieces. Summing over all phase i active full cells gives a total of $\tilde{\mathcal{O}}(n/\mu_i^{2/3})$ pieces. The time taken by a single CONSTRUCT call on one of these pieces $\square$ is $\tilde{\mathcal{O}}(\mu_{\lfloor 2i/3 \rfloor}|A_\square \cup B_\square| + \mu_{\lfloor 2i/3 \rfloor}^2) = \tilde{\mathcal{O}}(\mu_i^{2/3}|A_\square \cup B_\square| + \mu_i^{4/3})$. However, since $|A_\square \cup B_\square| \leq \mu_i^{2/3}$, we can rewrite the time taken for a single CONSTRUCT call as $\tilde{\mathcal{O}}(\mu_i^{4/3})$. Summing over all $\tilde{\mathcal{O}}(n/\mu_i^{2/3})$ pieces of category (c) gives a total time of $\tilde{\mathcal{O}}(n\mu_i^{2/3})$ as desired.

Finally, we bound the time taken for category (d). The time taken for a single CONSTRUCT call on a cell $\square$ of category (d) is $\tilde{\mathcal{O}}(|A_\square \cup B_\square|\mu_i^{2/3} + \mu_i^{4/3})$. However, since $|A_\square \cup B_\square| > \mu_i^{2/3}$, the first term dominates, and we can rewrite the time taken by CONSTRUCT on $\square$ as $\tilde{\mathcal{O}}(|A_\square \cup B_\square|\mu_i^{2/3})$. Summing over all such cells of category (d) gives a total time of $\tilde{\mathcal{O}}(n\mu_i^{2/3})$.

6.8.2 GENERATEDUALS Operation The GENERATEDUALS procedure simply consists of recursively calling the SYNC procedure on all non-root cells of $\square^*$, processing cells closest to the root of $\mathcal{T}_{\square^*}$ first. This process generates a set of dual weights $y(\cdot)$ for the vertices of $A_{\square^*} \cup B_{\square^*}$. It can be shown that after executing this GENERATEDUALS procedure, $M_{\square^*}, y(\cdot)$ are Q-feasible, meaning (I1) and (I2) hold. Since CONSTRUCT and SYNC have the same time bounds from Lemmas 6.6 and 6.8, the time taken by GENERATEDUALS can be bounded by the time taken by BUILD.

6.8.3 HUNGARIANSEARCH Operation This procedure takes a compressed feasible matching $M_{\square^*}, y(\cdot)$ that also satisfies (J) as input. It then conducts a search identical to Hungarian search on the associated graph of $\square^*$. The search procedure adjusts the dual weights of the vertices of $V_{\square^*}$ so that we have a path consisting of admissible edges. Once an admissible path is found in $\mathcal{AG}_{\square^*}$, the procedure projects this path to find an augmenting path of admissible edges in $\mathcal{G}_{\square^*}$ by recursively applying the SYNC procedure. We describe the details of the procedure in two parts. First, we describe the dual adjustments conducted by the HUNGARIANSEARCH, and then we describe how the procedure projects the path. We show that (J) continues to hold after the execution of HUNGARIANSEARCH.

Dual Adjustments: Recall that $\mathcal{A}_{\square^*}$ (resp. $\mathcal{B}_{\square^*}$) denotes the set of vertices of type A (resp. type B) in $V_{\square^*}$. Let $\mathcal{A}_F$ (resp. $\mathcal{B}_F$) be the set of free vertex clusters of $\mathcal{A}_{\square^*}$ (resp. $\mathcal{B}_{\square^*}$). We add a vertex s to the graph $\mathcal{AG}'_{\square^*}$ and add an edge from s to every free cluster of $\mathcal{B}_F$. The weight associated with this edge is 0. We set $\ell_{\max} = \mu_i^2 - \max_{X \in \mathcal{B}_{\square^*}} y(X)$. We then execute a Dijkstra's search to compute the shortest path distance from s to every vertex in $V_{\square^*}$. For any $v \in V_{\square^*}$, let ℓ_v be the shortest path distance from s. Let $\ell = \min_{X \in \mathcal{A}_F} \ell_X$. If $\ell > \ell_{\max}$, we set $\ell = \ell_{\max}$ and continue. For every vertex $v \in V_{\square^*}$ with $\ell_v \leq \ell$, we update the dual weight as follows. If $v \in \mathcal{B}_{\square^*}$, we increase the dual weight $y(v) \leftarrow y(v) + \ell - \ell_v$. Otherwise, if $v \in \mathcal{A}_{\square^*}$, we reduce the dual weight $y(v) \leftarrow y(v) - \ell + \ell_v$. This completes the description of the dual weight changes. These dual adjustments will make some of the edges on the shortest path tree have a slack of zero. If $\ell = \ell_{\max}$, the dual weight of every free cluster of type B would be updated to μ_i^2 and we return without finding an augmenting path. Otherwise, the dual adjustments will maintain compressed feasibility and create an admissible path P from a free cluster $Z \in \mathcal{B}_F$ to a free cluster Z' of $\mathcal{A}_F$ inside the associated graph $\mathcal{AG}_{\square^*}$. Using a relatively straight-forward argument, one can show that these dual adjustments do not violate the compressed feasibility conditions.

Projecting an Augmenting Path: The dual adjustment ensures that there is some admissible augmenting path P in $\mathcal{AG}_{\square^*}$. We create an augmenting admissible augmenting path in $\mathcal{G}_{\square^*}$ from some free vertex $b \in Z$ to $a \in Z'$ as follows: For any internal edge (U, V) in P, we can recursively use SYNC (Lemma 6.7) to retrieve an admissible path $\Pi_{u',v',\square^*}$ where $u' \in U$ and $v' \in V$. We make u' (resp. v') the representative of U (resp. V) and denote it by $r(U)$ (resp. $r(V)$). For every vertex Y on the path P that does not have a representative, we choose an arbitrary vertex $p \in Y$ as its representative, $p = r(Y)$. Note that P cannot have any vertex with two internal edges incident on it. Next, for any bridge edge (x, y) in P, we show how to connect their representatives. Suppose the bridge edge (x, y) is non-local edge. Then, we connect $r(x)$ and $r(y)$ directly by a non-local edge in $\mathcal{G}_{\square^*}$. Otherwise, suppose (x, y) is a local bridge edge. In this case, if $r(x)$ is matched to $r(y)$, we simply add the matching edge between them. Otherwise, if $r(x)$ is matched to x' and $r(y)$ is matched to y', the edges $(r(x), x')$, (x', y') and $(y', r(y))$ are all local and admissible. We add them the three edges in this order to connect $r(x)$ to $r(y)$. The resulting path obtained is a compact admissible path from a free vertex in B_F to

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

a free vertex in A_F as desired.

Note that the input compressed feasible matching satisfied (J) and the dual weight of every free cluster v in $\mathcal{B}_{\square^*}$ is $y_{\max}$. The dual adjustments conducted by the HungarianSearch procedure will not decrease the dual weights of any vertex $v \in \mathcal{B}_{\square^*}$ and will not increase the dual weight of any vertex $v \in \mathcal{A}_{\square^*}$. Furthermore, each dual adjustment conducted by the HungarianSearch procedure increases the dual weight of all free clusters of $\mathcal{B}_{\square^*}$ by ℓ which is the largest increase among all clusters. Therefore, the new dual weight of free clusters is $y_{\max} + \ell$ which is the largest among all vertices of $\mathcal{B}_{\square^*}$. Finally, by definition, every free vertex cluster v of $\mathcal{A}_{\square^*}$ has $\ell_v \geq \ell$ and, therefore, $y(v)$ remains 0. In conclusion, after the execution of HungarianSearch procedure (J) continues to hold. It can also be shown that the projected augmenting path is simple; for a proof, see the full version of the paper [21].

Efficiency of HungarianSearch: Next, we bound the time taken by the HungarianSearch procedure. First, we bound the time taken for the Dijkstra search over $\mathcal{AG}'_{\square^*}$ during some phase i. The root cell $\square^*$ has a diameter of $\tilde{\mathcal{O}}(\mu_i^2)$, and each of its pieces have a diameter of $\tilde{\mathcal{O}}(\mu_{\lfloor 2i/3 \rfloor}^2) = \tilde{\mathcal{O}}(\mu_i^{4/3})$. Therefore, there are $\tilde{\mathcal{O}}((\mu_i^2/\mu_i^{4/3})^2) = \tilde{\mathcal{O}}(\mu_i^{4/3}))$ pieces of $\square^*$. Each piece contains $\tilde{\mathcal{O}}(\mu_i^{2/3})$ vertices in $V_{\square^*}$ and $\tilde{\mathcal{O}}(\mu_i^{4/3})$ internal edges in $E_{\square^*}$. The number of bridge edges in $E_{\square^*}$ could be much higher, but we observe that, by using the WSPD, the bridge edges incident on every vertex of $V_\square$ can be divided into only $\tilde{\mathcal{O}}(1)$ groups where the edges of each group have the same net-cost and direction. A similar technique is used for the Hungarian search described in Section 5. Therefore, the Dijkstra search over $\mathcal{AG}'_{\square^*}$ can be executed in time near-linear in the number of internal edges and vertices of $\mathcal{AG}'_{\square^*}$, i.e., $\tilde{\mathcal{O}}(\mu_i^{8/3})$ time.

After executing the Dijkstra search over $\mathcal{AG}'_{\square^*}$, the HungarianSearch procedure executes the Sync procedure to produce an admissible augmenting path P in $\mathcal{G}_M$. During this process, Sync only needs to be executed once per affected cell $\square \in \mathcal{A}(P)$. From Lemma 6.8, each execution of Sync on a cell of level j takes $\tilde{\mathcal{O}}(\mu_j^3)$ time. Recall that Sync is not called on any cell with level higher than $\lfloor 2i/3 \rfloor$, i.e., the level of the pieces of $\square^*$. Therefore, the total time taken by the executions of the Sync procedure can be expressed as:

$$\tilde{\mathcal{O}}\left(\sum_{j=0}^{\lfloor 2i/3 \rfloor} |\mathcal{A}_j(P)| \mu_j^3\right).$$

Combining this with the time taken by the Dijkstra search gives the following bound on the time taken by the HungarianSearch procedure:

$$\tilde{\mathcal{O}}\left(\mu_i^{8/3} + \sum_{j=0}^{\lfloor 2i/3 \rfloor} |\mathcal{A}_j(P)| \mu_j^3\right).$$

6.8.4 Augment Operation

The Augment procedure accepts an admissible augmenting path P in $\mathcal{G}_M$. It then augments M along P, and updates the data structure accordingly. To augment M along P, we set $M \leftarrow M \oplus P$ and perform very similar dual weight changes to those described in Section 5. For any edge (a, b) that was non-local prior to augmentation and became local after augmentation, let $\square$ be the least common ancestor of a and b in Q. If there is a local bridge edge $(X, Y) \in E_\square$ prior to augmentation such that a enters X and b enters Y through augmentation, we simply set $y(a) \leftarrow y(X)$ and $y(b) \leftarrow y(Y)$. Otherwise, if no such local edge existed, we set $y(a) \leftarrow y(a) - \mu_{ab}^2$. Using similar arguments to those given in Section 5, it can be shown that this dual weight assignment only decreases the dual weights of $y(a)$ and $y(b)$.

After augmenting along P, the data structure must perform updates to account for the changes to the matching. Recall that the set $\mathcal{A}(P)$ of affected cells contains all non-root cells of $\mathcal{T}_{\square^*}$ that contain at least one vertex of P. To update the data structure, the procedure executes the Construct procedure on all cells of $\mathcal{A}(P)$, processing cells at lower layers of Q first.

Efficiency of Augment: To bound the efficiency of the Augment procedure, we consider the most expensive portion, which is the time taken for the calls to the Construct procedure on all affected pieces. Consider an execution of Augment that produced an augmenting path P. Recall that, from Lemma 6.6, the time taken for a single call to Construct on a cell of level j is $\tilde{\mathcal{O}}(\mu_j^3)$, which matches the time taken for the calls to the Sync procedure during the execution of HungarianSearch that generated P. Using an identical argument, we can conclude that the total time taken by Augment is:

$$\tilde{\mathcal{O}}\left(\sum_{j=0}^{\lfloor 2i/3 \rfloor} |\mathcal{A}_j(P)| \mu_j^3\right).$$

It can be shown that the Augment procedure will not violate compressed feasibility; see the full version [21].

References

[1] P. K. Agarwal and K. R. Varadarajan, A near-linear constant-factor approximation for Euclidean bipartite matching?, *Proc. 20th Annual ACM Symposium on Computational Geometry*, 2004, pp. 247–252.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

[2] J. Altschuler, F. Bach, A. Rudi, and J. Weed, Approximating the Quadratic Transportation Metric in Near-Linear Time, *arXiv preprint arXiv:1810.10046*, (2018).
[3] J. Altschuler, J. Weed, and P. Rigollet, Near-linear time approximation algorithms for optimal transport via sinkhorn iteration, *Proc. Advances in Neural Information Processing Systems 30*, 2017, pp. 1964–1974.
[4] M. Arjovsky, S. Chintala, and L. Bottou, Wasserstein generative adversarial networks, *Proc. 34th International Conference on Machine Learning*, 2017, pp. 214–223.
[5] 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.
[6] M. Cuturi, Sinkhorn distances: Lightspeed computation of optimal transport, *Proc. Advances in Neural Information Processing Systems 26*, 2013, pp. 2292–2300.
[7] F. De Goes, D. Cohen-Steiner, P. Alliez, and M. Desbrun, An optimal transport approach to robust reconstruction and simplification of 2D shapes, *Computer Graphics Forum*, Vol. 30, Wiley Online Library, 2011, pp. 1593–1602.
[8] P. Dvurechensky, A. Gasnikov, and A. Kroshnin, Computational optimal transport: Complexity by accelerated gradient descent is better than by sinkhorn's algorithm, *Proc. 35th International Conference on Machine Learning*, 2018, pp. 1367–1376.
[9] A. Efrat, A. Itai, and M. J. Katz, Geometry helps in bottleneck matching and related problems, *Algorithmica*, 31 (2001), 1–28.
[10] J. Fakcharoenphol and S. Rao, Planar graphs, negative weight edges, shortest paths, and near linear time, *Journal of Computer and System Sciences*, 72 (2006), 868–889.
[11] K. Fox and J. Lu, A near-linear time approximation scheme for geometric transportation with arbitrary supplies and spread, *Proc. 36th Annual Symposium on Computational Geometry*, 2020, pp. 45:1–45:18.
[12] H. N. Gabow and R. Tarjan, Faster scaling algorithms for network problems, *SIAM J. Comput.*, 18 (1989), 1013–1036.
[13] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, Improved training of wasserstein GANs, *Proc. Advances in Neural Information Processing Systems 30*, 2017, pp. 5767–5777.
[14] S. Har-Peled, *Geometric approximation algorithms*, American Mathematical Soc., 2011.
[15] P. Indyk, A near linear time constant factor approximation for Euclidean bichromatic matching (cost), *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007, pp. 39–42.
[16] A. B. Khesin, A. Nikolov, and D. Paramonov, Preconditioning for the geometric transportation problem, *Proc. 35th Annual Symposium on Computational Geometry*, 2019, pp. 15:1–15:14.
[17] H. Kuhn, Variants of the hungarian method for assignment problems, *Naval Research Logistics*, 3 (1956), 253–258.
[18] N. Lahn, D. Mulchandani, and S. Raghvendra, A graph theoretic additive approximation of optimal transport, *Proc. Advances in Neural Information Processing Systems 32*, 2019, pp. 13813–13823.
[19] N. Lahn and S. Raghvendra, A faster algorithm for minimum-cost bipartite matching in minor free graphs, *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2019, pp. 569–588.
[20] N. Lahn and S. Raghvendra, A weighted approach to the maximum cardinality bipartite matching problem with applications in geometric settings, *Proc. 35th Annual Symposium on Computational Geometry*, 2019, pp. 48:1–48:13.
[21] N. Lahn and S. Raghvendra, An $\tilde{O}(n^{5/4})$ time ε-approximation algorithm for RMS matching in a plane, *arXiv preprint arXiv:2007.07720*, (2020).
[22] H. Liu, X. Gu, and D. Samaras, Wasserstein gan with quadratic transport cost, *Proc. IEEE International Conference on Computer Vision*, 2019, pp. 4832–4841.
[23] J. M. Phillips and P. K. Agarwal, On bipartite matching under the RMS distance, *Proc. 18th Annual Canadian Conference on Computational Geometry*, 2006, pp. 143–146.
[24] K. Quanrud, Approximating optimal transport with linear programs, *Proc. 2nd SIAM Symposium on Simplicity in Algorithms*, 2019, pp. 6:1–6:9.
[25] J. Rabin and G. Peyré, Wasserstein regularization of imaging problem, *Proc. 18th IEEE International Conference on Image Processing*, 2011, pp. 1541–1544.
[26] S. Raghvendra and P. K. Agarwal, A near-linear time ϵ-approximation algorithm for geometric bipartite matching, *J. ACM*, 67 (2020), 18:1–18:19.
[27] R. Sharathkumar, A sub-quadratic algorithm for bipartite matching of planar points with bounded integer coordinates, *Proc. 29th Annual ACM Symposium on Computational Geometry*, 2013, pp. 9–16.
[28] R. Sharathkumar and P. K. Agarwal, Algorithms for transportation problem in geometric settings, *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012, pp. 306–317.
[29] J. Sherman, Generalized preconditioning and undirected minimum-cost flow, *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2017, pp. 772–780.
[30] J. Solomon, F. De Goes, G. Peyré, M. Cuturi, A. Butscher, A. Nguyen, T. Du, and L. Guibas, Convolutional wasserstein distances: Efficient optimal transportation on geometric domains, *ACM Transactions on Graphics*, 34 (2015), 66.
[31] K. R. Varadarajan, A divide-and-conquer algorithm for min-cost perfect matching in the plane, *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science*, 1998, pp. 320–331.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited