A WEIGHTED APPROACH TO THE MAXIMUM CARDINALITY BIPARTITE MATCHING PROBLEM WITH APPLICATIONS IN GEOMETRIC SETTINGS

Nathaniel Lahn[*], *Sharath Raghvendra*[†]

Abstract.

We present a weighted approach to compute a maximum cardinality matching in an arbitrary bipartite graph. Our main result is a new algorithm that takes as input a weighted bipartite graph $G(A \cup B, E)$ with edge weights of 0 or 1. Let $w \leq n$ be an upper bound on the weight of any matching in G. Consider the subgraph induced by all the edges of G with a weight 0. Suppose every connected component in this subgraph has $\mathcal{O}(r)$ vertices and $\mathcal{O}(mr/n)$ edges. We present an algorithm to compute a maximum cardinality matching in G in $\tilde{\mathcal{O}}(m(\sqrt{w} + \sqrt{r} + \frac{wr}{n}))$ time.[1]

When all the edge weights are 1 (symmetrically when all weights are 0), our algorithm will be identical to the well-known Hopcroft-Karp (HK) algorithm, which runs in $\mathcal{O}(m\sqrt{n})$ time. However, if we can carefully assign weights of 0 and 1 on its edges such that both w and r are sub-linear in n and $wr = \mathcal{O}(n^\gamma)$ for $\gamma < 3/2$, then we can compute a maximum cardinality matching in $o(m\sqrt{n})$ time. Using our algorithm, we obtain a new $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$ time algorithm to compute an ε-approximate bottleneck matching of $A, B \subset \mathbb{R}^2$ and a $\frac{1}{\varepsilon^{\mathcal{O}(d)}} n^{1+\frac{d-1}{2d-1}} \operatorname{poly}\log n$ time algorithm for computing an ε-approximate bottleneck matching in d-dimensions. All previous algorithms take $\Omega(n^{3/2})$ time.

Our algorithm also applies to any graph $G(A \cup B, E)$ that has an easily computable balanced vertex separator of size $|V'|^\delta$, for every subgraph $G'(V', E')$ where $\delta \in [1/2, 1)$. By applying our algorithm, we can compute a maximum matching in $\tilde{\mathcal{O}}(mn^{\frac{\delta}{1+\delta}})$ time improving upon the $\mathcal{O}(m\sqrt{n})$ time taken by the HK-Algorithm.

1 Introduction

We consider the classical matching problem in an arbitrary unweighted bipartite graph $G(A \cup B, E)$ with $|A| = |B| = n$ and $E \subseteq A \times B$. A *matching* $M \subseteq E$ is a set of vertex-disjoint edges. We refer to a largest cardinality matching M in G as a *maximum matching*. A maximum matching is *perfect* if $|M| = n$. Now suppose the graph is weighted and every edge $(a, b) \in E$ has a *weight* specified by $\mathtt{c}(a, b)$. The weight of any subset of edges $E' \subseteq E$ is given by $\sum_{(a,b)\in E} \mathtt{c}(a, b)$. A *minimum-weight maximum matching* is a maximum matching

[*] *Radford University*, nlahn@radford.edu.

[†] *Virginia Tech*, sharathr@vt.edu. This research is supported by NSF Grant CCF 1909171.

[1] We use $\tilde{\mathcal{O}}$ to suppress poly-logarithmic terms.

with the smallest weight. In this paper, we present an algorithm to compute a maximum matching faster by carefully assigning weights of 0 and 1 to the edges of G.

Maximum matching in graphs: In an arbitrary bipartite graph with n vertices and m edges, Ford and Fulkerson's algorithm [7] iteratively computes, in each phase, an augmenting path in $\mathcal{O}(m)$ time, leading to a maximum cardinality matching in $\mathcal{O}(mn)$ time. Hopcroft and Karp's algorithm (HK-Algorithm) [12] reduces the number of phases from n to $\mathcal{O}(\sqrt{n})$ by computing a maximal set of vertex-disjoint shortest augmenting paths in each phase. A single phase can be implemented in $\mathcal{O}(m)$ time leading to an overall execution time of $\mathcal{O}(m\sqrt{n})$. In weighted bipartite graphs with n vertices and m edges, the well-known Hungarian method computes a minimum-weight maximum matching in $\mathcal{O}(mn+n^2\log n)$ time [15]. Gabow and Tarjan designed a weight-scaling algorithm (GT-Algorithm) to compute a minimum-weight perfect matching in $\mathcal{O}(m\sqrt{n}\log(nC))$ time, provided all edge weights are integers bounded by C [11]. Their method, like the Hopcroft-Karp algorithm, computes a maximal set of vertex-disjoint shortest (for an appropriately defined augmenting path cost) augmenting paths in each phase. For the maximum matching problem in arbitrary graphs (not necessarily bipartite), a weighted approach has been applied to achieve a simple $\mathcal{O}(m\sqrt{n})$ time algorithm [10].

For planar graphs, the maximum-flow problem with multiple sources and multiple sinks can be solved in $\mathcal{O}(n\log^3 n)$ time [5]. By applying a standard reduction from the maximum cardinality bipartite matching problem to the maximum flow problem, this result of [5] also implies an $\mathcal{O}(n\log^3 n)$ time algorithm for the maximum cardinality matching problem on bipartite planar graphs. Recently Lahn and Raghvendra [16] gave $\tilde{\mathcal{O}}(n^{6/5})$ and $\tilde{\mathcal{O}}(n^{7/5})$ time algorithms for finding a minimum-weight perfect bipartite matching in planar and K_h-minor[2] free graphs respectively, overcoming the $\Omega(m\sqrt{n})$ barrier; see also Asathulla *et al.* [4]. Both these algorithms are based on the existence of an r-clustering which, for a parameter $r>0$, is a partitioning of G into edge-disjoint clusters $\{\mathcal{R}_1,\ldots,\mathcal{R}_k\}$ such that $k=\tilde{\mathcal{O}}(n/\sqrt{r})$, every cluster $\mathcal{R}_j$ has $\mathcal{O}(r)$ vertices, and each cluster has $\tilde{\mathcal{O}}(\sqrt{r})$ *boundary* vertices. A boundary vertex has edges from two or more clusters incident on it. Furthermore, the total number of boundary vertices, counted with multiplicity, is $\tilde{\mathcal{O}}(n/\sqrt{r})$. The algorithm of Lahn and Raghvendra extends to any graph that admits an r-clustering. There are also algebraic approaches for the design of fast algorithms for bipartite matching; see for instance [19, 20].

Matching in geometric settings: In geometric settings, A and B are points in a fixed d-dimensional space and G is a complete bipartite graph on A and B. For a fixed integer $p\geq 1$, the weight of an edge between $a\in A$ and $b\in B$ is $\|a-b\|^p$, where $\|a-b\|$ denotes the Euclidean distance between a and b. The weight of a matching M is given by $\left(\sum_{(a,b)\in M}\|a-b\|^p\right)^{1/p}$. For any fixed $p\geq 1$, we wish to compute a perfect matching with the minimum weight. When $p=1$, the problem is the well-studied *Euclidean bipartite matching* problem. A minimum-weight perfect matching for $p=\infty$ will minimize the largest-weight edge in the matching and is referred to as a *bottleneck matching*. The Euclidean bipartite

[2] They assume $h=\mathcal{O}(1)$.

matching in a plane can be computed in $\tilde{\mathcal{O}}(n^{3/2+\delta})$ [24] time for an arbitrary small $\delta > 0$; see also Sharathkumar and Agarwal [25]. Efrat *et al.* present an algorithm to compute a bottleneck matching in the plane in $\tilde{\mathcal{O}}(n^{3/2})$ [6] time. Both these algorithms use geometric data structures in a non-trivial fashion to speed up classical graph algorithms.

When $p = 1$, for any $0 < \varepsilon \leq 1$, there is an ε-approximation algorithm for the Euclidean bipartite matching problem that runs in $\tilde{\mathcal{O}}(n/\varepsilon^d)$ time [23]. For $p = 2$ and $A, B \subset \mathbb{R}^2$, Lahn and Raghvendra gave an $\tilde{\mathcal{O}}(n^{5/4}\text{poly}(1/\varepsilon))$ time algorithm for computing an ε-approximation [17]. For $p > 2$, all known ε-approximation algorithms take $\Omega(n^{3/2}/\varepsilon^d)$ time. We note that it is possible to find a $\Theta(1)$-approximate bottleneck matching in 2-dimensional space by reducing the problem to finding maximum flow in a planar graph and then finding the flow using an $\tilde{\mathcal{O}}(n)$ time max-flow algorithm [5]. There are numerous other results; see also [1, 2, 8, 14, 22]. Designing exact and approximation algorithms that break the $\Omega(n^{3/2})$ barrier remains an important research challenge in computational geometry.

Our results: We present a weighted approach to compute a maximum cardinality matching in an arbitrary bipartite graph. Our main result is a new matching algorithm that takes as input a weighted bipartite graph $G(A \cup B, E)$ with every edge having a weight of 0 or 1. Let $w \leq n$ be an upper bound on the weight of any matching in G. Consider the subgraph induced by all the edges of G with a weight 0. Let $\{K_1, K_2, \ldots, K_l\}$ be the connected components in this subgraph and let, for any $1 \leq i \leq l$, V_i and E_i be the vertices and edges of K_i. We refer to each connected component K_i as a *piece*. Suppose $|V_i| = \mathcal{O}(r)$ and $|E_i| = \mathcal{O}(mr/n)$. Given G, we present an algorithm to compute a maximum matching in G in $\tilde{\mathcal{O}}(m(\sqrt{w} + \sqrt{r} + \frac{wr}{n}))$ time. Consider any graph for which the removal of a sub-linear number of "separator" vertices partitions the graph into connected components with $\mathcal{O}(r)$ vertices and $\mathcal{O}(mr/n)$ edges. We can apply our algorithm to any such graph by simply setting the weight of every edge incident on any separator vertex to 1 and weights of all other edges to 0.

When all the edge weights are 1 or all edge weights are 0, our algorithm will be identical to the HK-Algorithm algorithm and runs in $\mathcal{O}(m\sqrt{n})$ time. However, if we can carefully assign weights of 0 and 1 on the edges such that both w and r are sub-linear in n and for some constant $\gamma < 3/2$, $wr = \mathcal{O}(n^\gamma)$, then we can compute a maximum matching in G in $o(m\sqrt{n})$ time. Using our algorithm, we obtain the following results:

- Our algorithm can be applied to any graph class with a sub-linear vertex separator. Given any graph $G(A \cup B, E)$ that has an easily computable balanced vertex separator S' for every subgraph $G'(V', E')$ where $|S'| = \mathcal{O}(|V'|^\delta)$, for $\delta \in [1/2, 1)$, there is a 0/1 weight assignment on edges of the graph so that the weight of any matching is $\mathcal{O}(n^{\frac{2\delta}{1+\delta}})$ and $r = \mathcal{O}(n^{\frac{1}{1+\delta}})$. As a result, we obtain an algorithm that computes the maximum cardinality matching in $\tilde{\mathcal{O}}(mn^{\frac{\delta}{1+\delta}})$ time (Section 5). Many classes of graphs including planar graphs, K_h-minor free graphs, and subgraphs of d-dimensional grids have a sub-linear sized balanced vertex separator.
- Notably, it is known how to efficiently compute separators of size $\tilde{\mathcal{O}}(\sqrt{n})$ on K_h minor-free graphs when $h = \mathcal{O}(1)$ [13, 26]. For such graphs, our result gives an $\tilde{\mathcal{O}}(n^{4/3})$ time

algorithm. This improves upon the algorithm of Lahn and Raghvendra [16] in the case where the graph is unweighted. While asymptotically better algorithms exist for this situation, using matrix multiplication [13, 20, 21, 26], such algorithms are randomized and use algebraic methods, while our algorithm is combinatorial and deterministic. To our knowledge, our $\tilde{\mathcal{O}}(n^{4/3})$ minor-free graph algorithm is the best known deterministic result for this problem. Furthermore, the core of our algorithm seems to be practical.

- Given two point sets $A, B \subset \mathbb{R}^2$ and an $0 < \varepsilon \leq 1$, we reduce the problem of computing an ε-approximate bottleneck matching to computing a maximum cardinality matching in a subgraph $\mathcal{G}$ of the complete bipartite graph on A and B. We can, in $\mathcal{O}(n)$ time assign 0/1 weights to the $\mathcal{O}(n^2)$ edges of $\mathcal{G}$ so that any matching has a weight of $\mathcal{O}(n^{2/3})$. Despite possibly $\Theta(n^2)$ edges in $\mathcal{G}$, we present an efficient implementation of our graph algorithm with $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$ execution time that computes an ε-approximate bottleneck matching for $d = 2$; all previously known algorithms take $\Omega(n^{3/2})$ time. Our algorithm, for any fixed $d \geq 2$ dimensional space, computes an ε-approximate bottleneck matching in $\frac{1}{\varepsilon^{\mathcal{O}(d)}} n^{1+\frac{d-1}{2d-1}} \operatorname{poly}\log n$ time (Section 6).

Our approach: Initially, we compute, in $\mathcal{O}(m\sqrt{r})$ time, a maximum matching within all pieces. Similar to the GT-Algorithm, the rest of our algorithm is based on a primal-dual method and executes in phases. Each phase consists of two stages. The first stage conducts a Hungarian search and finds at least one augmenting path containing only zero slack (with respect to the dual constraints) edges. Let the admissible graph be the subgraph induced by the set of all zero slack edges. Unlike in the GT-Algorithm, the second stage of our algorithm computes augmenting paths in the admissible graph that are not necessarily vertex-disjoint. In the second stage, the algorithm iteratively initiates a DFS from every free vertex. When a DFS finds an augmenting path P, the algorithm will augment the matching immediately and terminate this DFS. Let all pieces of the graph that contain the edges of P be *affected*. Unlike the GT-Algorithm, which deletes all edges visited by the DFS, our algorithm deletes only those edges that were visited by the DFS and did not belong to an affected piece. Consequently, we allow for visited edges from an affected piece to be reused in another augmenting path. As a result, our algorithm computes several more augmenting paths per phase than the GT-Algorithm, leading to a reduction of number of phases from $\mathcal{O}(\sqrt{n})$ to $\mathcal{O}(\sqrt{w})$. Note, however, that the edges of an affected piece may now be visited multiple times by different DFS searches within the same phase. This increases the cumulative time taken by all the DFS searches in the second stage. However, we are able to bound the total number of affected pieces across all phases of the algorithm by $\mathcal{O}(w \log w)$. Since each piece has $\mathcal{O}(mr/n)$ edges, the total time spent revisiting these edges is bounded by $\mathcal{O}((mrw \log w)/n)$. The total execution time can therefore be bounded by $\tilde{\mathcal{O}}(m(\sqrt{w} + \sqrt{r} + \frac{wr}{n}))$.

2 Preliminaries

We are given a bipartite graph $G(A \cup B, E)$, where any edge $(a, b) \in E$ has a weight $\mathsf{c}(a, b)$ of 0 or 1. Given a matching M, a vertex is *free* if it is not matched in M. An *alternating path* (resp. cycle) is a simple path (resp. cycle) that alternates between edges in M and not

in M. An *augmenting path* is an alternating path that begins and ends at a free vertex.

To assist in describing our algorithm, we define a residual network, an augmented residual network, and a set of feasibility conditions. A *residual network* G_M with respect to a matching M is a directed graph where every edge $(a, b) \in E$ is directed from b to a if $(a, b) \notin M$ and from a to b if $(a, b) \in M$. We use the notation $u \to v$ to represent an edge in the residual graph directed from a vertex u to a vertex v. Any directed edge $u \to v$ will inherit the cost $\mathsf{c}(u, v)$ from the corresponding undirected edge (u, v). For any matching M and its residual graph G_M, the edges of M correspond to a set of directed edges in G_M. For convenience in presentation, we will use M to also denote the set of directed edges in G_M that correspond to the matching edges, i.e., we will use $u \to v \in M$ and $u \to v \notin M$ to denote whether $u \to v$ corresponds to a matching edge or a nonmatching edge in E.

Next, we define a set of feasibility conditions, which relate to the directions of edges in the residual graph. A matching M and an assignment of dual weights $y(\cdot)$ on the vertices of G are *feasible* if for any edge $(a, b) \in (A \times B) \cap E$:

$$y(b) - y(a) \leq \mathsf{c}(a, b) \qquad \text{if } (a, b) \notin M, \tag{1}$$

$$y(a) - y(b) = \mathsf{c}(a, b) \qquad \text{if } (a, b) \in M. \tag{2}$$

We say that any edge $u \to v$ of the residual graph is feasible if the corresponding condition (1) or (2) holds. The weight $s(u, v)$ of any edge $u \to v$ in the residual graph is given by the slack of the edge with respect to feasibility conditions (1) and (2), i.e., if $u \to v \notin M$, then $u \in B$, $v \in A$, and $s(u, v) = \mathsf{c}(u, v) + y(v) - y(u)$. Otherwise, $(u, v) \in M$, $u \in A$, $v \in B$, and we have $s(a, b) = 0$. An *augmented residual network* is obtained by adding to the residual network an additional vertex s and additional directed edges from s to every free vertex of B, each having a weight of 0. We denote the augmented residual network as G'_M.

3 Our algorithm

Throughout this section we will use M to denote the current matching maintained by the algorithm and A_F and B_F to denote the vertices of A and B that are free with respect to M. Initially $M = \emptyset$, $A_F = A$, and $B_F = B$. Our algorithm consists of two steps. The first step, which we refer to as the *preprocessing step*, will execute the Hopcroft-Karp algorithm and compute a maximum matching within every piece. Any maximum matching M_{OPT} has at most w edges with a weight of 1 and the remaining edges have a weight of 0. Therefore, $|M_{\text{OPT}}| - |M| \leq w$. The time taken by the preprocessing step for K_i is $\mathcal{O}(|E_i|\sqrt{|V_i|}) = \mathcal{O}(|E_i|\sqrt{r})$. Since the pieces are vertex disjoint, the total time taken across all pieces is $\mathcal{O}(m\sqrt{r})$. After this step, no augmenting path with respect to M is completely contained within a single piece. We set the dual weight $y(v)$ of every vertex $v \in A \cup B$ to 0. The matching M along with the dual weights $y(\cdot)$ satisfies (1) and (2) and is feasible.

The second step of the algorithm is executed in *phases*. We describe phase k of the algorithm. This phase consists of two stages.

First stage: In the first stage, we construct the augmented residual network G'_M and execute Dijkstra's algorithm with s as the source. Let ℓ_v for any vertex v denote the shortest path distance from s to v in G'_M. If a vertex v is not reachable from s, we set ℓ_v to ∞. Let

$$\ell = \min_{v \in A_F} \ell_v. \tag{3}$$

Suppose M is a perfect matching or $\ell = \infty$, then this algorithm returns with M as a maximum matching. Otherwise, we update the dual weight of any vertex $v \in A \cup B$ as follows. If $\ell_v \geq \ell$, we leave its dual weight unchanged. Otherwise, if $\ell_v < \ell$, we set $y(v) \leftarrow y(v) + \ell - \ell_v$. After updating the dual weights, we construct the *admissible graph* which consists of a subset of edges in the residual network G_M that have zero slack. After the first stage, the matching M and the updated dual weights are feasible (see Lemma 5). Furthermore, there is at least one augmenting path in the admissible graph. This completes the first stage of the phase.

Second stage: In the second stage, we initialize G' to be the admissible graph and execute DFS to identify augmenting paths. For any augmenting path P found during the DFS, we refer to the pieces that contain its edges as *affected pieces* of P.

Similar to the HK-Algorithm, the second stage of this phase will iteratively initiate a partial DFS for each free vertex $b \in B_F$ in G'. If the DFS does not lead to an augmenting path, we delete all edges that were visited by the DFS. On the other hand, if the DFS finds an augmenting path P, then the matching is augmented along P, all edges that are visited by the DFS and do not lie in an affected piece of P are deleted, and the DFS initiated at b will terminate.

Now, we describe in detail the DFS initiated for a free vertex $b \in B_F$. Initially $P = \langle b = v_1 \rangle$. Every edge of G' is marked unvisited. At any point during the execution of DFS, the algorithm maintains a simple path $P = \langle b = v_1, v_2, \ldots, v_k \rangle$. The DFS continues from the last vertex of this path as follows:

- If there are no unvisited edges that are going out of v_k in G',
  - If $P = \langle v_1 \rangle$, remove all edges that were marked as visited from G' and terminate the execution of DFS initiated at b.
  - Otherwise, delete v_k from P and continue the DFS from v_{k-1}.
- If there is an unvisited edge going out of v_k, let $v_k \to v$ be this edge. Mark $v_k \to v$ as visited. If v is on the path P, continue the DFS from v_k. If v is not on the path P, add $v_k \to v$ to P, set v_{k+1} to v, and,
  - Suppose $v \in A_F$, then P is an augmenting path from b to v. Execute the AUGMENT procedure, defined below, which augments M along P. Delete from G' every visited edge that does not belong to any affected piece of P and terminate the execution of DFS initiated at b.
  - Otherwise, $v \in (A \cup B) \setminus A_F$. Continue the DFS from v_{k+1}.

The AUGMENT procedure receives a feasible matching M, a set of dual weights $y(\cdot)$, and an augmenting path P as input. For any $b \rightarrow a \in P \setminus M$, where $a \in A$ and $b \in B$, set $y(b) \leftarrow y(b) - 2\mathsf{c}(a,b)$. Then augment M along P by setting $M \leftarrow M \oplus P$. The residual graph is updated accordingly to reflect the new matching. Every edge of M after augmentation satisfies the feasibility condition (2) (see Lemma 5). This completes the description of our algorithm. The algorithm maintains the following invariants during its execution:

(I1) The matching M and the set of dual weights $y(\cdot)$ are feasible. Let $y_{\max} = \max_{v \in B} y(v)$. The dual weight of every vertex $v \in B_F$ is $y_{\max}$ and the dual weight for every vertex $v \in A_F$ is 0.

(I2) For every phase that is fully executed prior to obtaining a maximum matching, at least one augmenting path is found and the dual weight of every free vertex of B_F increases by at least 1.

Comparison with the GT-Algorithm: In the GT-Algorithm, the admissible graph does not have any alternating cycles. Also, every augmenting path edge can be shown to not participate in any future augmenting paths that are computed in the current phase. By using these facts, one can show that the edges visited unsuccessfully by a DFS will not lead to an augmenting path in the current phase. In our case, however, admissible cycles can exist. Also, some edges on the augmenting path that have zero weight remain admissible after augmentation and may participate in another augmenting path in the current phase. We show, however, that any admissible cycle must be completely inside a piece and cannot span multiple pieces (Lemma 2). Using this fact, we show that edges visited unsuccessfully by the DFS that do not lie in an affected piece will not participate in any more augmenting paths (Lemma 7 and Lemma 9) in the current phase. Therefore, we can safely delete them.

Correctness: From Invariant (I2), each phase of our algorithm will increase the cardinality of M by at least 1 and so, our algorithm terminates with a maximum matching.

Efficiency: We use the following notations to bound the efficiency of our algorithm. Let $\{P_1, \ldots, P_t\}$ be the t augmenting paths computed in the second step of the algorithm. Let $\mathbb{K}_i$ be the set of affected pieces with respect to the augmenting path P_i. Let M_0 be the matching at the end of the first step of the algorithm. Let, for $1 \leq i \leq t$, $M_i = M_{i-1} \oplus P_i$, i.e., M_i is the matching after the ith augmentation in the second step of the algorithm.

The first stage is an execution of Dijkstra's algorithm which takes $\mathcal{O}(m + n \log n)$ time [9]. Suppose there are λ phases; then the cumulative time taken across all phases for the first stage is $\mathcal{O}(\lambda m + \lambda n \log n)$. In the second stage, each edge visited by a DFS is discarded for the remainder of the phase, provided it is not in an affected piece. Since each affected piece has $\mathcal{O}(mr/n)$ edges, the total time taken by all the DFS searches across all the λ phases is bounded by $\mathcal{O}(m\lambda + (mr/n)\sum_{i=1}^{t} |\mathbb{K}_i|)$. In Lemma 3, we bound λ by $\mathcal{O}(\sqrt{w})$ and $\sum_{i=1}^{t} |\mathbb{K}_i|$ by $\mathcal{O}(w \log w)$. Therefore, the total time taken by the algorithm including the time taken by the preprocessing step is $\mathcal{O}(m\sqrt{r} + m\sqrt{w} + n\sqrt{w}\log n + \frac{mrw \log w}{n}) = \tilde{\mathcal{O}}(m(\sqrt{w} + \sqrt{r} + \frac{wr}{n}))$.

Theorem 1. *Let $G(A \cup B, E)$ be a bipartite graph with $n = |A| + |B|$ and $m = |E|$. Let $\mathsf{c}(\cdot,\cdot)$ be a 0/1 weight assignment on the edges of E. Assume that any matching on G has weight at most w, and any connected subgraph of G consisting of only weight 0 edges has $\mathcal{O}(r)$ vertices and $\mathcal{O}(mr/n)$ edges. Then a maximum cardinality matching on G can be computed in $\mathcal{O}(m\sqrt{r} + m\sqrt{w} + n\sqrt{w}\log n + \frac{mrw\log w}{n})$ time.*

Lemma 1. *For any feasible matching $M, y(\cdot)$ maintained by the algorithm, let $y_{\max}$ be the dual weight of every vertex of B_F. For any augmenting path P with respect to M directed from a free vertex $b' \in B_F$ to a free vertex $a' \in A_F$,*

$$\mathsf{c}(P) = y_{\max} + \sum_{u \to v \in P} s(u,v).$$

Proof. The weight of P is

$$\mathsf{c}(P) = \sum_{u \to v \in P} \mathsf{c}(u,v) = \sum_{b \to a \in P \setminus M} (y(b) - y(a) + s(b,a)) + \sum_{a \to b \in P \cap M} (y(a) - y(b)).$$

Since every vertex on P except for b' and a' participates in one edge of $P \cap M$ and one edge of $P \setminus M$, we can write the above equation as

$$\mathsf{c}(P) = y(b') - y(a') + \sum_{b \to a \in P \setminus M} s(b,a) = y(b') - y(a') + \sum_{u \to v \in P} s(u,v).$$

The last equality follows from the fact that edges of $P \cap M$ satisfy (2) and have a slack of zero. From (I1), we get that $y(b') = y_{\max}$ and $y(a') = 0$, which gives,

$$\mathsf{c}(P) = y_{\max} + \sum_{u \to v \in P} s(u,v).$$

□

Lemma 2. *For any feasible matching $M, y(\cdot)$ maintained by the algorithm, and for any directed alternating cycle C with respect to M, if $\mathsf{c}(C) > 0$, then*

$$\sum_{u \to v \in C} s(u,v) > 0,$$

i.e., C is not a cycle in the admissible graph.

Proof. The weight of C is

$$\mathsf{c}(C) = \sum_{u \to v \in C} \mathsf{c}(u,v) = \sum_{b \to a \in C \setminus M} (y(b) - y(a) + s(b,a)) + \sum_{a \to b \in C \cap M} (y(a) - y(b)).$$

Since every vertex on C participates in one edge of $C \cap M$ and one edge of $C \setminus M$, we can write the above equation as

$$\mathsf{c}(C) = \sum_{b \to a \in C \setminus M} s(b,a) = \sum_{u \to v \in C} s(u,v).$$

The last equality follows from the fact that edges of $C \cap M$ satisfy (2) and have a slack of zero. Since $\mathsf{c}(C) > 0$, we also have $\sum_{u \to v \in C} s(u, v) > 0$, implying the claim. □

Lemma 3. *The total number of phases is $\mathcal{O}(\sqrt{w})$ and the total number of affected pieces is $\mathcal{O}(w \log w)$, i.e., $\sum_{i=1}^{t} |\mathbb{K}_i| = \mathcal{O}(w \log w)$.*

Proof. Let M_{OPT} be a maximum matching, which has weight at most w. Consider the state of the algorithm at any point in time. The symmetric difference of M and M_{OPT} contains $j = |M_{\text{OPT}}| - |M|$ vertex-disjoint augmenting paths. While the symmetric difference also contains even-length alternating paths and even-length alternating cycles, we are only concerned with the augmenting paths. Let $\{\mathcal{P}_1, \ldots, \mathcal{P}_j\}$ be these augmenting paths. These paths contain edges of M_{OPT} and M, both of which are of weight at most w. Therefore, the sum of weights of these paths is

$$\sum_{i=1}^{j} \mathsf{c}(\mathcal{P}_i) \leq 2w.$$

Recall that, at any point in time, every free vertex of B_F has a dual weight of $y_{\max}$. By Lemma 1 and the fact that the slack on every edge is non-negative, we immediately get,

$$j \cdot y_{\max} \leq 2w. \tag{4}$$

Next, we use equation (4) to bound the number of phases executed by the algorithm. Consider the state maintained by the algorithm after the completion of phase k. From (I2), $y_{\max} \geq k$. When $\sqrt{w} \leq k < \sqrt{w} + 1$, it follows from equation (4) that $j = |M_{\text{OPT}}| - |M| \leq 2\sqrt{w}$. From (I2), we will compute at least one augmenting path in each phase and so the remaining j unmatched vertices are matched in at most $2\sqrt{w}$ phases. This bounds the total number of phases by $3\sqrt{w}$.

Next, we use equation (4) to bound the total augmenting path length. Recollect that $\{P_1, \ldots, P_t\}$ are the augmenting paths computed by the algorithm. The matching M_0 has $|M_{\text{OPT}}| - t$ edges. Let $y^l_{\max}$ correspond to the dual weight of the free vertices of B_F when the augmenting path P_l is found by the algorithm. From Lemma 1, and the fact that P_l is an augmenting path consisting of zero slack edges, we have $y^l_{\max} = \mathsf{c}(P_l)$. Before augmenting along P_l, there are $|M_{\text{OPT}}| - t + l - 1$ edges in M_{l-1} and $j = |M_{\text{OPT}}| - |M_{l-1}| = t - l + 1$. Plugging this in to (4), we get $\mathsf{c}(P_l) = y^l_{\max} \leq \frac{2w}{t-l+1}$. Summing over all $1 \leq l \leq t$, and recalling that $t = |M_{\text{OPT}}| \leq w$, we get,

$$\sum_{l=1}^{t} \mathsf{c}(P_l) \leq w \sum_{l=1}^{t} \frac{2}{t - l + 1} = \mathcal{O}(w \log t) = \mathcal{O}(w \log w). \tag{5}$$

For any augmenting path P_l, the number of affected pieces is upper bounded by one plus the number of non-zero weight edges on P_l, i.e., $|\mathbb{K}_l| \leq \mathsf{c}(P_l) + 1$. Therefore,

$$\sum_{l=1}^{t} |\mathbb{K}_l| \leq \sum_{l=1}^{t} (\mathsf{c}(P_l) + 1) = \mathcal{O}(w \log w).$$

□

4 Proof of invariants

We now prove (I1) and (I2). Consider any phase k in the algorithm. Assume inductively that at the end of phase $k-1$, (I1) and (I2) hold. We will show that (I1) and (I2) also hold at the end of the phase k. We establish a lemma that will help us prove (I1) and (I2).

Lemma 4. *For any edge $a \to b \in M$, let ℓ_a and ℓ_b be the distances returned by Dijkstra's algorithm during the first stage of phase k, then $\ell_a = \ell_b$.*

Proof. The only edge directed towards b is an edge from its match a. Therefore, any path from s to b in the augmented residual network, including the shortest path, should pass through a. Since the slack on any edge of M is 0, $\ell_b = \ell_a + s(a,b) = \ell_a$. □

Lemma 5. *Any matching M and dual weights $y(\cdot)$ maintained during the execution of the algorithm are feasible.*

Proof. We begin by showing that the dual weight modifications in the first stage of phase k will not violate dual feasibility conditions (1) and (2). Let $\tilde{y}(\cdot)$ denote the dual weights after the execution of the first stage of the algorithm. Consider any edge $u \to v$. There are the following possibilities:

If both ℓ_u and ℓ_v are greater than or equal to ℓ, then $y(u)$ and $y(v)$ remain unchanged and the edge remains feasible.

If both ℓ_u and ℓ_v are less than ℓ, suppose $u \to v \in M$. Then, from Lemma 4, $\ell_u = \ell_v$. We have, $\tilde{y}(u) = y(u) + \ell - \ell_u$, $\tilde{y}(v) = y(v) + \ell - \ell_v$, and $\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) + \ell_v - \ell_u = \mathsf{c}(u,v)$ implying $u \to v$ satisfies (2).

If ℓ_u and ℓ_v are less than ℓ and $u \to v \notin M$, then $u \in B$ and $v \in A$. By definition, $y(u) - y(v) + s(u,v) = \mathsf{c}(u,v)$. By the properties of shortest paths, for any edge $u \to v$, $\ell_v - \ell_u \leq s(u,v)$. The dual weight of u is updated to $y(u) + \ell - \ell_u$ and dual weight of v is updated to $y(v) + \ell - \ell_v$. The difference in the updated dual weights $\tilde{y}(u) - \tilde{y}(v) = (y(u) + \ell - \ell_u) - (y(v) + \ell - \ell_v) = y(u) - y(v) + \ell_v - \ell_u \leq y(u) - y(v) + s(u,v) = \mathsf{c}(u,v)$. Therefore, $u \to v$ satisfies (1).

If $\ell_u < \ell$ and $\ell_v \geq \ell$, then, from Lemma 4, $u \to v \notin M$, and so $u \in B$ and $v \in A$. From the shortest path property, for any edge $u \to v$, $\ell_v - \ell_u \leq s(u,v)$. Therefore,

$$\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) + \ell - \ell_u \leq y(u) - y(v) + \ell_v - \ell_u \leq y(u) - y(v) + s(u,v) = \mathsf{c}(u,v),$$

implying $u \to v$ satisfies (1).

If $\ell_u \geq \ell$ and $\ell_v < \ell$, then, from Lemma 4, $u \to v \notin M$, and so $u \in B$ and $v \in A$. Since $\ell_v < \ell$, we have,

$$\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) - \ell + \ell_v < y(u) - y(v) \leq \mathsf{c}(u,v),$$

implying $u \to v$ satisfies (1).

In the second stage of the algorithm, when an augmenting path P is found, the dual weights of some vertices of B on P decrease and the directions of edges of P change. We

argue these operations do not violate feasibility. Let $\tilde{y}(\cdot)$ be the dual weights after these operations. Consider any edge (a, b) where $a \in A$ and $b \in B$. If b is not on P, then the feasibility of (a, b) is unchanged. If b is on P and a is not on P, then $\tilde{y}(b) \leq y(b)$, and $\tilde{y}(b) - \tilde{y}(a) \leq y(b) - y(a) \leq \mathsf{c}(a, b)$, implying (1) holds. The remaining case is when both a and b are on P. Consider if $(a, b) \in M$ after augmentation. Prior to augmentation, (a, b) was an admissible edge not in M, and we have $y(b) - y(a) = \mathsf{c}(a, b)$ and $\tilde{y}(b) = y(b) - 2\mathsf{c}(a, b)$. So, $\tilde{y}(a) - \tilde{y}(b) = y(a) - (y(b) - 2\mathsf{c}(a, b)) = y(a) - y(b) + 2\mathsf{c}(a, b) = \mathsf{c}(a, b)$, implying (2) holds. Finally, consider if $(a, b) \notin M$ after augmentation. Then, prior to augmentation, (a, b) was in M, and $y(a) - y(b) = \mathsf{c}(a, b)$. So, $\tilde{y}(b) - \tilde{y}(a) \leq y(b) - y(a) = -\mathsf{c}(a, b) \leq \mathsf{c}(a, b)$, implying (1) holds. We conclude the second stage maintains feasibility. □

Next we show that the dual weights A_F are zero and the dual weights of all vertices of B_F are equal to y_{max}. At the start of the second step, all dual weights are 0. During the first stage, the dual weight of any vertex v will increase by $\ell - \ell_v$ only if $\ell_v < \ell$. By (3), for every free vertex $a \in A_F$, $\ell_a \geq \ell$, and so the dual weight of every free vertex of A remains unchanged at 0. Similarly, for any free vertex $b \in B_F$, $\ell_b = 0$, and the dual weight increases by ℓ, which is the largest possible increase. This implies that every free vertex in B_F will have the same dual weight of $y_{\max}$. In the second stage, matched vertices of B undergo a decrease in their dual weights, which does not affect vertices in B_F. Therefore, the dual weights of vertices of B_F will still have a dual weight of $y_{\max}$ after stage two. This completes the proof of (I1).

Before we prove (I2), we will first establish a property of the admissible graph after the dual weight modifications in the first stage of the algorithm.

Lemma 6. *After the first stage of each phase, there is an augmenting path consisting of admissible edges.*

Proof. Let $a \in A_F$ be a free vertex whose shortest path distance from s in the augmented residual network is ℓ, i.e., $\ell_a = \ell$. Let P be the shortest path from s to a and let P_a be the path P with s removed from it. Note that P_a is an augmenting path. We will show that after the dual updates in the first stage, every edge of P_a is admissible. Consider any edge $u \to v \in P_a \cap M$, where $u \in A$ and $v \in B$. From Lemma 4, $\ell_u = \ell_v$. Then the updated dual weights are $\tilde{y}(u) = y(u) + \ell - \ell_u$ and $\tilde{y}(v) = y(v) + \ell - \ell_v$. Therefore, $\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) - \ell_u + \ell_v = \mathsf{c}(u, v)$, and $u \to v$ is admissible. Otherwise, consider any edge $u \to v \in P_a \setminus M$, where $u \in B$ and $v \in A$. From the optimal substructure property of shortest paths, for any edge $u \to v \in P_a$, we have $\ell_v - \ell_u = s(u, v)$. Therefore, the difference of the new dual weights is

$$\tilde{y}(u) - \tilde{y}(v) = y(u) + \ell - \ell_u - y(v) - \ell + \ell_v = y(u) - y(v) - \ell_u + \ell_v = y(u) - y(v) + s(u, v) = \mathsf{c}(u, v),$$

implying that $u \to v$ is admissible. □

Proof of (I2): From Lemma 6, there is an augmenting path of admissible edges at the end of the first stage of any phase. Since we execute a DFS from every free vertex $b \in B_F$ in the second stage, we are guaranteed to find an augmenting path. Next, we show in Corollary 1

that there is no augmenting path of admissible edges at the end of stage two of phase k, i.e., all augmenting paths in the residual network have a slack of at least 1. This will immediately imply that the first stage of phase $k+1$ will have to increase the dual weight of every free vertex by at least 1 completing the proof for (I2).

Edges that are deleted during a phase do not participate in any admissible augmenting path for the rest of the phase. We show this in two steps. First, we show that at the time of deletion of an edge $u \to v$, there is no path in the admissible graph that starts from the edge $u \to v$ and ends at a free vertex $a \in A_F$ (Lemma 9). In Lemma 7, we show that any such edge $u \to v$ will not participate in any admissible alternating path to a free vertex of A_F for the rest of the phase.

We use DFS(b, k) to denote the DFS initiated from b in phase k. Let P_u^b denote the path maintained by DFS(b, k) when the vertex u was added to the path.

Lemma 7. *Consider some point during the second stage of phase k where there is an edge $u \to v$ that does not participate in any admissible alternating path to a vertex of A_F. Then, for the remainder of phase k, this edges $u \to v$ does not participate in any admissible alternating path to a vertex of A_F.*

Proof. Assume for the sake of contradiction that at some later time during phase k, the edge $u \to v$ becomes part of an admissible path $P_{y,z}$ from a vertex y to a vertex $z \in A_F$. Consider the first time this occurs for $u \to v$. During the second stage, the dual weights of some vertices of B may decrease just prior to augmentation; however, this does not create any new admissible edges. Therefore, $P_{y,z}$ must have become an admissible path due to augmentation along a path $P_{a,b}$ from some $b \in B_F$ to some $a \in A_F$. Specifically, $P_{y,z}$ must intersect $P_{a,b}$ at some vertex x. Therefore, prior to augmenting along $P_{a,b}$, there was an admissible path from y to a via x. This contradicts the assumption that $u \to v$ did not participate in any admissible path to a vertex of A_F prior to this time. □

Lemma 8. *Consider the execution of DFS(b, k) and the path P_u^b. Suppose the DFS(b, k) marks an edge $u \to v$ as visited. Let P_v be an admissible alternating path from v to any free vertex $a \in A_F$ in G'. Suppose P_v and P_u^b are vertex-disjoint. Then, DFS(b, k) will find an augmenting path that includes the edge $u \to v$.*

Proof. P_v and P_u^b are vertex-disjoint and so, v is not on the path P_u^b. Therefore, DFS(b, k) will add $u \to v$ to the path and we get the path $P = P_v^b$. We will show that all edges of P_v are unvisited by DFS(b, k) at this time, and so the DFS procedure, when continued from v, will discover an augmenting path.

We show, through a contradiction, that all edges of P_v are not yet visited by DFS(b, k). Consider, for the sake of contradiction, among all the edges of P_v, the edge $u' \to v'$ that was marked visited first. We claim the following:

(i) *$u' \to v'$ is visited before $u \to v$*: This follows from the assumption that when $u \to v$ was marked as visited, $u' \to v'$ was already marked as visited by the DFS.

(ii) *$u \to v$ is not a descendant of $u' \to v'$ in the DFS*: If $u' \to v'$ was an ancestor of $u \to v$ in the DFS, then P_u^b contains $u' \to v'$. By definition, P_v also contains $u' \to v'$, which contradicts the assumption that P_u^b and P_v are disjoint paths.

(iii) *When $u' \to v'$ is marked visited, it will be added to the path by the DFS*: The only reason why $u' \to v'$ is visited but not added is if v' is already on the path $P_{u'}^b$. In that case, P_v and $P_{u'}^b$ will share an edge that was visited before $u' \to v'$ contradicting the assumption that $u' \to v'$ was the earliest edge of P_v to be marked visited.

From (iii), when $u' \to v'$ was visited, it was added to the path $P_{v'}^b$. Since $u' \to v'$ was the edge on P_v that was marked visited first by DFS(b, k), all edges on the subpath from v' to a are unvisited. Therefore, the DFS(b, k), when continued from v', will not visit $u \to v$ (from (ii)), will find an augmenting path, and terminate. From (i), $u \to v$ will not be marked visited by DFS(b, k) leading to a contradiction. □

Lemma 9. *Consider a DFS initiated from some free vertex $b \in B_F$ in phase k. Let M be the matching at the start of this DFS and M' be the matching when the DFS terminates. Suppose the edge $u \to v$ was deleted during DFS(b, k). Then there is no admissible path starting with $u \to v$ and ending at a free vertex $a \in A_F$ in $G_{M'}$.*

Proof. At the start of phase k, G' is initialized to the admissible graph. Inductively, we assume that all the edges discarded in phase k prior to the execution of DFS(b, k) do not participate in any augmenting path of admissible edges with respect to M. Therefore, any augmenting path of admissible edges in G_M remains an augmenting path in G'. There are two possible outcomes for DFS(b, k). Either, (i) the DFS terminates without finding an augmenting path, or (ii) the DFS terminates with an augmenting path $\tilde{P}$ and $M' = M \oplus \tilde{P}$.

In case (i), $M = M'$ and any edge $u \to v$ visited by the DFS(b, k) is marked for deletion. For the sake of contradiction, let $u \to v$ participate in an admissible path P to a free vertex $a' \in A_F$. Since u is reachable from b and a' is reachable from u in G_M, a' is reachable from b. This contradicts the fact that DFS(b, k) did not find an augmenting path. Therefore, no edge $u \to v$ marked for deletion participates in an augmenting path with respect to M.

In case (ii), $M' = M \oplus \tilde{P}$. DFS(b, k) marks two kinds of edges for deletion.

(a) Any edge $u \to v$ on the augmenting path $\tilde{P}$ such that $\mathsf{c}(u, v) = 1$ is deleted, and,

(b) Any edge $u \to v$ that is marked visited by DFS(b, k), does not lie on $\tilde{P}$, and does not belong to any affected piece is deleted.

In (a), there are two possibilities (1) $u \to v \in \tilde{P} \cap M$ or (2) $u \to v \in \tilde{P} \setminus M$. If $u \to v \in M$ (case (a)(1)), then, after augmentation along $\tilde{P}$, $s(u, v)$ increases from 0 to at least 2, and $u \to v$ is no longer admissible. Therefore, $u \to v$ does not participate in any admissible alternating paths to a free vertex in A_F with respect to $G_{M'}$. If $u \to v \notin M$ (case (a)(2)), then the AUGMENT procedure reduces the dual weight of $u \in B$ by 2. So, every edge going out of u will have a slack of at least 2. Therefore, $u \to v$ cannot participate in any admissible path P to a free vertex in A_F. This completes case (a).

For (b), we will show that $u \to v$, even prior to augmentation along $\tilde{P}$, did not participate in any path of admissible edges from v to any free vertex of A_F. For the sake of contradiction, let there be a path P_v from v to $a' \in A_F$. We claim that P_v and P_u^b are not vertex-disjoint. Otherwise, from Lemma 8, the path $\tilde{P}$ found by DFS(b, k) includes $u \to v$. However, by our assumption for case (b), $u \to v$ does not lie on $\tilde{P}$. Therefore, we safely assume that P_v intersects P_u^b. There are two cases:

- $\mathsf{c}(u, v) = 1$: We will construct a cycle of admissible edges containing the edge $u \to v$. Since $\mathsf{c}(u, v) = 1$, our construction will contradict Lemma 2. Let x be the first vertex common to both P_v and P_u^b as we walk from v to a' on P_v. To create the cycle, we traverse from x to u along the path P_u^b, followed by the edge $u \to v$, followed by the path from v to x along P_v. All edges of this cycle are admissible including the edge $u \to v$.

- $\mathsf{c}(u, v) = 0$: In this case, $u \to v$ belongs to some piece K_i that is not an affected piece. Among all edges visited by DFS(b, k), consider the edge $u' \to v'$ of K_i, the same piece as $u \to v$, such that v' has a path to the vertex $a' \in A_F$ with the fewest number of edges. Let $P_{v'}$ be this path. We claim that $P_{v'}$ and $P_{u'}^b$ are not vertex-disjoint. Otherwise, from Lemma 8, the path $\tilde{P}$ found by DFS(b, k) includes $u' \to v'$ and K_i would have been an affected piece. Therefore, we can safely assume that $P_{v'}$ intersects with $P_{u'}^b$. Let z be the first intersection point with $P_{u'}^b$ as we walk from v' to a' and let z' be the vertex that follows after z in $P_{u'}^b$. There are two possibilities:

  - *The edge $z \to z' \in K_i$:* In this case, $z \to z'$ is also marked visited by DFS(b, k), and z' has path to a' with fewer number of edges than v'. This contradicts our assumption about $u' \to v'$.
  - *The edge $z \to z' \notin K_i$:* In this case, consider the cycle obtained by walking from z to u' along the path $P_{u'}^b$ followed by the edge $u' \to v'$ and the path from v' to z along $P_{v'}$. Since $u' \to v' \in K_i$ and $z \to z' \notin K_i$, the admissible cycle contains at least one edge of weight 1. This contradicts Lemma 2.

This concludes case (b) which shows that $u \to v$ did not participate in any augmenting paths with respect to M. From Lemma 7, it follows that $u \to v$ does not participate in any augmenting path with respect to $G_{M'}$ as well. □

Corollary 1. *At the end of any phase, there is no augmenting path of admissible edges.*

5 Algorithm for graphs with small balanced vertex separators

A *balanced vertex separator* for any graph $G(V, E)$ is a set $S \subseteq V$ of vertices whose removal from G disconnects the graph into two pieces, each piece having at least a third of the vertices in V. For any $\delta \in [1/2, 1)$, we say that a graph has a balanced vertex separator of size n^δ if every subgraph $G'(V', E')$ of G has a balanced vertex separator S' such that $|S'| = \mathcal{O}(|V'|^\delta)$. We assume that these balanced vertex separators are *efficiently computable*, meaning the time taken to compute the separators is dominated by the time taken by our

algorithm, given the separators. Consider any graph $G(V, E)$ with n vertices, m edges, and a balanced vertex separator of size n^δ. Using these separators, we describe a procedure for assigning weights of 0 and 1 to the edges so that the cost of any matching is no more than $w = \mathcal{O}(n^{\frac{2\delta}{1+\delta}})$ and the value of $r = \mathcal{O}(n^{\frac{1}{1+\delta}})$. Assuming that the separators can be computed efficiently, such a weight assignment can also be produced efficiently, and our algorithm will compute a maximum cardinality matching in $\tilde{\mathcal{O}}(mn^{\frac{\delta}{1+\delta}})$ time leading to Theorem 2.

Next, we describe how to compute a separator set $\mathcal{S}$ of size $\mathcal{O}(n/r^{1-\delta})$ whose removal disconnects the graph into disjoint pieces, each with at most r vertices and $\mathcal{O}(mr/n)$ edges. We accomplish this by using a recursive procedure SEPARATE(G'), which we describe next. SEPARATE(G') accepts as input a subgraph (or piece) $G'(V', E')$ of G. If $|V'| < r$ and $|E'| < mr/n$, then the procedure SEPARATE(G') returns immediately; this is the base case. Otherwise, the procedure SEPARATE(G') further subdivides G' by computing a separator S' of size $\mathcal{O}(|V'|^\delta)$. The procedure adds all of these vertices to the overall separator set by setting $\mathcal{S} \leftarrow \mathcal{S} \cup S'$. Next, let G_ℓ and G_r be the two pieces of G' formed by removing the vertices of S'. The procedure SEPARATE(G') is invoked recursively on both G_ℓ and G_r. Initially, we set $\mathcal{S} \leftarrow \emptyset$ and call SEPARATE on G. The set $\mathcal{S}$ returned by this call is a vertex set whose removal from G creates pieces each with at most r vertices and at most mr/n edges.

Next, we show that the total size of $\mathcal{S}$ is $\mathcal{O}(n/r^{1-\delta})$. We partition the set $\mathcal{S}$ into subsets $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_k$ as follows: The set $\mathcal{S}_0$ contains all separator vertices added to $\mathcal{S}$ during some recursive call to SEPARATE($G'(V', E')$) such that $|V'| \geq r$. For any $0 < i \leq k$, the set $\mathcal{S}_i$ contains all separator vertices added to $\mathcal{S}$ as part of a recursive call SEPARATE(G') with $\frac{r}{2^i} \leq |V'| < \frac{r}{2^{i-1}}$.

Recollect that the procedure SEPARATE is called on some piece of the original graph G. We define a set $\{\mathcal{K}_0, \ldots, \mathcal{K}_{k+1}\}$ where, for $0 \leq i \leq k+1$, each $\mathcal{K}_i$ is a subset of pieces on which the procedure SEPARATE is called. Let $\mathcal{K}_0$ be a singleton set containing the graph G. For $0 < i \leq k+1$, let $\mathcal{K}_i$ be the set of those pieces that contain at least mr/n edges and that are formed after removing $\mathcal{S}_{i-1}$ from the pieces of $\mathcal{K}_{i-1}$. Let n_i be the subset of vertices that participate in any piece of $\mathcal{K}_i$. We bound the quantity n_i for any $i > 0$. Removing $\mathcal{S}_{i-1}$ from pieces of $\mathcal{K}_{i-1}$ forms disjoint pieces, each containing at most $\frac{r}{2^{i-1}}$ vertices. Recall that a piece is not further subdivided unless it contains at least mr/n edges. Since there are at most m edges, each participating in at most one piece, the total number of such pieces requiring further subdivision is at most $\frac{n}{r}$. Each of these pieces contains at most $\frac{r}{2^{i-1}}$ vertices, so the number of vertices in $\mathcal{K}_i$ is at most $\frac{n}{2^{i-1}}$.

Using standard arguments, it can be shown that, for a sufficiently large constant c, by recursively applying SEPARATE on G the size of $\mathcal{S}_0$ is at most $\frac{cn}{r^{1-\delta}}$. For $0 < i \leq k$, we can apply similar arguments for each $\mathcal{K}_i$ and use the bound on n_i to upper bound $|\mathcal{S}_i|$ by $\frac{cn}{2^{i-1}r^{1-\delta}}$. Summing over all $0 \leq i \leq k$ gives a bound of $\mathcal{O}(n/r^{1-\delta})$ for the size of $\mathcal{S}$.

To generate our weight assignment, we simply assign each edge incident on a vertex of $\mathcal{S}$ a weight of 1 and any other edge a weight of 0. It is easy to see that $w = |\mathcal{S}| = \mathcal{O}(\frac{n}{r^{1-\delta}})$ is an upper bound on the cost of any matching. We choose $r = \mathcal{O}(n^{\frac{1}{1+\delta}})$ and so $w = \mathcal{O}(n^{\frac{2\delta}{1+\delta}})$ as desired. Applying Theorem 1 gives the following:

Theorem 2. *For any $\delta \in [1/2, 1)$ and any bipartite graph $G(A \cup B, E)$ with an efficiently-computable balanced vertex separator of size $\mathcal{O}(n^\delta)$ on each subgraph, a maximum cardinality matching in G can be computed in $\mathcal{O}(mn^{\frac{\delta}{1+\delta}} \log n)$ time.*

6 Minimum bottleneck matching

We are given two sets A and B of n d-dimensional points. Consider a weighted and complete bipartite graph on points of A and B. The weight of any edge $(a, b) \in A \times B$ is given by its Euclidean distance and denoted by $\|a - b\|$. For any matching M of A and B let its largest weight edge be its *bottleneck edge*. In the *minimum bottleneck matching* problem, we wish to compute a matching M_{OPT} of A and B with the smallest weight bottleneck edge. We refer to this weight as the *bottleneck distance* of A and B and denote it by β^*. An *ε-approximate bottleneck matching* of A and B is any matching M with a bottleneck edge weight of at most $(1 + \varepsilon)\beta^*$. We present an algorithm that takes as input A, B, and a value δ such that $\beta^* \leq \delta \leq (1 + \varepsilon/3)\beta^*$, and produces an ε-approximate bottleneck matching. For simplicity in presentation, we describe our algorithm for the 2-dimensional case when all points of A and B are in a bounding square S. The algorithm easily extends to any arbitrary fixed dimension d. For 2-dimensional case, given a value δ, our algorithm executes in $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$ time.

Although the value of δ is not known to the approximation algorithm, we can first find a value α that is guaranteed to be an n-approximation of the bottleneck distance [3, Lemma 2.2] and then select $\mathcal{O}(\log n/\varepsilon)$ values from the interval $[\alpha/n, \alpha]$ of the form $(1 + \varepsilon/3)^i\alpha/n$, for $0 \leq i \leq \mathcal{O}(\log n/\varepsilon)$. We will then execute our algorithm for each of these $\mathcal{O}(\log n/\varepsilon)$ selected values of δ. Our algorithm returns a maximum matching whose edges are of length at most $(1 + \varepsilon/3)\delta$ in $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$ time. At least one of the δ values chosen will be a $\beta^* \leq \delta \leq (1 + \varepsilon/3)\beta^*$. The matching returned by the algorithm for this value of δ will be perfect ($|M| = n$) and have a bottleneck edge of weight at most $(1+\varepsilon/3)^2\beta^* \leq (1+\varepsilon)\beta^*$ as desired. Among all executions of our algorithm that return a perfect matching, we return a perfect matching with the smallest bottleneck edge weight. We can reasonably assume that $1/\varepsilon$ is a polynomial in n; therefore, the number of guesses is $\mathcal{O}(\log n)$, and the total time taken to compute the ε-approximate bottleneck matching is $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$.

Given the value of δ, the algorithm will construct a graph as follows: Let $\mathbb{G}$ be a grid on the bounding square S. The side-length of every square in this grid is $\varepsilon\delta/(6\sqrt{2})$. For any cell ξ in the grid $\mathbb{G}$, let $N(\xi)$ denote the subset of all cells ξ' of $\mathbb{G}$ such that the minimum distance between ξ and ξ' is at most δ. By the use of a simple packing argument, it can be shown that $|N(\xi)| = \mathcal{O}(1/\varepsilon^2)$.

For any point $v \in A \cup B$, let ξ_v be the cell of grid $\mathbb{G}$ that contains v. We say that a cell ξ is *active* if $(A \cup B) \cap \xi \neq \emptyset$. Let A_ξ and B_ξ denote the points of A and B in the cell ξ. We construct a bipartite graph $\mathcal{G}(A \cup B, \mathcal{E})$ on the points in $A \cup B$ as follows: For any pair of points $(a, b) \in A \times B$, we add an edge in the graph if $\xi_b \in N(\xi_a)$. Note that every edge (a, b) with $\|a - b\| \leq \delta$ will be included in $\mathcal{G}$. Since δ is at least the bottleneck distance, $\mathcal{G}$ will have a perfect matching. The maximum distance between any cell ξ and a cell in $N(\xi)$ is $(1 + \varepsilon/3)\delta$. Therefore, no edge in $\mathcal{G}$ will have a length greater than $(1 + \varepsilon/3)\delta$. This implies

that any perfect matching in $\mathcal{G}$ will also be an ε-approximate bottleneck matching. We use our algorithm for maximum matching to compute this perfect matching in $\mathcal{G}$. Note, that $\mathcal{G}$ can have $\Omega(n^2)$ edges. For the sake of efficiency, our algorithm executes on a compact representation of $\mathcal{G}$ that is described later. Next, we assign weights of 0 and 1 to the edges of $\mathcal{G}$ so that the any maximum matching in $\mathcal{G}$ has a small weight w.

For a parameter[3] $r > 1$, we will carefully select another grid $\mathbb{G}'$ on the bounding square S, each cell of which has a side-length of $\sqrt{r}(\varepsilon\delta/(6\sqrt{2}))$ and encloses $\sqrt{r} \times \sqrt{r}$ cells of $\mathbb{G}$. For any cell ξ of the grid $\mathbb{G}$, let $\square_\xi$ be the cell in $\mathbb{G}'$ that contains ξ. Any cell ξ of $\mathbb{G}$ is a *boundary cell* with respect to $\mathbb{G}'$ if there is a cell $\xi' \in N(\xi)$ such that $\square_{\xi'} \neq \square_\xi$. Equivalently, if the minimum distance from ξ to the boundary of $\square_\xi$ is at most δ, then ξ is a boundary cell. For any boundary cell ξ of $\mathbb{G}$ with respect to grid $\mathbb{G}'$, we refer to all points of A_ξ and B_ξ that lie in ξ as boundary points. All other points of A and B are referred to as internal points. We carefully construct this grid $\mathbb{G}'$ such that the total number of boundary points is $\mathcal{O}(n/\varepsilon\sqrt{r})$ as follows: First, we will generate the vertical lines for $\mathbb{G}'$, and then we will generate the horizontal lines using a similar construction. Consider the vertical line y_{ij} to be the line $x = i(\varepsilon\delta)/(6\sqrt{2}) + j\sqrt{r}(\varepsilon\delta/(6\sqrt{2}))$. For any fixed integer i in $[1, \sqrt{r}]$, consider the set of vertical lines $\mathbb{Y}_i = \{y_{ij} \mid y_{ij}$ intersects the bounding square $S\}$. We label all cells ξ of $\mathbb{G}$ as boundary cells with respect to $\mathbb{Y}_i$ if the distance from ξ to some vertical line in $\mathbb{Y}_i$ is at most δ. We designate the points inside the boundary cells as boundary vertices with respect to $\mathbb{Y}_i$. For any given i, let A_i and B_i be the boundary vertices of A and B with respect to the lines in $\mathbb{Y}_i$. We select an integer $\kappa = \arg\min_{1\leq i\leq\sqrt{r}} |A_i \cup B_i|$ and use $\mathbb{Y}_\kappa$ as the vertical lines for our grid $\mathbb{G}'$. We use a symmetric construction for the horizontal lines.

Lemma 10. *Let A_i and B_i be the boundary points with respect to the vertical lines $\mathbb{Y}_i$. Let $\kappa = \arg\min_{1\leq i\leq\sqrt{r}} |A_i \cup B_i|$. Then, $|A_\kappa \cup B_\kappa| = \mathcal{O}(n/(\varepsilon\sqrt{r}))$.*

Proof. For any fixed cell ξ in $\mathbb{G}$, of the $\sqrt{r}$ values of i, there are $\mathcal{O}(1/\varepsilon)$ values for which $\mathbb{Y}_i$ has a vertical line at a distance at most δ from ξ. Therefore, each cell ξ will be a boundary cell in only $\mathcal{O}(1/\varepsilon)$ shifts out of $\sqrt{r}$ shifts. So, A_ξ and B_ξ will be counted in $A_i \cup B_i$ for $\mathcal{O}(1/\varepsilon)$ different values of i. Therefore, if we take the average over choices of i, we get

$$\min_{1\leq i\leq\sqrt{r}} |A_i \cup B_i| \leq \frac{1}{\sqrt{r}} \sum_{i=1}^{\sqrt{r}} |A_i \cup B_i| \leq \mathcal{O}(n/(\varepsilon\sqrt{r})).$$

□

Using a similar construction, we guarantee that the boundary points with respect to the horizontal lines of $\mathbb{G}'$ is also at most $\mathcal{O}(n/(\varepsilon\sqrt{r}))$.

Corollary 2. *The grid $\mathbb{G}'$ that we construct has $\mathcal{O}(n/(\varepsilon\sqrt{r}))$ many boundary points.*

Let ξ and ξ' be two cells of the grid $\mathbb{G}$ such that $\xi' \in N(\xi)$ and $\square_\xi \neq \square_{\xi'}$. Then the weights of all edges of $A_\xi \times B_{\xi'}$ and of $B_\xi \times A_{\xi'}$ are set to 1. All other edges have a weight of 0. We do not make an explicit weight assignment as it is expensive to do so. Instead,

[3] Assume r to be a perfect square.

we can always derive the weight of an edge when we access it. Only boundary points will have edges of weight 1 incident on them. From Corollary 2, it follows that any maximum matching will have a weight of $w = \mathcal{O}(n/(\varepsilon\sqrt{r}))$.

The edges of every piece in $\mathcal{G}$ have endpoints that are completely inside a cell of $\mathbb{G}'$. Note, however, that there is no straight-forward bound on the number of points and edges of $\mathcal{G}$ inside each piece. Moreover, the number of edges in $\mathcal{G}$ can be $\Theta(n^2)$. Consider any feasible matching $M, y(\cdot)$ in $\mathcal{G}$. Let $\mathcal{G}_M$ be the residual network. In order to obtain a running time of $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$, we use the grid $\mathbb{G}$ to construct a compact residual network $\mathcal{CG}_M$ for any feasible matching $M, y(\cdot)$ and use this compact graph to implement our algorithm. The following lemma assists us in constructing the compressed residual network.

Lemma 11. *Consider any feasible matching $M, y(\cdot)$ maintained by our algorithm on $\mathcal{G}$ and any active cell ξ in the grid $\mathbb{G}$. The dual weight of any two points $a, a' \in A_\xi$ can differ by at most 2. Similarly, the dual weights of any two points $b, b' \in B_\xi$ can differ by at most 2.*

Proof. We present our proof for two points $b, b' \in B_\xi$. A similar argument will extend for $a, a' \in A_\xi$. For the sake of contradiction, let $y(b) \geq y(b') + 3$. b' must be matched since $y(b') < y(b) \leq y_{\max}$. Let $m(b') \in A$ be the match of b' in M. From (2), $y(m(b')) - y(b') = \mathsf{c}(b', m(b'))$. Since both b and b' are in ξ, we have $\mathsf{c}(b, m(b')) = \mathsf{c}(b', m(b'))$. So, $y(b) - y(m(b')) \geq (y(b') + 3) - y(m(b')) = 3 - \mathsf{c}(b, m(b'))$. This violates (1) leading to a contradiction. □

For any feasible matching and any cell ξ of $\mathbb{G}$, we divide points of A_ξ and B_ξ based on their dual weight into at most three clusters. Let A^1_ξ, A^2_ξ and A^3_ξ be the three clusters of points in A_ξ and let B^1_ξ, B^2_ξ and B^3_ξ be the three clusters of points in B_ξ. We assume that points with the largest dual weights are in A^1_ξ (resp. B^1_ξ), the points with the second largest dual weights are in A^2_ξ (resp. B^2_ξ), and the points with the smallest dual weights are in A^3_ξ (resp. B^3_ξ).

Compact residual network: Given a feasible matching M, we construct a compact residual network $\mathcal{CG}_M$ to assist in the fast implementation of our algorithm. The vertex set $\mathcal{A} \cup \mathcal{B}$ for the compact residual network is constructed as follows. First we describe the vertex set $\mathcal{A}$. For every active cell ξ in $\mathbb{G}$, we add a vertex a^1_ξ (resp. a^2_ξ, a^3_ξ) to represent the set A^1_ξ (resp. A^2_ξ, A^3_ξ) provided $A^1_\xi \neq \emptyset$ (resp. $A^2_\xi \neq \emptyset, A^3_\xi \neq \emptyset$). We designate a^1_ξ (resp. a^2_ξ, a^3_ξ) as a *free* vertex if $A^1_\xi \cap A_F \neq \emptyset$ (resp. $A^2_\xi \cap A_F \neq \emptyset, A^3_\xi \cap A_F \neq \emptyset$). Similarly, we construct a vertex set $\mathcal{B}$ by adding a vertex b^1_ξ (resp. b^2_ξ, b^3_ξ) to represent the set B^1_ξ (resp. B^2_ξ, B^3_ξ) provided $B^1_\xi \neq \emptyset$ (resp. $B^2_\xi \neq \emptyset, B^3_\xi \neq \emptyset$). We designate b^1_ξ (resp. b^2_ξ, b^3_ξ) as a *free* vertex if $B^1_\xi \cap B_F \neq \emptyset$ (resp. $B^2_\xi \cap B_F \neq \emptyset, B^3_\xi \cap B_F \neq \emptyset$). Each active cell ξ of the grid $\mathbb{G}$ therefore has at most six points. Each point in $\mathcal{A} \cup \mathcal{B}$ will inherit the dual weights of the points in its cluster; for any vertex $a^1_\xi \in \mathcal{A}$ (resp. $a^2_\xi \in \mathcal{A}, a^3_\xi \in \mathcal{A}$), let $y(a^1_\xi)$(resp. $y(a^2_\xi), y(a^3_\xi)$) be the dual weight of all points in A^1_ξ (resp. A^2_ξ, A^3_ξ). We define $y(b^1_\xi)$, $y(b^2_\xi)$, and $y(b^3_\xi)$ as dual weights of points in B^1_ξ, B^2_ξ, and B^3_ξ respectively. Since there are at most n active cells, $|\mathcal{A} \cup \mathcal{B}| = \mathcal{O}(n)$.

Next, we create the edge set for the compact residual network $\mathcal{CG}_M$. Like before, we use $u \to v$ to denote an edge directed from u to v in the compact residual network. For any active cell ξ in the grid $\mathbb{G}$ and for any cell $\xi' \in N(\xi)$,

- We add a directed edge from a^i_ξ to $b^j_{\xi'}$, for $i, j \in \{1, 2, 3\}$ if there is an edge $(a, b) \in (A^i_\xi \times B^j_{\xi'}) \cap M$. We define the weight of $a^i_\xi \to b^j_{\xi'}$ to be $\mathsf{c}(a, b)$. We also define the slack $s(a^i_\xi, b^j_{\xi'})$ to be $\mathsf{c}(a^i_\xi, b^j_{\xi'}) - y(a^i_\xi) + y(b^j_{\xi'})$ which is equal to $s(a^i_\xi, b^j_{\xi'}) = \mathsf{c}(a, b) - y(a) + y(b) = s(a, b) = 0$.
- We add a directed edge from b^i_ξ to $a^j_{\xi'}$, for $i, j \in \{1, 2, 3\}$ if $(B^i_\xi \times A^j_{\xi'}) \setminus M \neq \emptyset$. Note that the weight and slack of every directed edge in $B^i_\xi \times A^j_{\xi'}$ are identical. We define the weight of $b^i_\xi \to a^j_{\xi'}$ to be $\mathsf{c}(a, b)$ for any $(a, b) \in A^j_{\xi'} \times B^i_\xi$. We also define the slack $s(b^i_\xi, a^j_{\xi'}) = \mathsf{c}(b^i_\xi, a^j_{\xi'}) - y(b^i_\xi) + y(a^j_{\xi'})$ which is equal to the slack $s(a, b)$.

For each vertex in $\mathcal{A} \cup \mathcal{B}$, we added at most two edges to every cell $\xi' \in N(\xi)$. Since $N(\xi) = \mathcal{O}(1/\varepsilon^2)$, the total number of edges in $\mathcal{E}$ is $\mathcal{O}(n/\varepsilon^2)$. For a cell $\square$ in $\mathbb{G}'$, let $\mathcal{A}_\square$ be the points of $\mathcal{A}$ generated by cells of $\mathbb{G}$ that are contained inside the cell $\square$. A piece $K_\square$ has $\mathcal{A}_\square \cup \mathcal{B}_\square$ as the vertex set and $\mathcal{E}_\square = (\mathcal{A}_\square \times \mathcal{B}_\square \cup \mathcal{B}_\square \times \mathcal{A}_\square) \cap \mathcal{E}$ as the edge set. Note that the number of vertices in any piece $K_\square$ is $\mathcal{O}(r)$ and the number of edges in $K_\square$ is $\mathcal{O}(r/\varepsilon^2)$. Every edge (u, v) of any piece $K_\square$ has a weight $\mathsf{c}(u, v) = 0$ and every edge (u, v) with a weight of zero belongs to some piece of $\mathcal{CG}_M$.

The following straight-forward lemma implies that the compact graph $\mathcal{CG}_M$ preserves all minimum slack paths in $\mathcal{G}_M$.

Lemma 12. *For any directed path $\mathcal{P}$ in the compact residual network $\mathcal{CG}_M$, there is a directed path P in the residual network such that $\sum_{u \to v \in P} s(u, v) = \sum_{u \to v \in \mathcal{P}} s(u, v)$. For any directed path P in $\mathcal{G}_M$, there is a directed path $\mathcal{P}$ in the compact residual network such that $\sum_{u \to v \in P} s(u, v) \geq \sum_{u \to v \in \mathcal{P}} s(u, v)$.*

Preprocessing step: At the start, $M = \emptyset$ and all dual weights are 0. Consider any cell $\square$ of the grid $\mathbb{G}'$ and any cell ξ of $\mathbb{G}$ that is contained inside $\square$. Suppose we have a point a^1_ξ. We assign a demand $d_{a^1_\xi} = |A^1_\xi| = |A_\xi|$ to a^1_ξ. Similarly, suppose we have a point b^1_ξ, we assign a supply $s_{b^1_\xi} = |B^1_\xi| = |B_\xi|$. The preprocessing step reduces to finding a maximum matching of supplies to demand. This is an instance of the unweighted transportation problem which can be solved using the algorithm of [18] in $\tilde{\mathcal{O}}(|\mathcal{E}_\square|\sqrt{|\mathcal{A}_\square \cup \mathcal{B}_\square|}) = \tilde{\mathcal{O}}(|\mathcal{E}_\square|\sqrt{r})$. Every edge of $\mathcal{E}$ participates in at most one piece. Therefore, the total time taken for preprocessing across all pieces is $\tilde{\mathcal{O}}(|\mathcal{E}|\sqrt{r}) = \tilde{\mathcal{O}}(n\sqrt{r}/\varepsilon^2)$. We can trivially convert the matching of supplies to demands into a matching in $\mathcal{G}$.

Efficient implementation of the second step: Recollect that the second step of the algorithm consists of phases. Each phase has two stages. In the first stage, we execute Dijkstra's algorithm in $\mathcal{O}(n \log n/\varepsilon^2)$ time by using the compact residual network $\mathcal{CG}$. After adjusting the dual weight of nodes in the compact graph, in the second stage, we iteratively compute

augmenting paths of admissible edges by conducting a DFS from each free vertex of $\mathcal{B}$. Our implementation of DFS has the following differences from the one described in Section 3.

- Recollect that each free vertex $v \in \mathcal{B}$ may represent a cluster that has $t > 0$ free vertices. We will execute DFS from v exactly t times, once for each free vertex of $\mathcal{B}$.
- During the execution of any DFS, unlike the algorithm described in Section 3, the DFS will mark an edge as visited only when it backtracks from the edge. Due to this change, all edges on the path maintained by the DFS are marked as unvisited. Therefore, unlike the algorithm from Section 3, this algorithm will not discard weight 1 edges of an augmenting path after augmentation. From Lemma 3, the total number of these edges is $\mathcal{O}(w \log w)$. Note that this does not asymptotically increase the total number of edge revisits during the DFS.

Efficiency: The first stage is an execution of Dijkstra's algorithm which takes $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|) = \mathcal{O}(n \log n/\varepsilon^2)$ time. Suppose there are λ phases; then the cumulative time taken across all phases for the first stage is $\tilde{\mathcal{O}}(\lambda n/\varepsilon^2)$. In the second stage of the algorithm, in each phase, every edge is discarded once it is visited by a DFS, unless it is in an affected piece or it is an edge of weight 1 on an augmenting path. Since each affected piece has $\mathcal{O}(r/\varepsilon^2)$ edges, and since there are $\mathcal{O}(w \log w)$ edges of weight 1 on the computed augmenting paths, the total time taken by all the DFS searches across all the λ phases is bounded by $\tilde{\mathcal{O}}(n\lambda/\varepsilon^2 + r/\varepsilon^2 \sum_{i=1}^{t} |\mathbb{K}_i| + w \log w)$. In Lemma 3, we bound λ by $\mathcal{O}(\sqrt{w})$ and $\sum_{i=1}^{t} |\mathbb{K}_i|$ by $\mathcal{O}(w \log w)$. Therefore, the total time taken by the algorithm including the time taken by the preprocessing step is $\tilde{\mathcal{O}}((n/\varepsilon^2)(\sqrt{r} + \sqrt{w} + \frac{wr}{n}))$. Setting $r = n^{2/3}$, we get $w = \mathcal{O}(n/(\varepsilon\sqrt{r})) = \mathcal{O}(n^{2/3}/\varepsilon)$, and the total running time of our algorithm is $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$. To obtain the bottleneck matching, we execute this algorithm on $\mathcal{O}(\log n)$ guesses of δ; therefore, the total time taken to compute an ε-approximate bottleneck matching, accounting for these guesses, is $\tilde{\mathcal{O}}(n^{4/3}/\varepsilon^3)$.

Finally, we give the bounds for our algorithm when $d > 2$. Like the $d = 2$ case, each cell of the grid $\mathbb{G}'$ contains r cells of the grid $\mathbb{G}$. However, now the width of each cell in $\mathbb{G}'$ is $r^{1/d}$ times the width of each cell in $\mathbb{G}$. Lemma 10 can be easily generalized for d dimensions; the only change necessary is the reduction in number of shifts from $\sqrt{r}$ to $r^{1/d}$. As a result, the number of boundary vertices due to each dimension is $\mathcal{O}(\frac{n}{\varepsilon r^{1/d}})$ and we get $w = \mathcal{O}(\frac{dn}{\varepsilon r^{1/d}})$. It is also worth noting that, for any cell $\xi \in \mathbb{G}$, the size of $N(\xi)$ becomes $1/\varepsilon^d$, and so the total number of edges in $\mathcal{E}$ becomes $\mathcal{O}(n/\varepsilon^d)$. Setting $r = n^{\frac{d}{2d-1}}$ gives a running time of $\frac{1}{\varepsilon^{\mathcal{O}(d)}} n^{1+\frac{d-1}{2d-1}} \operatorname{poly}\log n$.

Theorem 3. *Given $A, B \subset \mathbb{R}^d$ and a parameter $\varepsilon > 0$, we can compute a $(1+\varepsilon)$-approximate bottleneck matching of A and B in $\frac{1}{\varepsilon^{\mathcal{O}(d)}} n^{1+\frac{d-1}{2d-1}} \operatorname{poly}\log n$ time.*

References

[1] Pankaj K. Agarwal, Kyle Fox, Debmalya Panigrahi, Kasturi R. Varadarajan, and Allen Xiao. Faster algorithms for the geometric transportation problem. In *33rd International*

Symposium on Computational Geometry, pages 7:1–7:16, 2017.

[2] Pankaj K. Agarwal and R. Sharathkumar. Approximation algorithms for bipartite matching with metric and geometric costs. In *Proceedings of the 46th ACM Symposium on Theory of Computing Conference*, pages 555–564, 2014.

[3] Pankaj K. Agarwal and Kasturi R. Varadarajan. A near-linear constant-factor approximation for euclidean bipartite matching? In *20th International Symposium on Computational Geometry*, pages 247–252, 2004.

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

[5] Glencora Borradaile, Philip N. Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM J. Comput.*, 46(4):1280–1303, 2017.

[6] Alon Efrat, Alon Itai, and Matthew J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

[7] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[8] Kyle Fox and Jiashuai Lu. A Near-Linear Time Approximation Scheme for Geometric Transportation with Arbitrary Supplies and Spread. In *36th International Symposium on Computational Geometry*, pages 45:1–45:18, 2020.

[9] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[10] Harold N. Gabow. The weighted matching approach to maximum cardinality matching. *Fundam. Inform.*, 154(1-4):109–130, 2017.

[11] Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[12] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[13] Ken-ichi Kawarabayashi and Bruce A. Reed. A separator theorem in minor-closed classes. In *51th Annual IEEE Symposium on Foundations of Computer Science*, pages 153–162, 2010.

[14] Andrey Boris Khesin, Aleksandar Nikolov, and Dmitry Paramonov. Preconditioning for the geometric transportation problem. In *35th International Symposium on Computational Geometry*, pages 15:1–15:14, 2019.

[15] Harold Kuhn. Variants of the Hungarian method for assignment problems. *Naval Research Logistics*, 3(4):253–258, 1956.

[16] Nathaniel Lahn and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite matching in minor-free graphs. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 569–588, 2019.

[17] Nathaniel Lahn and Sharath Raghvendra. An $\tilde{O}(n^{5/4})$ time ε-approximation algorithm for RMS matching in a plane. In *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 869–888, 2021.

[18] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}(\text{vrank})$ iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science*, pages 424–433, 2014.

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

[20] Marcin Mucha and Piotr Sankowski. Maximum matchings via gaussian elimination. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 248–255, 2004.

[21] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via gaussian elimination. *Algorithmica*, 45(1):3–20, 2006.

[22] Jeff M. Phillips and Pankaj K. Agarwal. On bipartite matching under the RMS distance. In *Proceedings of the 18th Annual Canadian Conference on Computational Geometry*, pages 143–146, 2006.

[23] Sharath Raghvendra and Pankaj K. Agarwal. A near-linear time ε-approximation algorithm for geometric bipartite matching. *J. ACM*, 67(3):18:1–18:19, 2020.

[24] R. Sharathkumar. A sub-quadratic algorithm for bipartite matching of planar points with bounded integer coordinates. In *29th International Symposium on Computational Geometry*, pages 9–16, 2013.

[25] R. Sharathkumar and P. K. Agarwal. Algorithms for transportation problem in geometric settings. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 306–317, 2012.

[26] Christian Wulff-Nilsen. Separator theorems for minor-free and shallow minor-free graphs with applications. In *52nd Annual IEEE Symposium on Foundations of Computer Science*, pages 37–46, 2011.