A Weighted Approach to the Maximum Cardinality Bipartite Matching Problem with Applications in Geometric Settings

Nathaniel Lahn  
Virginia Tech, Blacksburg, VA, USA  
lahnn@vt.edu

Sharath Raghvendra  
Virginia Tech, Blacksburg, VA, USA  
sharathr@vt.edu

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 $O(r)$ vertices and $O(mr/n)$ edges. We present an algorithm to compute a maximum cardinality matching in $G$ in $\tilde{O}(m(\sqrt{w} + \sqrt{f} + \frac{wr}{n}))$ 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 $O(m\sqrt{n})$ 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 $wr = O(n^\gamma)$ for $\gamma < 3/2$, then we can compute maximum cardinality matching in $G$ in $o(m\sqrt{n})$ time. Using our algorithm, we obtain a new $\tilde{O}(n^{\delta/3} \varepsilon^4)$ time algorithm to compute an $\varepsilon$-approximate bottleneck matching of $A, B \subseteq \mathbb{R}^2$ and an $\frac{1}{\varepsilon} n\left\lfloor \frac{1}{\varepsilon^4} \right\rfloor + \text{polylog} n$ time algorithm for computing $\varepsilon$-approximate bottleneck matching in $d$-dimensions. All previous algorithms take $\Omega(n^{3/2})$ time. Given any graph $G(A \cup B, E)$ that has an easily computable balanced vertex separator for every subgraph $G'(V', E')$ of size $|V'|^\delta$, for $\delta \in [1/2, 1)$, we can apply our algorithm to compute a maximum matching in $\tilde{O}(mn^{\frac{1}{\delta}})$ time improving upon the $O(m\sqrt{n})$ 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 $c(a, b)$. The weight of any subset of edges $E' \subseteq E$ is given by $\sum_{(a, b) \in E'} c(a, b)$. A minimum-weight maximum matching is a maximum matching 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 $G$.

---

$^1$ We use $\tilde{O}$ to suppress poly-logarithmic terms.
48:2 Weighted Approach to Maximum Cardinality Bipartite Matching

Maximum matching in graphs. In an arbitrary bipartite graph with \( n \) vertices and \( m \) edges, Ford and Fullerson’s algorithm [7] iteratively computes, in each phase, an augmenting path in \( O(m) \) time, leading to a maximum cardinality matching in \( O(mn) \) time. Hopcroft and Karp’s algorithm (HK-Algorithm) [10] reduces the number of phases from \( n \) to \( O(\sqrt{m}) \) by computing a maximal set of vertex-disjoint shortest augmenting paths in each phase. A single phase can be implemented in \( O(m) \) time leading to an overall execution time of \( O(m\sqrt{n}) \). In weighted bipartite graphs with \( n \) vertices and \( m \) edges, the well-known Hungarian method computes a minimum-weight maximum matching in \( O(m\sqrt{n}) \) time [11]. Gabow and Tarjan designed a weight-scaling algorithm (GT-Algorithm) to compute a minimum-weight perfect matching in \( O(m\sqrt{n}\log(nC)) \) time, provided all edge weights are integers bounded by \( C \) [8]. 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 \( O(m\sqrt{n}) \) time algorithm [9].

Recently Lahn and Raghvendra [12] gave \( \tilde{O}(n^{6/5}) \) and \( \tilde{O}(n^{7/5}) \) time algorithms for finding a minimum-weight perfect bipartite matching in planar and \( K_h \)-minor-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 \( \{R_1, \ldots, R_k\} \) such that \( k = \tilde{O}(n/\sqrt{r}) \), every cluster \( R_j \) has \( O(r) \) vertices, and each cluster has \( \tilde{O}(\sqrt{r}) \) 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 \( \tilde{O}(n/\sqrt{r}) \). 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 [14, 15].

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 \( (\sum_{(a,b) \in M} \|a - b\|^p)^{1/p} \). 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 matching in a plane can be computed in \( \tilde{O}(n^{3/2+\delta}) \) [17] time for an arbitrary small \( \delta > 0 \); see also Sharathkumar and Agarwal [19]. Efrat et al. present an algorithm to compute a bottleneck matching in the plane in \( \tilde{O}(n^{3/2}) \) [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 \( \varepsilon \)-approximation algorithm for the Euclidean bipartite matching problem that runs in \( \tilde{O}(n/\varepsilon^d) \) time [18]. However, for \( p > 1 \), all known \( \varepsilon \)-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 \( \tilde{O}(n) \) time max-flow algorithm [5]. There are numerous other results; see also [2, 3, 16]. Designing exact and approximation algorithms that break the \( \Omega(n^{3/2}) \) barrier remains an important research challenge in computational geometry.

\[\text{They assume } h = O(1).\]
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 $|V_i| = O(r)$ and $|E_i| = O(mr/n)$. Given $G$, we present an algorithm to compute a maximum matching in $G$ in $\tilde{O}(m(\sqrt{n} + \sqrt{r} + \frac{mr}{n}))$ time. Consider any graph in which removal of sub-linear number of "separator" vertices partitions the graph into connected components with $O(r)$ vertices and $O(mr/n)$ 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 $O(m\sqrt{n})$ 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 $\gamma < 3/2$, $wr = O(n^\gamma)$, then we can compute a maximum matching in $G$ in $o(m\sqrt{n})$ time. Using our algorithm, we obtain the following result for bottleneck matching:

Given two point sets $A, B \subset \mathbb{R}^2$ and an $0 < \varepsilon \leq 1$, we reduce the problem of computing an $\varepsilon$-approximate bottleneck matching to computing a maximum cardinality matching in a subgraph $G$ of the complete bipartite graph on $A$ and $B$. We can, in $O(n)$ time assign 0/1 weights to the $O(n^2)$ edges of $G$ with so that any matching has a weight of $O(n^{2/3})$. Despite possibly $O(n^2)$ edges in $G$, we present an efficient implementation of our graph algorithm with $O(n^{3/2}/\varepsilon^4)$ execution time that computes an $\varepsilon$-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 $\varepsilon$-approximate bottleneck matching in $rac{1}{\varepsilon^2} \cdot n^{\frac{1}{d} + \frac{1}{2d} + \sqrt{\frac{d}{2}}} \log n$ time. (See Section 5).

The algorithm of Lahn and Raghvendra [12] for $K_h$-minor free graphs requires the clusters to have a small number of boundary vertices, which is used to create a compact representation of the residual network. This compact representation becomes prohibitively large as the number of boundary vertices increase. For instance, their algorithm has an execution time of $\Omega(m\sqrt{n})$ for the case where $G$ has a balanced vertex separator of $\Theta(n^{2/3})$. Our algorithm, on the other hand, extends to any graph with a sub-linear vertex separator. Given any graph $G'(A \cup B, E')$ that has an easily computable balanced vertex separator for every subgraph $G'(V'_i, E'_i)$ of size $|V'_i|^d$, for $d \in [1/2, 1)$, there is a 0/1 weight assignment on edges of the graph so that the weight of any matching is $O(n^{\frac{1}{d-1}})$ and $r = O(n^{\frac{1}{d-1}})$. This assignment can be obtained by simply recursively sub-dividing the graph using balanced separators until each piece has $O(r)$ vertices and $O(mr/n)$ edges. All edges incident on the separator vertices are then assigned a weight of 1 and all other edges are assigned a weight of 0. As a result, we obtain an algorithm that computes the maximum cardinality matching in $\tilde{O}(mn^{\frac{1}{d-1}})$ time.

Our approach. Initially, we compute, in $O(m\sqrt{n})$ 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 \( \mathcal{O}(w \log w) \). Since each piece has \( \mathcal{O}(mr/n) \) edges, the total time spent revisiting these edges is bounded by \( \mathcal{O}(m r w \log(w)/n) \). The total execution time can therefore be bounded by \( \tilde{\mathcal{O}}(m(\sqrt{w} + \sqrt{r} + \frac{w r}{n})) \).

2 Preliminaries

We are given a bipartite graph \( G(A \cup B, E) \), where any edge \((a, b) \in E\) has a weight \( c(a, b) \) 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.

A matching \( M \) and an assignment of dual weights \( y(\cdot) \) on the vertices of \( G \) is feasible if for any \((a, b) \in A \times B\):

\[
\begin{align*}
y(b) - y(a) &\leq c(a, b) & \text{if } (a, b) \notin M, \quad (1) \\
y(a) - y(b) &\leq c(a, b) & \text{if } (a, b) \in M. \quad (2)
\end{align*}
\]

To assist in describing our algorithm, we first define a residual network and an augmented residual network with respect to a feasible matching \( M, y(\cdot) \). A residual network \( G_M \) with respect to a feasible matching \( M \) is a directed graph where every edge \((a, b) \notin M\) is directed from \( b \) to \( a \) if \((a, b) \notin M \) and from \( a \) to \( b \) if \((a, b) \in M \). The weight \( s(a, b) \) of any edge is given by the slack of this edge with respect to feasibility conditions (1) and (2), i.e., if \((a, b) \notin M \), then \( s(a, b) = c(a, b) + y(a) - y(b) \) and \( s(a, b) = 0 \) otherwise. An augmented residual network is obtained by adding to the residual network an additional vertex \( s \) and additional directed edges from \( s \) to every vertex in \( B_F \), each of 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, \( |M_{OPT}| - |M| \leq w \). 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 \( y(\cdot) \) 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 \( \ell_v \) 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 \( \ell_v \) to \( \infty \). Let

\[
\ell = \min_{v \in A_F} \ell_v. 
\] (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. 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 initiate a DFS from every 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 \( P = \{b = v_1\} \). Every edge of \( G' \) is marked unvisited. At any point during the execution of DFS, the algorithm maintains a simple path \( P = \{b = v_1, v_2, \ldots, v_k\} \). The DFS search 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 \( P = \{v_1\} \), 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 search from \( v_k-1 \).
- If there is an unvisited edge going out of \( v_k \), let \( (v_k, v) \) be this edge. Mark \( (v_k, v) \) as visited. If \( v \) is on the path \( P \), continue the DFS from \( v_k \). If \( v \) is not on the path \( P \), add \( (v_k, v) \) 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 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 \( (b, a) \in P \setminus M \), where \( a \in A \) and \( b \in B \), set \( y(b) \leftarrow y(b) - 2c(a, b) \). Then augment \( M \) along \( P \) by setting \( M \leftarrow M \oplus P \). By doing so, every edge of \( M \) after augmentation satisfies the feasibility condition (2). This completes the description of our algorithm. The algorithm maintains the following invariants during its execution:

\text{SoCG 2019}
(11) The matching \( M \) and the set of dual weights \( y(\cdot) \) are feasible. Let \( y_{\text{max}} = \max_{v \in B} y(v) \). The dual weight of every vertex \( v \in B_F \) is \( y_{\text{max}} \) and the dual weight for every vertex \( v \in A_F \) is 0.

(12) 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 6 and Lemma 7) in the current phase. Therefore, we can safely delete them.

**Correctness.** From Invariant (12), 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 \( K_i \) 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 \( i \)th 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. Suppose there are \( \lambda \) 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 \( \lambda \) phases is bounded by \( \mathcal{O}((m + n \log n)\lambda + (mr/n)\sum_{i=1}^{t} |K_i|) \). In Lemma 3, we bound \( \lambda \) by \( \sqrt{w} \) and \( \sum_{i=1}^{t} |K_i| \) by \( \mathcal{O}(w \log w) \). Therefore, the total time taken by the algorithm including the time taken by preprocessing step is \( \mathcal{O}(m\sqrt{r} + m\sqrt{w} + n\sqrt{w} \log n + \frac{mrw \log w}{n}) = \tilde{\mathcal{O}}(m(\sqrt{w} + \sqrt{r} + \frac{wr}{n})) \).

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

\[
c(P) = y_{\text{max}} + \sum_{(a,b) \in P} s(a, b).
\]

**Proof.** The weight of \( P \) is

\[
c(P) = \sum_{(a,b) \in P} c(a,b) = \sum_{(a,b) \in P \cap M} (y(b) - y(a) + s(a, b)) + \sum_{(a,b) \in P \setminus M} (y(a) - y(b)).
\]

Since every vertex on \( P \) except for \( u \) and \( v \) participates in one edge of \( P \cap M \) and one edge of \( P \setminus M \), we can write the above equation as

\[
c(P) = y(u) - y(v) + \sum_{(a,b) \in P \setminus M} s(a, b) = y(u) - y(v) + \sum_{(a,b) \in P} s(a, b).
\]
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 \( y(u) = y_{\text{max}} \) and \( y(v) = 0 \), which gives,

\[
c(P) = y_{\text{max}} + \sum_{(a,b) \in P} s(a,b).
\]

\( \triangleright \)

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

\[
\sum_{(a,b) \in P} s(a,b) > 0,
\]

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

**Proof.** The claim follows from (4) and the fact that the first vertex \( u \) and the last vertex \( v \) in a cycle are the same. \( \triangleright \)

**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., \( \sum_{i=1}^{t} |K_i| = \mathcal{O}(w \log w) \).

**Proof.** Let \( M_{\text{OPT}} \) be a maximum matching, which has weight at most \( w \). Consider any phase \( k \) of the algorithm. By (I2), the dual weight \( y_{\text{max}} \) of every free vertex in \( B_F \) is at least \( k \). The symmetric difference of \( M \) and \( M_{\text{OPT}} \) will contain \( j = |M_{\text{OPT}}| - |M| \) vertex-disjoint augmenting paths. Let \( \{P_1, \ldots, P_j\} \) be these augmenting paths. These paths contain edges of \( M_{\text{OPT}} \) and \( M \), both of which are of weight at most \( w \). Therefore, the sum of weights of these paths is

\[
\sum_{i=1}^{j} c(P_i) \leq 2w.
\]

Let \( y_{\text{max}} \) be the dual weight of every vertex \( b \) of \( B \) that is free with respect to \( M \), i.e., \( b \in B_F \). From (I2), \( y_{\text{max}} \geq k \). From Lemma 1 and the fact that the slack on every edge is non-negative, we immediately get,

\[
2w \geq \sum_{i=1}^{j} c(P_i) \geq jy_{\text{max}} \geq jk.
\]

(5)

When \( \sqrt{w} \leq k < \sqrt{w} + 1 \), it follows from the above equation 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 \( 3\sqrt{w} \).

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 \( y_{\text{max}}' \) correspond to the dual weight of the free vertices of \( B_F \) when the augmenting path \( P_1 \) is found by the algorithm. From Lemma 1, and the fact that \( P_1 \) is an augmenting path consisting of zero slack edges, we have \( y_{\text{max}}' = c(P_1) \). Before augmenting along \( P_1 \), 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 (5), we get \( c(P_1) = y_{\text{max}}' \leq \frac{2w}{t-l+1} \). Summing over all \( 1 \leq l \leq t \), we get,

\[
\sum_{l=1}^{t} c(P_l) \leq w \sum_{l=1}^{t} \frac{2}{t-l+1} = \mathcal{O}(w \log t) = \mathcal{O}(w \log w).
\]

(6)
For any augmenting path $P_l$, the number of affected pieces is upper bounded by the number of non-zero weight edges on $P_l$, i.e., $|K_l| \leq c(P_l)$. Therefore,

$$\sum_{l=1}^{t} |K_l| \leq \sum_{l=1}^{t} c(P_l) = O(w \log w).$$

\section{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 $k$.

\begin{lemma}
Any matching $M$ and dual weights $y(\cdot)$ maintained during the execution of the algorithm are feasible.
\end{lemma}

Next we show that the dual weights $A_F$ are zero and the dual weights of all vertices of $B_F$ are equal to $y_{\text{max}}$. 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 $\ell$, which is the largest possible increase. This implies that every free vertex in $B_F$ will have the same dual weight of $y_{\text{max}}$. 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 $y_{\text{max}}$ 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.

\begin{lemma}
After the first stage of each phase, there is an augmenting path consisting of admissible edges.
\end{lemma}

\begin{proof}[(I2)] From Lemma 5, 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 8 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 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 $(u, v)$, there is no path in the admissible graph that starts from the edge $(u, v)$ and ends at a free vertex $a \in A_F$ (Lemma 7). In Lemma 6, we show that any such edge $(u, v)$ will not participate in any admissible alternating path to a free vertex of $A_F$ for the rest of the phase.

\begin{lemma}
Consider some point during the second stage of phase $k$ where there is an edge $(u, v)$ that does not participate in any admissible alternating path to a vertex of $A_F$. Then, for the remainder of phase $k$, $(u, v)$ does not participate in any admissible alternating path to a vertex of $A_F$.
\end{lemma}
Lemma 7. 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 $(u,v)$ was deleted during this DFS. Then there is no admissible path starting with $(u,v)$ and ending at a free vertex $a \in A_F$ in $G_M$.

By combining Lemmas 6 and 7, we get the following corollary.

Corollary 8. At the end of any phase, there is no augmenting path of admissible edges.

5 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 $\beta^*$. An $\varepsilon$-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 $\delta$ such that $\beta^* \leq \delta \leq (1 + \varepsilon/3)\beta^*$, and produces an $\varepsilon$-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 $\delta$, our algorithm executes in $\tilde{O}(n^{4/3}/\varepsilon^3)$ time.

Although, the value of $\delta$ is not known to the algorithm, we can first find a value $\alpha$ that is guaranteed to be an $n$-approximation of the bottleneck distance [1, Lemma 2.2] and then select $O(\log n/\varepsilon)$ values from the interval $[\alpha/n, \alpha]$ of the form $(1 + \varepsilon/3)^i \alpha/n$, for $0 \leq i \leq O(\log n/\varepsilon)$. We will then execute our algorithm for each of these $O(\log n/\varepsilon)$ selected values of $\delta$. Our algorithm returns a maximum matching whose edges are of length at most $(1 + \varepsilon/3)\delta$ in $O(n^{4/3}/\varepsilon^3)$ time. At least one of the $\delta$ values chosen will be a $\beta^* \leq \delta \leq (1 + \varepsilon/3)\beta^*$. The matching returned by the algorithm for this value of $\delta$ 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. Therefore, the total time taken to compute the $\varepsilon$-approximate bottleneck matching is $\tilde{O}(n^{4/3}/\varepsilon^4)$.

Given the value of $\delta$, the algorithm will construct a graph as follows: Let $G$ 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 $\xi$ in the grid $G$, let $N(\xi)$ denote the subset of all cells $\xi'$ of $G$ such that the minimum distance between $\xi$ and $\xi'$ is at most $\delta$. By the use of a simple packing argument, it can be shown that $|N(\xi)| = O(1/\varepsilon^2)$.

For any point $v \in A \cup B$, let $\xi_v$ be the cell of grid $G$ that contains $v$. We say that a cell $\xi$ is active if $(A \cup B) \cap \xi \neq \emptyset$. Let $A_\xi$ and $B_\xi$ denote the points of $A$ and $B$ in the cell $\xi$. We construct a bipartite graph $G(A \cup B, E)$ 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 $G$. Since $\delta$ is at least the bottleneck distance, $G$ will have a perfect matching. The maximum distance between any cell $\xi$ and a cell in $N(\xi)$ is $(1 + \varepsilon/3)\delta$. Therefore, no edge in $G$ will have a length greater than $(1 + \varepsilon/3)\delta$. This implies that any perfect matching in $G$ will also be an $\varepsilon$-approximate bottleneck matching. We use our algorithm for maximum matching to compute this perfect matching in $G$. Note, that $G$ can have $\Omega(n^2)$ edges. For the sake of efficiency, our algorithm executes on a compact representation of $G$ that is described later. Next, we assign weights of 0 and 1 to the edges of $G$ so that the any maximum matching in $G$ has a small weight $w$. 
For a parameter\(^3 r > 1\), we will carefully select another grid \(G'\) 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 \(G\). For any cell \(\xi\) of the grid \(G\), let \(\square_{\xi}\) be the cell in \(G'\) that contains \(\xi\). Any cell \(\xi\) of \(G\) is a boundary cell with respect to \(G'\) if there is a cell \(\xi' \in \mathcal{N}(\xi)\) such that \(\square_{\xi'} \neq \square_{\xi}\). Equivalently, if the minimum distance from \(\xi\) to \(\square_{\xi}\) is at most \(\delta\), then \(\xi\) is a boundary cell. For any boundary cell \(\xi\) of \(G\) with respect to grid \(G'\), we refer to all points of \(A_{\xi}\) and \(B_{\xi}\) that lie in \(\xi\) as boundary points. All other points of \(A\) and \(B\) are referred to as internal points. We carefully construct this grid \(G'\) such that the total number of boundary points is \(O(n/\varepsilon\sqrt{r})\) as follows: First, we will generate the vertical lines for \(G'\), and then we will generate the horizontal lines using a similar construction. Consider the vertical line \(y_{ij}\) to be the line \(x = \frac{i(\varepsilon\delta)}{(6\sqrt{2})} + \frac{j\sqrt{r}(\varepsilon\delta)}{(6\sqrt{2})}\). For any fixed integer \(i\) in \([1, \sqrt{r}]\), consider the set of vertical lines \(\mathcal{V}_i = \{y_{ij} | y_{ij} \text{ intersects the bounding square } S\}\). We label all cells \(\xi\) of \(G\) as boundary cells with respect to \(\mathcal{V}_i\) if the distance from \(\xi\) to some vertical line in \(\mathcal{V}_i\) is at most \(\delta\). We designate the points inside the boundary cells as boundary vertices with respect to \(\mathcal{V}_i\). For any given \(i\), let \(A_i\) and \(B_i\) be the boundary vertices of \(A\) and \(B\) with respect to the lines in \(\mathcal{V}_i\). We select an integer \(\kappa = \min_{1 \leq i \leq \sqrt{r}} |A_i \cup B_i|\) and use \(\mathcal{V}_\kappa\) as the vertical lines for our grid \(G'\). We use a symmetric construction for the horizontal lines.

\textbf{Lemma 9.} Let \(A_i\) and \(B_i\) be the boundary points with respect to the vertical lines \(\mathcal{V}_i\). Let \(\kappa = \min_{1 \leq i \leq \sqrt{r}} |A_i \cup B_i|\). Then, \(|A_\kappa \cup B_\kappa| = O(n/(\varepsilon\sqrt{r}))\).

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

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

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

\textbf{Corollary 10.} The grid \(G'\) that we construct has \(O(n/(\varepsilon\sqrt{r}))\) many boundary points.

For any two cells \(\xi\) and \(\xi' \in \mathcal{N}(\xi)\) of the grid \(G\), suppose \(\square_{\xi} \neq \square_{\xi'}\). 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, 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 10, it follows that any maximum matching will have a weight of \(w = O(n/(\varepsilon\sqrt{r}))\).

The edges of every piece in \(G\) have endpoints that are completely inside a cell of \(G'\). Note, however, that there is no straight-forward bound on the number of points and edges of \(G\) inside each piece. Moreover, the number of edges in \(G\) can be \(\Theta(n^2)\). Consider any feasible matching \(M, y(\cdot)\) in \(G\). Let \(G_M\) be the residual network. In order to obtain a running time of \(O(n^{4/3}/\varepsilon^{3})\), we use the grid \(G\) to construct a compact residual network \(CG_M\) 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.

\(^3\) Assume \(r\) to be a perfect square.
Lemma 11. Consider any feasible matching $M$, $y(\cdot)$ maintained by our algorithm on $G$ and any active cell $\xi$ in the grid $G$. 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 $y(b') < y(b) \leq y_{\text{max}}$. Let $m(b') \in A$ be the match of $b'$ in $M$. From (2), $y(m(b')) - y(b') = c(b', m(b'))$. Since both $b$ and $b'$ are in $\xi$, the distance $c(b, m(b')) = c(b', m(b'))$. So, $y(b) - y(m(b')) \geq (y(b') + 3) - y(m(b')) = 3 - c(b, m(b'))$. This violates (1) leading to a contradiction.

For any feasible matching and any cell $\xi$ of $G$, we divide points of $A_\xi$ and $B_\xi$ based on their dual weight into at most three clusters. Let $A_\xi^1, A_\xi^2$ and $A_\xi^3$ be the three clusters of points in $A_\xi$ and let $B_\xi^1, B_\xi^2$ and $B_\xi^3$ be the three clusters of points in $B_\xi$. We assume that points with the largest dual weights are in $A_\xi^1$ (resp. $B_\xi^1$), the points with the second largest dual weights are in $A_\xi^2$ (resp. $B_\xi^2$), and the points with the smallest dual weights are in $A_\xi^3$ (resp. $B_\xi^3$).

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

Next, we create the edge set for the compact residual network $CG$. For any active cell $\xi$ in the grid $G$ and for any cell $\xi' \in N(\xi)$,

- We add a directed edge from $a^i_\xi$ to $b^i_\xi$, for $i, j \in \{1, 2, 3\}$ if there is an edge $(a, b) \in (A_\xi^i \times B_\xi^j) \cap M$. We define the weight of $(a^i_\xi, b^j_\xi)$ to be $c(a, b)$. We also define the slack $s(a^i_\xi, b^j_\xi)$ to be $c(a^i_\xi, b^j_\xi) - y(a^i_\xi) + y(b^j_\xi)$ which is equal to $c(a, b) - (y(a) + y(b) = s(a, b) = 0)$.

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

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

The following lemma shows that the compact graph $CG$ preserves all minimum slack paths in $G_M$. 

\[ \text{Lemma 11.} \] Consider any feasible matching $M$, $y(\cdot)$ maintained by our algorithm on $G$ and any active cell $\xi$ in the grid $G$. 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.
Lemma 12. For any directed path $\mathcal{P}$ in the compact residual network $\mathcal{CG}$, there is a directed path $\mathcal{P}$ in the residual network such that $\sum_{(u,v) \in \mathcal{P}} s(u,v) = \sum_{(u,v) \in \mathcal{P}} s(u,v)$. For any directed path $\mathcal{P}$ in $\mathcal{GM}$, there is a directed path $\mathcal{P}$ in the compact residual network such that $\sum_{(u,v) \in \mathcal{P}} s(u,v) \geq \sum_{(u,v) \in \mathcal{P}} s(u,v)$.

Preprocessing step. At the start, $M = \emptyset$ and all dual weights are 0. Consider any cell $\Box$ of the grid $\mathcal{G}'$ and any cell $\xi$ of $\mathcal{G}$ that is contained inside $\Box$. Suppose we have a point $a^1_{\xi}$. We assign a demand $d^1_{\xi} = |A^1_{\xi}| = |A^1_{\xi}|$ to $a^1_{\xi}$. Similarly, suppose we have a point $b^1_{\xi}$, we assign a supply $s^1_{\xi} = |B^1_{\xi}| = |B^1_{\xi}|$. 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 [13] in $\tilde{O}(\sqrt{|E|}) = \tilde{O}(\sqrt{r})$. Every edge of $\mathcal{E}$ participates in at most one piece. Therefore, the total time taken for preprocessing across all pieces is $\tilde{O}(\sqrt{r}) = \tilde{O}(\log(n/\varepsilon^2))$. We can trivially convert the matching of supplies to demand to 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 $\mathcal{CG}$ in $\tilde{O}(n\log n/\varepsilon^2)$ 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 vertex. 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 of the free vertices 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 $\tilde{O}(w \log w)$.

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 $\lambda$ phases; then the cumulative time taken across all phases for the first stage is $\tilde{O}(\lambda n/\varepsilon^2)$. 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 $\lambda$ phases is bounded by $\mathcal{O}(n/\varepsilon^2 + r/\varepsilon^2 \sum_{i=1}^\lambda |K_i| + w \log w)$. In Lemma 3, we bound $\lambda$ by $\sqrt{w}$ and $\sum_{i=1}^\lambda |K_i|$ by $\mathcal{O}(w \log w)$. Therefore, the total time taken by the algorithm including the time taken by preprocessor step is $\tilde{O}((n/\varepsilon^2)(\sqrt{r} + \sqrt{w} + \frac{w}{n}))$. Setting $r = n^{2/3}$, we get $w = \mathcal{O}(n(\sqrt{r})) = \mathcal{O}(n^{2/3}/\varepsilon)$, and the total running time of our algorithm is $\tilde{O}(n^{4/3}/\varepsilon^3)$. To obtain the bottleneck matching, we execute this algorithm on $\mathcal{O}(\log(n/\varepsilon))$ guesses; therefore, the total time taken to compute an $\varepsilon$-approximate bottleneck matching is $\tilde{O}(n^{4/3}/\varepsilon^3)$. For $d > 2$, we choose $r = \frac{n\lambda}{\sqrt{d - 1}}$ and $w = \mathcal{O}(n/(d\varepsilon^{1/d}))$. With these values, the execution time of our algorithm is $\frac{1}{\varepsilon^{2(d-1)}} n^{1 + \frac{2d}{d-1}} \log n$. 


References


