ARTICLE IN PRESS

Discrete Applied Mathematics xxx (xxxx) xxx

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

A 2/3-approximation algorithm for vertex-weighted matching[☆]

Ahmed Al-Herz, Alex Pothen [*]

Computer Science Department, Purdue University, West Lafayette, IN 47907, USA

ARTICLE INFO

Article history:
Received 26 October 2018
Received in revised form 16 September 2019
Accepted 25 September 2019
Available online xxxx

Keywords:
Vertex-weighted matching
Approximation algorithm
Heaviest unmatched neighbor
Augmenting path
Weight-increasing path

ABSTRACT

We consider the maximum vertex-weighted matching problem (MVM) for non-bipartite graphs in which non-negative weights are assigned to the vertices of a graph and a matching that maximizes the sum of the weights of the matched vertices is desired. In earlier work we have described a 2/3-approximation algorithm for the MVM on bipartite graphs (Florin Dobrian et al., 2019). Here we show that a 2/3-approximation algorithm for MVM on non-bipartite graphs can be obtained by restricting the length of augmenting paths to at most three. The algorithm has time complexity $O(m \log \Delta + n \log n)$, where n is the number of vertices, m is the number of edges, and Δ is the maximum degree of a vertex.

The approximation ratio of the algorithm is obtained by considering *failed vertices*, i.e., vertices that the approximation algorithm fails to match but the exact algorithm does. We show that there are two distinct heavier matched vertices that we can charge each failed vertex to. Our proof techniques characterize the structure of augmenting paths in a novel way.

We have implemented the 2/3-approximation algorithm and show that it runs in under a minute on graphs with tens of millions of vertices and hundreds of millions of edges. We compare its performance with five other algorithms: an exact algorithm for MVM, an exact algorithm for the maximum edge-weighted matching (MEM) problem, as well as three approximation algorithms. The approximation algorithms include a 1/2-approximation algorithm for MVM, and $(2/3-\epsilon)$- and $(1-\epsilon)$-approximation algorithms for the MEM. In our test set of nineteen problems, there are graphs on which the exact algorithms fail to terminate in 100 hours. In addition, the new 2/3-approximation algorithm for MVM outperforms the other approximation algorithms by either being faster (often by orders of magnitude) or obtaining better weights.

1. Introduction

We consider a variant of the matching problem in non-bipartite graphs in which weights are assigned to the vertices of a graph, the weight of a matching is the sum of the weights of the matched vertices, and we find a matching of maximum weight. We call this the maximum vertex-weighted matching problem (MVM). In this paper we describe a 2/3-approximation algorithm for the MVM that has $O(m \log \Delta + n \log n)$ time complexity, where n is the number of

☆ This work was supported in part by U.S. National Science Foundation grant CCF-1637534; the U.S. Department of Energy through grant DE-SC0010205; and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE Office of Science and the NNSA.

* Corresponding author.
E-mail addresses: aalherz@purdue.edu (A. Al-Herz), apothen@purdue.edu (A. Pothen).

https://doi.org/10.1016/j.dam.2019.09.013
0166-218X/© 2019 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

vertices, m is the number of edges, and Δ is the maximum degree of a vertex. We implement this algorithm as well as a few other approximation algorithms for this problem, and show that the 2/3-approximation algorithm runs fast on large graphs, obtains weights that are close to optimal, and is faster or obtains greater weights than the other algorithms on our test set.

Consider an undirected vertex weighted graph $G = (V, E, \phi)$, where $|V| \equiv n$ is the number of vertices, $|E| \equiv m$ is the number of edges, and $\phi : V \mapsto R_{\geq 0}$ is a non-negative weight function on the vertices. The MVM problem can be solved in polynomial time by an exact algorithm [32] with $O(\sqrt{n}m \log n)$ time complexity. We have designed and implemented an exact algorithm with $O(mn)$ time complexity [10], since it is easier to implement, and it is well-known that the practical performance of a matching algorithm does not necessarily correlate with its worst-case time complexity. We show that this exact algorithm can be slow for many large graphs with millions of vertices and edges, and can even fail to terminate in 100 hours. Thus, there is a need for faster approximation algorithms that can return a matching with a guaranteed fraction of the maximum weight.

Many linear time approximation algorithms have been designed for the maximum edge weighted matching problem (MEM), but we are not aware of earlier approximation algorithms for the MVM problem on non-bipartite graphs. We have designed and implemented a 2/3-approximation algorithm for MVM in bipartite graphs [10]. The MVM problem arises in applications such as the design of network switches [33], schedules for training of astronauts [7], computation of sparse bases for the null space or the column space of a rectangular matrix [8,27,28], etc.

A more recent application of vertex-weighted matching in bipartite graphs arises in internet advertising. In a simplified model, U is a set of advertisers known at the beginning of the algorithm, and V is a set of keyword searches, which arrive online during the execution of the algorithm. Each advertiser $u \in U$ expresses interest in placing ads for a subset of keywords, and will pay $\phi(u)$ units of money for placing the ad. The problem is to find a set of ad placements that maximizes the money spent. Here the order of arrival of the keywords V is unknown, and the problem is to design an online MVM algorithm that is as close to the optimal (when V is fully known) as possible. Aggrawal, Goel, Karande and Mehta [2] design an online algorithm for this problem that computes a weight that is at least $(1 - 1/e)$ of the optimal, assuming that the vertices in V arrive in random order.

Mehta [24] surveys several ad allocation problems and the online algorithms that have been designed for this problem. He states that "internet advertising constitutes perhaps the largest matching problems in the world, both in terms of [money] and numbers of items". He also asks for "a fast simple offline approximation algorithm [for non-bipartite matching] as opposed to the optimum algorithm, especially when the data is very big" (Section 10.1.2). Our work describes precisely such an off-line algorithm for MVM that beats the competitive ratio of this known on-line algorithm. Indeed, the MVM problem can be approximated arbitrarily close to 1 using an approximation algorithm for the maximum edge-weighted matching problem, although as we show later, its practical performance is not as good as the algorithm we describe here.

The MVM problem can be transformed to an MEM problem by assigning each edge a weight obtained by summing the weights at its endpoints. Hence algorithms for MEM can be used to solve MVM problems. However, this transformation can lead to increase in run times for an exact MEM algorithm by three orders of magnitude or more relative to the time for computing an MEM when the weights are random [10]. A simpler, more efficient, and faster exact algorithm is obtained by solving the MVM problem directly by processing vertices in non-increasing order of weights and then matching an unmatched vertex to a heaviest unmatched vertex it can reach by an augmenting path. In this sense the MVM problem is more similar to maximum cardinality matching than MEM.

The first 2/3-approximation algorithm for MVM on *bipartite graphs* was proposed by us and our coauthors [10]. The idea is to decompose the problem into two 'one-side-weighted' problems, solve them individually by restricting the length of augmenting paths to at most three, and then combine the two matchings into a final matching by invoking the Mendelsohn–Dulmage theorem [25]. Restricting augmenting paths to length three does not lead to 2/3-approximation algorithm for the MEM; however more sophisticated $(2/3 - \epsilon)$-algorithms are available [26].

The 2/3-approximation algorithm for non-bipartite graphs described here is similar to the 2/3-approximation algorithm for bipartite graphs: Both process unmatched vertices in non-increasing order of weights, and match an unmatched vertex to a heaviest unmatched vertex reachable by an augmenting path of length at most three. The approximation ratios of both algorithms are obtained by showing that for each failed vertex (a vertex that the approximation algorithm fails to match but the exact algorithm does) there are two distinct matched vertices with at least the weight of the failed vertex. However, the one-side weighted decomposition technique used for bipartite graphs cannot be applied to non-bipartite graphs. Hence the proof of correctness of the 2/3-approximation algorithm for non-bipartite graphs has to be different from the one for bipartite graphs.

For non-bipartite graphs, we need a more careful study of the structure of augmenting paths. We distinguish between the origin (the first vertex) and the terminus (the last vertex) of such paths. The origin and terminus of an augmenting path are *corresponding vertices* of each other, and this pairing is uniquely determined in our algorithm. We also introduce the concept of a heaviest unmatched neighbor of a matched vertex. We consider the symmetric difference of an optimal matching and an approximate matching, and then examine the first five vertices on a path that begins at a failed vertex and alternates between edges in the two matchings. We show that this alternating path does not change in future augmentation steps of the approximation algorithm, and prove that the weight of a failed vertex is no larger than the *corresponding vertices* of two of the vertices on this path. However, the corresponding vertices themselves may not be on the augmenting path. The proof makes use of heaviest unmatched neighbors to establish relationships among the weights of the vertices. Thus while the algorithm is simple to state, the proof that its approximation ratio is 2/3 requires several new concepts.

2. Background and related work

2.1. Background on matchings

We define the basic terms we use here, and additional background on matchings is available in several books, such as [31]. Only undirected graphs are considered in this paper; we use the notation (u, v) for an undirected edge, and its endpoints are the vertices u and v. A *matching* M in a graph $G = (V, E)$ is a set of vertex-disjoint edges; hence at most one edge from M is incident on each vertex in the graph. An edge $(u, v) \in E$ is a matching edge if it belongs to M, and a non-matching edge otherwise. A vertex $v \in V$ is matched if it is an end point of a matching edge, and otherwise it is unmatched. A *heaviest unmatched neighbor* of u is denoted by HUN(u); note that HUN(u) might not be unique, but its weight is.

A *path* P in G is a finite sequence of distinct vertices $\{v_i, v_{i+1}, ..v_{i+k}\}$ such that $(v_j, v_{j+1}) \in E$ for $i \le j \le i + k - 1$. The length of a path $|P|$ is the number of edges in the path. A *cycle* is a path concatenated with an edge joining its first and last vertices. An *alternating path* P with respect to M is a path whose edges alternate between edges in the matching M and edges not in the matching. If the first and last vertices on an M-alternating path P are unmatched, then it is an M-*augmenting path*, which necessarily has an odd number of edges. The matching M can be augmented by matching the edges in the symmetric difference $M \oplus P$.

When an augmentation is performed, we distinguish between the *origin* (the vertex from which an augmenting path search is initiated), and the *terminus* (the vertex at which the augmenting path search ends). We will denote the origin of the $i-$th augmentation step by o_i, and the terminus by t_i, where $i \ge 1$. When a vertex is not explicitly denoted by o_i or t_i, then it could be either an origin or a terminus, or neither, unless mentioned otherwise. Note that if we begin with the empty matching, then each matching edge is obtained through an augmentation step, so that $|M|$ is equal to the number of origins or termini. We will say that the ith origin and the ith terminus *correspond* to each other, so that $o_i(t_i)$ is the corresponding vertex of the vertex $t_i(o_i)$. An *alternating path* P with respect to two matchings, $M_1 \oplus M_2$, is a path whose edges alternate between edges in the matchings M_1 and M_2.

2.2. Related work

We now describe more fully the work we have done earlier with our colleagues on exact algorithms for MVM on general graphs, and a 2/3-approximation algorithm for this problem on bipartite graphs [10]. When vertex weights are non-negative, we can choose a maximum vertex-weighted matching to be one of maximum cardinality, and from now on we assume that this choice has been made.

For a matching M, an M-*reversing path* is an alternating path with an even number of edges consisting of an equal number of matching and non-matching edges. An M-*increasing path* is an M-reversing path whose unmatched endpoint has higher weight than its matched endpoint. By switching the matching and non-matching edges on this path, we can increase the weight of the matching, much as we would using an augmenting path. There are two ways of characterizing maximum vertex-weighted matchings in a general graph. The first is that a matching M is an MVM if and only if there is (i) neither an M-augmenting path (ii) nor an M-increasing path in the graph. The second is to list the weights of matched vertices in non-increasing order in a vector (this is the weight vector of the matching M). Then a matching M is an MVM if and only if its weight vector is lexicographically maximum among all the weight vectors of matchings.

These two characterizations lead to two extreme algorithms for computing an MVM. The first begins with the empty matching, and at each step matches a currently heaviest unmatched vertex to a heaviest unmatched vertex it can reach by augmenting path. In this algorithm once a vertex is matched, it will always remain matched, since augmentation does not change a matched vertex to an unmatched vertex. An algorithm with the approximation ratio of 2/3 for MVM on bipartite graphs was designed and implemented in [10].

The second exact algorithm for solving the MVM problem begins with a maximum cardinality matching. It then looks for increasing paths or cycles with respect to the current matching and terminates when there is none. This second, increasing path algorithm, has the advantage that it has more concurrency whereas the first algorithm has to process vertices in a specified order. We will discuss the increasing path algorithm in future work.

Now we turn to approximation algorithms that have been designed for the maximum edge weighted matching problem (MEM). The well-known Greedy algorithm [5] iteratively adds a heaviest edge to the matching, and deletes all edges incident on the endpoints of the added edge. This algorithm is 1/2-approximate and requires $O(m \log n)$ time. Another 1/2-approximation algorithm, the Locally Dominant edge algorithm [30], avoids sorting the edges by choosing locally dominant edges (an edge that is a heaviest edge incident on both of its endpoints) to add to the matching. A more recent 1/2-approximation algorithm is the Suitor algorithm [20], which employs a proposal-based approach similar to the classical algorithms for stable matching. The Suitor and other algorithms have been extended to find 1/2-approximate b-Matchings [19]. Other papers improve the performance ratios: For any fixed $\epsilon > 0$, $(2/3 - \epsilon)$- and $(3/4 - \epsilon)$-approximation algorithms have been proposed [11,12,17,21,26]. Furthermore, a $(1 - \epsilon)$-approximation algorithm, based on a scaling approach has been proposed by Duan and Pettie [13] which has time complexity $O(m\,\epsilon^{-1} \log \epsilon^{-1})$. We show that the $(1 - \epsilon)$-approximation algorithm when applied to the MVM problem is significantly slower than the 1/2- and 2/3-approximation algorithms, and surprisingly, does not compute greater matching weights for relevant values of ϵ. The MEM problem appears in applications such as placing large elements on the diagonal of sparse matrices [15,16], multilevel graph partitioning [18], scheduling, etc.

3. A two-third approximation algorithm for MVM

In this section we describe a 2/3-approximation algorithm for MVM, discuss its relation to an exact algorithm, and then prove its correctness.

3.1. Exact and 2/3-approximation algorithms

Recall that we consider a graph $G = (V, E)$, with $\phi(v)$ denoting the non-negative weight on a vertex v. The approximation algorithm, described in Algorithm 1, sorts the vertices in non-increasing order of weights, and inserts the sorted vertices into a queue Q. The algorithm begins with the empty matching, and attempts to match the vertices in Q in the given order. Each unmatched vertex u is removed from Q, and beginning at u the algorithm searches for a heaviest unmatched vertex v reachable by an augmenting path of length at most *three*. If an augmenting path is found, then the matching is augmented by the path that leads to a heaviest unmatched vertex v, and the vertex v is also removed from Q. If no augmenting path of length at most three is found, we search from the next heaviest unmatched vertex (even though longer augmenting paths might exist in the graph). The algorithm terminates when all vertices are processed.

The 2/3-approximation algorithm may be viewed as one obtained from an exact algorithm for MVM. In the exact algorithm for MVM, at each step we search from a currently heaviest unmatched vertex for a heaviest unmatched vertex reachable by an augmenting path of any length. If an augmenting path is found, we choose the path that leads to a heaviest unmatched vertex, and then augment by this path. If no augmenting path is found, we search from the next heaviest unmatched vertex. This algorithm was proved correct by Dobrian, Halappanavar, Pothen and Al-Herz in [10]. The time complexity of this algorithm is $O(nm)$.

Consider running the exact algorithm and the 2/3-approximation algorithm simultaneously using the vertices in the same queue Q. Both consider vertices in non-increasing order of weights, and break ties among weights consistently. If a vertex u is matched by the exact algorithm but not by the approximation algorithm (because the augmenting path is longer than three), then we call u a *failure* or a failed vertex, because the approximation algorithm failed to match it while the exact algorithm succeeded.

Algorithm 1 Input: A graph G with weights ϕ on the vertices. Output: A matching M. Effect: Computes a 2/3-approximation to a maximum vertex-weighted matching.

```
1:  procedure TWOTHIRD-APPROX($G = (V, E, \phi)$)
2:      $M \leftarrow \emptyset$;
3:      $Q \leftarrow V$;
4:      while $Q \neq \emptyset$ do
5:          $u \leftarrow heaviest(Q)$;
6:          $Q \leftarrow Q - u$;
7:          Let $v$ denote a heaviest unmatched vertex reachable from $u$ by an augmenting path $P$ of length at most three;
8:          if $P$ is found then
9:              $M \leftarrow M \oplus P$;  $Q \leftarrow Q - v$;
10:         end if
11:     end while
12: end procedure
```

3.2. Time complexity of the two-thirds approximation algorithm

Let Δ denote the maximum degree of a vertex.

Theorem 3.1. *The time complexity of the Two-Thirds approximation algorithm is $O(m \log \Delta + n \log n)$, when the vertex-weights are real-valued.*

Proof. We sort the adjacency list of each vertex in non-increasing order of weights, and maintain a pointer to a heaviest unmatched neighbor of each vertex. Since the adjacency list is sorted, each list is searched once from highest to lowest weight in the algorithm.

Let $N(u)$ be the set of neighbors of a vertex u and $d(u) = |N(u)|$. In each iteration of the while loop, we choose an unmatched vertex u and examine all vertices in $N(u)$ to find a heaviest unmatched neighbor, if one exists. If u has a matched neighbor v, then we form an augmenting path of length three by taking the matching edge (v, w), and finding a heaviest unmatched neighbor x of w. All neighbors of u, unmatched and matched, can be found in $O(d(u))$ time, and finding the matched vertex w and a heaviest unmatched neighbor x can be done in constant time, since the adjacency lists are sorted. Thus the search for augmenting paths in the algorithm takes $O(m)$ time. Sorting the adjacency lists takes time proportional to

$$\sum_u d(u) \log d(u) \leq \sum_u d(u) \log \Delta = m \log \Delta.$$

Sorting the vertices in non-increasing order of weights takes $O(n \log n)$ time. □

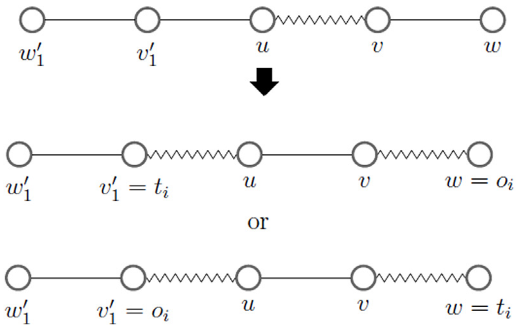

Fig. 1. Lemma 3.2: Base case.

When the vertex-weights are integer values belonging to the range $[0, K]$, then we use an $O(n+K)$-time counting sort, and in this case, the second term in the complexity should be replaced by this expression.

3.3. Correctness of the algorithm

In this subsection we will prove (Theorem 3.7) that Algorithm 1 computes a $\frac{2}{3}$-approximate MVM, M_A. Let $\phi(F)$ denote the sum of the weights of the failures, $\phi(M_A)$ the weight of the approximate matching, and $\phi(M_{opt})$ the weight of an optimal matching. In order to prove the theorem, it suffices to prove that $\phi(F) \leq \frac{1}{2}\phi(M_A)$, since $\phi(M_{opt}) \leq \phi(M_A) + \phi(F)$.

To prove that $\phi(F) \leq \frac{1}{2}\phi(M_A)$, we show that for every failure there are two distinct vertices that are matched in M_A, with weight at least as heavy as the failure. This is achieved in Lemma 3.6 by considering $M_{opt} \oplus M_A$-alternating paths, using a charging technique in which each failure charges two distinct vertices matched in M_A. Each failure is an endpoint of the $M_{opt} \oplus M_A$-alternating path. The two distinct vertices are obtained as the *corresponding vertices* (the other ends of the augmenting paths) of two of the first three vertices on the $M_{opt} \oplus M_A$-alternating path.

We prove the approximation ratio by means of several lemmas. The key Lemma 3.6 is proved using Lemmas 3.2–3.5.

Lemma 3.2. *Let (u, v) be an edge in a matching M at some step in the $2/3$-approximation algorithm, and let $w = \text{HUN}(v)$ be a heaviest unmatched neighbor of v. Suppose (u, v) is changed to a matching edge (u, v') in a future augmentation step, and let $w' = \text{HUN}(v')$ denote a heaviest unmatched neighbor of v', then $\phi(w) \geq \phi(w')$.*

Proof. The proof is by induction on i, the number of augmentation steps that include u on the augmenting path. Let v_i' be the matched neighbor of u after i augmentation steps involving u, and let w_i' be its heaviest unmatched neighbor $\text{HUN}(v_i')$. There are two possible augmentation steps that include the matching edge (u, v). (1) $\{o_i, u, v, t_i\}$, and (2) $\{o_i, v, u, t_i\}$, where o_i (t_i) is the origin (terminus) of the augmenting path.

For the base case, ($i = 1$), consider Fig. 1. If the augmentation path is $\{o_1 = w, v, u, t_1 = v_1'\}$, clearly $\phi(w) \geq \phi(w_1')$, since the algorithm processes vertices in non-increasing order of weights. If the augmenting path is $\{o_1 = v_1', u, v, t_1 = w\}$, then $\phi(w) \geq \phi(w_1')$ because w was matched in preference to w_1'.

Assume the claim is true for k augmentation steps. By using the same argument as in the base case we have $\phi(w_k') \geq \phi(w_{k+1}')$ at the $k+1$-st augmentation step. Now by the inductive hypothesis we have $\phi(w) \geq \phi(w_k')$, and by combining the two inequalities, we obtain $\phi(w) \geq \phi(w_{k+1}')$. □

Lemma 3.3. *Let M_A^x denote the $2/3$-approximate matching at the xth failure f_x, and let $P = \{f_x, v_1, v_2, \ldots\}$ be an alternating path that begins with f_x in $M_{opt} \oplus M_A^x$.*

(1) If v_1 is an origin o_i of some prior augmentation step, then $\phi(t_i) \geq \phi(f_x)$.

(2) $\phi(v_2) \geq \phi(f_x)$.

Proof. (1) If v_1 is an origin o_i, then we have $\phi(t_i) \geq \phi(f_x)$, because t_i was matched in preference to f_x.

(2) In this case, we have to consider three possibilities.

(a) The vertex v_2 is an origin, in which case $\phi(v_2) \geq \phi(f_x)$, since v_2 was processed before f_x.

(b) The vertex v_2 is a terminus that is matched by an augmenting path that includes v_1. An example of this case is shown in Fig. 2. In this case we have two possibilities: either v_1 is an origin and v_2 is the corresponding terminus, or v_1 is previously matched in which case we have an augmenting path $\{o_i, x, v_1, v_2\}$. In both possibilities v_2 was matched in preference to f_x, so $\phi(v_2) \geq \phi(f_x)$.

(c) The vertex v_2 is a terminus that is matched by an augmenting path that includes a vertex $u \neq v_1$, where u is adjacent to v_2. An example of this case is shown in Fig. 3. Let $\text{HUN}(u)$ be a heaviest unmatched neighbor of u after v_2 is matched. In this case, again we have two possibilities: u is an origin and v_2 is the corresponding terminus, or u is previously matched

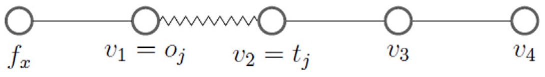

Fig. 2. Lemma 3.3 Case (b): v_2 is a terminus that is matched by an augmenting path that includes v_1.

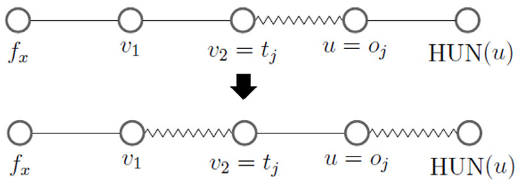

Fig. 3. Lemma 3.3 Case (c): v_2 is a terminus that is matched by an augmenting path that includes $u \neq v_1$.

in which case we have an augmenting path $\{o_i, x, u, v_2\}$. In both possibilities v_2 was matched in preference to HUN(u) so we have $\phi(v_2) \geq \phi(\text{HUN}(u))$. By Lemma 3.2 when the matching edge (u, v_2) is changed to the matching edge (v_1, v_2), we have $\phi(\text{HUN}(u)) \geq \phi(f_x)$. By combining these two inequalities, we obtain $\phi(v_2) \geq \phi(f_x)$. □

Lemma 3.4. *Let M_A^x denote the 2/3-approximate matching at the xth failure f_x, and let $P = \{f_x, v_1, v_2, v_3, \ldots\}$ be an $M_{opt} \oplus M_A^x$-alternating path that begins with f_x. If the vertex v_3 is an origin o_i of some prior augmentation step in the Approximation Algorithm, and if $\phi(t_i) < \phi(f_x)$, then (1) immediately prior to the step when the Approximation Algorithm matches the vertex v_3, the vertex v_2 is matched to a vertex $u \neq v_1$, and $\{v_2, v_3, u\}$ is a cycle.*

(2) the ith augmenting path is $\{v_3 = o_i, u, v_2, t_i\}$.

Proof. (1) First we will establish that v_2 is matched to some vertex u prior to the step when v_3 is matched. To obtain a contradiction, assume that v_2 is not matched to some vertex u prior to the step of matching v_3. Then after v_3 is matched, the terminus t_i is either v_2 or a vertex that is matched in preference to v_2. In both possibilities we have $\phi(t_i) \geq \phi(v_2)$. We know from Lemma 3.3 that $\phi(v_2) \geq \phi(f_x)$. Combining the two inequalities, we have $\phi(t_i) \geq \phi(f_x)$, which contradicts the assumption in the lemma.

Now we show that the vertex $u \neq v_1$. Assume for a contradiction that $u = v_1$, then at the step of matching v_3 there exists an augmenting path from v_3 to f_x of length three. After we match v_3, we have $\phi(t_i) \geq \phi(f_x)$, since it was matched in preference to f_x. This again contradicts the assumption in the lemma.

Now we show that $\{v_2, v_3, u\}$ is a cycle by showing that $v_3 = \text{HUN}(u)$. Assume $v_3 \neq \text{HUN}(u)$ and let some vertex $q = \text{HUN}(u)$, as shown in Fig. 4. Note that by Lemma 3.2 we have

$$\phi(q) \geq \phi(\text{HUN}(v_1)) \geq \phi(f_x), \tag{A}$$

since we know the matching edge (v_2, u) is changed to (v_2, v_1). Also, immediately prior to the step when v_3 is matched, there exists an augmenting path of length three from v_3 to q. So after we match v_3, t_i is either q or a vertex that is matched in preference to q, so

$$\phi(t_i) \geq \phi(q). \tag{B}$$

Combining (A) and (B) we get $\phi(t_i) \geq \phi(f_x)$. Thus, $v_3 = \text{HUN}(u)$. Hence $\{v_2, v_3, u\}$ is a cycle since we have established the existence of the edge (u, v_3) (the existence of the other two edges of the cycle were established earlier).

(2) We establish this result by contradiction as well. Suppose the augmenting path is not $\{o_i, u, v_2, t_i\}$. Then we have two cases:

Case 1: The augmenting path is $\{o_i, v_2, u, t_i\}$ as shown in Fig. 5. In this case there must exist an unmatched vertex w adjacent to v_3, since after matching the edge (v_2, v_3) it must be changed to (v_2, v_1) by an augmenting path of length three. After matching v_3, assume without loss of generality that w becomes HUN(v_3). After the augmentation step, we have

$$\phi(t_i) \geq \phi(w), \tag{C}$$

since there existed an augmenting path from v_3 to w when t_i was matched. Also, (v_2, v_3) was matched in this step, and it must be changed to the matching edge (v_2, v_1). By Lemma 3.2 we have

$$\phi(w = \text{HUN}(v_3)) \geq \phi(\text{HUN}(v_1)) \geq \phi(f_x). \tag{D}$$

Combining (C) and (D), we obtain $\phi(t_i) \geq \phi(f_x)$. Again we have a contradiction of the condition of the lemma.

Case 2: The augmentation step does not include the edge (v_2, u) as shown in Fig. 6. In this case there must exist an unmatched vertex q adjacent to u since the matching edge (v_2, u) must be changed to (v_2, v_1) by an augmenting path of

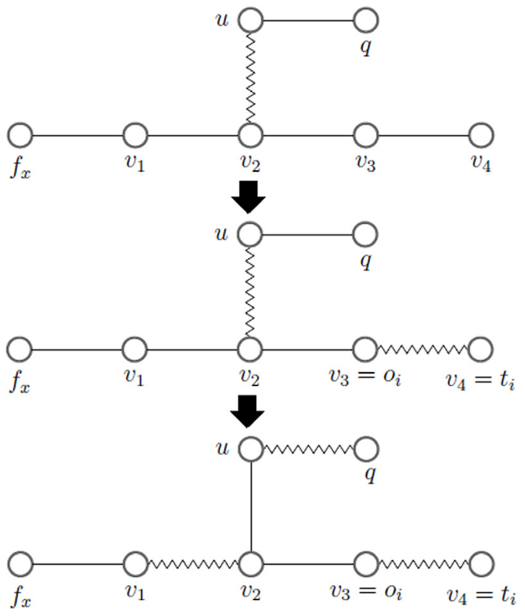

Fig. 4. Lemma 3.4: The case where $v_3 \neq \text{HUN}(u)$.

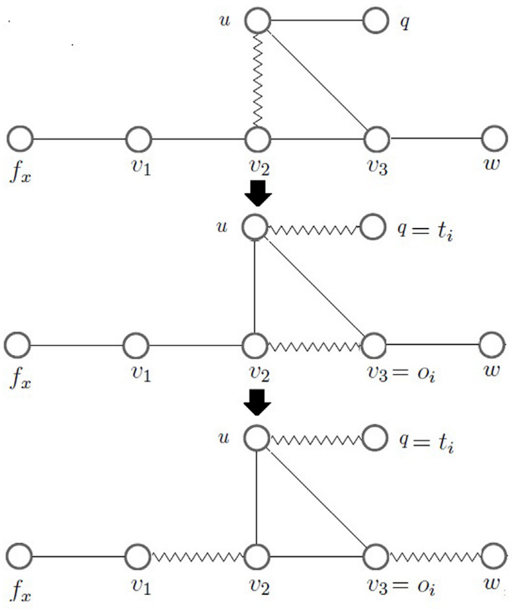

Fig. 5. Lemma 3.4, (2) Case 1: The augmentation step is $\{o_i, v_2, u, t_i\}$.

length three. After matching v_3, assume without loss of generality that q becomes HUN(u). After the augmentation step, we have

$$\phi(t_i) \geq \phi(q), \tag{E}$$

since there existed an augmenting path from v_3 to q. Note that (v_2, u) is still matched and must be changed to (v_2, v_1). By Lemma 3.2 we have

$$\phi(q = \text{HUN}(u)) \geq \phi(\text{HUN}(v_1)) \geq \phi(f_x). \tag{F}$$

Again, combining (E) and (F), we obtain $\phi(t_i) \geq \phi(f_x)$.

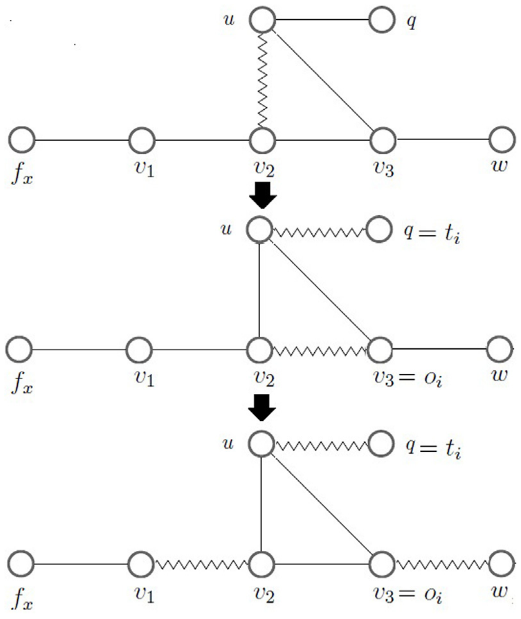

Fig. 6. Lemma 3.4 (2) Case 2: the augmentation step does not include the edge (v_2, u).

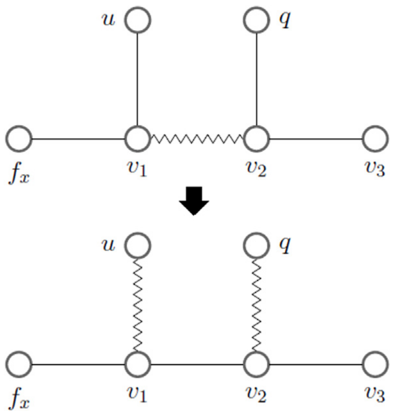

Fig. 7. Lemma 3.5: Augmenting the path $\{u, v_1, v_2, q\}$ after f_x is determined to be a failure.

In both cases we obtain $\phi(t_i) \geq \phi(f_x)$, a contradiction to the condition of the lemma. Therefore, the ith augmentation step must be $\{v_3 = o_i, u, v_2, t_i\}$. $\square$

Lemma 3.5. *Consider the symmetric difference $M_{opt} \oplus M_A^x$, corresponding to the 2/3-approximate matching at the xth failure. Let $P = \{f_x, v_1, v_2, v_3, v_4\}$ be an $M_{opt} \oplus M_A^x$-alternating path, then the alternating subpath $P = \{f_x, v_1, v_2, v_3\}$ will not change in future augmentation steps of the approximation algorithm.*

Proof. Assume for the sake of contradiction that after f_x is determined to be a failure, the edge (v_1, v_2) is changed by a future augmenting path of length three, say $\{u, v_1, v_2, q\}$, as shown in Fig. 7. Then, the augmenting path $\{f_x, v_1, v_2, q\}$ must exist when f_x was determined as a failure, and in this case f_x could not have been a failure. Hence the matching edge (v_1, v_2) in the approximate matching M_A^x cannot be changed in future augmentations. $\square$

Lemma 3.6. *Consider the symmetric difference $M_{opt} \oplus M_A$, where M_A is the matching computed by the 2/3-Approximation algorithm. For every failure f there are two distinct matched vertices in M_A that are at least as heavy as f.*

Proof. First run the approximation algorithm and at the ith augmentation step label the origin by o_i and the terminus by t_i. Recall that we denote o_i as the corresponding vertex of t_i, and vice versa. Consider the symmetric difference between

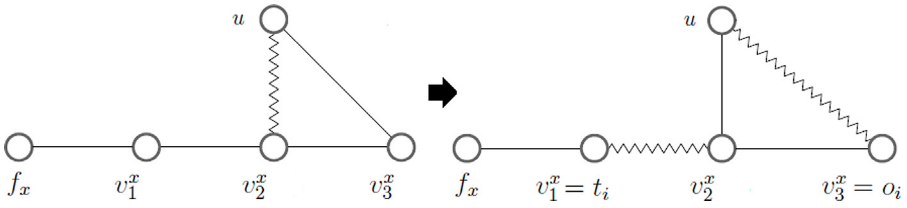

Fig. 8. Lemma 3.6, $i-2$: The corresponding terminus is strictly lighter than the failure f_x.

M_{opt} and M_A which results in alternating paths and cycles. We can ignore alternating cycles since every vertex in a cycle is matched in both M_{opt} and M_A. Since failures are matched by the optimal matching but not the approximate matching, they are at the ends of alternating paths.

By Lemma 3.5 the first four vertices of an alternating path beginning with a failure do not change, which makes it possible to identify the origins and termini which are used to construct the alternating path. We will number each failure f_x in the order that it was discovered in the approximation algorithm. A failure f_x could be an end of an alternating path which has one failure or two failures. We will consider these two types of alternating paths in the following.

(i) First consider an alternating path with one failure, and denote the path as $P = \{f_x, v_1^x, v_2^x, v_3^x, v_4^x\}$. We charge two distinct vertices for f_x as follows:
($i-1$) If the vertex v_1^x is a terminus, then charge the corresponding origin, which must be at least as heavy as the failure f_x since it was processed before f_x. If v_1^x is an origin then charge the corresponding terminus, which by Lemma 3.3 (1) must be at least as heavy as f_x.
($i-2$) If the vertex v_3^x is a terminus, then charge the corresponding origin which must be at least as heavy as the failure f_x since it was processed before f_x. If v_3^x is an origin, and the corresponding terminus is at least as heavy as f_x, then charge the corresponding terminus. If the corresponding terminus is strictly lighter than f_x, then by Lemma 3.4 we have immediately prior to the step in which v_3^x is matched, the vertex v_2^x is matched to some vertex u, $u \neq v_1^x$ such that $\{v_2^x, v_3^x, u\}$ is a cycle, as shown in Fig. 8. In this case we consider v_2^x instead of v_3^x to find a vertex to charge. If the vertex v_2^x is a terminus (in a prior augmentation step), then charge the corresponding origin which must be at least as heavy as f_x, since it was processed before the latter. If v_2^x is an origin in the prior augmentation step, then charge the corresponding terminus which must be at least as heavy as f_x since it was matched in preference to v_3^x which is an origin.

(ii) Now we consider an alternating path with two failures f_x and f_y as its endpoints. We assume without loss of generality that $\phi(f_x) \geq \phi(f_y)$.

For the failure f_x we charge two distinct vertices as we did in Part (i) of this lemma. Now we consider charging for the failure f_y. If the length of the alternating path is at least seven edges, then we can label two alternating subpaths $\{f_x, v_1^x, v_2^x, v_3^x\}$ and $\{f_y, v_1^y, v_2^y, v_3^y\}$, and these do not overlap. Hence we can charge two distinct vertices for f_y as we did in Part (i) of the lemma.

If the length of the alternating path is five then $\{v_2^x, v_3^x\}$ and $\{v_2^y, v_3^y\}$ overlap. Thus $v_2^x = v_3^y$, and $v_3^x = v_2^y$. So, we charge one vertex v_1^y for f_y as we did in ($i-1$) and we will charge the other distinct vertex as follows.
Case 1: If f_x charged the corresponding vertex of v_2^x then f_y must charge the corresponding vertex of $v_2^y = v_3^x$. Referring to ($i-2$), the vertex f_x charged the corresponding vertex of v_2^x because $v_3^x = v_2^y$ must be an origin and the corresponding terminus is strictly lighter than f_x. Let the origin v_3^x be denoted by o_i, and the corresponding terminus be t_i, for some augmentation step i. By Lemma 3.4 we have (1) at the step of matching v_3^x but before it is matched, v_2^x is matched to some u, where $u \neq v_1^x$, and $\{v_2^x, v_3^x, u\}$ is a cycle; (2) the augmenting path is $\{v_3^x = o_i, u, v_2^x, t_i\}$.

We will show that $\phi(t_i) \geq \phi(f_y)$, and thus f_y can be charged to t_i. We consider two subcases:
Subcase 1: f_y is adjacent to u, as shown in Fig. 9. Note that $\phi(t_i) \geq \phi(f_y)$, since at the step of matching v_3^x there existed an augmenting path from v_3^x to f_y.
Subcase 2: The failure f_y is not adjacent to u as shown in Fig. 10. Note there must exist some unmatched vertex q that is adjacent to u because after augmenting by the path $\{v_3^x = o_i, u, v_2^x, t_i\}$ the matching edge $(v_2^y = v_3^x, u)$ must be changed to (v_2^y, v_1^y), which can be done with an augmenting path of length three. After the augmentation step, we have

$$\phi(t_i) \geq \phi(q), \tag{G}$$

because there existed an augmenting path from v_2^y to q. After v_2^y is matched, assume without loss of generality that $q = \mathrm{HUN}(u)$. By Lemma 3.2, after (v_2^y, u) is changed to (v_2^y, v_1^y) we have

$$\phi(q = \mathrm{HUN}(u)) \geq \phi(\mathrm{HUN}(v_1^y)) \geq \phi(f_y). \tag{H}$$

Combining (G) and (H) we obtain $\phi(t_i) \geq \phi(f_y)$.
Case 2: If f_x charged the corresponding vertex of $v_3^x = v_2^y$, then f_y must charge the corresponding vertex of $v_3^y = v_2^x$. We will show that the corresponding vertex of v_3^y is at least as heavy as f_y. Suppose that the corresponding vertex is strictly lighter than f_y which is true if it is a terminus, say t_i in the ith augmenting step. By Lemma 3.4 we have (1) at the step

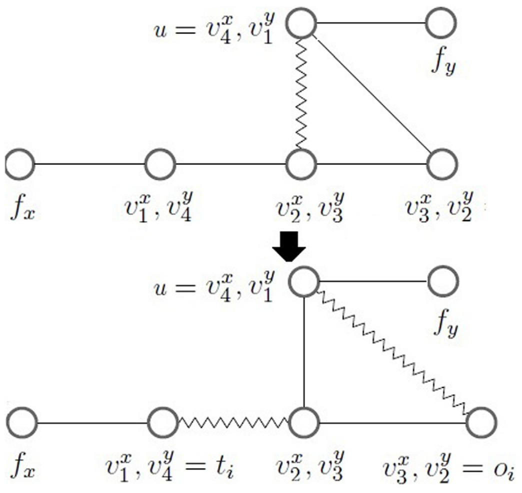

Fig. 9. Lemma 3.6, Case 1, Subcase 1: The failure f_y is adjacent to u.

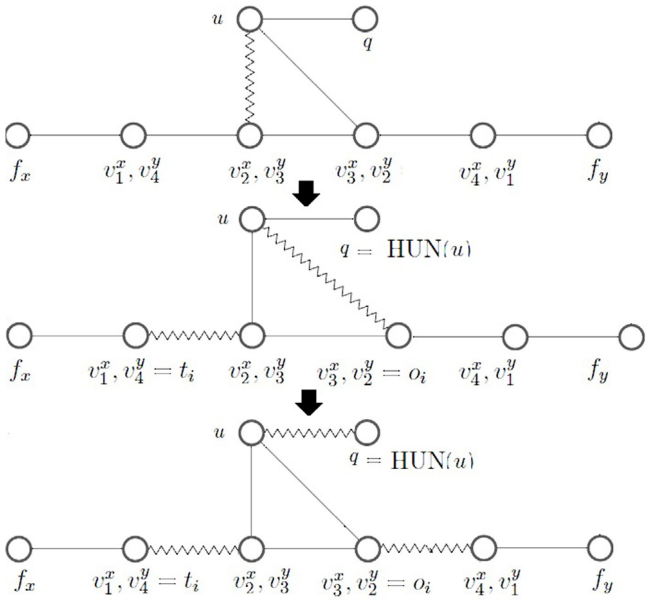

Fig. 10. Lemma 3.6, Case 1, Subcase 2: The failure f_y is not adjacent to u.

when the vertex v_3^y is matched but prior to matching it, the vertex v_2^y is matched to some u, with $u \neq v_1^y$, such that $\{v_2^y, v_3^y, u\}$ is a cycle; and (2) the augmenting path is $\{v_3^y = o_i, u, v_2^y, t_i\}$. By symmetry and using the same argument as in Case 1 we get $\phi(t_i) \geq \phi(f_x)$. Since by assumption we have $\phi(f_x) \geq \phi(f_y)$, it follows that $\phi(t_i) \geq \phi(f_y)$.

Note that each matched vertex has a unique corresponding vertex, since once they (the vertex and its corresponding vertex) are matched they will not be unmatched. So, to charge a vertex twice, a vertex u must be considered by two failures (and the corresponding vertex of u must be charged twice). But two failures cannot consider the same vertex. This is not possible for two failures in different alternating paths, since the alternating paths are vertex disjoint. This is also not possible for two failures in the same alternating path, since by our charging method they do not consider the same vertices to charge. □

Theorem 3.7. *Algorithm 1 computes a 2/3-approximation for the MVM problem.*

Proof. Let M_A be the matching computed by the approximation algorithm, and M_{opt} be a matching of maximum vertex weight. Consider all paths in the symmetric difference between M_A and M_{opt}. Let $\phi(F)$ denote the sum of weights of all the failures, let $\phi(M_{opt})$ denote the weight of the maximum-weighted matching, and let $\phi(M_A)$ denote the weight of the approximate matching. Then, $\phi(M_{opt}) = \phi(M_A) + \phi(F) - \phi(M_A \setminus M_{opt}) \leq \phi(M_A) + \phi(F)$, and we know from Lemma 3.6 that $\phi(F) \leq \frac{1}{2}\phi(M_A)$ since for every failure we have two distinct vertices that are at least as heavy as the failures. Hence $\phi(M_{opt}) - \phi(M_A) \leq \phi(F) \leq \frac{1}{2}\phi(M_A)$. Thus we have $\phi(M_{opt}) \leq \frac{3}{2}\phi(M_A)$. This completes the proof. □

4. A $(1-\epsilon)$- scaling algorithm

In this section we describe a scaling algorithm for computing a $(1-\epsilon)$-approximate algorithm for MEM, designed by Duan and Pettie [13], since we will compare the performance of the 2/3-MVM algorithm with an implementation of it. This is the first implementation of the former algorithm that we know about. Our treatment of this algorithm will be brief, and we refer the reader to Duan and Pettie's paper for further details.

The algorithm is based on the scaling technique, and employs a primal dual formulation of the problem. We assume that the reader is familiar with matchings in non-bipartite graphs; see [14] for an introduction.

A blossom with respect to a current matching is a cycle of odd length whose edges are alternately matching and non-matching edges. When discovered, blossoms may be shrunk to a super-vertex, and such vertices may become vertices in another blossom. A root blossom is one that is not contained in any other blossom. An outer vertex (or blossom) is a vertex that is reachable from an unmatched vertex by an even length alternating path. An inner vertex (or blossom) is a vertex that is reachable from an unmatched vertex by an odd length alternating path.

Let W be the maximum edge weight. Given an $\epsilon' > 0$, we define $\delta_0 = W\epsilon'$, and $\delta_i = \delta_0/2^i$; scaled weights $\phi_i(e) = \delta_i\lfloor\phi(e)/\delta_i\rfloor$; and $\gamma = \log 1/\epsilon'$. The dual variables y are defined over the vertices, and z over the blossoms. We define the variable yz over the edge $e = (u, v)$ as

$$yz_e = y_u + y_v + \sum_{(u,v)\in E(B)} z_B,$$

where B is a blossom. The dual variables satisfy the following properties:

1. z_B is non-negative multiple of δ_i for all discovered blossoms B, and the value of y_u is a non-negative multiple of $\delta_i/2$ for all $u \in V$.
2. $z_B > 0$ for all root blossoms.
3. $yz_e \geq \phi_i(e) - \delta_i$ for all $e \in E$.
4. $yz_e \leq \phi_i(e) + (\delta_j - \delta_i)$ for all $e \in M$, where e is matched in scale j and $j \leq i$.

At scale i an edge e is eligible to be considered for finding an augmenting path if at least one of the following holds:

1. e is in a blossom.
2. e is matched, $yz_e - \phi_i(e)$ is a non-negative integer multiple of δ_i, and $\log_2 \phi(e) \geq i - \gamma$.
3. e is not matched, $yz_e = \phi_i(e) - \delta_i$, and $\log_2 \phi(e) \geq i - \gamma$.

At each scale only eligible edges are considered for finding an augmenting path.

The algorithm starts with an empty matching, and sets $\delta_0 = \epsilon' W$ and $y_u = W/2 - \delta_0/2$, for all $u \in V$. At the ith scale, the algorithm performs the following four steps: it finds and augments a maximal set of disjoint augmenting paths using eligible edges; it shrinks discovered blossoms and sets the dual variables of discovered blossoms to zero; it updates dual variables of vertices and blossoms reached by unmatched vertices in the search, and expands inner blossoms whose dual variables are equal to zero. The four steps are repeated until the dual variables of unmatched vertices are equal to $W/2^{i+2} - \delta_i/2$, (zero at the last scale). After the end of each scale i (except the last one), the dual variables of all vertices are incremented by $\delta_i/2$.

The time complexity of this algorithm is $O(m\epsilon^{-1}\log\epsilon^{-1})$.

5. Experiments and results

5.1. Experimental setup and algorithms being compared

We used an Intel Xeon E5-2660 processor-based system (part of the Purdue University Community Cluster), called *Rice*[1] for the experiments. The machine consists of two processors, each with ten cores running at 2.6 GHz (20 cores in total) with 25 MB unified L3 cache and 64 GB of memory. The operating system is Red Hat Enterprise Linux release 6.9. All code was developed using C++ and compiled using the g++ compiler (version: 4.4.7) using the -O3 flag. Our test set consists of nineteen real-world graphs taken from the University of Florida Matrix collection [9] covering several application areas. Table 1 gives some statistics on our test set. The graphs are listed in increasing order of the number of vertices. The largest number of vertices of any graph is nearly 51 million, and the largest number of edges is nearly 216 million. For each graph we list the maximum and average vertex degrees and the ratio of the standard deviation of the degrees and the mean degree. The average degrees vary from 2 to 118, and the graphs are diverse with respect to their

[1] https://www.rcac.purdue.edu/compute/rice/.

Algorithm 2 $(1-\epsilon)$-approximation for maximum edge weighted matching.

```
1:  $M \leftarrow \emptyset$;
2:  $\delta_0 \leftarrow \epsilon' W$; // $W$ is the maximum edge weight, and $\epsilon' = \Theta(\epsilon)$
3:  $y_u \leftarrow W/2 - \delta_0/2$ for all $u \in V$;
4:  for Scale $i = 0$ to $\log W$ do
5:      while there are unmatched vertices $u$ with $y_u > W/2^{i+2} - \delta_i/2$ or ($y_u \neq 0$ and $i = \log W$) do
6:          Find a maximal set $\mathscr{P}$ of vertex disjoint augmenting paths using eligible edges;
7:          $M \leftarrow M \oplus \mathscr{P}$;
8:          Shrink blossoms found in step 6;
9:          $z_B \leftarrow 0$ for all blossoms $B$ found in step 6;
10:         Update the duals:
11:             $y_u \leftarrow y_u - \delta_i/2$ for all $u \in$ outer vertices;
12:             $y_u \leftarrow y_u + \delta_i/2$ for all $u \in$ inner vertices;
13:             $z_B \leftarrow z_B + \delta_i$ for all $B \in$ outer root blossoms;
14:             $z_B \leftarrow z_B - \delta_i$ for all $B \in$ inner root blossoms;
15:             Dissolve all inner root blossoms whose $z_B = 0$;
16:     end while
17:     if $i < \log W$ then
18:         $\delta_{i+1} \leftarrow \delta_i/2$;
19:         $y_u \leftarrow y_u + \delta_{i+1}$ for all $u \in V$;
20:     end if
21: end for
```

Table 1
The set of test problems.

Graph	$\|V\|$	Degree			$\|E\|$
		Max.	Mean	SD/Mean	
G34	2,000	4	4.00	0.00	4,000
G39	2,000	210	11.8	1.17	11,778
de2010	24,115	45	4.81	0.62	58,028
shipsec8	114,919	131	56.9	0.25	3,269,240
kron_g500-logn17	131,072	29,935	94.8	4.40	5,113,985
mt2010	132,288	139	4.83	0.74	319,334
fe_ocean	143,437	6	5.71	0.12	409,593
tn2010	240,116	89	4.97	0.60	596,983
kron_g500-logn19	524,288	80,674	106	5.76	21,780,787
tx2010	914,231	121	4.87	0.63	2,228,136
kron_g500-logn21	2,097,152	213,904	118	7.47	91,040,932
M6	3,501,776	10	5.99	0.14	10,501,936
hugetric-00010	6,592,765	3	2.99	0.01	9,885,854
rgg_n_2_23_s0	8,388,608	40	15.1	0.26	63,501,393
hugetrace-00010	12,057,441	3	2.99	0.01	18,082,179
nlpkkt200	16,240,000	27	26.6	0.09	215,992,816
hugebubbles-00010	19,458,087	3	2.99	0.01	29,179,764
road_usa	23,947,347	9	2.41	0.39	28,854,312
europe_osm	50,912,018	13	2.12	0.23	54,054,660

degree distributions. The three `kron_g500` graphs of different sizes have high maximum degrees, and high ratios of the standard deviation of the degrees and mean degree, but most problems have low values.

We compare the 2/3-approximation algorithm for MVM (for brevity we will call this the Two-thirds algorithm) with a number of other algorithms.

The Exact algorithm for MVM is similar to Algorithm 1 except that there is no restriction on the augmenting path length, and it is discussed in Section 3, and in more detail in [10]. The complexity of the Exact algorithm we have implemented is $O(nm)$. The Spencer and Mayr algorithm [32] has $O(m\sqrt{n}\,\log n)$ time complexity, but is more complicated to implement; it is not clear if it would lead to better practical performance, and our focus in this paper is on approximation algorithms for MVM with much lower time complexity. We improved the practical performance of the Exact algorithm for MVM by two modifications:

1. If a search for an augmenting path fails, we mark all visited vertices, and when these vertices are encountered in a future search, the algorithm quits searching along those paths. Azad, Buluç and Pothen [6] prove that there will not exist an augmenting path from such vertices in future steps of the algorithm.

2. At the step of matching a vertex u_i we keep track of a heaviest unmatched vertex u_j (with $j > i$) among the remaining unmatched vertices. If an augmenting path from u_i to a vertex v is found such that $\phi(v) = \phi(u_j)$, then the algorithm immediately stops the search and augments the matching.

We have included an exact algorithm for the maximum edge-weighted matching problem (MEM) implemented in LEDA [1,22] in our comparisons. This is a primal–dual algorithm implemented with advanced priority queues and efficient dual weight updates, with time complexity $O(nm \log n)$ [23]. Since this is commercial software, we can only run the object code, and we ran it with no initialization and with a fractional matching initialization. The latter first computes a $\{0, 1/2, 1\}$ solution to the linear programming formulation of maximum weighted matching by ignoring the odd-set constraints (this solution is computed combinatorially), and then rounds the solution to $\{0, 1\}$ values [4]. We call these two variants LEDA1 and LEDA2, respectively.

The Greedy 1/2-approximation algorithm for MVM (called Half here) matches the vertices in non-increasing order of weights; it matches an unmatched vertex to a heaviest unmatched neighbor, and then deletes other edges incident on the endpoints of the matching edge. Its time complexity is $O(m + n \log n)$ [10].

We used two implementations the Random Augmentation Matching Algorithm (RAMA) and the Random Order Augmentation Matching Algorithm (ROMA) of the $(2/3 - \epsilon)$ approximation algorithm for MEM due to Pettie and Sanders [26], and Maue and Sanders [21] with $\epsilon = 0.01$. Before describing each implementation we will describe a 2-augmentation centered at a vertex v, which is an operation used in both implementations.

We define an *arm* of v to be either $\{v, u\}$ or $\{v, u, u'\}$, where (v, u) is a non-matching edge, and (u, u') is a matching edge. The *gain* of an augmentation or exchange of edges is the increase in weight obtained by the transformation. There are two cases:
Case (1) v is unmatched: find an arm of v with the highest positive gain.
Case (2) v is matched to a vertex v': find the highest positive gain by checking the gains of the following paths or cycles:
(1) Alternating cycles of length four that include the edge (v, v').
(2) Alternating paths of length at most four, which is done as follows: Find two vertex disjoint arms of v with the highest gains, P and P', then find an arm of v' with highest gain Q. If P and Q are vertex disjoint then $P \cup (v, v') \cup Q$ is a highest gain alternating path; otherwise choose $P' \cup (v, v') \cup Q$ as a highest gain alternating path.

There are two implementations of this algorithm. The *RAMA implementation* chooses a random vertex v and performs a 2-augmentation centered at v with the highest-gain. This is repeated $k = \frac{1}{3} \log \epsilon^{-1}$ times. The *ROMA implementation* randomly permutes the order of vertices, and for each vertex v in the permuted order performs 2-augmentation with the highest-gain arm centered at v. This is repeated for $k = \frac{1}{3} \log \epsilon^{-1}$ phases; for $\epsilon = 0.01$, we have $k = 2$. If no further improvement can be achieved after finishing a phase then the algorithm terminates.

The algorithm can be initialized with the 1/2-approximation algorithm called the Global Paths algorithm (GPA) [21], which sorts the edges in non-increasing order of their weights. It constructs sets of paths and cycles *of even length* by considering the edges in non-increasing order of their weights. Then it computes a maximum weight matching for each path and cycle by dynamic programming, and it deletes the matching edges and their adjacent edges. The algorithm repeats until all edges are deleted. The time complexity of the GPA algorithm is $O(m \log n)$, and that of the ROMA $(2/3 - \epsilon)$-approximation algorithm is $O(m \log \epsilon^{-1})$. Maue and Sanders [21] have reported that the ROMA implementation with GPA initialization computed heavier matchings than the other three variants albeit at the expense of higher running times; we have obtained similar results, and find that the ROMA implementation with no initialization was the fastest among the four variants. Hence we report results from these two variants, called ROMA and GPA-ROMA, respectively.

The final algorithm we implemented is a $(1 - \epsilon)$-approximate scaling algorithm for MEM (Scaling) due to Duan and Pettie [13], with the choice of $\epsilon = 1/3$, $1/4$, and $1/6$. The algorithm is described in Section 4.

In total, we have two exact algorithms for MEM and MVM, and four approximation algorithms. The exact MEM algorithm and the $(2/3 - \epsilon)$-approximation algorithm have two options for initialization.

Integer weights of vertices were generated uniformly at random in the range [1 1000], and real-valued weights were chosen randomly in the range [1.0 1.3]. The reported results are average of ten trials of randomly generated weights. The standard deviations for run-time, weight ratio, and cardinality ratio are close to zero, so there is not much variation on these metrics for each algorithm.

When the weights are integers in a range [0 K], we employ a counting sort with $O(n + K)$-time complexity for sorting the weights, and observed that it is two to three orders of magnitude faster than the sort function in C++ STL. For real weights, we have used the latter sort function.

5.2. Performance of the algorithms

In Table 2 we group the problems into three sets based on our results. In the first set, the time taken by the Exact MEM algorithm from LEDA without initialization (LEDA1), and the relative performance of the other algorithms (the ratio of the time taken by LEDA1 to the time taken by the other algorithm), are reported. Numbers greater than one indicate that the latter algorithms are faster. For the second set of problems, the LEDA algorithm with no initialization did not complete in four hours. Hence we report the time taken by LEDA2, the code with fractional matching initialization, and relative performance for the other algorithms. For the third set consisting of one problem, none of the exact algorithms completed in 100 h, and we report the run times of the approximation algorithms.

Table 2
Running times (seconds) and relative performance of several exact and approximation algorithms for the MEM and MVM problems. The exact algorithms include the LEDA implementations for MEM, with no initialization and a fractional matching initialization, and the exact MVM algorithm described in this paper. The approximation algorithms include the 1/2-MVM, the 2/3-MVM, ROMA $(2/3-\epsilon)$ MEM with $\epsilon = 0.01$ with and without GPA initialization, and the $(1-\epsilon)$-Scaling MEM approximation algorithm with $\epsilon = 1/3$. Vertex weights are random integers in the range [1 1000]. The three groups of problems indicate those for which the LEDA implementation with no initialization terminated in under four hours; those for which the LEDA implementation with a fractional matching initialization terminated in under four hours but the first algorithm did not; and a problem for which none of the exact algorithms terminated in 100 h.

Graph	Time (s)	Relative Performance						
	Exact	Exact	Exact	$1-\epsilon$	GPA-ROMA	ROMA	2/3-	1/2-
	LEDA1	LEDA2	MVM	Scal. $\epsilon = 1/3$	$2/3-\epsilon$ $\epsilon = 0.01$		MVM	MVM
G34	0.137	1.877	13.10	18.72	21.64	39.97	808.7	2673.7
G39	1.195	99.6	46.24	93.65	109.7	154.2	2826.4	13301.5
de2010	3.865	13.51	7.583	26.86	45.96	80.27	1094.7	3232.3
kron_g500-logn17	120.6	14.08	31.96	29.85	38.55	69.19	1814.9	7056.5
mt2010	15.41	10.26	10.90	18.62	26.11	61.75	605.0	1432.9
fe_ocean	653.0	0.776	5.831	873.7	941.4	2040	20564	45136.0
tn2010	232.5	22.78	17.08	91.90	185.4	428.8	4442.9	9807.1
kron_g500-logn19	612.5	15.21	35.24	22.47	40.72	75.99	2081.7	7586.2
tx2010	1914.4	29.17	20.70	148.9	315.0	723.2	6880.6	17652.2
kron_g500-logn21	3329.0	15.87	30.88	18.36	35.99	62.17	1877.1	7516.3
road_usa	1151.0	9.019	18.27	3.490	8.493	18.48	161.3	273.0
europe_osm	4076.4	15.84	53.01	7.035	13.98	35.08	293.3	463.1
Geo. Mean		11.68	19.73	34.81	56.46	112.47	1616.9	4480.6
		Time (s)	Relative Performance					
shipsec8		5.920	0.210	3.258	2.533	4.686	10.07	191.7
M6		1190.3	0.648	13.45	39.26	88.218	677.6	1941.8
hugetric-00010		372.8	0.408	3.892	9.529	21.226	188.3	338.2
rgg_n_2_23_s0		5952.0	2.693	37.21	45.84	86.433	401.9	2184.5
hugetrace-00010		641.4	0.437	3.916	8.890	19.905	177.0	317.3
hugebubbles-00010		1799.8	0.574	5.950	14.72	32.064	289.2	519.5
Geo. Mean			0.578	7.272	13.36	28.02	172.6	597.1
				Time(s)				
nlpkkt200				263.6	348.3	185.5	50.21	6.591

On the first set of problems, in geometric mean, the exact algorithms LEDA2 and MVM are 12 and 20 times, respectively, faster than LEDA1; the Scaling algorithm and the GPA-ROMA algorithms are about 35 and 56 times faster, respectively; and the ROMA algorithm is 112 times faster; the Two-thirds MVM algorithm is 1600 times faster, and the Half algorithm is almost 4450 times faster, all relative to LEDA1.

On the second set of problems, the Exact MVM algorithm is slower than the exact MEM algorithm LEDA2 by a factor of about 1.7. The approximation algorithms are all faster than LEDA2, the fastest again being the Half algorithm (by a factor of 600), and the Two-thirds algorithm is faster by a factor of 170. The scaling and the ROMA algorithms are 7 and 28 times faster than LEDA2.

For the `nlpkkt200` problem, the Two-thirds algorithm computed the matching in 50 s on the integer weights; the Half approximation algorithm took about 7 s, while the Scaling algorithm solved the same problem in 264 s. The GPA-ROMA and ROMA algorithms took 348 and 186 s, respectively. This graph has an interesting structure. It comes from a nonlinear programming problem (it is a symmetric Kuhn–Tucker-Karush matrix), which can be partitioned into two subsets of vertices V_1 and V_2; vertices in the set V_1 are connected to each other and to vertices in the set V_2, but the latter is an independent set of vertices, i.e., no edge joins a vertex in V_2 to another vertex in V_2. There are 8240,000 vertices in V_1 and 8000,000 vertices in V_2. This structure creates a large number of augmenting paths for the exact algorithms, and we conjecture this is why the exact algorithms do not terminate.

We also report the maximum time taken by an algorithm over all problems on which it terminated. For LEDA1, it is 4076 s on the `europe_osm` problem; for LEDA2, 5952 s on the `rgg` problem; the Exact MVM algorithm needed 3136 s on the `huge_bubbles` problem. The Scaling algorithm took 579 s on the `europe_osm` problem, and the Half algorithm took 9 s on the same problem. The problem `nlpkkt200` needed the most time for the other approximation algorithms: 348 s for the GPA-ROMA, 186 s for the ROMA algorithm, and 50 s for the Two-thirds algorithm.

The runtimes per edge of some of these algorithms are plotted against various graphs in a semi-logarithmic plot in Fig. 11. it is clearly seen that the Two-thirds algorithm is the fastest among these algorithms for all problems. The exact algorithms LEDA2 and Exact MVM are the slowest, followed by the Scaling algorithm and then the GPA-ROMA algorithm.

We compare the weight of the matching computed by the algorithms with integer weights in range [1 1000] in Table 3. All the exact algorithms compute the same maximum weight, which is reported in the first column; the approximation algorithms compute nearly optimal weights, and in order to differentiate among them, we report the gap to optimality

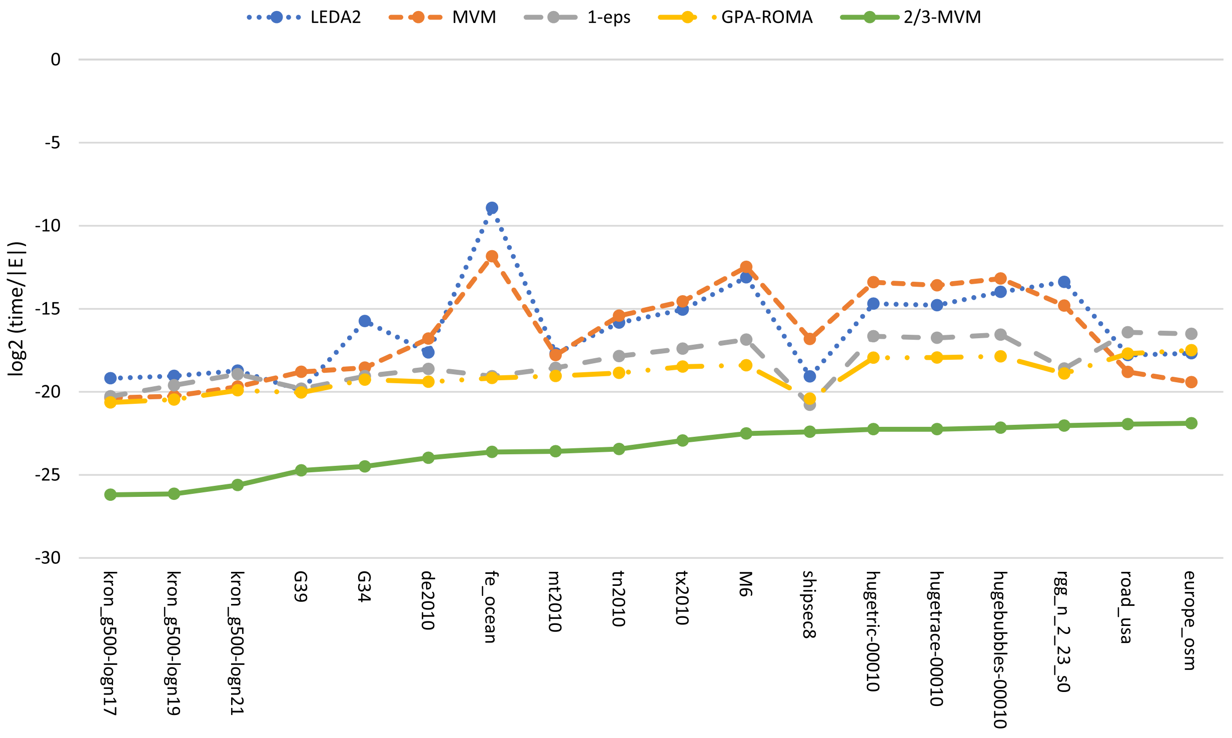

Fig. 11. Running time per edge (time for computing the matching scaled by the number of edges in a graph) for different algorithms plotted on a logarithmic scale. The vertex weights are integers in [1 1000].

Table 3
The weights computed by the exact MEM and MVM algorithms, and the gap to optimality of the weights of the matching obtained from the approximation algorithms. For the last problem, the weights are shown since the Exact algorithms did not terminate. Random integer weights in the range [1 1000] are used.

Graph	Weight	Gap to optimal weight (%)				
	Exact algs.	$1-\epsilon$-Scal. $\epsilon = 1/3$	GPA-ROMA $2/3-\epsilon$ $\epsilon = 0.01$	ROMA	2/3-MVM	1/2-MVM
G34	1.0E+6	1.89	0.31	0.30	0.47	2.88
G39	1.0E+6	1.63	0.04	0.06	0.06	2.92
de2010	1.2E+7	3.46	0.90	0.93	0.99	6.75
shipsec8	5.7E+7	0.02	0.00	0.00	0.00	0.05
kron_g500-logn17	4.4E+7	5.82	1.97	2.09	2.13	14.47
mt2010	6.5E+7	3.86	1.10	1.14	1.21	7.61
fe_ocean	7.2E+7	1.06	0.14	0.15	0.21	2.26
tn2010	1.2E+8	3.46	0.95	0.99	1.04	7.02
kron_g500-logn19	1.6E+8	5.29	1.72	1.81	1.95	14.33
tx2010	4.5E+8	2.25	0.83	0.87	0.94	6.40
kron_g500-logn21	5.5E+8	5.05	1.52	1.61	1.79	14.09
M6	1.8E+9	0.70	0.16	0.16	0.21	2.39
hugetric-00010	3.3E+9	1.57	0.50	0.64	0.81	4.43
rgg_n_2_23_s0	4.2E+9	0.20	0.03	0.03	0.03	0.56
hugetrace-00010	6.0E+9	1.56	0.49	0.62	0.79	4.39
hugebubbles-00010	9.7E+9	1.57	0.50	0.63	0.81	4.42
road_usa	1.2E+10	3.35	1.16	1.50	1.74	7.81
europe_osm	2.5E+10	3.08	0.53	1.81	2.00	6.77
Geom. Mean		1.64	0.33	0.39	0.46	3.88
		Weights				
nlpkkt200		8.06E+09	8.07E+09	8.07E+09	8.08E+09	8.04E+09

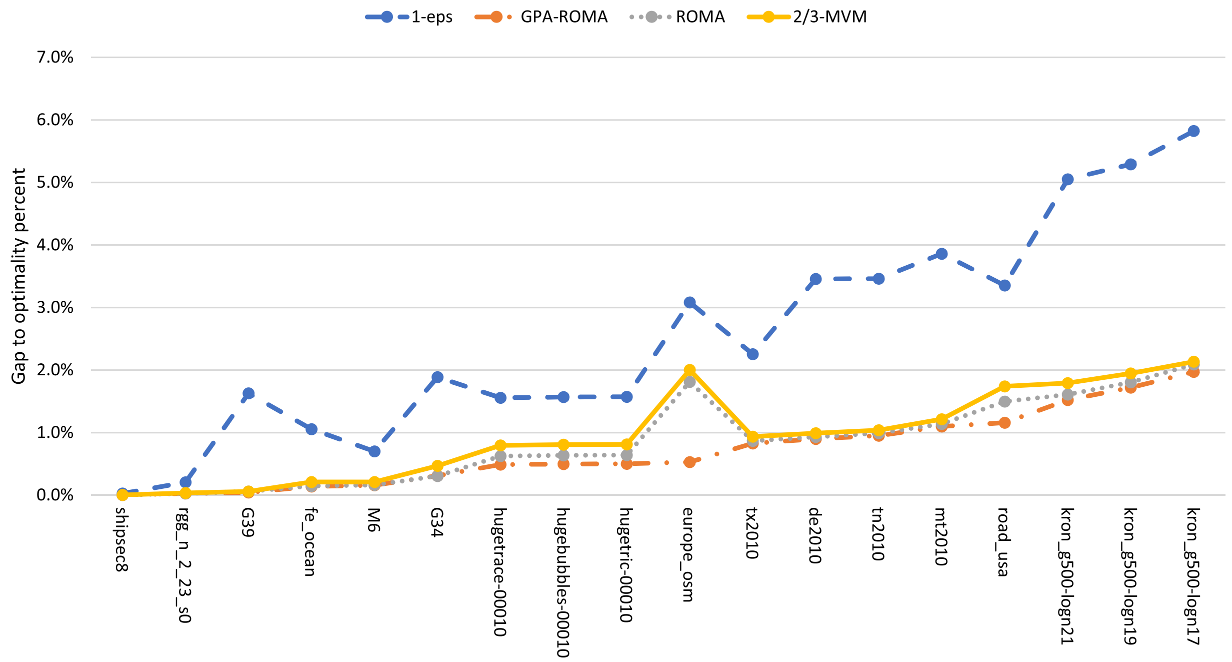

Fig. 12. Gaps to optimal weights for different algorithms, with integer weights in [1 1000].

as a percent. Hence we report $100(1-\phi(M_A)/\phi(M_{opt}))$, where $\phi(M_A)$ is the weight computed by an algorithm A and $\phi(M_{opt})$ is the optimal weight computed by the exact algorithms. The Half algorithm computes weights higher than 96% of the optimal, and the Scaling algorithm computes weights higher than 98% of the optimal. The other approximation algorithms obtain weights higher than 99.99% of the optimal. The best among these is GPA-ROMA, but it requires run times at least 20 times higher than the Two-thirds algorithm. Note that the weights obtained in practice are much better than the worst-case approximation guarantees. These results are plotted in Fig. 12.

We consider the cardinality of the matchings in Table 4. The exact algorithm for MVM computes a maximum cardinality matching when the vertex weights are positive, and since the MEM algorithms are derived from MVM problems by summing the weights, the MEM algorithms also compute maximum cardinality matchings. The Half approximation algorithm is about ten percent off the maximum cardinality, and the Scaling algorithm is about 6 percent off. The other approximation algorithms obtain cardinalities that are about 2 percent lower than the maximum, with the GPA-ROMA algorithm the best performer. For eight of the nineteen problems, the exact algorithms obtained perfect matchings (cardinality equal to $n/2$ or $(n-1)/2$).

To see how the Two-third algorithm fares against the Scaling algorithm for a smaller ϵ, we compared it with 3/4- and 5/6-Scaling approximation algorithms. For integer weights in the range [1 1000] the Two-third algorithm is 37 times faster than the 3/4-approximation, and 54 times faster than the 5/6-Scaling approximation algorithm. In geometric mean the Two-thirds algorithm obtained greater weight by 1.2% and 1.9%, and higher cardinality by 3.1% and 3.8%, relative to the 3/4- and 5/6-approximate Scaling algorithm.

In Table A.6, we report run times from the Exact MVM algorithm and the relative performance of the approximation algorithms when the vertex weights are real-valued in the range [1 1.3]. These weights are favorable to the Scaling approximation algorithm, since the number of scales needed is low. LEDA unfortunately does not work with real-valued weights. In geometric mean, the Half algorithm is faster than the Exact MVM algorithm by a factor of 110; the Two-thirds algorithm by a factor of 54; and the other approximation algorithms are faster by factor less than 8. On the `nlpkkt200` problem, the Exact MVM algorithm did not terminate; notice that the Scaling algorithm is faster with the smaller range of weights here when compared to the integer weights with a larger range. The run times of the other approximation algorithms are consistent with the rankings discussed earlier.

Table A.7 includes results for the real-valued weights in the range [1 1.3]. The Half approximation algorithm obtains about 89% of the maximum weight (geometric mean of these problems), and is the worst performer. The other approximation algorithms are all comparable in the weights they compute, two or three percent off the optimal. Again the best performer is the GPA-ROMA algorithm, which it achieves taking about a factor of nine more time than the Two-thirds algorithm.

In Table A.8, we show that cardinalities of matchings obtained with real weights with a smaller range are similar to the ones obtained with integer weights, except that this time the Scaling algorithm finds higher cardinalities.

Table 4
The cardinality of the matchings obtained by the exact algorithms and the gap to optimality of the approximation algorithms. For the last problem, cardinalities are shown since the Exact algorithm did not terminate. Random integer weights in [1 1000].

Graph	Card.	Gap to optimality (%)				
	Exact algs.	$1-\epsilon$-Scal. $\epsilon = 1/3$	GPA-ROMA $2/3-\epsilon$ $\epsilon = 0.01$	ROMA	2/3-MVM	1/2-MVM
G34	1,000	7.59	2.53	2.53	3.58	9.01
G39	1,000	7.29	1.03	1.13	1.22	9.74
de2010	11,853	9.86	3.98	4.05	4.42	14.07
shipsec8	57,459	0.91	0.14	0.15	0.17	1.13
kron_g500-logn17	38,823	6.82	2.90	3.06	2.85	16.93
mt2010	63,685	9.96	4.08	4.15	4.56	14.52
fe_ocean	71,718	6.10	1.75	1.77	2.41	8.19
tn2010	117,989	9.84	4.06	4.17	4.50	14.41
kron_g500-logn19	136,770	5.65	2.40	2.51	2.45	16.24
tx2010	449,167	7.35	3.73	3.81	4.26	13.55
kron_g500-logn21	482,339	5.16	2.10	2.21	2.21	15.69
M6	1,750,888	4.75	1.92	1.97	2.36	8.48
hugetric-00010	3,296,382	6.96	3.34	3.54	4.63	11.61
rgg_n_2_23_s0	4,194,303	2.57	0.77	0.79	0.93	3.85
hugetrace-00010	6,028,720	6.88	3.26	3.45	4.55	11.51
hugebubbles-00010	9,729,043	6.94	3.32	3.52	4.61	11.58
road_usa	11,325,669	7.09	3.34	3.81	4.59	13.37
europe_osm	25,149,787	8.22	1.91	5.23	6.29	13.57
Geom. Mean		6.01	2.12	2.34	2.72	10.14
		Cardinality				
nlpkkt200		7.88E+06	7.99E+06	7.99E+06	7.99E+06	7.82E+06

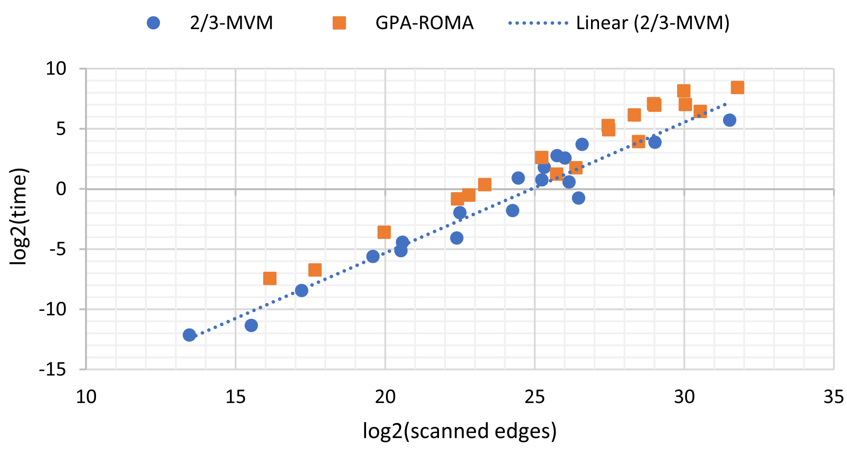

Fig. 13. Running time plotted against the number of edges scanned by the 2/3-MVM and GPA-ROMA algorithms (plotted on a log–log scale). The edge weights are integers in [1 1000].

For real-valued weights in [1 1.3] the Two-third algorithm is 9.6 and 12.6 times faster than the 3/4 and 5/6-Scaling approximation algorithms, respectively. In geometric mean the Two-third algorithm obtained greater weights by 4.7% than the 3/4-approximation; it was worse by 0.9% than the 5/6-approximation; the cardinality was higher by 4.7% over the 3/4-approximation, and worse by 1.1% relative to the 5/6-approximation.

In Fig. 13 we show the run times of the Two-thirds and the GPA-ROMA algorithms against the number of edges scanned by the algorithms, in a log–log plot. A near-linear relationship is seen, showing that the run times are determined by the number of edges scanned by the algorithms. We also show in Table 5 the breakdown of time taken by the various steps in these two algorithms. For GPA-ROMA, the time is dominated by the augmenting path searches in the ROMA algorithm and

Table 5
Percentage of time taken by the major steps in the 2/3-MVM and GPA-ROMA approximation algorithms. Random integer weights in [1 1000]. The remaining time is spent in variable declarations and initializations.

Graph	GPA-ROMA					2/3-MVM	
	GPA			ROMA		Sort	Aug.
	Sort	Paths and cycles search	Optimal edges by DP	Random permutation	Max 2-aug. search		paths search
G34	1.081	27.85	0.901	1.379	63.40	5.110	66.17
G39	3.016	15.99	0.547	0.772	76.43	2.361	82.77
de2010	3.159	34.92	1.707	0.933	53.54	6.393	87.02
shipsec8	9.873	28.15	0.708	0.201	58.61	0.130	99.70
kron_g500-logn17	11.10	25.63	0.316	0.163	59.96	1.640	96.09
mt2010	1.430	42.49	2.460	0.892	48.69	3.890	90.98
fe_ocean	1.524	41.76	1.914	0.779	50.38	3.159	93.34
tn2010	2.000	48.02	2.028	0.784	43.45	3.334	93.07
kron_g500-logn19	9.277	26.67	0.274	0.154	61.18	1.277	97.10
tx2010	2.575	43.94	1.656	0.643	47.35	2.429	94.72
kron_g500-logn21	7.104	23.54	0.213	0.104	67.01	1.471	97.12
M6	2.281	41.70	1.682	0.599	49.94	2.826	95.28
hugetric-00010	1.498	38.72	2.009	1.269	49.32	4.735	90.30
rgg_n_2_23_s0	3.199	31.85	0.706	0.552	60.81	0.940	98.52
hugetrace-00010	1.637	39.69	2.169	1.698	48.81	5.213	90.88
nlpkkt200	4.244	28.62	0.699	0.531	63.48	0.476	99.20
hugebubbles-00010	1.532	38.56	2.209	1.869	50.11	4.804	91.91
road_usa	1.359	39.81	1.971	2.181	48.12	5.005	91.49
europe_osm	1.266	46.37	1.694	2.596	41.23	4.227	89.66
Geom. Mean	2.709	33.76	1.084	0.684	54.16	2.333	91.52

the construction of paths and cycles in the GPA algorithm. Other steps such as the counting sort, the dynamic programming selection of matched edges, and the random permutation of vertices, all take a few percent of the total time. For the Two-thirds algorithm the augmenting path searches dominate the runtime. For the scaling approximation algorithm (not reported in the table), on geometric mean 85% of time is spent on finding a maximal set of augmenting paths and blossoms and 5.7% of the time on dual updates. Similar results are obtained when the weights are real; see Table A.9.

We also studied the influence of the distribution of weights on the performance of the algorithms. We generated three sets of random integer weights from the Gaussian distribution with mean 500 and standard deviations equal to 10, 100 and 400. For the Two-thirds, ROMA and GPA-ROMA algorithms there were no significant differences in running time over choosing random weights from a uniform distribution. In geometric mean there were slight improvements in running time by 5%, 5%, and 1% respectively, compared with running time on uniform random weights. The scaling algorithm showed more improvements in running time by 13%, and it ran up to 4 times faster when the standard deviation was equal to 10. The reason is that fewer scales are needed by the algorithm when the weights are concentrated around the mean.

We ran the Two-thirds algorithm and then employed the ROMA algorithm in a post processing step. ROMA was run for one, two and three phases. No significant improvement in weights was observed. On geometric mean the matching weight improved by 0.07% and the cardinality improved by 0.2%, but the time was significantly slower. The Two-thirds algorithm without this post-processing is 7, 13 and 18 times faster than the Two-thirds with ROMA using one, two and three phases, respectively.

6. Conclusions

We have described an augmentation-based 2/3-approximation algorithm for MVM on non-bipartite graphs whose time complexity is $O(m \log \Delta + n \log n)$, whereas the time complexity of an exact algorithm is $O(n^{1/2} m \log n)$. The approximation algorithm is derived in a natural manner from an exact algorithm for computing maximum weighted matchings by restricting the length of augmenting paths to at most three.

The 2/3-MVM algorithm has been implemented efficiently in C++, and on a set of nineteen graphs, some with hundreds of millions of edges, it computes the approximate matchings in less than 52 s. The weight of the approximate matching is greater than 98% (94%) of the weight of the Optimal matching for these problems on integer weights in [1 1000] (real weights in [1.0 1.3]). A Greedy Half-approximation algorithm is faster than the 2/3-MVM algorithm by about a factor of two, but the weight it computes is lower, and can be as low as 84% on the worst problem. All of these algorithms obtain weights that are much higher than the worst-case approximation guarantees.

In addition, on geometric mean the 2/3-MVM algorithm is faster than a Scaling based $(1-\epsilon)$ approximation algorithm by a factor of 37 on the integer weights in range [1 1000], which is expected due to the large overheads needed for

Table A.6
Running times (seconds) of the Exact MVM algorithm, and relative performance of five algorithms: the 1/2-MVM; the 2/3-MVM; ROMA, GPA followed by ROMA; $(2/3-\epsilon)$-approximation MEM for $\epsilon = 0.01$; and the $1-\epsilon$-Scaling MEM approximation algorithm with $\epsilon = 1/3$. Vertex weights are random reals in the range [1 1.3]. The last row shows the times of the approximation algorithms for a problem on which the exact algorithm did not terminate.

Graph	Time (s)	Relative Performance				
	Exact	$1-\epsilon$	GPA-ROMA	ROMA	2/3-	1/2-
	MVM	Scal.	$2/3-\epsilon$		MVM	MVM
		$\epsilon = 1/3$	$\epsilon = 0.01$			
G34	0.009	6.427	1.419	2.788	31.72	41.08
G39	0.017	6.964	1.527	2.509	35.62	71.71
de2010	0.225	6.462	2.435	5.103	49.65	72.28
shipsec8	6.499	13.78	2.733	5.089	11.33	162.5
kron_g500-logn17	3.439	3.561	1.127	2.002	50.47	118.9
mt2010	0.725	3.492	1.166	2.870	21.95	33.09
fe_ocean	60.62	453.8	82.71	187.60	1446.4	2299.3
tn2010	4.902	8.723	3.803	8.956	71.48	108.6
kron_g500-logn19	17.55	2.396	1.166	2.173	53.60	134.8
tx2010	33.62	11.34	5.448	12.54	99.90	183.0
kron_g500-logn21	97.97	1.967	1.028	1.860	53.46	159.1
M6	643.0	33.54	20.63	46.65	323.1	704.3
hugetric-00010	340.4	16.53	8.499	19.56	137.5	199.5
rgg_n_2_23_s0	645.2	15.81	4.631	9.057	46.31	178.6
hugetrace-00010	566.2	16.50	7.595	17.36	122.1	179.5
hugebubbles-00010	1182.3	18.38	8.969	20.94	147.5	218.3
road_usa	48.15	0.581	0.323	0.772	5.171	7.163
europe_osm	72.07	0.505	0.227	0.639	3.806	5.053
Geom. Mean		7.656	2.854	6.088	53.99	109.0
		Time (s)				
nlpkkt200		43.47	373.4	190.2	46.08	8.026

the handling of blossoms and dual variable updates. While the Scaling algorithm is faster on real-valued weights in a narrower range [1.0 1.3] since there are fewer scales, the 2/3-MVM algorithm is still faster than it on average by a factor of 7. The 2/3-approximate MVM algorithm obtains better matching weight than the Scaling approximation algorithm for relevant values of ϵ in all instances on the integer weights, and on geometric mean it obtains better matching weight on the real-valued weights.

The $(2/3-\epsilon)$-approximation algorithm for MEM, with ROMA selection of augmentations and initialization with the Global Paths algorithm, computes higher weights than the 2/3-approximation algorithm for MVM, but at a cost of an order of magnitude or more time. The weight differences are quite small for integer weights in a range [1 1000], but are about 0.7% for real-valued weights in the range [1 1.3].

We have also compared our algorithms with exact algorithms for the MEM problem from LEDA with a fractional matching initialization, and show that the exact MVM algorithm is quite competitive with it. The 2/3-approximation algorithm for MVM is two to three orders of magnitude faster than these exact algorithms, and there are problems on which the exact algorithms do not terminate in hundreds of hours.

Half-approximation algorithms for MEM (e.g., the Locally Dominant edge and Suitor algorithms) do not require sorting and can be used or adapted to obtain 1/2-approximate matchings for the MVM. The 2/3-approximation algorithm for MVM designed here processes the vertices in non-increasing order of weights, searching only for augmenting paths. An algorithm that searches for augmenting paths of length at most three *and* weight-increasing paths of length at most four, can process unmatched vertices in *any order*, leading to a parallel algorithm. We have designed and implemented a parallel 2/3-approximation algorithm for MVM based on this idea in recent work [3].

Finally, we mention that a survey of approximation algorithms for several variant maximum matching problems (cardinality, edge-weighted matching, vertex-weighted matching, b-matching) and the related minimum edge cover problems (cardinality, edge-weighted, and b-edge cover) is provided in [29].

Acknowledgments

We thank Jens Maue (Zurich) and Peter Sanders of the Karlsruhe Institute of Technology for sharing the code for the GPA and $(2/3-\epsilon)$-approximation algorithms with us.

Appendix. Tables with random real weights in the range [1 1.3]

See Tables A.6–A.9.

Table A.7
The weight obtained by the exact MEM and MVM algorithms, and the gaps to optimality of the matching computed by the approximation algorithms. For the last problem, weights are shown since the exact algorithm did not terminate. Random real-valued weights in the range [1 1.3] are used.

Graph	Weight	Gap to exact weight (%)				
	Exact algs.	$1-\epsilon$-Scal. $\epsilon = 1/3$	GPA-ROMA $2/3-\epsilon$ $\epsilon = 0.01$	ROMA	2/3-MVM	1/2-MVM
G34	2.30E+03	1.89	2.27	2.23	3.17	8.16
G39	2.30E+03	1.90	0.99	1.22	1.05	8.78
de2010	2.73E+04	6.88	3.54	3.65	4.07	12.64
shipsec8	1.32E+05	0.00	0.13	0.13	0.14	0.99
kron_g500-logn17	9.08E+04	5.71	2.72	2.87	2.74	16.33
mt2010	1.47E+05	6.29	3.69	3.77	4.09	13.27
fe_ocean	1.65E+05	1.44	1.55	1.56	2.13	7.23
tn2010	2.72E+05	6.43	3.66	3.74	4.05	13.07
kron_g500-logn19	3.20E+05	5.92	2.31	2.43	2.38	15.76
tx2010	1.03E+06	3.71	3.34	3.44	3.82	12.28
kron_g500-logn21	1.13E+06	5.51	2.02	2.13	2.15	15.26
M6	4.03E+06	0.82	1.70	1.74	2.08	7.50
hugetric-00010	7.58E+06	2.73	2.97	3.17	4.13	10.30
rgg_n_2_23_s0	9.65E+06	0.07	0.68	0.70	0.81	3.34
hugetrace-00010	1.39E+07	2.65	2.91	3.09	4.06	10.20
hugebubbles-00010	2.24E+07	2.76	2.95	3.15	4.11	10.26
road_usa	2.62E+07	6.62	3.05	3.51	4.21	12.25
europe_osm	5.79E+07	6.33	1.73	4.79	5.73	12.15
Geom. Mean		3.79	2.35	2.64	3.06	10.63
		Weights				
nlpkkt200		1.84E+7	1.84E+7	1.84E+7	1.84E+7	1.81E+7

Table A.8
The cardinality of the matchings obtained by the exact algorithms and the gap to optimality of the approximation algorithms. For the last problem, cardinalities are shown since the Exact algorithm did not terminate. Random real weights in [1 1.3].

Graph	Card.	Gap to optimality (%)				
	Exact algs.	$1-\epsilon$-Scal. $\epsilon = 1/3$	GPA-ROMA $2/3-\epsilon$ $\epsilon = 0.01$	ROMA	2/3-MVM	1/2-MVM
G34	1,000	2.08	2.57	2.52	3.58	9.25
G39	1,000	2.04	1.13	1.39	1.20	9.76
de2010	11,853	7.26	3.94	4.06	4.53	13.96
shipsec8	57,459	0.00	0.15	0.15	0.17	1.15
kron_g500-logn17	38,823	4.99	2.86	3.01	2.85	16.92
mt2010	63,685	6.54	4.09	4.18	4.53	14.56
fe_ocean	71,718	1.58	1.76	1.77	2.41	8.20
tn2010	117,989	6.75	4.07	4.16	4.50	14.41
kron_g500-logn19	136,770	5.17	2.41	2.53	2.46	16.24
tx2010	449,167	3.84	3.73	3.82	4.26	13.57
kron_g500-logn21	482,339	4.70	2.10	2.22	2.21	15.67
M6	1,750,888	0.90	1.93	1.98	2.36	8.49
hugetric-00010	3,296,382	2.91	3.34	3.55	4.63	11.62
rgg_n_2_23_s0	4,194,303	0.08	0.78	0.80	0.93	3.85
hugetrace-00010	6,028,720	2.83	3.27	3.46	4.55	11.51
hugebubbles-00010	9,729,043	2.95	3.32	3.52	4.61	11.58
road_usa	11,325,669	6.73	3.35	3.82	4.59	13.38
europe_osm	25,149,787	6.67	1.91	5.24	6.29	13.57
Geom. Mean		3.81	2.60	2.91	3.38	11.63
		Cardinality				
nlpkkt200		8.00E+06	7.99E+06	7.99E+06	7.99E+06	7.82E+06

Table A.9
Percentage of time taken by the major steps in the 2/3-MVM and GPA-ROMA approximation algorithms. Random real weights in [1.0 1.3]. The remaining time is spent in variable declarations and initializations.

Graph	GPA-ROMA					2/3-MVM	
	GPA			ROMA		Sort	Aug.
	Sort	Paths and cycles search	Optimal edges by DP	Random permutation	Max 2-aug. search		paths search
G34	3.574	25.88	0.822	1.328	62.18	32.84	46.34
G39	6.970	16.04	0.570	0.703	71.34	19.10	69.99
de2010	4.749	29.93	1.453	1.004	57.77	34.28	61.65
shipsec8	12.76	28.75	0.656	0.167	56.76	1.311	98.51
kron_g500-logn17	13.80	27.43	0.322	0.142	57.52	13.33	84.98
mt2010	3.893	46.37	2.238	0.764	44.14	29.89	67.32
fe_ocean	4.466	40.36	1.934	0.738	49.51	25.33	72.28
tn2010	3.727	45.30	1.826	0.707	45.72	27.90	69.56
kron_g500-logn19	12.70	27.58	0.242	0.125	57.59	12.64	85.67
tx2010	3.396	43.25	1.623	0.585	46.68	20.64	77.43
kron_g500-logn21	10.04	25.53	0.217	0.089	63.89	10.49	88.29
M6	3.287	59.84	2.081	0.547	49.45	14.88	83.52
hugetric-00010	2.673	41.49	2.317	1.202	48.61	22.88	74.55
rgg_n_2_23_s0	4.583	34.21	0.668	0.486	57.61	4.546	94.94
hugetrace-00010	2.452	40.49	2.056	1.571	48.00	21.71	75.51
nlpkkt200	5.913	31.12	0.675	0.463	59.30	2.380	97.36
hugebubbles-00010	2.216	41.04	1.981	1.742	47.90	20.24	77.41
road_usa	1.926	43.91	1.902	1.966	44.76	21.22	76.50
europe_osm	1.772	51.48	1.467	2.321	37.73	21.64	75.44
Geom. Mean	4.501	35.27	1.047	0.618	52.36	14.75	76.68

References

[1] Anonymous, LEDA 6.5 Description. http://www.algorithmic-solutions.com/leda/index.htm. (Accessed 22 October 2018).
[2] Gagan Aggarwal, Gagan Goel, Chinmay Karande, Aranyak Mehta, Online vertex-weighted bipartite matching and single-bid budgeted allocations, in: SODA, SIAM, 2011, pp. 1253–1264.
[3] Ahmed Al-Herz, Alex Pothen, A parallel 2/3-approximation algorithm for vertex-weighted matching, in: Proceedings of SIAM Workshop on Combinatorial Scientific Computing, 2020, 10pp., To appear.
[4] David Applegate, William Cook, Solving Large Scale Matching Problems, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 12, American Mathematical Society, 1993, pp. 557–576.
[5] David Avis, A survey of heuristics for the weighted matching problem, Networks 13 (4) (1983) 475–493.
[6] Ariful Azad, Aydin Buluç, Alex Pothen, Computing maximum cardinality matchings in parallel on bipartite graphs via tree grafting, IEEE Trans. Parallel Distrib. Syst. 28 (1) (2017) 44–59.
[7] Colin E. Bell, Weighted matching with vertex weights: An application to scheduling training sessions in NASA space shuttle cockpit simulators, European J. Oper. Res. 73 (3) (1994) 443–449.
[8] Thomas F. Coleman, Alex Pothen, The null space problem II. Algorithms, SIAM J. Algebr. Discrete Methods 8 (4) (1987) 544–563.
[9] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Software 38 (1) (2011) 1:1–1:25.
[10] Florin Dobrian, Mahantesh Halappanavar, Alex Pothen, Ahmed Al-Herz, A 2/3-approximation algorithm for vertex-weighted matching in bipartite graphs, SIAM J. Sci. Comput. 41 (1) (2019) A566–A591.
[11] Doratha E. Drake, Stefan Hougardy, Improved linear time approximation algorithms for weighted matchings, in: Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques, in: Lecture Notes in Computer Science, vol. 2129, Springer, 2003, pp. 14–23.
[12] R. Duan, S. Pettie, Approximating maximum weight matching in near-linear time, in: Proceedings 51st IEEE Symposium on Foundations of Computer Science (FOCS), 2010, pp. 673–682.
[13] R. Duan, S. Pettie, Linear time approximation for maximum weight matching, J. ACM 61 (1) (2014) Article 1.
[14] R. Duan, S. Pettie, Hsin hao Xu, Scaling algorithms for weighted matching in general graphs, ACM Trans. Algorithms 14 (1) (2018) Article 8.
[15] Iain S. Duff, Jacko Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, SIAM J. Matrix Anal. Appl. 22 (4) (2000) 973–996.
[16] Iain S. Duff, Bora Uçar, Combinatorial problems in solving linear systems, in: Uwe Naumann, Olaf Schenk (Eds.), Combinatorial Scientific Computing, CRC Press, 2009, pp. 21–68.
[17] Sven Hanke, Stefan Hougardy, New Approximation Algorithms for the Weighted Matching Problem, Research Report 101010, Research Institute for Discrete Mathematics, University of Bonn, 2010.
[18] George Karypis, Vipin Kumar, Analysis of multilevel graph partitioning, in: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, ACM, 1995, p. 29.
[19] Arif Khan, Alex Pothen, Mostofa Patwary, Nadathur Satish, Narayanan Sunderam, Fredrik Manne, Mahantesh Halappanavar, Pradeep Dubey, Efficient approximation algorithms for weighted b-matching, SIAM J. Sci. Comput. 38 (2016) S593–S619.
[20] Fredrik Manne, Mahantesh Halappanavar, New effective multithreaded matching algorithms, in: IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 519–528.
[21] Jens Maue, Peter Sanders, Engineering algorithms for approximate weighted matching, in: International Workshop on Experimental and Efficient Algorithms, in: Lecture Notes in Computer Science, vol. 4525, Springer Verlag, 2007, pp. 242–255.
[22] Kurt Mehlhorn, Stefan Naher, Stefan Näher, LEDA: a platform for combinatorial and geometric computing, Cambridge University Press, 1999.
[23] Kurt Mehlhorn, Guido Schäefer, Implementation of $O(nm \log n)$ algorithms for matchings in general graphs: the power of data structures, ACM J. Exp. Algorithmics 7 (2002) 4–23.

[24] Aranyak Mehta, Online matching and ad allocation, Found. Trends Theor. Comput. Sci. 8 (4) (2012) 265–368.
[25] N.S. Mendelsohn, A.L. Dulmage, Some generalizations of the problem of distinct representatives, Canad. J. Math. 10 (1958) 230–241.
[26] Seth Pettie, Peter Sanders, A simpler linear time 2/3- ε approximation for maximum weight matching, Inform. Process. Lett. 91 (6) (2004) 271–276.
[27] Ali Pinar, Edmond Chow, Alex Pothen, Combinatorial algorithms for computing column space bases that have sparse inverses, Electron. Trans. Numer. Anal. 22 (2006) 122–145.
[28] Alex Pothen, Chin-Ju Fan, Computing the block triangular form of a sparse matrix, ACM Trans. Math. Software 16 (4) (1990) 303–324.
[29] Alex Pothen, S.M. Ferdous, Fredrik Manne, Approximation algorithms in combinatorial scientific computing, Acta Numer. 28 (2019) 541–633.
[30] Robert Preis, Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs, in: STACS 99, Springer, 1999, pp. 259–269.
[31] Alexander Schrijver, Combinatorial Optimization: Polyhedra and Efficiency. Volume A: Paths, Flows and Matchings, Springer, Berlin, 2003.
[32] Thomas H. Spencer, Ernst W. Mayr, Node weighted matching, in: Proceedings of the 11th Colloquium on Automata, Languages and Programming, Springer-Verlag, London, UK, 1984, pp. 454–464.
[33] Vahid Tabatabaee, Leonidas Georgiadis, Leandros Tassiulas, QoS provisioning and tracking fluid policies in input queueing switches, IEEE/ACM Trans. Netw. 9 (5) (2001) 605–617.