APPROXIMATION ALGORITHMS FOR MAXIMUM VERTEX-WEIGHTED MATCHING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ahmed Al-Herz

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Alex Pothen, Chair

Department of Computer Science

Dr. Greg Frederickson

Department of Computer Science

Dr. Hemanta Maji

Department of Computer Science

Dr. Elena Grigorescu

Department of Computer Science

Dr. Seth Pettie

Department of Electrical Engineering and Computer Science, University of Michigan

Approved by:

Dr. Clifton Bingham

Head of the Department Graduate Program

In Memory of My Father,

Ibrahim Al-Herz

1956 - 2015

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere appreciation to my advisor Alex Pothen for his support, patience, motivation, and immense knowledge. This dissertation would not be possible without his guidance, and the great amount of time he dedicated to me and this research.

I would like to thank the members of my thesis committee: Greg Frederickson, Hemanta Maji, Seth Pettie, and Elena Grigorescu for taking the time to review my dissertation and for all of their insights and valuable comments. I would also 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.

I also thank the research group at Purdue University: Arif Khan, Mu Wang, Yu Hong Yeung, Xin Cheng, S. M. Ferdous, Shivaram Gopal, Abida Shemonti, Caitlin Whitter, Ahammed Ullah, and Hanjing Xu for their helpful comments and discussions on this research and presentations.

I would like also to thank my sponsor, King Fahd University of Petroleum and Minerals, for the financial support throughout my Ph.D. study.

Last but not least, I wish to express my gratefulness to my family. My grateful thanks go to my parents, sister, and brothers, their prayers and support are always with me. My deepest gratitude goes to my wife Rabab and our children Ibrahim, Maryam, Elias and Muhammed for their patience and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Al-Herz, Ahmed Ph.D., Purdue University, December 2019. Approximation Algorithms for Maximum Vertex-Weighted Matching. Major Professor: Alex Pothen.

We consider the maximum vertex-weighted matching problem (MVM), in which non-negative weights are assigned to the vertices of a graph, and the weight of a matching is the sum of the weights of the matched vertices. Vertex-weighted matchings arise in many applications, including internet advertising, facility scheduling, constraint satisfaction, the design of network switches, and computation of sparse bases for the null space or the column space of a matrix. Let m be the number of edges, n number of vertices, and Δ the maximum degree of a vertex in the graph. We design two exact algorithms for the MVM problem with time complexities of $O(mn)$ and $O(\Delta mn)$. The new exact algorithms use a maximum cardinality matching as an initial matching, after which the weight of the matching is increased using weight-increasing paths.

Although MVM problems can be solved exactly in polynomial time, exact MVM algorithms are still slow in practice for large graphs with millions and even billions of edges. Hence we investigate several approximation algorithms for MVM in this thesis. First we show that a maximum vertex-weighted matching can be approximated within an approximation ratio arbitrarily close to one, to $k/(k+1)$, where k is related to the length of augmenting or weight-increasing paths searched by the algorithm. We identify two main approaches for designing approximation algorithms for MVM. The first approach is direct; vertices are sorted in non-increasing order of weights, and then the algorithm searches for augmenting paths of restricted length that reach a heaviest vertex. (In this approach each vertex is processed once). The second approach repeatedly searches for augmenting paths and increasing paths, again of

restricted length, until none can be found. In this second, *iterative* approach, a vertex may need to be processed multiple times. We design two approximation algorithms based on the direct approach with approximation ratios of 1/2 and 2/3. The time complexities of the 1/2-approximation algorithm is $O(m + n \log n)$, and that of the 2/3-approximation algorithm is $O(m \log \Delta)$. Employing the second approach, we design 1/2- and 2/3-approximation algorithms for MVM with time complexities of $O(\Delta m)$ and $O(\Delta^2 m)$, respectively. We show that the iterative algorithm can be generalized to find a $k/(k+1)$-approximate MVM with a time complexity of $O(\Delta^k m)$. In addition, we design parallel 1/2- and 2/3-approximation algorithms for a shared memory programming model, and introduce a new technique for locking augmenting paths to avoid deadlock and related problems.

MVM problems may be solved using algorithms for the maximum edge-weighted matching (MEM) by assigning to each edge a weight equal to the sum of the vertex weights on its endpoints. However, our results will show that this is one way to generate MEM problems that are difficult to solve. On such problems, exact MEM algorithms may require run times that are a factor of a thousand or more larger than the time of an exact MVM algorithm. Our results show the competitiveness of the new exact algorithms by demonstrating that they outperform MEM exact algorithms. Specifically, our fastest exact algorithm runs faster than the fastest MEM implementation by a factor of 37 and 18 on geometric mean, using two different sets of weights on our test problems. In some instances, the factor can be higher than 500. Moreover, extensive experimental results show that the MVM approximation algorithm outperforms an MEM approximation algorithm with the same approximation ratio, with respect to matching weight and run time. Indeed, our results show that the MVM approximation algorithm outperforms the corresponding MEM algorithm with respect to these metrics in both serial and parallel settings.

1 INTRODUCTION

1.1 Matching Problems and Applications

Matching in graphs is one of the most studied problems in combinatorics due to its importance as a representative problem in computer science as well its applications in many domains. One of the earliest publications on matching algorithms in the last century dates to Kuhn [1] in 1955, which was originally motivated by the need to optimally assign personnel to jobs. Subsequently matching problems have been considered in many applications.

Given a graph $G = (V, E)$ with a set of vertices V, and a set of edges E, a matching M is a subset of edges such that no two edges in M meet at the same vertex. A graph may be unweighted, edge-weighted or vertex-weighted. Thus, we have three basic variations of the matching problem:

1. Maximum cardinality matching (MCM).

2. Maximum edge-weighted matching (MEM).

3. Maximum vertex-weighted matching (MVM).

Most of the previous work has focused on cardinality and edge-weighted matchings, while little attention has been paid to vertex-weighted matchings despite the recent increase in the use of MVM in many applications.

In this thesis we study the maximum vertex-weighted matching problem on non-bipartite graphs and present efficient exact and approximation algorithms.

Vertex-weighted matchings arise in many applications, including internet advertising [2], the design of network switches [3–10], facility scheduling [11], the computation of sparse bases for the null space or the column space of a rectangular matrix [12–14], drawing permutations [15], reverse spanning trees [16], and constraint satisfaction [17, 18]. To provide motivation for our study, we will briefly discuss the role of MVM in solving four problems from different areas.

Vertex-weighted matching in bipartite graphs has recently been applied to 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 finds a matching with weight as close to the optimal as possible (when V is fully known). Aggrawal et al. [19] design an online algorithm for this problem that computes a weight that is at least $(1 - 1/e) \approx 0.632$ of the optimal, assuming that the vertices in V arrive in random order. The best approximation ratio for the online vertex-weighted matching problem is 0.6534 [20], assuming that the vertices in V arrive in random order.

Mehta [2] surveys several ad allocation problems as well as the online algorithms that have been designed to address this problem. He states that "internet advertising constitutes perhaps the largest matching problem 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 one using one of the approximation algorithms we have designed.

Another recent application is network switch scheduling [4], where a switch is modeled as a bipartite graph. Here U is a set of input ports and V is set of output ports. There is an edge (u, v) if a packet in $u \in U$ needs to be routed to $v \in V$. The weight of $u \in U$ is the number of packets at port u; the weight of $v \in V$ is the number of packets that need to be sent to v. The objective of the scheduling policy is to maximize the number of packets sent in one time slot. This is achieved by finding a maximum vertex-weighted matching, where for each matched edge (u, v) the packet is routed from port u to port v. It has been shown that modeling network switch scheduling as a vertex-weighted matching problem is better than modeling based on edge-weighted matching since the vertex-weighted model evacuates all packets in the system in the minimum amount of time given that there are no more new arrivals. The vertex-weighted models are also throughput-optimal.

Vertex-weighted matching can also be applied to scheduling astronaut training sessions [11]. Let T be a set of time periods, R a set of requests from astronauts for resources, and S a set of shared resources. Each astronaut provides a subset of time periods in which he/she is available and a subset of the shared resources needed. The objective is to schedule as many astronauts as possible with no conflicts regarding shared resources. The problem can be solved in two phases. In phase one, a graph is constructed as follows: R is a set of vertices representing the requests, S is a set of vertices representing the shared resources, and there is an edge between $r \in R$ and $s \in S$ if there is at least one time period at which both r and s can be accommodated. Computing a maximum vertex-weighted matching results in a set of matching edges and unmatched vertices, denoted as a group set Gr. In phase two, a graph is constructed with a set of vertices Gr and T, where gr is a vertex in Gr, t is a vertex in T and there is an edge between gr and t if gr can be accommodated at time t. A vertex weight is assigned to a group based on its priorities. The maximum vertex-weight matching results in a matching of groups to periods of time such that the total weight is maximized.

Finally, we will discuss an MVM application in solving the sparsest column-space basis (SCB) problem [13]. Let A be a matrix with k rows and n columns, with $n > k$, and full row rank k. The maximum number of linearly independent columns (or rows) of A is the *numerical rank* of A. The maximum number of nonzeroes in a diagonal is the *structural rank* of A. The numerical rank of a matrix is less than or equal to its structural rank of A. A *basis* for the column-space of A is a linearly independent set of columns of maximum cardinality. A *sparsest basis* for the column-space of A is a basis with the fewest nonzeroes. An SCB problem can be modeled as an MVM problem. An n by k A matrix can be represented as a bipartite graph $G = (S, T, E, \phi)$ with weight function $\phi : S \mapsto R_{\geq 0}$, where set S represents the n columns, set T represents the k rows, and there is an edge $(s, t) \in E$ if the entry $A(s, t)$ is nonzero. The vertex weights in S are given by $\phi(s) = k + 1 - d(s)$, where $d(s)$ equals the number of nonzero elements in column s. A matching M in G is equivalent to a subset of nonzero elements in A, such that no two nonzero elements share a column and a row. The subset of nonzeroes that corresponds to matched edges can be placed on the diagonal of A by permuting the rows and columns of A. Assuming that the numerical rank of A is equal to the structural rank of A and given the weights described above, a sparsest basis is obtained by a maximum vertex-weighted matching in G.

1.2 Approaches for Solving MVM Problems

The MVM problem can be transformed into an MEM problem by assigning each edge a weight obtained by summing the weights at its endpoints, and thus an MEM algorithm can be used to solve the MVM problem. A simpler and more efficient exact algorithm is obtained by solving an MVM problem directly. The first MVM polynomial time algorithm was presented by Spencer and Mayer [21] with time complexity of $O(m\sqrt{n} \log n)$. The Spencer-Mayer algorithm has not been implemented to the best our our knowledge; we have not done so because it is a sophisticated algorithm, and we believe that it would be slower than the simpler algorithms we have designed. Our focus is also on faster approximation algorithms for MVM. A more recent polynomial time algorithm was devised by Dobrian et al. [22] and Halappanavar [23], with time complexity of $O(mn)$. This algorithm was implemented, and we found it is indeed faster than exact MEM algorithms by three to four orders of magnitude. However, there were some instances in which the algorithm did not terminate within 200 hours. In this thesis, we present two exact algorithms with time complexity $O(mn)$ and $O(\Delta mn)$, where Δ is the maximum degree. In practice, the new exact algorithms outperform the earlier exact algorithms. Furthermore, in some instances, the new exact MVM algorithms can be 100 to 1000 times faster than exact MEM algorithms.

Although MVM problems can be solved exactly in polynomial time, on recent big data problems featuring massive graphs with millions and even billions of edges, exact MVM algorithms are slow, taking hours, and occasionally failing to terminate within hundreds of hours. In light of this, there is a demand for fast approximation algorithms. For instance, LEDA [24, 25] (a commercial software library for solving many combinatorial problems) failed to solve the nlpkkt200 problem in 200 hours, whereas our new exact algorithm found the matching in 160 hours. Our fastest 2/3-approximation algorithm computes a matching in under few minutes! It is worth mentioning that semi-streaming algorithms for MCM and MEM have been studied since they were introduced in [26] where a limited space of $O(n \text{ polylog } n)$ can be used and the edges arrive one by one. The best approximation ratio for MEM using a semi-streaming algorithm is $\frac{1}{2} + c$ [27], assuming that the edges arrive in random order, where $c > 0$ is an absolute constant. Many approximation algorithms for solving the MEM problem have been proposed in the last several years. The approximation algorithm with the best approximation ratio guarantees a (1-ϵ) fraction of the maximum edge weight, where ϵ is a positive real number [28]. In contrast, approximation algorithms for MVM have not been well-investigated till our work. While MEM approximation algorithms can be used, their performance in terms of time and matching weight relative to approximation algorithms for the MVM problem has not been studied.

In this thesis we identify two techniques, direct and iterative, for designing approximation algorithms. The direct approach begins with an empty matching, and at each step matches a currently heaviest unmatched vertex to a heaviest unmatched vertex that it can reach by an augmenting path of a restricted length. In this algorithm, once a vertex is matched, it will always remain matched because augmentation does not change a matched vertex to an unmatched vertex. The second approach processes the vertices in arbitrary order. It looks for any augmenting and increasing paths with respect to the current matching and terminates when there is none. This second iterative algorithm has the advantage that it has more concurrency, whereas the first algorithm has to process vertices in a specified order.

We designed two approximation algorithms based on the first approach with approximation ratios of 1/2 and 2/3. The technique for proving the approximation ratio of the 2/3-approximation algorithm requires several new concepts.

Unfortunately there are three drawbacks to the above approach for designing approximation algorithms. First, we do not know how to generalize the algorithm to obtain an approximation ratio of $k/(k+1)$. Second, it is not suitable for parallel implementation since vertices must be processed in a particular order. Third, the algorithm must start with an empty matching and cannot be initialized, which is critical for obtaining fast runtimes in practice.

1.3 Significant Contributions in This Thesis

One of our significant contributions is a theorem that leads to an algorithm with approximation ratio arbitrarily close to one for the MVM problem. We show that if a matching does not admit an augmenting path of length less than or equal to $2k-1$ or a weight-increasing path of length less than or equal to $2k$, then it is a $k/(k+1)$-approximate matching. We present 1/2- and 2/3- approximation algorithms based on this approach that obtain nearly optimal weights while also being fast in practice.

The key advantages of the iterative approach lie in its abilities to initialize the matching and to process vertices in any order, which makes it suitable for parallel implementation. We have designed parallel 1/2- and 2/3-approximation algorithms for a shared memory programming model. In order to compute the correct matching in parallel, locking and synchronization methods must be used. The previous methods for parallelization were based on locking the neighbor of a vertex for non-bipartite graphs [29]. While this suffices for 1/2-approximate matching, it will lead to deadlock or livelock if the augmenting or weight-increasing paths are longer than one edge. We successfully designed a new technique for locking augmenting paths; it identifies the vertices on these paths that need to be locked, and locks them in a specific order. To the best of our knowledge, this is the first parallel algorithm with approximation ratio greater than half for a weighted matching problem.

In addition to the theoretical results, we present extensive experimental results to show that the MVM approximation algorithm outperforms an MEM approximation algorithm with the same approximation ratio. We show that, indeed, the MVM approximation algorithm outperforms the corresponding MEM algorithm in terms of time and matching weight in both serial and parallel settings.

We summarize our contributions in this dissertation as follows:

1. Given a vertex-weighted graph, we can transform it into an edge-weighted graph by summing the vertex weights to obtain edge weights. We prove that the exact MEM and MVM algorithms find the same matching in this graph, provided ties in weights are broken consistently in the two algorithms (Theorem 3.3.1).

2. We obtain a new theorem (Theorem 4.1.1) that identifies sufficient conditions to obtain a $k/(k+1)$-approximation ratio for the MVM problem.

3. We design two new exact MVM algorithms on non-bipartite graphs (Algorithms 15 and 16) and prove their correctness (Theorems 3.5.2 and 3.5.4).

4. We design new 1/2- and 2/3-approximation algorithms for MVM on non-bipartite graphs based on the direct method (Algorithms 17 and 18).

5. We prove the approximation ratio of the new 2/3-approximation direct algorithm using a new proof technique (Theorem 4.2.12).

6. We present new 1/2- and 2/3-approximation algorithms (Algorithms 19 and 20) based on the iterative approach, where an unmatched vertex might have to

be processed multiple times since it could be matched and unmatched in the course of the algorithm.

7. Given a transformed edge-weighted graph obtained from a vertex-weighted graph, we prove that the 1/2-approximation MEM algorithm based on locally dominant edges and the 1/2-approximation MVM algorithm based on the direct method find the same matching (Theorem 4.2.1).

8. We implement serial 1/2- and 2/3-approximation algorithms based on the direct approach.

9. We implement serial and parallel 1/2- and 2/3-approximation algorithms based on the iterative approach on a shared memory parallel computer.

10. We implement the $(1 - \epsilon)$-approximation algorithm for MEM.

11. We evaluate the performance of the new 2/3-approximation iterative algorithm by comparing its running time, matching weight, and cardinality with several MEM approximation algorithms.

12. We show that the new 2/3-approximation algorithms obtain better weight and cardinality than all approximation algorithms for MEM. Moreover, the new 1/2-approximation iterative algorithm runs faster than the fastest 1/2-approximation algorithm for MEM.

13. We show that the new 1/2- and 2/3-approximation parallel algorithms scale very well on a shared memory machine with a modest number of cores; they out-perform the parallel Suitor algorithm.

14. We present an open-source library of C++ routines to compute several variants of matchings called Matchbox.

Some of these results have been published in the following papers. Dobrian et al. [22] describe a 2/3-approximation algorithm for MVM in bipartite graphs in a paper in the *SIAM Journal of Scientific Computing.* Al-Herz and Pothen [30] describe a 2/3-approximation algorithm for MVM in non-bipartite graphs in a paper in *Discrete and Applied Mathematics.* Finally, Al-Herz and Pothen [31] describe a parallel 2/3-approximation algorithm for MVM in non-bipartite graphs in a paper in the Proceedings of the *SIAM Workshop on Combinatorial Scientific Computing*, 2020.

We now describe how the rest of the dissertation is structured.

In Chapter 2, we will provide definitions and background information regarding matching. We will discuss relevant previous work on exact and approximation algorithms for maximum cardinality matching, maximum edge-weighted matching, and maximum vertex-weighted matching.

In Chapter 3, we will describe important concepts related to MVM. First we will prove sufficient and necessary conditions for obtaining an exact MVM. Second, we will prove that when a transformed edge-weighted graph is obtained from a vertex-weighted graph, the MEM and MVM algorithms find the same matching. Third, we will describe the new exact algorithms, prove the algorithms correct, and obtain time complexities for the algorithms. Finally, we will discuss practical improvements to the exact algorithms.

In Chapter 4, we will prove a theorem stating sufficient conditions for obtaining a $k/(k+1)$-approximation ratio for the MVM problem. Then, we will describe the approximation algorithms, prove their time complexity, and prove their correctness. We will prove that the 1/2-approximation algorithm based on the direct approach finds the same matching as the edge-weighted 1/2-approximation algorithm based on the locally dominant approach. Lastly, we will show that the iterative algorithm can indeed be used to find a $k/(k+1)$-approximate matching.

In Chapter 5, we will discuss how the iterative approximation algorithm could be parallelized. We will introduce a new method of locking vertices on augmenting paths in order to augment only along a vertex-disjoint subset of the paths, and prove the correctness of the parallel algorithm.

In Chapter 6, we will present extensive experimental results. We will compare the new approximation algorithms with a set of approximation algorithms for the MEM problem. In particular, we will compare running time, matching weight, and matching cardinality. Additionally, we will compare the number of edges scanned by these algorithms, since it is a metric that does not depend on the machine specifications. We will present a break-down of time for each major step of the approximation algorithms in order to see what percentage of time the algorithm spends on each step. In addition, we will present parallel experimental results using 20 threads of a shared memory parallel processor, and we will report on the speedup and scalability of each algorithm.

Lastly we will summarize our contributions and discuss future work in the concluding Chapter 7.

2 BACKGROUND AND RELATED WORK

In this chapter, we will provide the basic foundations of matching algorithms. More comprehensive coverage of matching theory and matching algorithms can be found in [32–36]. We begin by introducing definitions, notations, and basic theorems in matching, and then we will highlight representative previous work. In particular, we will discuss exact algorithms for maximum cardinality, maximum edge-weighted, and maximum vertex-weighted matching, on both bipartite and non-bipartite graphs. We will keep the descriptions of algorithms brief, since the main goal is to provide basic background and an overview of the development of matching algorithms.

2.1 Foundations

Let $G = (V, E)$ be a simple graph where V is a set of vertices and E is a set of edges; each edge constitutes an unordered binary relation on V. We will use the notations (u, v) and uv for an undirected edge with endpoints at the vertices u and v. The number of vertices and edges are denoted by n and m, respectively. The degree of a vertex u is the number of edges that are incident on u, denoted $d(u)$. The maximum degree of a vertex in a graph will be denoted by Δ. The set of neighbors of a vertex u will be denoted by $N(u)$. A graph can be unweighted or weights could be associated with its edges or vertices. Weights on vertices can be represented as $\phi : V \mapsto R_{\geq 0}$ for weighted vertices and $\phi : E \mapsto R_{\geq 0}$ for weighted edges. A bipartite graph $G = (S, T, E)$ is a graph in which the set of vertices $S \cup T$ can be partitioned into two disjoint sets S and T, such that no edge joins any two vertices in S, and no edge joins any two vertices in T. Since edges only connect vertices in S and vertices in T, bipartite graphs do not contain a cycle of odd length. A matching M in a graph $G = (V, E)$ is a subset of edges such that no two edges in the subset meet at the same vertex. A matching can be seen as an independent set of edges or a set of pairs of vertices. If u is matched in M let $Mate(u) = v$ where $(u, v) \in M$; otherwise $Mate(u) =$ NULL. The cardinality of a matching M is the number of edges in M and is denoted by $|M|$. Based on whether the graph is weighted or not, we have three types of maximum matching problems, which are defined below.

Definition 2.1.1 *Given a graph $G = (V, E)$, the Maximum Cardinality Matching problem is to find a matching M of maximum cardinality in G.*

Definition 2.1.2 *Given a graph $G = (V, E, \phi)$ and a weight function $\phi : E \mapsto R_{\geq 0}$, we define the weight of a matching as $\sum_{e \in M} \phi(e)$, the sum of the weights of the matching edges. In the Maximum Edge-Weighted Matching problem we find a matching M of maximum weight in G.*

Definition 2.1.3 *Given a graph $G = (V, E, \phi)$ and a weight function $\phi : V \mapsto R_{\geq 0}$, we define the weight of a matching as $\sum_{e \in M, e=(u,v)} (\phi(u) + \phi(v))$, the sum of the weights on the endpoints of matching edges. The Maximum Vertex-Weighted Matching problem is to find a matching M of maximum vertex weight in G.*

There are several variants of matching problems. A maximal matching is a matching such that another edge cannot be added to it without violating the matching constraints on the vertices. A perfect matching is matching in which every vertex in the graph is matched. A maximum (or minimum) weighted perfect matching is a perfect matching with maximum (or minimum) weight. A maximal matching is not necessarily a maximum cardinality matching but a perfect matching is always a maximum cardinality matching. A graph might not have a perfect matching. Note that a maximum edge-weighted matching is not necessarily a maximum cardinality matching, whereas a maximum vertex-weighted matching can be chosen to be a maximum cardinality matching when the vertex weights are non-negative.

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 $i \leq j \leq i + k - 1$. 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. A subgraph induced by a subset of vertices X is the graph that includes all edges that join vertices in X. A subgraph induced by a subset of edges F is the graph obtained from the edges in F and all vertices which are its endpoints. We refer to the set of edges of a subgraph H as $E(H)$ and similarly its set of vertices as $V(H)$.

Definition 2.1.4 *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.*

Definition 2.1.5 *An* augmenting path *P with respect to M is an M-alternating path of odd length where the first and last vertices on the M-alternating path P are unmatched.*

Definition 2.1.6 *In the context of vertex-weighted matching, a weight-increasing path P with respect to M is an M-alternating path of even length where one endpoint is unmatched and the other endpoint is matched such that the weight of the unmatched vertex is heavier than that of the matched vertex.*

Definition 2.1.7 *A vertex v on a path P is an outer vertex if the number of edges from an unmatched vertex to v is even, and an inner vertex if the number is odd. All unmatched vertices are considered as outer vertices.*

The matching M can be augmented by matching the edges in the symmetric difference $M' = M \oplus P$, which is a matching of cardinality $|M| + 1$. Augmenting paths may used to find a maximum cardinality matching by repeatedly searching an augmenting path from an unmatched vertex, if it exists. When no augmenting path can be found with respect to M, then the cardinality of the matching is maximum. The following theorem from Berge [37] is a cornerstone to many matching algorithms.

Theorem 2.1.1 *A matching M in a graph G is a maximum matching if and only if there is no M-augmenting path in G.*

Proof If there exists an augmenting path P in G with respect to M, then the cardinality of M can be increased by matching the edges in $M \oplus P$, and hence M cannot be a maximum matching. Conversely, assume M is not a maximum matching and M' is a maximum matching. Consider the symmetric difference $M \oplus M'$, which results in $M \oplus M'$-alternating paths and even cycles. In the case of an alternating path or an alternating cycle of even lengths, both M and M' match the same number of edges. Since $|M'| > |M|$, there must exist an $M \oplus M'$-alternating path P of an odd length such that $|P \cap M'| > |P \cap M|$ where the first and the last edges belong to M'. Thus, P is augmenting path with respect to M in G. ■

2.2 Exact Algorithms

Here, we will give generic descriptions of the three types of maximum matching. Moreover, we will describe the basic combinatorial algorithms for cardinality and vertex-weighted matching and primal-dual algorithms for edge-weighted matching. There are scaling and randomized algebraic techniques [38–42], that are not discussed here. For each type of maximum matching we will start by describing a generic bipartite maximum matching algorithm because it is simpler; we will then describe the non-bipartite maximum matching algorithm.

2.2.1 Maximum Cardinality Matching

Bipartite Graphs

Let $G = (S, T, E)$ be a bipartite graph and M an empty matching. The algorithm picks an unmatched vertex $s \in S$ and searches for an augmenting path P. If P is found then M is augmented by $P \oplus M$, and the algorithm repeats until all vertices in S are considered. if we fail to find an augmenting path from an unmatched vertex during the algorithm, then it does not need to be considered again. The search for an augmenting path takes $O(m)$ time, and since we have n vertices, the time complexity is $O(mn)$.

Algorithm 1 Exact Algorithm for MCM on Bipartite Graphs.

procedure EXACT-MCM-BIP($G = (S, T, E)$)
  $M \leftarrow \emptyset$;
  for all $s \in S$ **do**
    Starting from s search for an augmenting path P;
    if P is found **then**
      $M \leftarrow M \oplus P$;
    end if
  end for
end procedure

Hopcroft and Karp [43] presented an algorithm that drastically reduces the number of searches by finding a maximal set of vertex-disjoint augmenting paths of shortest length in one $O(m)$ search. The aim is to divide the searches into phases, and in each phase, a maximal set of shortest length vertex-disjoint M-augmenting paths is found. It can be shown that the number of phases is bounded by $O(\sqrt{n})$. Since each phase takes $O(m)$, the time complexity for such an approach in bipartite graphs is $O(m\sqrt{n})$. We refer the reader to [43] for a proof.

Non-bipartite Graphs

Solving the maximum cardinality matching on non-bipartite graphs is more complicated. If the bipartite algorithm described above is used on a non-bipartite graph, the search for an augmenting path may not succeed because of the existence of odd length cycles. Odd length cycles which have the maximum cardinality of matched edges are called blossoms.

Definition 2.2.1 *Let M be a matching in G and B be an odd set of vertices. B is a blossom if the set of vertices in B form a cycle C and the number of matching edges in C is $(|B|-1)/2$, that is, $|C \cap M| = |(|B|-1)/2|$.*

Consider the example in Figure 2.1, where the search for an augmenting path starts from v_1 and there exists an augmenting path $\{v_1, v_2, v_3, v_4, v_6, v_7, v_5, v_8\}$. If a breadth-first search is used, then two alternating paths are found: $\{v_1, v_2, v_3, v_4, v_6\}$ and $\{v_1, v_2, v_3, v_5, v_7\}$, and both fail to find the augmenting path. If a depth-first search is used, then one alternating path could be $\{v_1, v_2, v_3, v_5, v_7, v_6, v_4\}$, which also fails to find the augmenting path.

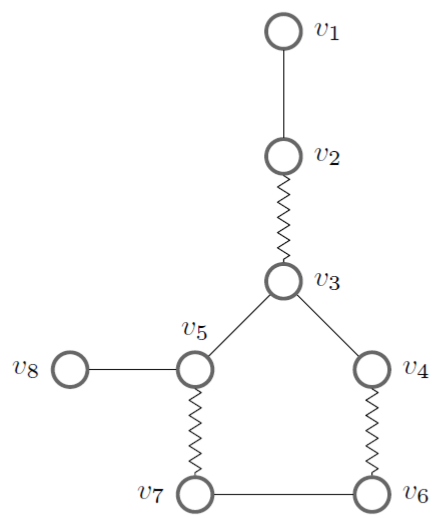

Figure 2.1. A correct augmenting path is $\{v_1, v_2, v_3, v_4, v_6, v_7, v_5, v_8\}$.

Edmonds [44] discovered a remarkable solution to the problem caused by odd cycles. His idea was to shrink a blossom, replace it with a super vertex v_B, and replace the edge set E_1 incident on vertices in B with the set $E_2 = \{(v_B, j) | \exists (i, j) \in E_1,\ i \in B, j \notin B\}$.

Now, we can describe the maximum matching algorithm. Let $G = (V, E)$ be a graph and M an empty matching. The algorithm picks an unmatched vertex u and searches for an augmenting path. If a blossom is discovered, then the algorithm shrinks the blossom, updates G, and recursively starts searching for an augmenting path from u. If an augmenting path is found, then the algorithm expands all blossoms on the augmenting path recursively to recover the augmenting path in the original graph. Then, the algorithm augments the matching. The algorithm repeats until all vertices are considered.

Searching for an augmenting path takes $O(m)$ time, updating a graph that is caused by discovering a blossom costs $O(m)$ time, and there can be at most $O(n)$ recursive shrunken blossoms. Thus, each search in Edmonds' algorithm takes $O(mn)$ time, and since we have n stages, the time complexity is $O(mn^2)$. Gabow [45] improved the time complexity to $O(n^3)$ by avoiding explicitly shrinking blossoms and reconstructing the graph. A blossom membership array is used in order to find which blossom a vertex belongs to, and when a blossom is discovered, the inner vertices are inserted into a queue for future searches, which already includes the outer vertices. Another improvement is using a union-find data structure for managing blossom membership [45]. Finding which blossom a vertex belongs to takes $O(m\alpha(n,m))$ per stage, so the total time complexity is $O(nm\alpha(n,m))$, where α is the inverse of Ackermann's function. The cost per stage can be reduced to $O(m)$ by using the incremental tree set union algorithm [46].

Like the bipartite case, the MCM problem in non-bipartite graphs can be solved in $O(\sqrt{n}m)$ using $O(\sqrt{n})$ phases [47]. In each phase the algorithm finds a maximal set of shortest length vertex disjoint augmenting paths. We refer the reader to [47,48] for a detailed description.

2.2.2 Maximum Edge-Weighted Matching

The MEM problem can be formulated as a linear programming problem to which the theory of duality can be applied. In this section, we will discuss primal-dual solutions to the MEM problem in bipartite and non-bipartite graphs. We refer the

Algorithm 2 Exact Algorithm for MCM on Non-Bipartite Graphs.

procedure EXACT-MCM($G = (V, E)$)
    $M \leftarrow \emptyset$;
    for all $u \in V$ **do**
        Search for an augmenting path P starting from u;
        if a blossom B is found **then**
            Shrink B and continue the search for an augmenting path P from u;
        end if
        if P is found **then**
            Expand all blossoms recursively that P goes through;
            $M \leftarrow M \oplus P$;
        end if
    end for
end procedure

reader to [32] for more detailed discussions about linear programming and primal-dual methods.

Bipartite Graphs

The primal-dual solution for the MEM problem on bipartite graphs is known as the Hungarian method, and it was first applied to the assignment problem proposed by Harold W. Kuhn [1]. Consider a bipartite graph $G = (S, T, E, \phi)$ with weight function $\phi : E \mapsto R_{\geq 0}$. The primal-dual formulation for the MEM problem is given by:

$$
\begin{array}{lll}
\text{Primal problem:} & & \\
\max & \sum\limits_{st \in E} \phi(st)x(st) & \\
\text{s.t.} & 0 \leq x(st) \leq 1 & \text{for} \quad \forall st \in E, \\
& \sum\limits_{t \in T} x(st) \leq 1 & \text{for} \quad \forall s \in S, \\
& \sum\limits_{s \in S} x(st) \leq 1 & \text{for} \quad \forall t \in T.
\end{array}
$$

$$
\begin{array}{lll}
\text{Dual problem:} & & \\
\min & \sum\limits_{s \in S} y(s) + \sum\limits_{t \in T} y(t) & \\
\text{s.t.} & y(s) + y(t) \geq \phi(st) & \text{for} \quad \forall st \in E, \\
& y(s) \geq 0 & \text{for} \quad \forall s \in S, \\
& y(t) \geq 0 & \text{for} \quad \forall t \in T.
\end{array}
$$

A primal variable $x(st)$ is assigned to each edge $st \in E$, and can take a value of 1 (for a matching edge) or 0 (for a non-matching edge). A dual variable $y(v)$ is assigned to each vertex $v \in S \cup T$. The dual variables are used to guide the graph search procedure. Let $\pi(st)$ be a slack variable for each edge $st \in E$, such that $\pi(st) = y(s) + y(t) - \phi(st)$. If $\pi(st) = 0$, we say the edge st is tight. A primal-dual solution is optimal if the following complimentary slackness conditions hold:

1. $\pi(st) \geq 0$, $\forall st \in E$.

2. If st is a matching edge, then $\pi(st) = 0$.

3. If s is an unmatched vertex, then $y(s) = 0$.

The main idea of the primal-dual algorithm is to start with an initial solution that satisfies conditions 1 and 2 but violates condition 3. For instance consider the following initialization: $y(s)$ is assigned the maximum weight of an edge incident on s for all $s \in S$, and $y(t)$ is assigned 0 for all $t \in T$. This assignment satisfies conditions 1 and 2 but violates condition 3, and during the course of the algorithm, the number of violations of condition 3 is reduced while maintaining conditions 1 and 2.

The algorithm picks an unmatched vertex s, and then searches for an augmenting path that starts from s and uses tight edges. If an augmenting path is found, then the current matching is augmented with this path. If no tight edges can be found, the duals are adjusted by the minimum positive slack $\delta_{min} = \min(\pi(st))$ such that $\pi(st) > 0\ \forall st \in E$ as follows:

- $y(s) \leftarrow y(s) - \delta_{min}$.

- $y(t) \leftarrow y(t) + \delta_{min}$.

After the duals are adjusted, new tight edges could cause an augmenting path to be found. The steps repeat until the current vertex s is matched or the dual variable $y(s)$ becomes 0. The time complexity of the algorithm is influenced by the method of finding the minimum slack δ_{min} and updating the duals. A simple array search leads to $O(n^3)$ time complexity [49]. Using a binary heap will reduce the time complexity to $O(mn \log n)$ [50, 51], and using a Fibonacci heap will result in $O(mn + n^2 \log n)$ time [52]. The fastest exact MEM algorithm on bipartite graphs using integer weights is presented in [53] and has a time complexity of $O(\sqrt{n}m \log(W))$, where W is the maximum edge weight.

Algorithm 3 Exact Algorithm for MEM on Bipartite Graphs.

procedure EXACT-MEM-BIP($G = (S, T, E, \phi)$)
    $M \leftarrow \emptyset$;
    $\forall s \in S$ $y(s) = \max(\phi(st))$;
    $\forall t \in T$ $y(t) = 0$;
    for all $s \in S$ **do**
        while $\exists$ a tight edge **do**
            Search for an augmenting path P starting from s using tight edges;
            if P is found **then**
                $M \leftarrow M \oplus P$;
                break;
            else
                $\delta_{min} \leftarrow$ minimum positive slack;
                $y(s) \leftarrow y(s) - \delta_{min}$;
                $y(t) \leftarrow y(t) + \delta_{min}$;
            end if
        end while
    end for
end procedure

Non-bipartite Graphs

If we solve the non-bipartite problem using the primal-dual formulation used for bipartite graphs, the matching may not be correct since the solution to the linear program may have fractional values. Consider a triangle graph and assume the weight of each edge is 1. In this case, $\sum_{uv \in E} x(uv) = 1.5$ is an optimal solution to the linear program, where each $x(uv) = 0.5$. Edmonds [54] proposed an ingenious way to solve this problem by adding constraints of the form

$$\sum_{uv \in E(B)} x(uv) \leq (|B| - 1)/2, \ \forall B \in V_{odd},$$

where V_{odd} is the set of all odd size subsets of V. The primal-dual formulation is as follows:

Primal problem:

$$\begin{array}{lll} \max & \sum_{uv \in E} \phi(uv)x(uv) & \\ \text{s.t.} & 0 \leq x(uv) \leq 1 & \text{for } \forall uv \in E \\ & \sum_{uu' \in E} x(uu') \leq 1 & \text{for } \forall u \in V \\ & \sum_{uv \in E(B)} x(uv) \leq (|B| - 1)/2 & \text{for } \forall B \in V_{odd}. \end{array}$$

Dual problem:

$$\begin{array}{lll} \min & \sum_{u \in V} y(u) + \sum_{B \in V_{odd}} z(B)(|B - 1|)/2 & \\ \text{s.t.} & y(u) + y(v) + \sum_{B \in V_{odd}: uv \in B} z(B) \geq \phi(uv) & \text{for } \forall uv \in E, \\ & y(u) \geq 0 & \text{for } \forall u \in V, \\ & z(B) \geq 0 & \text{for } \forall B \in V_{odd}. \end{array}$$

The slack is given by $\pi(uv) = y(u) + y(v) + \sum_{B \in V_{odd}: uv \in B} z(B) - \phi(uv)$. The optimality of the primal-dual solution is given if the following complimentary slackness conditions hold:

1. $\pi(uv) \geq 0 \ \forall uv \in E$.

2. If uv is a matching edge, then $\pi(uv) = 0$.

3. If u is an unmatched vertex, then $y(u) = 0$.

4. $z(B) > 0$ for all shrunken blossoms.

Let V_{outer} be the set of outer vertices, V_{inner} be the set of inner vertices, $V_{non} = V \setminus V_{outer} \cup V_{inner}$. Additionally, let B_{outer} be the set of outer blossoms and B_{inner} be the set of inner blossoms. The outer and inner blossoms are defined as the outer and inner vertices in Definition 2.1.7. The algorithm consists of $O(n)$ stages. In each stage, we search for an augmenting path using tight edges. If an augmenting path is found, then augment the matching. If a blossom B is discovered, then shrink it and set $z(B) = 0$. If an inner blossom B is visited and its dual $z(B)$ equals 0, then expand B. If there are no tight edges, then update dual variables by δ_{min} to make a new tight edge. We choose

$$\delta_{min} = \min\{\delta_1, \delta_2, \delta_3\}, \quad \text{where}$$

- $\delta_1 = \min(\pi(uv))$, if $u \in V_{outer}$ and $v \in V_{non}$.

- $\delta_2 = \min(\pi(uv)/2)$, if $u, v \in V_{outer}$.

- $\delta_3 = \min(z(B))$, if $B \in B_{inner}$.

Then we update the duals as follows:

- $y(u) \leftarrow y(u) - \delta_{min}$, if $u \in V_{outer}$.

- $y(u) \leftarrow y(u) + \delta_{min}$, if $u \in V_{inner}$.

- $z(B) \leftarrow z(B) + 2\delta_{min}$, if $B \in B_{outer}$.

- $z(B) \leftarrow z(B) - 2\delta_{min}$, if $B \in B_{inner}$.

If $\delta_{min} = \delta_3$ then expand all inner blossoms with $z(B) = 0$. If $\delta_{min} = \delta_1$ or δ_2 then the tight edges can be used to grow the search or to discover a new blossom. Again, as in the bipartite algorithm, the costliest part is finding the minimum slack and updating dual variables. The fastest exact MEM algorithm on non-bipartite graphs for integer weights is given in [53] and has a time complexity of $O(\sqrt{n}m\log(nW))$, where W is the maximum edge weight.

Algorithm 4 Exact Algorithm for MEM on Non-Bipartite Graphs.

procedure EXACT-MEM($G = (V, E, \phi)$)
  $M \leftarrow \emptyset$;
  $\forall u \in V \ y(u) = \max(\phi(e))/2$;
  for all $u \in V$ **do**
    while $\exists$ a tight edge $e = uv$ such that $u \in V_{outer}$ **do**
      One of the following steps is done;
      Growing a search tree step, if $v \in V_{non}$;
      Shrinking a blossom step, if $v \in V_{outer}$ and v is matched;
      Augmenting a path step, if $v \in V_{outer}$ and v is unmatched;
      Expanding a blossom step, if and $v \in B_{inner}$ and $z(B) = 0$;
      if $\nexists$ a tight edge **then**
        $\delta_{min} \leftarrow \min\{\delta_1, \delta_2, \delta_3\}$;
        $y(u) \leftarrow y(u) - \delta_{min}$ for $\forall u \in V_{outer}$;
        $y(u) \leftarrow y(u) + \delta_{min}$ for $\forall u \in V_{inner}$;
        $z(B) \leftarrow z(B) + \delta_{min}$ for $\forall B \in B_{outer}$;
        $z(B) \leftarrow z(B) - \delta_{min}$ for $\forall B \in B_{inner}$;
      end if
    end while
  end for
end procedure

2.2.3 Maximum Vertex-Weighted Matching

Bipartite Graphs

Tabatabaee et al. [3] proposed an algorithm that was used for designing network switches. The algorithm works as follows. First, it computes a maximum cardinality matching and then sorts the unmatched vertices in non-increasing order of weights. From each unmatched vertex in this order, it searches for a weight-increasing path. If an increasing path is found, then it updates the matching, and if not, it proceeds to the next unmatched vertex in the order. A maximum cardinality matching can be computed in $O(m\sqrt{n})$ time; searching for increasing paths takes $O(mn)$ time. Thus the time complexity is $O(mn)$.

Dobrian et al. [22] and Halappanavar [23] proposed an algorithm that exploits the structure of bipartite graphs. The premise is to sort the vertices in non-increasing order of weights and decompose the problem into two one-side-weighted problems. After this, the two problems are solved separately by finding augmenting paths from each vertex. The two matchings can be combined into a final matching by invoking the Mendelsohn-Dulmage Theorem [55]. Finding a maximum vertex-weighted matching in a one-side-weighted graph takes $O(mn)$ time, and combining the two matchings into the final matching takes $O(n)$ time. Thus the time complexity of this algorithm is $O(mn)$.

Non-bipartite Graphs

Spencer and Mayer [21] presented an exact algorithm for the MVM problem. Vertices are sorted in non-increasing order of their weights, and a non-bipartite graph is transformed into a bipartite graph by shrinking all blossoms and assigning a shrunken blossom the weight of the lightest vertex in a blossom. In addition, the bipartite weighted matching is solved as two one-side-weighted problems. Then the two matchings are combined using the Mendelsohn-Dulmage Theorem [55]. A divide and con-

Algorithm 5 Exact Algorithm for MVM on Bipartite Graphs (Dobrian et. al.).

```
procedure MVM-Bip(G = (S, T, E, φ))
    M ← ∅; M_S ← ∅; M_T ← ∅;
    Q ← S;
    while Q ≠ ∅ do
        u ← heaviest(Q);
        Q ← Q − u;
        Find an augmenting path P starting at u;
        if P found then
            M_S ← M_S ⊕ P;
        end if
    end while
    Q ← T;
    while Q ≠ ∅ do
        u ← heaviest(Q);
        Q ← Q − u;
        Find an augmenting path P starting at u;
        if P found then
            M_T ← M_T ⊕ P;
        end if
    end while
    M ← MendelsohnDulmage(M_S, M_T, M);
end procedure
```

quer strategy is used to recursively split the one-side-weighted bipartite graph into smaller sub-problems, in each of which a maximum cardinality matching is found. The divide and conquer strategy divides the graph to $\log n$ parts, and at each step a maximum cardinality matching takes at most $O(m\sqrt{n})$ time. Thus the Spencer and Mayer algorithm takes $O(m\sqrt{n}\log n)$ time.

Dobrian et al. [22] and Halappanavar [23] proposed a simpler MVM algorithm. Vertices are sorted in non-increasing order of weights. The algorithm starts with an empty matching M, and then it attempts to match a heaviest unmatched vertex u. From u the algorithm searches for a heaviest unmatched vertex v that it can reach by an augmenting path P. If it finds P, then the matching is augmented by forming the symmetric difference of the current matching M with P, and the vertices u and v are removed from the set of unmatched vertices. If it fails to find an augmenting path from u, then u is removed from the set of unmatched vertices. When all the unmatched vertices have been processed, the algorithm terminates. The time complexity of the algorithm is $O(mn)$. We will revisit this algorithm in the next chapter with a proof of its correctness, since one of our approximation algorithms is based on it with a restriction on the augmenting path length.

2.3 Approximation Algorithms

An α-approximation matching algorithm finds a matching that is within a factor of α of the weight of the exact matching. If M_α is a matching that is computed by an α-approximation algorithm and M_{opt} is the optimal matching, then $\sum_{u\in M_\alpha}\phi(u) \geq \alpha\sum_{u\in M_{opt}}\phi(u)$. Approximation algorithms are generally designed for NP-hard problems [56–59]. However, polynomial time exact matching algorithms are very slow for applications involving massive graphs. and ones that require especially fast computations. For many applications (e.g., big data) a matching needs to be computed fast on massive graphs, and the optimality of the matching is not crucial. This is one motivation for the development of fast approximation algorithms. Another motivation is the necessity for parallel algorithms on massive graphs, when a processor is not able to store the graph in memory; often the exact algorithms do not possess much concurrency but the approximation algorithms do.

A recent detailed survey of approximation algorithms for several variant maximum matching problems (cardinality, edge-weighted matching, vertex-weighted matching, and b-matching) and the related minimum edge cover problems (cardinality, edge-weighted, and b-edge cover) is provided in [60]. An earlier survey of cardinality and edge-weighted matching approximation algorithms is given in [61]. In this section we will briefly discuss some of the recent developments in approximation algorithms for edge- and vertex-weighted matching.

2.3.1 Edge-Weighted Matching

Avis [62] proposed a simple 1/2-approximation algorithm for maximum edge-weighted matching. Given a graph $G = (V, E, \phi)$ with weight function $\phi : E \mapsto R_{\geq 0}$, consider the edges in non-increasing order of weights. The algorithm picks a heaviest non-matching edge and adds it to the matching M. Then it deletes all the edges that are incident on the endpoints of the current matching edge. The algorithm repeats the process until all the edges have been considered. The cost of sorting edges is $O(m \log n)$, so the total time is $O(m \log n + m)$.

Algorithm 6 The Greedy 1/2-Approximation Algorithm for MEM.

```
procedure GREEDY(G = (V, E, φ))
    M ← ∅;
    while E ≠ ∅ do
        Pick a heaviest edge uv ∈ E;
        M ← M ∪ uv;
        Delete all edges incident on vertices u and v;
    end while
end procedure
```

The locally-dominant-edge approximation algorithm was proposed by Preis [63]. It guarantees a 1/2-approximation for maximum edge-weighted matching and runs in linear time $O(m)$. Given a graph $G = (V, E, \phi)$ with weight function $\phi : E \mapsto R_{\geq 0}$, the algorithm arbitrarily picks a non-matching edge $uv \in E$. It scans the edges that are incident to the vertices u and v. An edge uv is said to be a locally-dominant if it is at least as heavy as all other edges incident on the vertices u and v. If an edge ux (or vy) is found such that $\phi(ux) > \phi(uv)$ (or $\phi(vy) > \phi(uv)$) then the algorithm proceeds to the edge ux (or vy). The algorithm repeats recursively until a locally-dominant edge is found, and adds it to the current matching. After that the algorithm removes all the edges that are incident on the matching edge. The algorithm repeats until all the edges have been deleted. When all edges incident on a path have been deleted, the algorithm begins searching for a new path from another non-matching edge chosen arbitrarily.

Algorithm 7 The Locally Dominant Edge 1/2-Approximation Algorithm for MEM.

```
procedure LOCAL-DOM(G = (V, E, φ))
    M ← ∅;
    while E ≠ ∅ do
        Pick an arbitrary edge uv ∈ E;
        if uv is locally dominant edge then
            M ← M ∪ uv;
            Delete all edges incident on u and v;
        else
            Searching from u and v, find a locally dominant edge xy ∈ E;
            M ← M ∪ xy;
            Delete all edges incident on x and y;
        end if
    end while
end procedure
```

Drake and Hougardy [64] proposed a simpler algorithm based on the concept of growing a path in a graph. The path-growing algorithm guarantees a 1/2-approximation for maximum edge-weighted matching, and works as follows. Given a graph $G = (V, E, \phi)$ with weight function $\phi : E \mapsto R_{\geq 0}$ and two empty matching sets M_1 and M_2, the algorithm starts with an arbitrary unmatched vertex u. The algorithm searches for a heaviest edge $uv \in E$ incident on u and adds it to the matching set M_1. Then, the algorithm deletes u and all other edges incident on u from G. Next, the algorithm proceeds to v, and repeats the same steps, except, this time, it adds a heaviest edge $vw \in E$ to the matching set M_2. The algorithm repeats the process of adding new edges alternatively to sets M_1 and M_2. After all edges are deleted, the final matching is the heavier of M_1 and M_2. The time complexity of the path growing algorithm is clearly $O(m)$ since it requires scanning the adjacent edges for each vertex once.

The dynamic programming method can be used to find optimal matching edges from each path grown by the Drake-Hougardy algorithm. Yet another 1/2-approximation algorithm is the Global Paths algorithm (GPA) which was proposed by Maue and Sanders [65]. The algorithm 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. It then computes a maximum weight matching for each path and cycle using dynamic programming, and it deletes the matching edges as well as their adjacent edges. The algorithm repeats this process until all edges are deleted. The time complexity of the GPA algorithm is $O(m \log n)$.

A more recent 1/2-approximation algorithm is the Suitor algorithm [29], which employs a proposal-based approach similar to the classical algorithm for a stable matching [66]. Each vertex u proposes to a heaviest vertex v that still has not received a better proposal earlier. If v already has a proposal of lower weight from a vertex w, then v annuls it and accepts the new proposal from u; the annulled vertex w must propose again for another partner. An edge is matched when both vertices

Algorithm 8 The Path Growing 1/2-Approximation Algorithm for MEM.

```
procedure PG(G = (V, E, φ))
    M ← ∅;
    M_1 ← ∅; M_2 ← ∅;
    while E ≠ ∅ do
        i = 1;
        Pick an arbitrary vertex u ∈ V of degree at least 1;
        while u has degree at least 1 do
            Let uv be a heaviest neighbor of u;
            M_i ← M_i ∪ uv;
            Delete all edges incident on u;
            u = v;
            i = 3 − i
        end while
    end while
    M ← max {M_1, M_2};
end procedure
```

Algorithm 9 The Suitor 1/2-Approximation Algorithm for MEM.

```
procedure SUITOR(G = (V, E, φ))
    M ← ∅;
    suitor(u) ← NULL ∀u ∈ V;
    ws(u) ← 0 ∀u ∈ V;
    for each u ∈ V do
        x = u;
        done = False;
        while not done do
            partner = suitor(x);
            heaviest = ws(x);
            for each v ∈ N(x) do
                if φ(vx) > heaviest and φ(vx) > ws(v) then
                    partner = v;
                    heaviest = φ(vx);
                end if
            end for
            done = True;
            if heaviest > 0 then
                y = suitor(partner);
                suitor(partner) = x;
                ws(partner) = heaviest;
                if y ≠ NULL then
                    x = y;
                    done = False;
                end if
            end if
        end while
    end for
end procedure
```

propose to each other. The Suitor algorithm's time complexity is $O(\Delta m)$ in the worst case, where Δ is the maximum degree.

Two recent approaches have been used to achieve a better approximation ratios than 1/2. The first approach employs short augmenting paths and cycles repeatedly until a specific criterion is reached. The second approach is based on the primal-dual formulation. We will describe two $(2/3-\epsilon)$-approximation algorithms that are based on the first approach. Then, we will describe a $(1-\epsilon)$-approximation algorithm that is based the on the primal-dual formulation.

Before outlining the $(2/3-\epsilon)$-approximation algorithms we will describe a 2-augmentation centered at a vertex u. An arm of u is defined to be either an edge $\{u, v\}$ or a path $\{u, v, v'\}$, where (u, v) is a non-matching edge and (v, v') is a matching edge. The gain of an arm is defined to be the weight of the non-matching edge minus the weight of the matching edge and if the arm consists of a non-matching edge then the gain is the weight of the non-matching edge. We have two cases:

Case 1) u is unmatched: find an arm of u with the highest positive gain.

Case 2) u is matched to u': 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 (u, u'). (2) Alternating paths: the search is executed as follows: Find two vertex disjoint arms of u, with the highest gains P and P', then find an arm of u' with highest gain Q. If P and Q are vertex disjoint then $P \cup (u, u') \cup Q$ is a highest gain alternating path; otherwise choose $P \cup (u, u') \cup Q$ as a highest gain alternating path. Now we will consider the $(2/3-\epsilon)$-approximation algorithms. The random matching algorithm (RAMA) [67] chooses a random vertex u and performs a 2-augmentation centered at u with the highest-gain. This is repeated $k = n\frac{1}{3}\ln\frac{1}{\epsilon}$ times. The random ordered matching algorithm (ROMA) [65] permutes the order of vertices, and each vertex u in the permuted order performs 2-augmentation with the highest-gain centered at u. This is repeated for $k = \frac{1}{3}\ln\frac{1}{\epsilon}$ phases. If no further improvement can be achieved after finishing a phase, then the algorithm terminates.

The same technique has been used to achieve $(3/4-\epsilon)$-approximation [68, 69] by extending the length of augmentations.

Algorithm 10 The Random $(2/3-\epsilon)$-Approximation Algorithm for MEM.

```
procedure RAMA(G = (V, E, φ), k)
    M ← ∅ (or initialized with any matching);
    for i = 1 to k do
        Randomly pick a vertex u ∈ V;
        M ← M⊕ 2-augmentation(u);
    end for
end procedure
```

Algorithm 11 The Random Ordered $(2/3-\epsilon)$-Approximation Algorithm for MEM.

```
procedure ROMA(G = (V, E, φ),k)
    M ← ∅ (or initialized with any matching);
    for i = 1 to k do
        Permute the order of vertices;
        for each u ∈ V in the permuted order do
            M ← M⊕ 2-augmentation(u);
        end for
        if there is no improvement then
            break;
        end if
    end for
end procedure
```

A $(1-\epsilon)$-approximation algorithm has also been proposed [28,68]. The algorithm is based on the scaling technique, the primal dual formulation of the problem, and relaxed feasibility and complementary slackness. Let W be the maximum edge weight, $\epsilon' = \Theta(\epsilon)$, $\delta_0 = W/\epsilon'$, $\delta_i = \delta_0/2^i$, $\phi_i(e) = \delta_i\lfloor \phi(e)/\delta_i \rfloor$ and $\gamma = \log 1/\epsilon$. A root blossom is a blossom that is not contained in any other blossom. The dual variables y are defined over the vertices, and z over the blossoms. We define the variable yz over the edge $e = uv$ as

$$yz(e) = y(u) + y(v) + \sum_{uv \in E(B)} z(B),$$

where B is a blossom.

The approximation algorithm consists of $\log W + 1$ scales. At every scale i, the relaxed feasibility and complementary slackness conditions hold:

1. $z(B)$ is a non-negative multiple of δ_i $\forall B \in V_{odd}$ and $y(u)$ is a non-negative multiple of $\delta_i/2$ $\forall u \in V$.

2. $z(B) > 0$ for all root blossoms.

3. $yz(e) \geq \phi_i(e) - \delta_i$, $\forall e \in E$.

4. $yz(e) \leq \phi_i(e) + (\delta_j - \delta_i)$, $\forall e \in M$, where e becomes a matching edge in scale j and $j \leq i$.

At each scale only eligible edges are considered. At scale i an edge e is eligible if at least one of the following hold:

- e is in a blossom.

- If e is a matching edge, $yz(e) - \phi_i(e)$ is a non-negative integer multiple of δ_i, and $\log \phi(e) \geq i - \gamma$.

- If e is a non-matching edge, $yz(e) = \phi_i(e) - \delta_i$ and $\log \phi(e) \geq i - \gamma$.

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 i-th 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 duals of discovered blossoms to zero; it updates dual variables of vertices and blossoms reached by unmatched vertices in the search; and it expands inner blossoms whose dual variables are equal to zero. The four steps are repeated until the duals 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 $O(m\epsilon^{-1} \log \epsilon^{-1})$.

2.3.2 Vertex-Weighted Matching

A 2/3-approximation algorithm for MVM on bipartite graphs was proposed by Dobrian et al. [22,23]. The vertices are sorted in non-increasing order of weights, and the problem is decomposed into two one-side-weighted problems. These problems are solved individually by restricting the length of augmenting paths to at most three. The two matchings are then combined into a final matching by invoking the Mendelsohn-Dulmage Theorem. The time complexity is $O(m + n \log n)$.

Algorithm 12 $(1-\epsilon)$-Approximation Algorithm for MEM.

1: **procedure** SCALING($G = (V, E, \phi)$,ϵ)
2: $M \leftarrow \emptyset$;
3: $\delta_0 \leftarrow \epsilon' W$; // W is the maximum edge weight, and $\epsilon' = \Theta(\epsilon)$
4: $y(u) \leftarrow W/2 - \delta_0/2$ for all $u \in V$;
5: **for** Scale $i = 0$ to $\log W$ **do**
6: **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**
7: Find a maximal set $\mathscr{P}$ of vertex-disjoint augmenting paths using eligible edges;
8: $M \leftarrow M \oplus \mathscr{P}$;
9: Shrink blossoms found in step 6;
10: $z(B) \leftarrow 0$ for all blossoms B found in step 6;
11: Update the duals:
12: $y(u) \leftarrow y(u) - \delta_i/2$ for all $u \in V_{outer}$;
13: $y(u) \leftarrow y(u) + \delta_i/2$ for all $u \in V_{inner}$s;
14: $z(B) \leftarrow z(B) + \delta_i$ for all $B \in B_{outer}$ and B is a root blossom;
15: $z(B) \leftarrow z(B) - \delta_i$ for all $B \in B_{inner}$ and B is a root blossom;
16: Expand all inner root blossoms whose $z(B) = 0$;
17: **end while**
18: **if** $i < \log W$ **then**
19: $\delta_{i+1} \leftarrow \delta_i/2$;
20: $y(u) \leftarrow y(u) + \delta_{i+1}$ for all $u \in V$;
21: **end if**
22: **end for**
23: **end procedure**

Algorithm 13 2/3-Approximation Algorithm for MVM on Bipartite Graphs.

procedure 2/3-MVM-BIP($G = (S, T, E, \phi)$)
  $M \leftarrow \emptyset$; $M_S \leftarrow \emptyset$; $M_T \leftarrow \emptyset$;
  $Q \leftarrow S$;
  while $Q \neq \emptyset$ **do**
    $u \leftarrow \mathit{heaviest}(Q)$;
    $Q \leftarrow Q - u$;
    Find a shortest augmenting path P of length at most 3 starting at u;
    if P *found* **then**
      $M_S \leftarrow M_S \oplus P$;
    end if
  end while
  $Q \leftarrow T$;
  while $Q \neq \emptyset$ **do**
    $u \leftarrow \mathit{heaviest}(Q)$;
    $Q \leftarrow Q - u$;
    Find a shortest augmenting path P of length at most 3 starting at u;
    if P *found* **then**
      $M_T \leftarrow M_T \oplus P$;
    end if
  end while
  $M \leftarrow$MENDELSOHNDULMAGE(M_S, M_T, M);
end procedure

3 EXACT ALGORITHMS FOR MVM

In this chapter, we will present some important properties of the maximum vertex-weighted matching problem (MVM). First we will demonstrate necessary and sufficient conditions for an MVM to have the maximum weight. Then we will describe a lexicographical ordering property that characterizes an MVM. Next we will show how an MVM problem can be transformed to an MEM problem, and we will prove that the matchings computed by the MVM and MEM algorithms are the same, provided ties in weights are broken consistently. Additionally we will revisit an earlier algorithm that we call the Direct-Augmenting algorithm and provide a proof of correctness. Finally we will present two new exact algorithms for MVM and describe practical improvements.

3.1 Necessary and Sufficient Conditions for an Optimal MVM

Theorem 3.1.1 *Given a graph $G = (V, E, \phi)$ and a weight function $\phi : V \mapsto R_{\geq 0}$ such that the weights are all positive, a matching M is a maximum vertex-weighted matching if and only if there is neither an M-augmenting path nor an M-increasing path in G.*

Proof Let M be a maximum vertex-weighted matching in G and assume that the weights are all positive. For the sake of contradiction, assume that there exists an M-augmenting path or an M-increasing path P in G. Then, $M \oplus P$ will increase the matching weight, which contradicts M having maximum weight.

Let M be a maximum vertex-weighted matching in G such that there is neither an M-augmenting path nor an M-increasing path in G. For the sake of contradiction, assume that there exists a matching M' such that $\phi(M') > \phi(M)$. The symmetric difference $M' \oplus M$ results in $M \oplus M'$-alternating paths and even cycles. In the case of an even alternating cycle, both M' and M match the same vertices so we get $\phi(M') = \phi(M)$.

Now we have two cases if $\phi(M') > \phi(M)$:

Case 1: There exists at least one alternating path P such that $|E(M' \cap P)| = |E(M \cap P)| + 1$ as shown in Figure 3.1. Notice that the path P is an augmenting path with respect to M, which is a contradiction.

Case 2: (Let $V(M)$ denote the set of vertices that are matched in M.) There exists at least one alternating path P such that $|E(M' \cap P)| = |E(M \cap P)|$, such that $u \in V(M'), v \notin V(M')$, $u \notin V(M), v \in V(M)$, and $\phi(u) > \phi(v)$ as shown in Figure 3.2. Notice that the path P is an increasing path with respect to M, which contradicts our assumption. ■

Figure 3.1. $M \oplus M'$-alternating path P, where $|E(M' \cap P)| = |E(M \cap P)| + 1$.

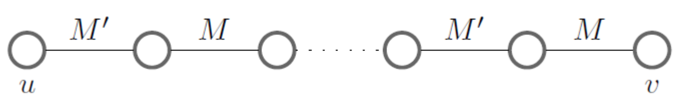

Figure 3.2. $M \oplus M'$-alternating path P, where $|E(M' \cap P)| = |E(M' \cap P)|$.

Corollary 3.1.1.1 *Given a graph $G = (V, E, \phi)$ and a weight function $\phi : V \mapsto R_{\geq 0}$, a maximum vertex-weighted matching M is also a maximum cardinality matching.*

Proof It follows from Theorem 3.1.1 that, if M is a maximum vertex-weighted matching, then there is no M-augmenting path in G. Thus, M is a maximum cardinality matching. ■

Notice that if a graph admits a perfect matching (all vertices are matched), then a maximum cardinality matching (MCM) algorithm will suffice to solve the MVM problem. However, if a graph does not admit a perfect matching, then a subset of vertices will not be matched and the MCM algorithm cannot be used, as we need to match vertices with highest weights. The following well-known theorem from Tutte states a necessary and sufficient condition for the existence of a perfect matching in a graph.

Theorem 3.1.2 ([70]) *Let $G = (V, E)$ be a graph, $X \subseteq V$ be a set of vertices, and $odd(G \setminus X)$ be the number of odd components in the subgraph induced by $V \setminus X$ (i.e., the number of components with an odd number of vertices). Then G has a perfect matching if and only if $odd(G \setminus X) \leq |X|$ for every $X \subseteq V$.*

The Gallai-Edmonds decomposition, stated in the following theorem, gives more information about the structure of an MCM.

Theorem 3.1.3 ([32]) *Let $G = (V, E)$ be a graph and let*

$D = \{v \in V$ *such that there exists an MCM in which* v *is unmatched* $\}$,

$A = \{v \in V$ *such that* $v \notin D, u \in N(v), u \in D\}$, *and*

$C = V \setminus (D \cup A)$.

Then: 1- D is the union of odd components from $G \setminus A$, and each component in D is a factor-critical subgraph. (A graph G is said to be factor critical if $G \setminus \{v\}$ has a perfect matching for each vertex v in the graph). 2- C is the union of even components from $G \setminus A$, and each component in C has a perfect matching.

If the Gallai-Edmonds decomposition of a graph G is available, then we need not consider the subgraph induced by C any further since any perfect matching of V belongs to the MVM of G.

3.2 Lexicographical Ordering

Mulmuley et al. [71] introduced the lexicographical ordering of vertex sets, an important concept in MVM. For a graph $G = (V, E, \phi)$ with weight function $\phi : V \mapsto$

$R_{\geq 0}$, let each vertex be assigned a distinct integer between 1 and $|V|$. A relationship between two vertices can be established by using both the weights and the labels associated with the vertices. A precedence operator $\succ$ can be defined as follows: given two vertices u and v, $u \succ v$ if and only if $\phi(u) < \phi(v)$, or $\phi(u) = \phi(v)$ and $l(u) < l(v)$, where $l(u)$ and $l(v)$ are distinct integer labels. The precedence relationship can be used to compare two vertex-weighted matchings. Given two matchings M and M' in a graph $G = (V, E, \phi)$, let $U = V(M)$ and $U' = V(M')$ be the set of vertices matched by M and M', respectively. Assuming that the cardinality of the two matchings are equal, if $U \succ U'$ (U is lexicographically greater than U'), then the first difference between the two sets, $u \in U$ and $v \in U'$ is such that $u \succ v$. The lexicographical order of a vertex set was used by Mulmuley et al. [71] to prove that some maximum cardinality matching is also a maximum vertex-weight matching in a graph.

Theorem 3.2.1 ([71]) *Given a graph $G = (V, E, \phi)$ and weight function $\phi : V \mapsto R_{\geq 0}$, a lexicographically largest matching of maximum cardinality is also a maximum vertex-weight matching in G.*

Proof Let M_L represent a lexicographically largest matching and M represent a maximum vertex-weight matching. Also, let M_L and M differ from each other in their sets of matched vertices. From Corollary 3.1.1.1, M is a maximum cardinality matching in G, and M_L is also a maximum cardinality matching by choice. Consider the matched vertices in M_L and M in non-increasing order of weights. Let $u \in V$ be the first vertex that is matched in M_L but not in M. The symmetric difference $M_L \oplus M$ will result in an alternating path P starting at u, matched only by M_L and ending with $v \in V$, matched only by M. Since u is the first vertex in the non-increasing order that is different, we have $\phi(u) \geq \phi(v)$. If $\phi(u) > \phi(v)$, the matching obtained by the symmetric difference $P \oplus M$ will have a weight larger than M, thereby contradicting the assumption that M is a maximum vertex-weight matching. If $\phi(u) = \phi(v)$, then by performing $M = P \oplus M$ we have brought the two matchings M_L and M closer to each other by one more edge. By continuing this process of considering the two matched sets of vertices where they differ, we either obtain a contradicton, or we can transform the matching M to the matching M_L without changing their weights. ■

3.3 Relationship between Exact MVM and MEM Algorithms

An MVM problem can be transformed to an MEM problem by transforming $G = (V, E, \phi)$ into $G' = (V, E, \phi')$ as follows: for each edge $e = (u, v) \in E$, add the vertex weights of its endpoints u and v, and assign that weight to the edge: thus $\phi'(e) = \phi(u) + \phi(v)$. The following theorem shows that exact MVM and MEM algorithms find the same matching if weights in ties are broken consistently.

Theorem 3.3.1 *Let $G = (V, E, \phi)$ be a vertex-weighted graph and $G' = (V, E, \phi')$ be an edge-weighted graph such that for all $e = (u, v) \in E$ we have $\phi'(e) = \phi(u) + \phi(v)$. Then, exact MVM and MEM algorithms find the same matching in G and G', respectively, if ties in weights are broken consistently in the two algorithms.*

Proof Let M_v be a maximum vertex-weighted matching in G and M_e be a maximum edge-weighted matching in G'. If the two matchings do not have the same weight, then first we assume for showing a contradiction that $\phi(M_v) < \phi'(M_e)$. (We will consider the case $\phi(M_v) > \phi(M_e)$ next.) Consider the symmetric difference $M_v \oplus M_e$, which results in $M_v \oplus M_e$-alternating paths and even cycles. We have four cases:
Case 1 (Figure 3.3): An $M_v \oplus M_e$-alternating path P where $|E(M_e \cap P)| = |E(M_v \cap P))|+1$. Here we find a contradiction as this creates an augmenting path with respect to M_v, but we know from Theorem 3.1.1 that if M_v is an MVM, then there is no M_v-augmenting path in G.

Case 2 (Figure 3.4): An $M_v \oplus M_e$-alternating path P where $|E(M_v \cap P)| = |E(M_e \cap P)| + 1$. Here, we have a contradiction to the assumption $\phi(M_v) < \phi'(M_e)$, since the weights of the edges matched in M_e are also included in the weights of vertices matched in M_v.

Case 3 (Figure 3.5): An even $M_v \oplus M_e$-alternating path P where $|E(M_v \cap P)| = |E(M_e \cap P)|$, assume without loss of generality that $P = \{u_1, u_2, ..., u_{k-1}, u_k\}$. The

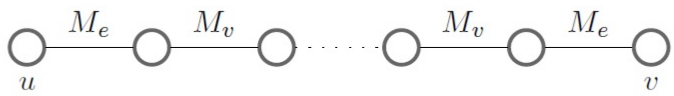

Figure 3.3. An $M_v \oplus M_e$-alternating path P, where $|E(M_e \cap P)| = |E(M_v \cap P)| + 1$.

Figure 3.4. An $M_v \oplus M_e$-alternating path P, where $|E(M_v \cap P)| = |E(M_e \cap P)| + 1$.

weight of $E(M_e \cap P)$ is $\phi(u_1) + \phi(u_2) + ... + \phi(u_{k-1})$ and the weight of $V(M_v \cap P)$ is $\phi(u_2) + \phi(u_3) + ... + \phi(u_k)$. Notice $V(M_v \cap P)$ and $E(M_e \cap P)$ have the same weights except u_1 and u_k, and because by assumption $\phi'(M_e) > \phi(M_v)$, $\phi(u_1) > \phi(u_k)$. However, the weight of M_v can be increased by flipping the matching edges along P, which contradicts the fact that M_v is an MVM.

Case 4: An even $M_v \oplus M_e$-alternating cycle C. Since all vertices are matched in the cycle C, $\phi'(E(M_e \cap C))$ must equal $\phi(V(M_v \cap C))$. In addition, because ties are broken consistently, $E(M_e \cap C)$ must equal $E(M_v \cap C)$.

Hence,$\phi(M_v) = \phi'(M_e)$ and both matchings are the same.

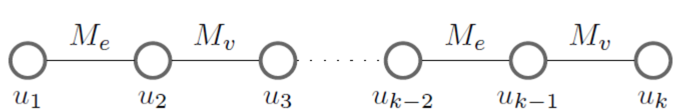

Figure 3.5. An $M_v \oplus M_e$-alternating path, where $|M_e(P)| = |M_v(P)|$.

Now we consider the case that $\phi'(M_e) < \phi(M_v)$ to obtain a contradiction.

Let $yz(u) + yz(v) = y(u) + y(v) + \sum_{(u,v)\in E(B)} z(B)$, where B is a blossom, be the dual variables in the linear programming formulation of the maximum edge-weighted matching. The following complimentary slackness conditions involving the weights and dual variables must hold:

1. $yz(u) + yz(v) \geq \phi(u) + \phi(v)\ \forall uv \in E$. (domination property)

2. If uv is an edge in the matching M_e, then $yz(u)+yz(v) = \phi(u)+\phi(v)$. (tightness property)

3. If u is an unmatched vertex in M_e, then $yz(u) = 0$.

Take the symmetric difference $M_v \oplus M_e$, which results in $M_v \oplus M_e$-alternating paths and even cycles. We have four cases:

Case 1: $M_v \oplus M_e$-alternating path where $|E(M_v \cap P)| = |E(M_e \cap P)| + 1$. Let $P = \{u_1, u_2, ...u_k\}$, where M_e matches vertices $u_2, u_3, ...u_{k-1}$. We know that matching edges must be tight, non-matching edges must be dominated, and the dual variables of unmatched vertices should be zero. Here, by domination we have

$$yz(u_1) + yz(u_2) + ... + yz(u_k) \geq \phi(u_1) + \phi(u_2) + ... + \phi(u_k), \tag{3.1}$$

and by tightness, we have

$$yz(u_2) + ... + yz(u_{k-1}) = \phi(u_2) + ... + \phi(u_{k-1}). \tag{3.2}$$

Using the value of $yz(u_2) + ... + yz(u_{k-1})$ in (3.2), the inequality (3.1) becomes

$$yz(u_1) + yz(u_k) + \phi(u_2) + ... + \phi(u_{k-1}) \geq \phi(u_1) + \phi(u_2) + ... + \phi(u_k)$$

$$\Longrightarrow$$

$$y(u_1) + y(u_k) \geq \phi(u_1) + \phi(u_k).$$

Since u_1 and u_k are unmatched vertices in M_e, their dual variables are equal to 0, which implies $\phi(u_1)+\phi(u_k) = 0$. Now, all non-matching edges in P with respect to M_e

are tight and M_e should have been augmented with the path P. Thus, $\phi'(E(M_e \cap P))$ equals $\phi(V(M_v \cap P))$ and $E(M_e \cap P)$ is equal to $E(M_v \cap P)$.
Case 2: An $M_v \oplus M_e$-alternating path where $|E(M_e \cap P)| = |E(M_v \cap P)| + 1$. This leads to a contradiction, since M_v is not maximum because there exists an augmenting path with respect to M_v.
Case 3: An even $M_v \oplus M_e$-alternating path where $|E(M_v \cap P)| = |E(M_e \cap P)|$. The weight of $E(M_e \cap P)$ is $\phi(u_1) + \phi(u_2) + ... + \phi(u_{k-1})$, and the weight of $V(M_v \cap P)$ is $\phi(u_2) + \phi(u_3) + ... + \phi(u_k)$. Because M_v is maximum we know that $\phi(u_k) > \phi(u_1)$. By domination, we have

$$yz(u_2) + ... + yz(u_k) \geq \phi(u_2) + ... + \phi(u_k), \tag{3.3}$$

and by tightness, we have

$$yz(u_1) + ... + yz(u_{k-1}) = \phi(u_1) + ... + \phi(u_{k-1}). \tag{3.4}$$

Using the value of $yz(u_2) + ... + yz(u_{k-1})$ in (3.4), we get

$$yz(u_k) - yz(u_1) + \phi(u_1) + ... + \phi(u_{k-1}) \geq \phi(u_2) + ... + \phi(u_k)$$
$$\Longrightarrow$$
$$yz(u_k) - yz(u_1) \geq \phi(u_k) - \phi(u_1).$$

Since u_k is unmatched, its dual variable is equal to 0, thereby implying $\phi(u_1) \geq y(u_1) + \phi(u_k)$, which contradicts $\phi(u_k) > \phi(u_1)$.
Case 4: An even $M_v \oplus M_e$-alternating cycle C. Since all vertices are matched in the cycle C, $\phi'(E(M_e \cap P))$ must equal $\phi(V(M_v \cap C))$. In addition, because ties are broken consistently, $E(M_e \cap C)$ must equal $E(M_v \cap C)$.
Thus,$\phi(M_v) = \phi'(M_e)$ and both matchings are the same.

This completes the proof. ■

3.4 Direct-Augmenting Algorithm for MVM

Dobrian et al. [22] and Halappanavar [23] proposed an exact algorithm that we call the Direct-Augmenting algorithm, since each vertex is considered for matching using an augmenting path only once. The new direct approximation algorithms are derived from this algorithm in a natural manner by restricting the augmenting path length.

The Direct-Augmenting algorithm is listed in Algorithm 14. This algorithm sorts vertices in non-increasing order of weights, and in each iteration it attempts to match a heaviest unmatched vertex u. From u the algorithm searches for a heaviest unmatched vertex v it can reach by an augmenting path P. If it finds P, then the matching is augmented by forming the symmetric difference of the current matching M with P, and the vertices u and v are removed from the set of unmatched vertices. If it fails to find an augmenting path from u, then u is removed from the set of unmatched vertices, since we do not need to search for an augmenting path from u again. When all the unmatched vertices have been processed, the algorithm terminates.

Algorithm 14 The Direct-Augmenting Exact Algorithm for MVM.

procedure DIRECT-AUGMENTING($G = (V, E, \phi)$)
    $M \leftarrow \phi$;
    $Q \leftarrow V$;
    while $Q \neq \emptyset$ **do**
        $u \leftarrow heaviest(Q)$;
        $Q \leftarrow Q - u$;
        Find an augmenting path P from u that reaches
                a heaviest unmatched vertex v;
        if P *found* **then**
            $M \leftarrow M \oplus P$;
            $Q \leftarrow Q - v$;
        end if
    end while
end procedure

The proof of correctness is taken from [22], and it is stated here for completeness.

Lemma 3.4.1 ([22]) *Let z be an unmatched vertex with respect to a matching M in a graph $G = (V, E, \phi)$. Suppose that there does not exist an M-augmenting path from the vertex z and that there is no M-increasing path (from any vertex) in the graph G. Let P be an M-augmenting path from a heaviest unmatched vertex u, whose other endpoint v is a heaviest unmatched vertex that can be reached from u by an M-alternating path. If $M' = M \oplus P$, then there does not exist an M'-augmenting path from the vertex z, nor an M'-increasing path (from any vertex) in the graph G.*

Proof When P is an augmenting path from some M-unmatched vertex u, u has to be distinct from the vertex z, as from the latter, there is no augmenting path by the condition of the Lemma. A proof that there is no M'-augmenting path from z can be found in [33]. Hence we prove that there is no M'-increasing path in G.

If there is no M'-reversing path in G, then there cannot be any M'-increasing path, and we are done. Thus, choose an arbitrary M'-reversing path P' that joins an M'-unmatched vertex w and an M'-matched vertex w'. Since every vertex on the M-augmenting path P is matched in M', the vertex w cannot belong to P, while the vertex w' can belong to P and does not need to be distinct from the vertices u or v. We will prove that $\phi(w) \leq \phi(w')$ and hence that the path P' is not M'-increasing.

If an M-reversing path also joins the vertices w and w', where w is M-unmatched and w' is M-matched, then since there is no M-increasing path in G, we have $\phi(w) \leq \phi(w')$. If no M-reversing path joins w and w', then the paths P' and P cannot be vertex-disjoint; for if they were, then P' would also be an M-reversing path, which we assumed does not exist in G. Thus the paths P and P' share at least one common vertex, and indeed, as we show now, it shares a matching edge. For, every vertex on the path P is M'-matched, and hence a vertex in x in $P' \setminus P$ that is adjacent to a vertex y in P must have the edge (x, y) as a non-matching edge in M'. Since P' is an M'-alternating path, the next edge on the path P' must be a matching edge incident on the vertex y, and hence this matching edge is common to both paths P' and P. (The paths P and P' could intersect more than once.)

Now we have two cases to consider.

The cases are illustrated in Figure 3.6 and 3.7. In the first case, there is an M-augmenting path between u and w, and there are two subcases: either v and w' are the same vertex, or there is an M-reversing path Q between v and w'. The second subcase corresponds to Figure 3.6. Now the path Q cannot be an M-increasing path by our assumption that no such path exists in G. Hence in both subcases, we can write $\phi(v) \leq \phi(w')$. Since we chose the path P to begin at u and end at the M-unmatched vertex v and not at the M-unmatched vertex w, we have $\phi(w) \leq \phi(v)$. Combining the two inequalities, we obtain $\phi(w) \leq \phi(w')$.

In the second case, there is an M-augmenting path between v and w, and again there are two subcases: either u and w' are the same vertex, or there is an M-reversing path Q' between u and w'. The second subcase is illustrated in Figure 3.7. As before, the path Q' cannot be M-increasing by supposition, and therefore $\phi(u) \leq \phi(w')$. Since u is a heaviest M-unmatched vertex by choice, and w is M-unmatched, we have $\phi(w) \leq \phi(u)$. Combining, we have $\phi(w) \leq \phi(w')$. ■

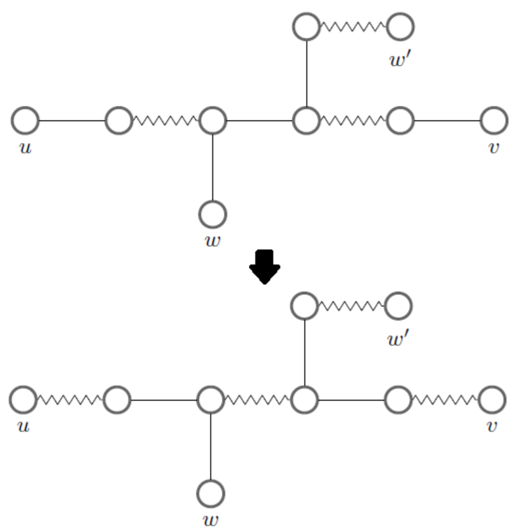

Figure 3.6. Construction used in the proof of Lemma 3.4.1.

Theorem 3.4.2 ([22]) *The Direct-Augmenting algorithm computes an MVM in a graph $G = (V, E, \phi)$.*

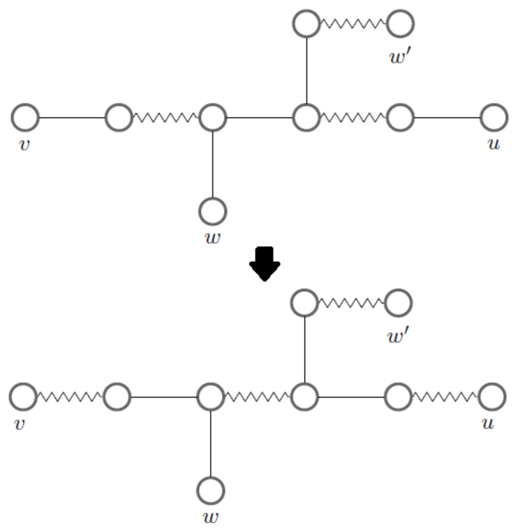

Figure 3.7. Construction used in the proof of Lemma 3.4.1.

Proof Let M be the matching computed by the Direct-Augmenting algorithm. We show by induction that there does not exist an M-augmenting path nor an M-increasing path in the graph G.

Let n_a be the number of augmenting operations in the Direct-Augmenting algorithm. The matching M is the last in a sequence of matchings M_i, for $i = 0, 1, \ldots,$ n_a, computed by the algorithm. For $0 \leq i < n_a$, let P_i denote the M_i-augmenting path used to augment M_i to the matching M_{i+1}, and let u_i denote the source of the augmenting path (the M_i-unmatched vertex from which we searched for an augmenting path), and let v_i denote its other end point. The induction is on the matching M_i, and the inductive claim is that

(1) there is no M_i-augmenting path from an unmatched vertex that has already been processed, i.e., a vertex from which we have searched for an augmenting path earlier and have failed to find one, and

(2) there is no M_i-increasing path from any vertex in G.

The basis of the induction is $i = 0$, when the result is trivially true. The first condition holds because no vertices have been processed yet, and the second condition holds since the matching is empty and hence there is no increasing path. Hence assume that the claim is true for some i, with $0 \leq i < n_a$. Now the result holds for the step $i+1$ by applying Lemma 3.4.1. ■

The time complexity of this algorithm is $O(nm + n \log n)$. The algorithm tries to match each vertex, and the search for augmenting paths from each vertex costs $O(m)$ time. The second term is the cost of sorting the vertex weights, when they are real-valued. If the weights are integers in a range $[0\ K]$, then a counting sort could be used with time complexity $O(n + K)$.

3.5 New Exact Algorithms for MVM

3.5.1 Direct-Increasing Exact Algorithm

In this section we describe a new exact algorithm for computing MVM. The new algorithm is based on sorting the vertices in non-increasing order of weights, then computing a maximum cardinality matching, and finally finding all increasing paths with respect to the former matching. It is called Direct-Increasing since each vertex is processed once, and the algorithm uses increasing paths to increase the weight of a maximum cardinality matching. The new approach consists of two phases. The first phase finds a maximum cardinality matching and the second phase finds increasing paths with the highest gain. The first phase of the exact algorithm, as shown in Algorithm 15, starts with an empty matching, then sorts the vertices in non-increasing order of weights and inserts the vertices in a queue Q. In each iteration the algorithm attempts to match a heaviest unmatched vertex u. From u the algorithm searches for an unmatched vertex v that it can reach via an augmenting path P. If it finds P, then the matching is augmented by forming the symmetric difference of the current matching M with P, and the vertices u and v are removed from the set of unmatched vertices. If it fails to find an augmenting path from u, then u is removed from the set of unmatched vertices and inserted into a queue Q', as we need to search for an increasing path from u in the next phase. When all the unmatched vertices have been processed, the algorithm proceeds to the next phase.

In the second phase Q' is the set of unmatched vertices from phase one, and in each iteration, the algorithm attempts to match a heaviest unmatched vertex u. From u, the algorithm searches for a lightest matched vertex v it can reach by an increasing path P such that $\phi(u) > \phi(v)$. If it finds P, then the matching is reversed by forming the symmetric difference of the current matching M with P, and the vertex u is removed from the set of unmatched vertices Q'. If it fails to find an increasing path from u, then u is removed from the set of unmatched vertices. When all the unmatched vertices in Q' have been processed, the algorithm terminates.

Theorem 3.5.1 *The time complexity of the Direct-Increasing algorithm is $O(mn + n \log n)$.*

Proof First, sorting vertices takes $O(n \log n)$ time. The maximum cardinality matching takes $O(m\sqrt{n})$ time. In the second phase, each unmatched vertex takes at most $O(m)$ time to search for an increasing path. Since we have $O(n)$ vertices, the second phase takes $O(mn)$ time. Thus the time complexity of the Direct-Increasing algorithm is $O(mn + n \log n)$. ■

Proof of Correctness

Theorem 3.5.2 *The Direct-Increasing algorithm computes an MVM in a graph $G = (V, E, \phi)$.*

Proof Let M be a matching obtained by the Direct-Increasing algorithm. We know that there does not exist an augmenting path in G with respect to M since M is a maximum cardinality matching and reversing an increasing path results in the same cardinality. Therefore it suffices to show that there do not exist increasing paths after the algorithm terminates. Suppose that after the algorithm terminates there exists an increasing path from some vertex u. There are two cases in which u is unmatched

Algorithm 15 The Direct-Increasing Exact Algorithm for MVM.

```
procedure Direct-Increasing(G = (V, E, φ))
    M ← φ;
    Q ← V;
    while Q ≠ ∅ do
        u ← heaviest(Q);
        Q ← Q − u;
        Find an augmenting path P from u to an unmatched vertex v;
        if P found then
            M ← M ⊕ P;
            Q ← Q − v;
        else
            Q′ ← Q′ ∪ u;
        end if
    end while
    while Q′ ≠ ∅ do
        u ← heaviest(Q′);
        Q′ ← Q′ − u;
        Find an increasing path P from u reaching a lightest matched vertex v;
        if P found then
            M ← M ⊕ P;
        end if
    end while
end procedure
```

in M:

Case 1: u is matched in phase one and then unmatched in phase two because it is a lightest vertex reachable by an increasing path P from some vertex v. In this case, P was reversed, which unmatched u.

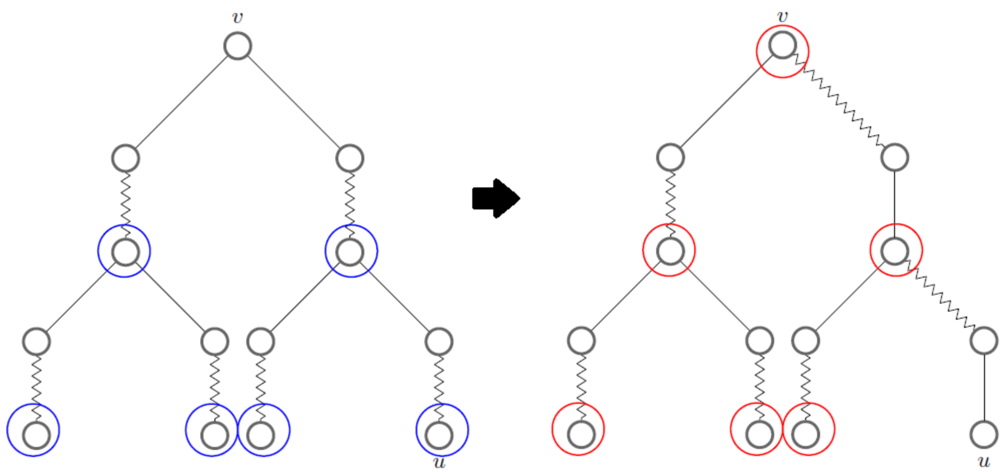

Figure 3.8. Before updating the matching, all outer vertices from v are circled. After the matching is updated the same circled vertices are outer vertices from u in addition to v.

As shown in Figure 3.8, before matching v and reversing P, all reachable outer vertices from v become reachable outer vertices from u after reversing P. The vertex v is also in this set of outer vertices. Since u is the lightest outer vertex reachable from v, all outer vertices reachable from u are at least as heavy as u. Thus there does not exist an increasing path from u.

Case 2: The algorithm failed to find an increasing path from u.

Suppose, during future steps, an increasing path is found and reversed; then an increasing path is made from u.

Notice that the Direct-Increasing algorithm considers unmatched vertices in non-increasing order of weights. Let $P_1 = \{u_1, ..., u_x\}$ be an alternating path visited by u that fails to find an increasing path. Let $P_2 = \{v_1, ..., v_y\}$ be an increasing path that is found from v and $P_1 \cap P_2 = \{u_i, ..., u_j\}$ where $\phi(u) > \phi(v)$. The case is illustrated in Figure 3.5.1. The lightest vertex v_y should have been reached by u when it was considered by the path $\{u_1....u_i, ..., u_j, ..., v_y\}$.

Thus an increasing path does not exist in G with respect to M, and M is a maximum vertex-weighted matching. ■

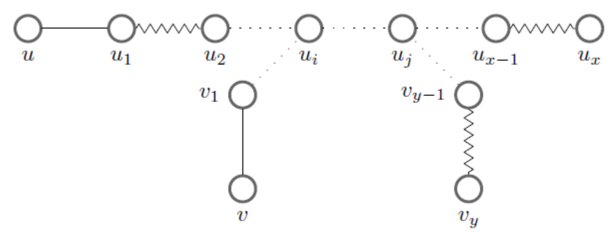

Figure 3.9. An increasing path from u to u_x and from v to v_y. This existence of overlapped alternating path from u_i to u_j implies there is an increasing path from u to v_y.

3.5.2 Iterative Exact Algorithm

In this section we describe another exact algorithm for computing MVM. The new algorithm is based on the iterative approach where a vertex may be processed several times. The algorithm consists of two phases: in the first phase, a maximum cardinality matching is computed, and, in the second phase, highest gain increasing paths are sought. Phase two is repeated until no increasing path is found. In the first phase of the exact algorithm, as shown in Algorithm 16, the algorithm processes unmatched vertices in an arbitrary order, and in each iteration the algorithm attempts to match an unmatched vertex u. From u, the algorithm searches for an unmatched vertex v that it can reach by an augmenting path P. If it finds P, then the matching is augmented by forming the symmetric difference of the current matching M with P. When all the unmatched vertices have been processed, the algorithm proceeds to the next phase.

In the second phase the algorithm searches for increasing paths. The algorithm processes unmatched vertices in an arbitrary order, and in each iteration it attempts to match an unmatched vertex u. From u, the algorithm searches for a lightest matched vertex v that it can reach by an increasing path P such that $\phi(u) > \phi(v)$. If it finds P, then the matching is reversed by forming the symmetric difference of the current matching M with P, and now v becomes unmatched. The algorithm terminates when it fails to find an increasing path during one pass over the unmatched vertices.

Algorithm 16 The Iterative Exact Algorithm for MVM.

procedure ITER($G = (V, E, \phi)$)
    $M \leftarrow \phi$;
    for each unmatched vertex u **do**
        Find an augmenting path P from u;
        if P *found* **then**
            $M \leftarrow M \oplus P$;
        end if
    end for
    do
        done = true;
        for each unmatched vertex u **do**
            Search for an increasing path P from u reaching a lightest matched vertex v;
            if P *found* **then**
                $M \leftarrow M \oplus P$;
                done =false;
            end if
        end for
    while not done
end procedure

Theorem 3.5.3 *The time complexity of the Iterative algorithm is $O(\Delta mn)$.*

Proof The maximum cardinality matching takes $O(m\sqrt{n})$ time. In the second phase, each unmatched vertex takes at most $O(m)$ time to search for an increasing path. Since we have $O(n)$ vertices, the second phase takes $O(mn)$. Note that a vertex can be unmatched at most $O(\Delta)$ times. Thus, the time complexity of the Iterative algorithm is $O(\Delta mn)$. ■

Proof of Correctness

Theorem 3.5.4 *The Iterative algorithm computes an MVM in a graph $G = (V, E, \phi)$.*

Proof Let M be a matching obtained by Algorithm 16. We know that there does not exist an augmenting path in G with respect to M because M is a maximum cardinality matching and reversing increasing paths results in the same cardinality. Therefore, it suffices to show that there do not exist increasing paths after the algorithm terminates. Since the algorithm keeps repeating phase two until no increasing paths can be found, there does not exist an increasing path when the algorithm terminates. Thus M is a maximum vertex-weighted matching. ■

3.6 Practical Improvements to Direct-Increasing Exact Algorithm

The Direct-Increasing exact algorithm can be further improved in practice. First, we want to avoid repeating searching along paths that do not lead to increasing paths. Notice that the algorithm processes vertices in non-increasing order of their weights, so if a vertex v_i fails to find an increasing path, the next vertex v_j using the same paths will also fail since all outer vertices (at an even distance) from v_i are at least as heavy as v_i, and this is true for v_j. To prevent repeating failed searches, we track all visited outer vertices from the source vertex. If the search fails we label each tracked vertex. If a search encounters a labeled vertex, the search along that vertex halts.

Another method to accelerate the algorithm is as follows: since we know the order of vertices in advance, we can check the lightest unmatched vertex. Now during the search if the lightest unmatched vertex is encountered, then the algorithm can quit further searching and return the increasing path.

A similar technique can be used to improve the performance of the Direct-Augmenting algorithm. This time we want to avoid repeating searching along paths that do not lead to augmenting paths. Again, the algorithm processes vertices in non-increasing order of their weights, so if a vertex v_i fails to find an augmenting path, the next vertex v_j using the same paths will also fail. We can track all visited outer vertices from the source vertex and if the search failed we label each tracked vertex. If any search encounters a labeled vertex, the search along that vertex halts. Azad et al. [72] prove that there will not exist an augmenting path from such vertices in future steps of the algorithm. Also, because we know the order of vertices in advance, we can check the second heaviest unmatched vertex. During the search, if an unmatched vertex (having the second heaviest weight) is encountered, then the algorithm ceases further searching and returns an augmenting path.

4 APPROXIMATION ALGORITHMS FOR MVM

Approximation algorithms were originally studied to address intractable problems, but the demand for fast approximation algorithms for tractable problems has arisen out of the following reasons:

- Some applications require a problem be solved multiple times as fast as possible (e.g., multilevel graph partitioning [73, 74] and real time switch scheduling [4]). While there are polynomial time algorithms for these problems, they fail to compute a solution within a reasonable time.
- Some applications require processing huge amounts of data (e.g., sparse matrix computations [12–14, 75, 76] in computational science and engineering applications).
- Many applications do not require an exact solution.
- Parallelizing exact algorithms is often not possible due to the sophistication of the algorithms, and their inherent lack of concurrency.

In this chapter we will prove a new theorem that states sufficient conditions for obtaining a $k/(k+1)$-approximate matching for the MVM problem. Similar results are known for cardinality matching [43] and edge-weighted matching [60], so our result extends it to the vertex-weighted context. Next we propose new approximation algorithms based on direct and iterative techniques.

The direct technique begins with an empty matching, and at each step, matches a currently heaviest unmatched vertex to a heaviest unmatched vertex that it can reach by a short augmenting path. In direct algorithms, each vertex is processed once (hence the appellation direct); once a vertex is matched, it will always remain matched, as augmentation does not change a matched vertex to an unmatched vertex in the MVM problem. Two approximation algorithms based on the direct method with approximation ratios of 1/2 and 2/3 are proposed.

The iterative technique begins with either an empty or initial matching. It then looks for increasing paths or augmenting paths (all paths are restricted in length) with respect to the current matching, and terminates when there is none; this algorithm may need to process a vertex multiple times (for this reason, it is called "iterative"). We will describe new approximation algorithms based on the iterative method that achieve 1/2, 2/3 and $k/(k+1)$-approximation ratios for the MVM problem.

4.1 Sufficient Conditions for $k/(k+1)$-approximation for MVM

Theorem 4.1.1 *Let $G = (V, E)$ be a graph and $\phi : V \mapsto R_{\geq 0}$ a weight function. If there does not exist an augmenting path of length $2k-1$ and an increasing path of length $2k$ with respect to a matching M, then M is $k/(k+1)$-approximation to the optimal MVM.*

To prove the Theorem we need the following Lemma.

Lemma 4.1.2 *Let M_{opt} be an optimal matching and M_A be an approximate matching in a graph $G = (V, E, \phi)$ such that there does not exist an augmenting path of length $2k-1$ and an increasing path of length $2k$ with respect to M_A. For every M-unmatched vertex u that is matched in M_{opt}, there are k distinct M-matched vertices that are at least as heavy as u.*

Proof Let U be the set of vertices that are matched in M_{opt} but unmatched in M_A. Consider the symmetric difference of the two matchings $M_{opt} \oplus M_A$, which consists of vertices with degrees 0, 1 or 2. Hence this subgraph is a union of alternating paths and alternating cycles. In each alternating cycle all vertices are matched in both M_{opt} and M_A; hence we need consider only alternating paths. Each vertex $u \in U$ has degree one in the subgraph, and therefore must be an end point of an alternating path. Since there does not exist an augmenting path of length $2k-1$, the length of each alternating path is at least $2k$. Since there are no weight-increasing paths of length $2k$, the k vertices at even-valued distances from u on the symmetric difference path must be at least as heavy as u. If both endpoints of an alternating path belong to U, then it has length at least $2k + 1$ as there are no augmenting paths of length $2k - 1$. Then each endpoint can choose k distinct vertices which are at even-valued distances from it. Note that the symmetric difference paths are vertex disjoint, and, thus, for every unmatched vertex u, we have found k distinct matched vertices in M_A that are at least as heavy as u. ■

Now we prove Theorem 4.1.1.

Proof Let M_A, M_{opt} and U all be as defined in the Lemma, and consider paths in the symmetric difference between M_A and M_{opt}. Each vertex $u \in U$ must be an endpoint of an alternating path in the symmetric difference. Thus

$$\begin{aligned}\phi(M_{opt}) &= \phi(M_A) + \phi(U) - \phi(M_A \setminus M_{opt}) \\ &\leq \phi(M_A) + \phi(U).\end{aligned}$$

By Lemma 4.1.2, we have $\phi(U) \leq \frac{1}{k}\phi(M_A)$. Hence,

$$\begin{aligned}\phi(M_{opt}) - \phi(M_A) &\leq \frac{1}{k}\phi(M_A) \\ \Rightarrow \phi(M_A) &\geq \frac{k}{k+1}\phi(M_{opt}).\end{aligned}$$

■

4.2 New Approximation Algorithms Based on the Direct Approach

In this section we present new 1/2- and 2/3-approximation algorithms which are derived from the Direct-Augmenting algorithm (described in the previous chapter) by restricting the augmenting path length to one and three, respectively. The approximation algorithms are called 1/2-Dir and 2/3-Dir for short. Additionally, we will show that if vertex weights in a graph are transformed into edge weights, then a matching obtained by the 1/2-Dir algorithm is identical to the matching obtained by the 1/2-approximation algorithms for MEM that match locally dominant edges, given that ties in weights are broken consistently in the two algorithms.

In the exact Direct-Augmenting 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 such an augmenting path is found then augment the matching using this path. If no augmenting path is found, we search from the next heaviest unmatched vertex. Consider running the exact Direct-Augmenting algorithm and the 1/2-Dir or 2/3-Dir 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 one or three), then we call u a *failure* or a failed vertex, because the approximation algorithm failed to match it, while the exact algorithm succeeded.

In this section we distinguish between the origin (the first vertex) and the terminus (the last vertex) of augmenting paths. The origin and terminus of an augmenting path are *corresponding vertices* of each other, and this pairing is uniquely determined in our algorithms.

4.2.1 A 1/2-Approximation Algorithm

The approximation algorithm sorts the vertices in non-increasing order of weights, and inserts the sorted vertices into a queue Q. The algorithm begins with an empty matching and attempts to match the vertices in Q in the given order. Each unmatched vertex u is removed from Q, and the algorithm searches for a heaviest unmatched neighbor v. If such neighbor exists, then we match the edge (u, v) and remove the vertex v from Q. If no unmatched neighbor is found, we search from the next heaviest unmatched vertex. The algorithm terminates when all vertices are processed.

Algorithm 17 The Direct 1/2-Approximation Algorithm for MVM.

```
procedure 1/2-Dir(G = (V, E, φ))
    M ← ∅;
    Q ← V;
    while Q ≠ ∅ do
        u ← heaviest(Q);
        Q ← Q − u;
        Let v denote a heaviest unmatched neighbor of u
        if v is found then
            M ← M ∪ (u, v);
            Q ← Q − v;
        end if
    end while
end procedure
```

We will now illustrate the relation between the 1/2-Dir algorithm and the 1/2-approximation algorithm for edge-weighted matching (1/2-MEM) which matches locally dominant edges (An edge (u, v) is said to be a locally dominant if it is at least as heavy as all other edges incident on the vertices u and v). Recall that an MVM problem can be transformed into an MEM problem by transforming $G = (V, E, \phi)$ into $G' = (V, E, \phi')$: for each edge $e = (u, v) \in E$, sum the vertex weights of adjacent vertices u and v, and assign that weight to the edge; hence $\phi'(e) = \phi(u) + \phi(v)$.

Theorem 4.2.1 *Let $G = (V, E, \phi)$ be a vertex-weighted graph, and $G' = (V, E, \phi')$ be an edge-weighted graph such that for all $e = (u, v) \in E$, we set $\phi'(e) = \phi(u) + \phi(v)$. The matching obtained by the 1/2-Dir algorithm on G is identical to the matching obtained by 1/2-MEM algorithm that matches locally dominant edges, given that ties in weights are broken consistently.*

Proof We run both algorithms step by step; the algorithms might match different edges at a step, but we continue running the algorithms until we find two different edges incident on a common vertex v matched by the algorithms (possibly at different steps). Let (u, v) be matched by the 1/2-Dir algorithm and (w, v) be matched by the 1/2-MEM algorithm. We know that the 1/2-Dir algorithm processes vertices in non-increasing order of weights and the 1/2-MEM algorithm matches edges that are locally dominant. We consider two cases, the first of which is $\phi(u) \geq \phi(v)$, then we have two subcases:

Subcase 1: $\phi(u) \geq \phi(w)$, so (w, v) is not locally dominant.

Subcase 2: $\phi(u) < \phi(w)$. Then, w must be matched by 1/2-Dir to, say, q, since 1/2-Dir processes vertices in non-increasing order of weights. Clearly, $\phi(q) \geq \phi(v)$, so (w, v) is not locally dominant.

Now consider the second case $\phi(v) > \phi(u)$. Here again we have two subcases:

Case 1: $\phi(u) \geq \phi(w)$; then (w, v) is not locally dominant.

Case 2: $\phi(u) < \phi(w)$. Then w must be matched to, say, q by the 1/2-Dir algorithm, as this algorithm processes vertices in non-increasing order of weights. Clearly, $\phi(q) \geq$

$\phi(v)$, so (w, v) is not locally dominant.

Since all cases where the two algorithms pick different edges incident on a common vertex contradicts the choice of the locally dominant edge matched by 1/2-MEM, the matching of 1/2-Dir and 1/2-MEM must be the same, given that ties are broken consistently. ■

Time Complexity

Theorem 4.2.2 *The time complexity of the* 1/2*-Dir approximation algorithm is* $O(m + n \log n)$*.*

Proof 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, which can be done in $O(d(u))$ time. Thus the search for a heaviest unmatched neighbor in the algorithm takes $O(m)$ time. Sorting the vertices in non-increasing order of weights takes $O(n \log n)$ time. ■

Proof of Correctness

In this subsection, we will prove the approximation ratio of the 1/2-Dir approximation algorithm. Let M_A be the approximate matching found by the 1/2-Dir algorithm. We will prove first the non-existence of increasing paths of length two, which is achieved by using Lemma 4.2.3.

Lemma 4.2.3 *Let* u *be a vertex matched in the optimal matching that failed to be matched by the approximation algorithm* M_A*, and let* $P = \{u, v_1, v_2, ...\}$ *be an* $M_{opt} \oplus M_A$*-alternating path that begins with* u*. Then* $\phi(v_2) \geq \phi(u)$*.*

Proof In this case, we must consider two possibilities:

(a) The vertex v_2 is an origin, in which case $\phi(v_2) \geq \phi(u)$, since v_2 was processed before u by the approximation algorithm.

(b) The vertex v_2 is a terminus and v_1 is an origin. In this case, v_2 was matched in preference to u, so $\phi(v_2) \geq \phi(u)$.

Note that if v_2 is a terminus and v_1 is not the corresponding origin, then the alternating patj $P = \{u, v_1, v_2, ...\}$ cannot exist; and v_2 is not at distance two from u because v_1 cannot be matched to v_2 in the approximate matching. ∎

Lemma 4.2.4 *After the 1/2-Dir algorithm terminates, there does not exist an augmenting path of length one and an increasing path of length two with respect to M_A.*

Proof Clearly there does not exist an augmenting path of length one, since the algorithm scans all neighbors, and if there is an unmatched neighbor it will always select one with the highest weight. For the increasing path of length two, by Lemma 4.2.3, we know all matched vertices at distance two from any unmatched vertex u are at least as heavy as u. Thus, an increasing path of length two does not exist with respect to M_A. ∎

Theorem 4.2.5 *The 1/2-Dir algorithm has the approximation ratio of* $1/2$.

Proof By Lemma 4.2.4, we know that there does not exist an augmenting path of length one and increasing path of length two. Hence, it follows from Theorem 4.1.1 that M_A is at least $\frac{1}{2}M_{opt}$. ∎

4.2.2 A 2/3-Approximation Algorithm

The approximation algorithm described in Algorithm 18 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 such an augmenting path is found, then the matching is augmented by the path that leads to a heaviest unmatched vertex, 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.

Algorithm 18 The Direct 2/3-Approximation Algorithm for MVM.

procedure 2/3-DIR($G = (V, E, \phi)$)
    $M \leftarrow \emptyset$;
    $Q \leftarrow V$;
    while $Q \neq \emptyset$ **do**
        $u \leftarrow heaviest(Q)$;
        $Q \leftarrow Q - u$;
        Let v denote a heaviest unmatched vertex reachable from u by an augmenting path P of length at most three;
        if P is found **then**
            $M \leftarrow M \oplus P$; $Q \leftarrow Q - v$;
        end if
    end while
end procedure

Time Complexity

Theorem 4.2.6 *The time complexity of the 2/3-Dir approximation algorithm is* $O(m \log \Delta + n \log n)$*, where* Δ *is the maximum degree.*

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. 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. ■

Proof of Correctness

In this subsection, we will prove (Theorem 4.2.12) that Algorithm 18 computes a 2/3-approximate MVM, M_A. Note that Theorem 4.1.1 cannot be used since the 2/3-Dir algorithm does not guarantee the non-existence of an increasing path of length four. Consider the following case illustrated in Figure 4.1, first the algorithm match v_2 with v_5 and v_3 with v_6. Then two augmenting paths of length three are found and augmented $\{v_7, v_5, v_2, v_1\}$ and $\{v_8, v_6, v_3, v_4\}$. Searches for an augmenting path from u_1 and u_2 will fail. As shown in Figure 4.2, there are two increasing paths of length four, in this case $\{u_1, v_1, v_2, v_3, v_4\}$ and $\{u_2, v_4, v_3, v_2, v_1\}$.

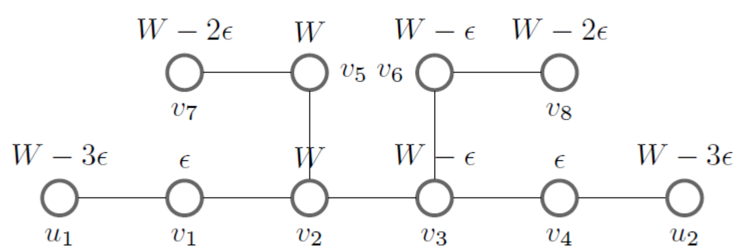

Figure 4.1. A case that leads to an increasing path of length four after the 2/3-Dir algorithm terminates. W and ϵ are real values, where $W \gg \epsilon$.

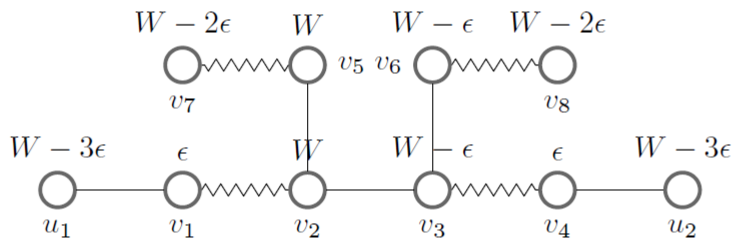

Figure 4.2. After the 2/3-Dir terminates, we have two increasing paths $\{u_1, v_1, v_2, v_3, v_4\}$ and $\{u_2, v_4, v_3, v_2, v_1\}$.

The proof of the approximation ratio for the 2/3-Dir algorithm requires several new concepts. First, we need a more careful study of the structure of augmenting paths. In addition to the concepts of the origin and terminus of an augmenting path, 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. A *heaviest unmatched neighbor* of u is denoted by HUN(u). Note that HUN(u) might not be unique, but the weight of HUN(u) is unique.

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 approximation ratio, 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 the weight at least as heavy as the failure. This is achieved in Lemma 4.2.11 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 4.2.11 is proved using Lemmas 4.2.7, 4.2.8, 4.2.9 and 4.2.10. We begin by proving each of the latter Lemmas.

Lemma 4.2.7 *Let (u, v) be a matching edge in a matching M at some step in the 2/3-Dir approximation algorithm, and let $w = \mathrm{HUN}(v)$ 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 $w' = \mathrm{HUN}(v')$ 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 $\mathrm{HUN}(v'_i)$. 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 Figure 4.3. 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 $\phi(w) \geq \phi(w'_{k+1})$. ■

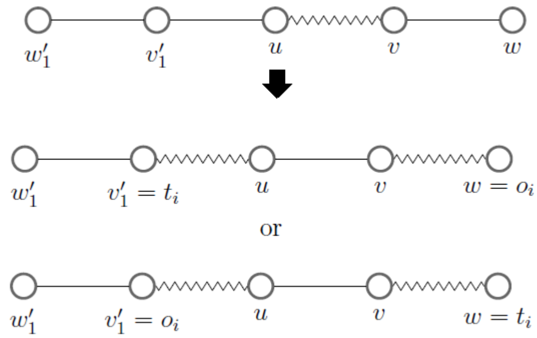

Figure 4.3. Lemma 4.2.7: Base case.

Lemma 4.2.8 *Let M_A^x denote the 2/3-Dir approximate matching at the x^{th} failure f_x, and let $P = \{f_x, v_1, v_2, ...\}$ 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 Figure 4.4. 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 is 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 Figure 4.5. Let 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 in which case we have an augmenting path

$\{o_i, x, u, v_2\}$. In both possibilities, v_2 is matched in preference to HUN(u), so we have $\phi(v_2) \geq \phi(\text{HUN}(u))$. By Lemma 4.2.7, 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 $\phi(v_2) \geq \phi(f_x)$. ■

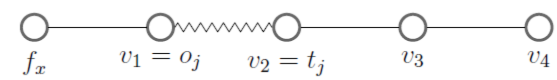

Figure 4.4. Lemma 4.2.8 Case (b): v_2 is a terminus that is matched by an augmenting path that includes v_1.

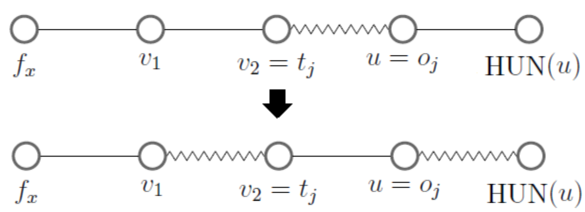

Figure 4.5. Lemma 4.2.8 Case (c): v_2 is a terminus that is matched by an augmenting path that includes $u \neq v_1$.

Lemma 4.2.9 *Let M_A^x denote the 2/3-Dir approximate matching at the x^{th} failure f_x, and let $P = \{f_x, v_1, v_2, v_3, ...\}$ 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 i-th 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 4.2.8 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 $v_3 = \mathrm{HUN}(u)$. Assume $v_3 \neq \mathrm{HUN}(u)$ and let some vertex $q = \mathrm{HUN}(u)$, as shown in Figure 4.6. Note that by Lemma 4.2.7 we have $\phi(q) \geq \phi(\mathrm{HUN}(v_1)) \geq \phi(f_x)$ (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 $\phi(t_i) \geq \phi(q)$ (B). Combining (A) and (B) we get $\phi(t_i) \geq \phi(f_x)$. Thus, $v_3 = \mathrm{HUN}(u)$. 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 Figure 4.7. 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 $\mathrm{HUN}(v_3)$. After the augmentation step, we have $\phi(t_i) \geq \phi(w)$ (A), 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 4.2.7 we have

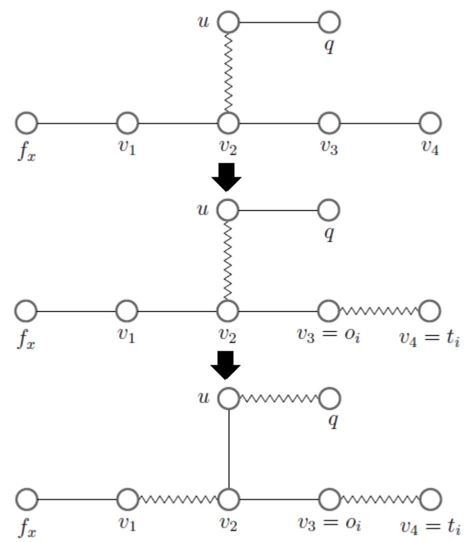

Figure 4.6. Lemma 4.2.9: The case where $v_3 \neq \mathrm{HUN}(u)$.

$\phi(w = \mathrm{HUN}(v_3)) \geq \phi(\mathrm{HUN}(v_1)) \geq \phi(f_x)$ (B). Combining (A) and (B), 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 Figure 4.8. 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 length three. After matching v_3, assume without loss of generality that q becomes $\mathrm{HUN}(u)$. After the augmentation step, we have $\phi(t_i) \geq \phi(q)$ (A), since there existed an augmenting path from v_3 to q. Note that (v_2, u) is still matching and must be changed to (v_2, v_1). By Lemma 4.2.7 $\phi(q = \mathrm{HUN}(u)) \geq \phi(\mathrm{HUN}(v_1)) \geq \phi(f_x)$ (B). Again, by combining (A) and (B), we obtain $\phi(t_i) \geq \phi(f_x)$.

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

Lemma 4.2.10 *Consider the symmetric difference $M_{opt} \oplus M_A^x$, corresponding to the 2/3-Dir approximate matching at the x-th failure. Let $P = \{f_x, v_1, v_2, v_3, v_4\}$ be an*

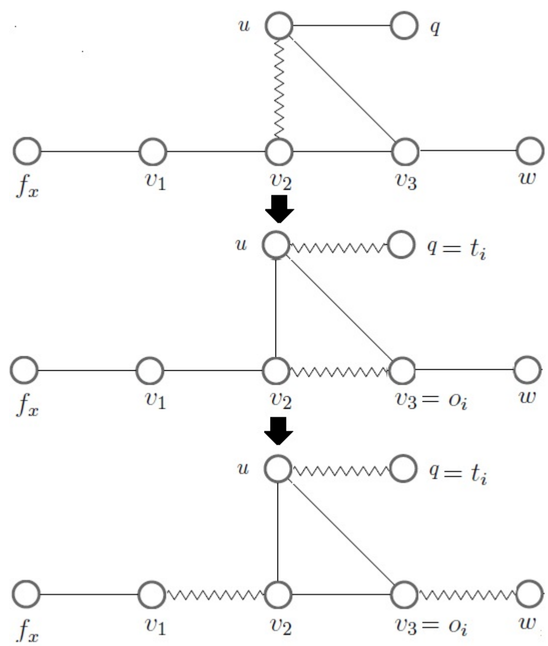

Figure 4.7. Lemma 4.2.9, (2) Case 1: The augmentation step is $\{o_i, v_2, u, t_i\}$.

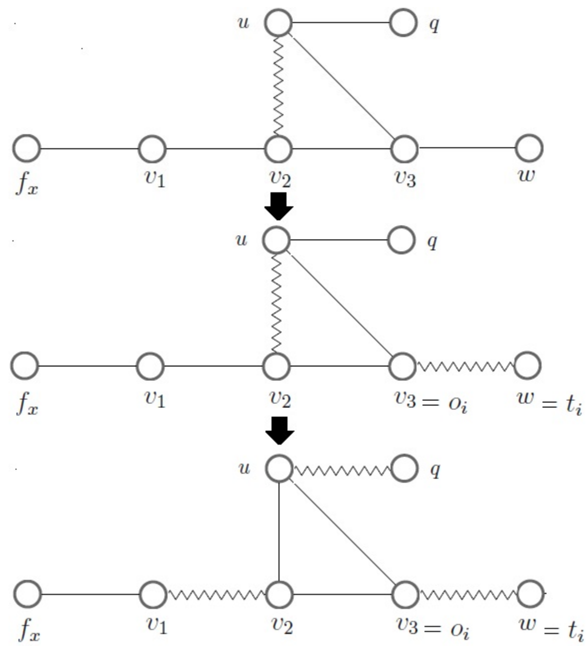

Figure 4.8. Lemma 4.2.9 (2) Case 2: the augmentation step does not include the edge (v_2, u).

$M_{opt} \oplus M_A^x$-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 Figure 4.9. 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. ■

Lemma 4.2.11 *Consider the symmetric difference $M_{opt} \oplus M_A$, where M_A is the matching computed by the 2/3-Dir approximation algorithm. For every failure f there are two distinct matched vertices in M_A that are at least as heavy as f.*

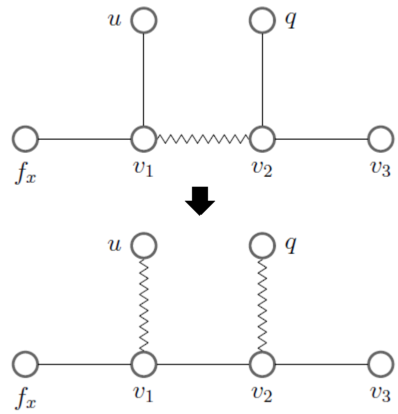

Figure 4.9. Lemma 4.2.10: Augmenting the path $\{u, v_1, v_2, q\}$ after f_x is determined to be a failure.

Proof First run the approximation algorithm and at the i-th 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 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 4.2.10, 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 4.2.8 (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 4.2.9 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 Figure 4.10. 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.

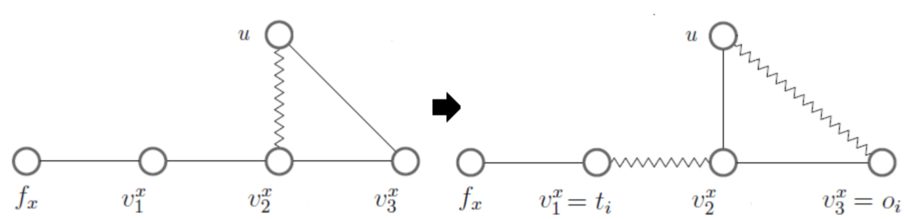

Figure 4.10. Lemma 4.2.11, $i-2$: The corresponding terminus is strictly lighter than the failure f_x.

(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 4.2.9 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 Figure 4.11. 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.

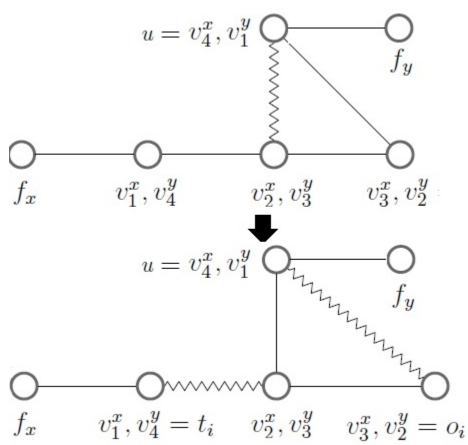

Figure 4.11. Lemma 4.2.11, Case 1, Subcase 1: The failure f_y is adjacent to u.

Subcase 2: The failure f_y is not adjacent to u as shown in Figure 4.12. 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 $\phi(t_i) \geq \phi(q)$ (A), because there existed an augmenting path from v_2^y to q. After v_2^y is matched, assume without loss of generality that $q = \text{HUN}(u)$. By Lemma 4.2.7, after (v_2^y, u) is changed to (v_2^y, v_1^y) we have $\phi(q = \text{HUN}(u)) \geq \phi(\text{HUN}(v_1^y)) \geq \phi(f_y)$ (B). Combining (A) and (B) we obtain $\phi(t_i) \geq \phi(f_y)$.

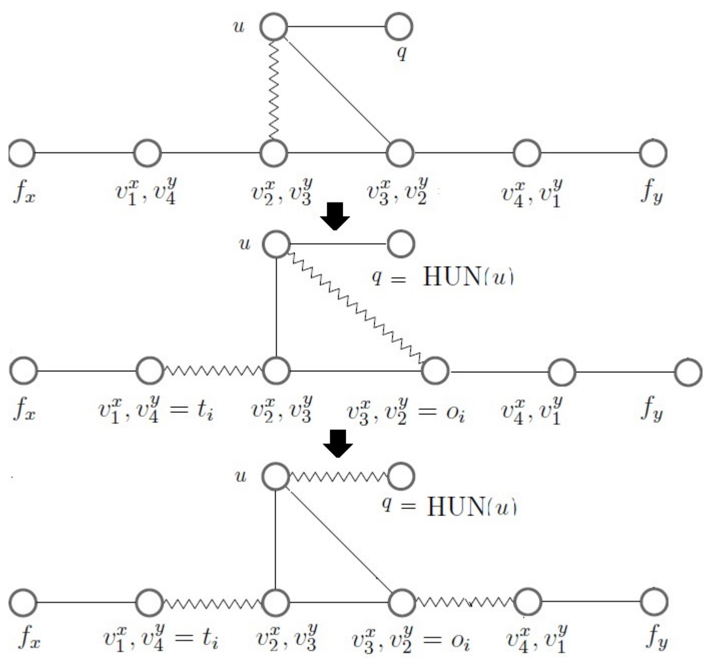

Figure 4.12. Lemma 4.2.11, Case 1, Subcase 2: The failure f_y is not adjacent to u.

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 4.2.9 we have (1) at the step 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 4.2.12 *Algorithm 18 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 4.2.11 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. ■

4.3 New Approximation Algorithms Based on the Iterative Approach

In this section, we propose new 1/2- and 2/3-approximation algorithms as well as a generalized $(k/k+1)$-approximation algorithm based on the iterative approach.

These are abbreviated to 1/2-Iter, 2/3-Iter and $(k/k+1)$-Iter. The iterative approach starts with an empty matching or initialized matching (naturally a cardinality matching with restricted length augmenting paths). Then, process the unmatched vertices in arbitrary order and improve the matching by finding restricted length increasing path or restricted length augmenting path. Unlike the previous technique, the iterative approach can be initialized, thus exploiting the structure of vertex-weighted graphs. In particular, an MVM has the same cardinality as a Maximum Cardinality Matching (MCM). Also, an α-approximate MVM has at least the same cardinality as an α-approximate MCM. In this way, the iterative approximation algorithms match very large numbers of vertices rapidly, without considering the weights of vertices by restricted length augmenting paths. Then, the algorithm finds increasing paths and augmenting paths that may result from prior increasing paths. One huge advantage of the iterative approach is its suitability for parallelization because the vertices are processed in arbitrary order, while the direct approach processes the vertices in non-increasing order of weights, making it unsuitable for parallelization.

4.3.1 A 1/2-Approximation Algorithm

In this section, we will describe the 1/2-Iter approximation algorithm. First, the algorithm starts with an empty matching or an initialized matching. The following steps are repeated as long as an augmenting or increasing path is found. The algorithm considers the vertices in arbitrary order. For each unmatched vertex u, the algorithm searches for an unmatched neighbor v. If it is found, then u and v are matched. If not, then we search for an increasing path of length two that reaches a lightest vertex v. If an increasing path is found, then the matching is updated by reversing the increasing path.

Algorithm 19 The Initialized Iterative 1/2-Approximation Algorithm for MVM.

```
procedure 1/2-Ite(G = (V, E, φ))
    M ← ∅;
    for each u ∈ V do
        Find from u an unmatched neighbor v;
        if v is found then
            M ← M ∪ (u, v);
        end if
    end for
    do
        done = true;
        for each u ∈ V do
            if u ∉ M then
                Find from u an unmatched neighbor v;
                if v is found then
                    M ← M ∪ (u, v);
                    done = false;
                else
                    Find a highest gain increasing path P′ from u s.t. |P′| = 2;
                    if P′ is found then
                        M ← M ⊕ P′;
                        done = false;
                    end if
                end if
            end if
        end for
    while not done
end procedure
```

Time Complexity

Theorem 4.3.1 *The time complexity of the 1/2-Iter approximation algorithm is* $O(\Delta m)$*, where* Δ *is the maximum degree.*

Proof If the matching is initialized with a maximal matching, then the maximal cardinality matching can be found in at most $O(m)$ time. In each iteration of the **for** inner loop, we choose an unmatched vertex u and examine all neighbors of u. If there is a matched neighbor v, then the mate of v is examined. Thus, for a vertex u, the algorithm examines at most a number of vertices equal to $d(u)$. Since a vertex can be unmatched at most $O(\Delta)$ times, a total of $O(d(u)\Delta)$ vertices are searched. Summing over all vertices, we have $\sum_{u \in V} d(u)\Delta = 2\Delta m = O(\Delta m)$. ■

Proof of Correctness

Now, we will prove that the 1/2-Iter approximation algorithm computes 1/2-approximation for MVM.

Theorem 4.3.2 *Let* $G = (V, E, \phi)$ *be a graph and* $\phi : V \mapsto R_{\geq 0}$ *a weight function. Then the* 1/2*-Iter approximation algorithm computes a* 1/2*- approximation for the maximum vertex-weighted matching problem on* G*.*

Proof Let M_A be the approximate matching computed by the 1/2-Iter algorithm and M_{opt} be an optimal MVM. By the algorithm design, there does not exist an augmenting path of length one and increasing path of length two with respect to M_A in G. It follows from Theorem 4.1.1 that M_A is at least $\frac{1}{2}M_{opt}$. ■

4.3.2 A 2/3-Approximation Algorithm

The description of the approximation algorithm is similar to the 1/2-Iter, but it finds augmenting paths of lengths at most three and increasing paths of lengths at most four.

Algorithm 20 The Initialized Iterative 2/3-Approximation Algorithm for MVM.

```
procedure 2/3-ITER(G = (V, E, φ))
    M ← ∅;
    for each u ∈ V do
        Find an augmenting path P s.t. |P| ≤ 3 from u;
        if P is found then
            M ← M ⊕ P;
        end if
    end for
    do
        done =true;
        for each u ∈ V do
            if u ∉ M then
                Find aug. path P from u s.t. |P| ≤ 3;
                if P is found then
                    M ← M ⊕ P;
                    done =false;
                else
                    Find a highest gain increasing path P′ from u s.t. |P′| ≤ 4;
                    if P′ is found then
                        M ← M ⊕ P′;
                        done =false;
                    end if
                end if
            end if
        end for
    while done
end procedure
```

Time Complexity

Theorem 4.3.3 *The time complexity of the 2/3-Iter approximation algorithm is* $O(\Delta^2 m)$*, where* Δ *is the maximum degree.*

Proof If the matching is initialized then the initial cardinality matching is found in $O(m)$ time. This is achieved by using a pointer to the first unmatched neighbor in the adjacency list. In this way, each vertex u requires at most $O(d(u))$ steps to find an augmenting path of length at most three. Summing over all vertices, we have $O(m)$.

In the do while loop, in each iteration of the **for** loop, we choose an unmatched vertex u and examine all neighbors of u. If there is a matched neighbor v, then all neighbors of $Mate(v)$ except v are examined. Therefore, for a vertex u, the algorithm examines at most a number of vertices equal to

$$d(u) + \sum_{v \in N(u)} d(Mate(v)) - 1 \leq d(u) + \sum_{v \in N(u)} \Delta - 1 = d(u) + d(u)\Delta - d(u) = d(u)\Delta.$$

Since a vertex can be unmatched at most $O(\Delta)$ times in total, $O(d(u)\Delta^2)$ vertices are searched. Summing over all vertices, we have $\sum_{u \in V} d(u)\Delta^2 = 2\Delta^2 m = O(\Delta^2 m)$.
■

Proof of Correctness

We will now prove the approximation ratio of the 2/3-Iter approximation algorithm.

Theorem 4.3.4 *Let* $G = (V, E, \phi)$ *be a graph and* $\phi : V \mapsto R_{\geq 0}$ *a weight function. Then, the* 2/3*-Iter approximation algorithm computes a* 2/3*- approximation for the maximum vertex-weighted matching problem on* G*.*

Proof Let M_A be the approximate matching computed by 2/3-Iter algorithm and M_{opt} be an optimal MVM. By the algorithm design, there does not exist an augmenting path of length three and increasing path of length four with respect to M_A in G. It follows from Theorem 4.1.1 that M_A is at least $\frac{2}{3}M_{opt}$. ■

We investigated the possibility of reducing the iterations of the iterative approximation algorithms to one by processing the vertices in non-increasing order of weights. We found that there is no advantage to sorting vertices in non-increasing order of their weights, as it still requires more than one iteration. Consider a case illustrated in Figure 4.13. Since we process the vertices in non-increasing order of weights, first we search for an increasing path from v_1, and the search fails. Next, a search starts from v_6 and finds an increasing path of length four $\{v_6, v_5, v_4, v_7, v_8\}$. After updating the matching as shown in Figure 4.14, an increasing path is created $\{v_1, v_2, v_3, v_4, v_7\}$. Thus, another iteration is required.

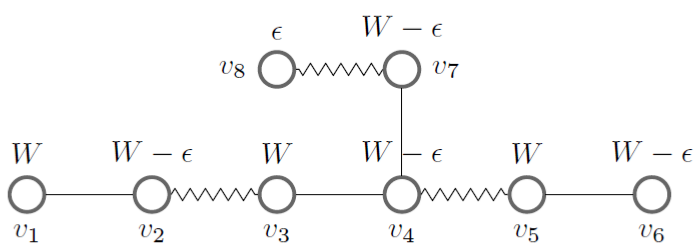

Figure 4.13. A search for an increasing path from v_1 fail. $W \gg \epsilon$.

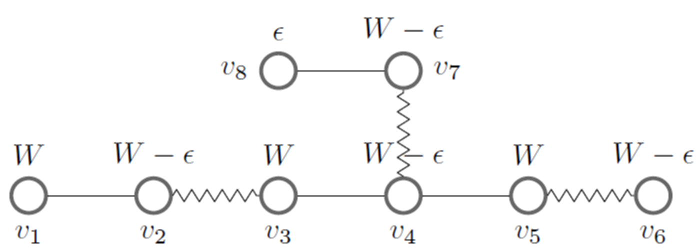

Figure 4.14. An increasing path from v_1 is created after the matching is updated. $W \gg \epsilon$.

For a 1/2-approximation algorithm, consider a case illustrated in Figure 4.15. Since we process the vertices in non-increasing order of weights, first we search for an increasing path from v_1, and the search fails. Next, a search starts from v_4 and finds an increasing path of length two $\{v_4, v_3, v_2\}$. After updating the matching, v_5 finds an unmatched neighbor and is matched to v_2. Now, an increasing path of length two is created $\{v_1, v_2, v_5\}$. Again, another iteration is required.

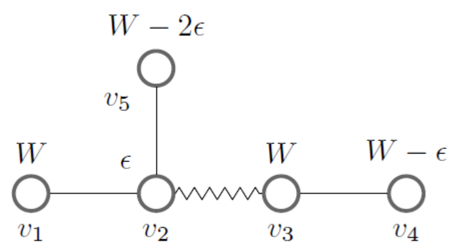

Figure 4.15. A search for an increasing path from v_1 fail. $W \gg \epsilon$.

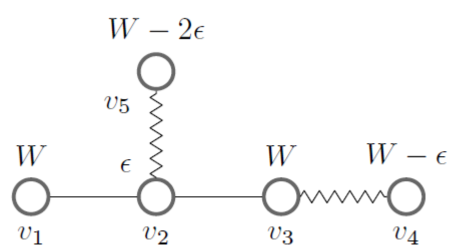

Figure 4.16. An increasing path from v_1 is created after the matching is updated twice. $W \gg \epsilon$.

4.3.3 A $(k/k+1)$-Approximation Algorithm

The approximation algorithm follows the same steps of the 1/2- and 2/3-Iter approximation algorithms, but it finds augmenting paths of lengths at most $2k - 1$

and increasing paths of lengths at most $2k$. Here, k is the maximum number of non-matching edges in augmenting and increasing paths.

Algorithm 21 The Initialized Iterative $k/k+1$-Approximation Algorithm for MVM.

procedure $\frac{k}{k+1}$-ITER($G = (V, E, \phi),\ k$)
  $M \leftarrow \emptyset$;
  $M \leftarrow (k/k+1)$-APPROXCARD($G = (V, E)$,k);
  do
    done = true;
    for each unmatched vertex u **do**
      Find an aug. path P from u s.t. $|P| \leq 2k-1$;
      if P is found **then**
        $M \leftarrow M \oplus P$;
        done = false;
      else
        Find a highest gain increasing path P' from u s.t. $|P'| \leq 2k$;
        if P' is found **then**
          $M \leftarrow M \oplus P'$;
          done = false;
        end if
      end if
    end for
  while done
end procedure

Time Complexity

Theorem 4.3.5 *The time complexity of the $k/(k+1)$-Iter approximation algorithms is $O(\Delta^k m)$, where Δ is the maximum degree of a vertex and k is the maximum number of non-matching edges in an augmenting or increasing path.*

Proof The $(\frac{k}{k+1})$-cardinality matching can be found in $O(mk)$ time, using the Micali and Vazirani algorithm [47] with k rounds. In each iteration of the inner **for** loop, we choose an unmatched vertex u. In the worst case, we search an alternating tree of height $2k$. We have the root u and $k-1$ levels of vertices of even depth called outer vertices (from which we search for an unmatched vertex and increasing path), and the total is k outer vertices levels. Now, we will upper bound the number of searched vertices. At the root, the algorithm examines all vertices in $N(u)$. If there is a matched neighbor v, then all neighbors of $Mate(v)$ except v are examined so we have at most $N(u)$ vertices searched. For each other outer level $i > 1$, we have $N(u)(\Delta-1)^i$ searched vertices. So for a vertex u, the algorithm examines at most a number of vertices equal to

$$\begin{aligned}
& d(u) + \sum_{i=1}^{k-1} d(u)(\Delta-1)^i \\
&= d(u)(\sum_{i=0}^{k-1}(\Delta-1)^i) \text{ This is a partial geometric series} \\
&= d(u)((\Delta-1)^k - 1)/((\Delta-1)-1) \\
&< d(u)(\Delta^{k-1}).
\end{aligned}$$

Since a vertex can be unmatched at most $O(\Delta)$ times in total, $O(d(u)\Delta^k)$ vertices are searched. Summing over all vertices, we have $\sum_{u\in V} O(d(u)\Delta^k) = O(\Delta^k m)$. ■

Proof of Correctness

Theorem 4.3.6 *Let $G = (V, E, \phi)$ be a graph and $\phi : V \mapsto R_{\geq 0}$ a weight function. Then Algorithm 21 computes a $(k/k+1)$- approximation for the maximum vertex-weighted matching problem on G.*

Proof Let M_A be the approximate matching computed by $(k/k+1)$-Iter algorithm and M_{opt} be an optimal MVM. By the algorithm design, there does not exist an augmenting path of length $2k-1$ and an increasing path of length $2k$ with respect to M_A in G. It follows from Theorem 4.1.1 that M_A is at least $\frac{k}{k+1}M_{opt}$. ■

5 PARALLEL APPROXIMATION ALGORITHMS FOR MVM

In this chapter we will describe how the iterative 2/3-approximation algorithm for MVM can be parallelized in shared memory multi-core machines. We will also discuss potential problems that arise when we augment and increase the weight of a matching in parallel. We propose a new locking technique to ensure the correctness of a matching, which we proved is free from livelock, deadlock and starvation states.

5.1 A Parallel 2/3-Approximation Algorithm

Now we will discuss the parallelization of Algorithm 20, the initialized iterative 2/3-approximation algorithm for MVM. While there are unmatched vertices, the algorithm searches for augmenting paths (of length at most three) or increasing paths (of length at most four). Once a thread finds one such path, it locks vertices on the path such that no other thread should augment or update the matching on these vertices since the augmenting and increasing paths discovered by two threads could overlap. If a thread cannot acquire all locks needed, then it releases all locks and proceeds to search from other unmatched vertices. There is an implicit synchronization barrier across all threads at the end of each iteration of the **for** loop.

Note that for the 1/2-approximation algorithm, the same method of parallelization may be employed by restricting the length of an augmenting path to one, and the length of an increasing path to two.

Now we discuss how the test and set locks are employed in the parallel algorithm. The lock is free if its value is zero and, not free otherwise. If a thread reads a value of zero for a lock, then it has atomic access to the lock variable and can set it to a nonzero value. If a thread reads a nonzero value for a lock, then it is unavailable. We allow an augmenting path joining the vertices u and v to augment the matching only

Algorithm 22 The Parallel Initialized Iterative 2/3-Approximation Algorithm for MVM.

procedure PAR-2/3-ITER($G = (V, E, \phi)$)
  $M \leftarrow$ PAR-2/3-APPROXCARD($G = (V, E, \phi)$);
  do
    done = true;
    for all unmatched $u \in V$ **do in parallel**
      Find an aug. path P from u to v s.t. $|P| \leq 3$;
      if P is found and $u < v$ **then**
        if LOCK(P,u) = true **then**
          $M \leftarrow M \oplus P$; done = false;
          release all locks;
        else
          continue;
        end if
      else
        Find a highest gain increasing path P' from u s.t. $|P'| \leq 4$;
        if P' is found **then**
          if LOCK(P',u) = true **then**
            $M \leftarrow M \oplus P'$; done =false;
            release all locks;
          else
            continue;
          end if
        end if
      end if
    end for
  while done = false
end procedure

if $u < v$, and in this way we prevent two threads from attempting to acquire locks and augmenting the same path from opposite directions.

For a single matching edge (u, v) on an augmenting path, we lock its lower-numbered endpoint; for two matching edges (u, v) and (x, y) on an increasing path, we need to lock the lower-numbered endpoint of both edges, but with the lowest numbered endpoint locked first. Hence the lock for first_min, the minimum among all four vertices is acquired first, and then the lock for second_min, the lower numbered endpoint of the other matching edge, is acquired.

We proceed to describe the locking procedure in more detail in Algorithm 23. When a thread finds an augmenting path of length one, $\{u, q\}$, then it tests $lock(u)$. If $lock(u) \neq 0$, then the algorithm continues to the next unmatched vertex. If it equals 0, it sets lock(u) with 1 and then tests the status of the lock on q. If a thread finds a $lock(u)$ value to be nonzero, then it abandons the attempt to lock the remaining vertices on the augmenting or increasing path, releases any locks that it has acquired on the path, and processes the next unmatched vertex. If $lock(q) = 0$, then it sets $lock(q)$ to 1. After a thread has acquired all locks for a path, then it checks to see if any other thread had already updated the matching using some of the vertices or edges on this path during the time it took to acquire the locks before updating the matching. If it has changed then the thread releases all acquired locks and continues to the next unmatched vertex. After augmenting the matching using the path, the thread then releases locks on u and q. To avoid repetition, from now on, we will assume that if a thread finds a $lock(v)$ value to be nonzero or a path has been changed, then it abandons the attempt to lock the remaining vertices on the augmenting or increasing path and update the matching, releases any locks that it has acquired on the path, and processes the next unmatched vertex.

For an augmenting path of length three $\{u, v_1, v_2, q\}$, the same technique is used. If $lock(u) = 0$, then it sets $lock(u)$ to 1 and tests the status of the lock on q. If $lock(q) = 0$, then $lock(q)$ is set to 1. After acquiring $lock(q)$, the thread finds $v = \min(v_1, v_2)$

and tests $lock(v)$. If its value is 0, it sets it to 1. If all three locks are acquired by the thread, then it augments the matching and releases the locks acquired.

Next, we discuss how vertices on an increasing path are locked in order to update the matching. First we describe this for an increasing path of length two, $\{u, v_1, v_2\}$. If $lock(u) = 0$, then the thread sets $lock(u)$ to 1. Now the thread finds $v = \min\{v_1, v_2\}$ and tests $lock(v)$; if $lock(v) = 0$, the thread sets $lock(v) = 1$. If the two required locks on the increasing path are acquired by the thread, then the matching is updated and the acquired locks are released. Now consider an increasing path of length four denoted by $\{u, v_1, v_2, v_3, v_4\}$. If $lock(u) = 0$, then the thread sets $lock(u)$ to 1. The thread finds $m_1 = \min\{v_1, v_2\}$ and $m_2 = \min\{v_3, v_4\}$. Let $first\_min = \min\{m_1, m_2\}$ and $second\_min = max\{m_1, m_2\}$. The thread tests $lock(first\_min)$, and if its value is 0, the thread sets it to 1. Next, the thread tests $lock(second\_min)$; if its value is also 0, then the thread sets $lock(second\_min) = 1$. If the three required locks on the increasing path are acquired by the thread, then the matching is updated, after which the acquired locks are released.

In Algorithm 22, we must consider the possibility that, in an iteration of the **for** loop, none of the threads is able to augment or update the matching because they are unable to acquire the locks. This can happen in the case of a cyclic wait, where each thread is unable to acquire all the locks it needs because other threads have acquired some of the locks, causing a cyclic dependence on a subset of threads. We illustrate this in Fig. 5.1 for a set of increasing paths of length four that overlap with each other and induce a cycle in the graph. A thread T_i processing the unmatched vertex u_i needs to lock endpoints of two consecutive matching edges (we consider the increasing paths in the clockwise direction). Thus T_1 needs to lock (v_1, v_2) and (v_3, v_4) and so on, with the last thread T_k needing to lock (v_{k-1}, v_k) and (v_1, v_2). If each thread T_i succeeds in acquiring only one lock on v_{2i-1}, we could have **livelock** as shown in Figure 5.2, and none of the threads would be able to update the matching.

Similarly, an illustrattion of this is in Figure 5.3 for a set of augmenting paths of length three that overlap with each other and induce a cycle in the graph. A thread

Algorithm 23 Locking procedure.

```
procedure LOCK(P,u)
    if lock(u)=0 then
        lock(u)=1;
        if P is an augmenting path then
            if lock(q)=0 then                    ▷ q > u is other unmatched end point
                lock(q)=1;
            else
                release any locks;
                return false;
            end if
        end if
        for each matching edge e = (v_i, v_j) on P do
            min_v_in_e = min(v_i, v_j);
        end for
        for each min_v_in_e in increasing order do
            if lock(min_v_in_e)=0 then
                lock(min_v_in_e)=1;
            else
                release any locks;
                return false;
            end if
        end for
        if P has not changed then
            return true;
        else
            release any locks;
            return false;
        end if
    else
        return false;
    end if
end procedure
```

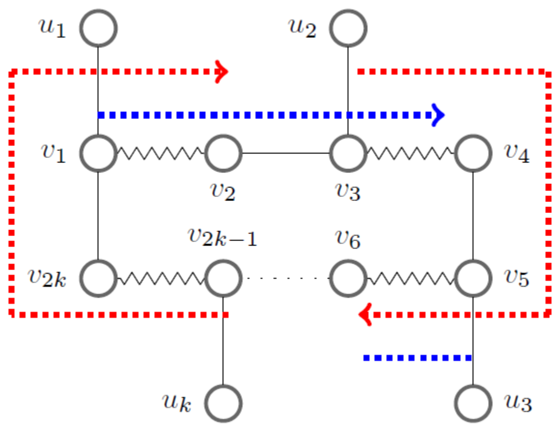

Figure 5.1. A set of increasing paths of length four that could induce a cyclic wait among threads.

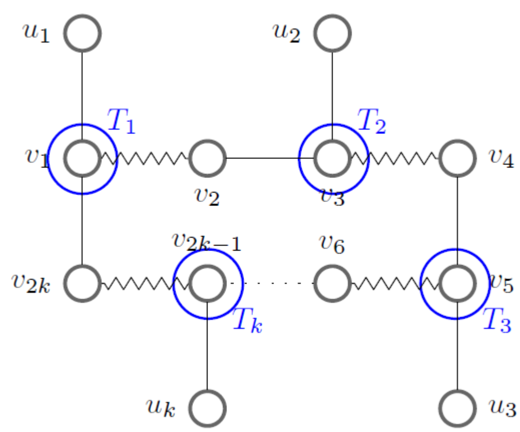

Figure 5.2. If each thread T_i locks v_{2i-1}, we have a cyclic wait.

T_i processing the unmatched vertex u_i needs to lock endpoints of two consecutive unmatched matched vertices (we consider the augmenting paths in the clockwise direction). Thus, T_1 needs to lock u_1 and (u_2) and so on, with the last thread T_k needing to lock (u_k) and (u_1). If each thread T_i succeeds in acquiring only one lock on u_i, we could have **livelock** as shown in Figure 5.4, and none of the threads would be able to update the matching.

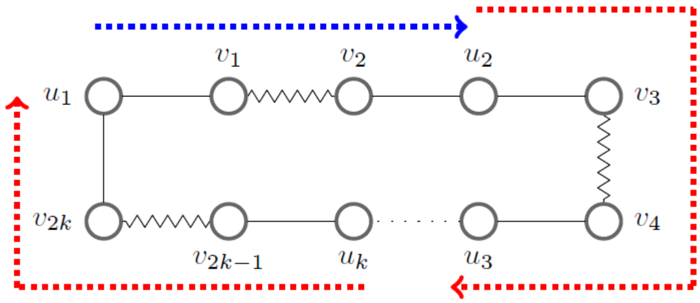

Figure 5.3. A set of augmenting paths of length three that could induce a cyclic wait among threads.

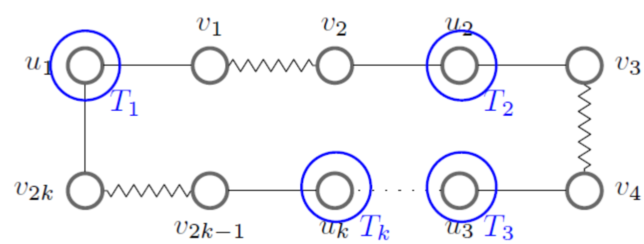

Figure 5.4. If each thread T_i locks u_i, we have a cyclic wait.

5.2 Proof of Correctness

Next we will prove the correctness of the parallel algorithm.

Theorem 5.2.1 *In each iteration of the* **for** *loop in Algorithm 22, at least one thread among a set of threads competing for locks will be able to acquire the locks it needs and update the matching.*

Proof We distinguish between the locks for unmatched vertices and matched vertices and say they are locks of different types. In any iteration of the **for** loop, there will be no dependence between locks of two distinct types. Cyclic dependencies among threads $\{T_1, ...T_k\}$ occur when these threads need to acquire two locks of the same type with one lock acquired by a thread T_i and the other by another thread T_j, in such a way that these dependencies are cyclic. In this case, these threads fail to acquire the locks they need and release them, and no thread can update the matching.

We consider cases where such cyclic dependencies may occur, and hence we do not need to consider the following cases:

1. A set of increasing paths of length two, since each thread requires a lock of a distinct unmatched vertex and a lock of a distinct matched vertex, and these locks are disjoint.

2. A set of increasing paths consisting of both lengths two and four for the same reason as above.

3. A set including increasing paths of length four and augmenting paths of length three, since a thread that locks an augmenting path needs two locks of unmatched vertices and one lock for a matched vertex, whereas the thread locking an increasing path needs one lock of an unmatched vertex and two locks of matched vertices.

We will consider the following four cases.

Case 1: An overlapping set of augmenting paths of length three that induce a (longer) path. Let the k augmenting paths be listed as $\{u_i, v_{2i-1}, v_{2i}, u_{i+1}\}$, for $1 \leq i \leq k$. Let thread T_i be assigned to augment $\{u_i, v_{2i-1}, v_{2i}, u_{i+1}\}$. Then, since there is no contention for the vertices u_1 and u_{k+1}, the thread T_1 can lock the former and T_k can lock the latter. If all threads lock their first unmatched vertices, then thread T_k both acquires its locks and can augment. If not, some thread T_j for $j \geq 2$ cannot lock its first unmatched vertex since thread T_{j-1} has acquired it. Choose T_j to be the lowest numbered such thread. By choice of j, T_{j-1} has acquired its first unmatched vertex also, and hence the latter thread can augment.

Case 2: An overlapping set of augmenting paths of length three that induce a cycle. Let the k augmenting paths be $\{u_i, v_{2i-1}, v_{2i}, u_{i+1}\}$ for $1 \leq i < k$ and $\{u_k, v_{2k-1}, v_{2k}, u_1\}$ for $i = k$. Let thread T_i be assigned to augment $\{u_i, v_{2i-1}, v_{2i}, u_{i+1}\}$ and T_k be assigned to augment $\{u_k, v_{2k-1}, v_{2k}, u_1\}$. Consider the lowest numbered unmatched vertex u_i in this cycle. The previous and the next unmatched vertices in the cycle, u_{i-1} and u_{+1}, are numbered higher than u_i. Because in Algorithm 22 we augment only from a lower-numbered unmatched vertex to a higher-numbered unmatched vertex, such a cyclic set of dependencies among augmenting paths requiring locks cannot exist. Thus, this case reduces to a non-cyclic set of augmenting paths, and from Case 1, one thread must succeed.

Case 3: An overlapping set of increasing paths of length four that induce a path. Let the k increasing paths be $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$, for $1 \leq i \leq k$. Let thread T_i be assigned to update $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$. We denote (v_{2i-1}, v_{2i}) as the first matching edge of T_i and (v_{2i+1}, v_{2i+2}) as the second matching edges of T_i. If all threads lock a vertex in the first matching edge first, then T_k (the last thread) will lock the vertex in the second matching edge since it is not shared. If not, there is some thread T_i such that its neighbor thread T_{i+1} locks a vertex in its second matching edge first. Choose T_i to be the lowest numbered such thread. If T_{i+1} succeeds in locking a vertex in its first matching edge also, then it can augment the matching. If it fails, then by choice of i thread, T_i has acquired its second matching edge and can augment.

Case 4: An overlapping set of increasing paths of length four that induce a cycle in the graph (see Figure 5.1). Let the k increasing paths be denoted by $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$ for $1 \leq i < k$ and $\{u_k, v_{2k-1}, v_{2k}, v_1, v_2\}$ for $i = k$. Let thread T_i be assigned to the path $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$ and T_k be assigned to the path $\{u_k, v_{2k-1}, v_{2k}, v_1, v_2\}$. Consider the lowest numbered matched vertex v_m in the cycle and denote the two threads competing for it by T_i and T_{i+1}. The one that fails to lock v_m will not seek to lock any other vertex; thus the cyclic dependence is now broken. Again, we have reduced this case to Case 3, and hence, one thread must succeed in acquiring locks and updating the matching.

This completes the proof. ■

In the context of Algorithm 22, there are three potentially adverse things that could happen. The first is deadlock, when some thread cannot acquire the locks it needs and cannot execute another instruction. This does not happen here by design, since when a thread fails to acquire a lock, it proceeds to the next unmatched vertex or to the next iteration. The second is starvation, when one or more threads are not able to acquire locks because other threads have higher priority. This also cannot happen here because at least one thread responsible for augmenting or updating the matching overlapping paths succeeds in each iteration, and thus, in at most n iterations, all vertices will be processed. The third is livelock, when cyclic wait makes every thread unable to acquire the locks it needs; we have proven that this cannot happen in this parallel algorithm.

6 EXPERIMENTS AND RESULTS

6.1 Experimental Setup

We used an Intel Xeon E5-2660 processor-based system (part of the Purdue University Community Cluster), called *Rice* and *Snyder*[1]. 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, 64 GB of memory for Rice, 256 GB for Snyder. The operating system is Red Hat Enterprise Linux release 6.7. All code was developed using C++ and compiled using the g++ compiler (version: 6.3.0) using the -O3 flag. Our test set consists of nineteen real-world graphs taken from the University of Florida Matrix collection [77], covering several application areas and synthetic datasets that were generated by the RMAT graph generator [78]. We generated three different synthetic datasets varying the RMAT parameters. These are (i) rmat-G500 representing graphs with skewed degree distribution from the Graph 500 benchmark [79] with parameter set (0.57, 0.19, 0.19, 0.05), (ii) rmat-SSCA from HPCS Scalable Synthetic Compact Applications graph analysis benchmark [80], with parameter set (0.6, 0.133, 0.133, 0.133), and (iii) rmat-ER Erdos-Renyi random graphs with uniform degree distributions, with parameter set (0.25, 0.25, 0.25, 0.25). Table 6.1 gives some statistics regarding our test set. The graphs are listed in increasing order of the total number of vertices. The largest number of vertices of any graph is nearly 134 million, and the largest number of edges is nearly 2 billion. For each graph, we list the maximum, average vertex degrees and standard deviation (SD) over the mean degrees. The average degrees vary from 2.12 to 117.92, and SD/Mean vary from 0.0 to 15.5. Hence, the graphs are diverse with respect to their degree distributions. We have three types of weights. Integer weights of vertices were generated uniformly at random in the

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

range [1 1000]; real-valued weights were chosen randomly in the range $[1.0, 1.3]$; and the degree of a vertex v is used as the weight of v. The reported results are the averages of ten trials of randomly generated weights, and for the degree weights, the time is the average of ten trials using the same weights. The standard deviations for run-time, weight ratio, and cardinality ratio are close to zero, so there is not much variation in 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, We 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.

6.2 Serial Algorithms Results

6.2.1 Exact Algorithms

In this section we compare the Direct-Augmenting, Direct-Increasing and Iterative maximum vertex-weighted matching (MVM) algorithms. We have included an exact algorithm for the maximum edge-weighted matching (MEM) implemented in LEDA [24, 25] in our comparisons. This is a primal-dual algorithm implemented with advanced priority queues and efficient dual weight updates, with time complexity $O(mn \log n)$ [81]. Since this is commercial software, we can only run the object code, and we ran it with no initialization, Greedy 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 [82]. We call these three variants LEDA1, LEDA2 and LEDA3, respectively. Because all exact algorithms find the same weight and cardinality, we will compare the running time and report weight and cardinality in Table 6.5. Unfortunately, LEDA code runs on integer weights only, and because of data types issues, it does not run on rmat-graphs with 2 billion edges. The cutoff time is set to be four

Table 6.1.
The set of test problems.

Graph	$\lvert V \rvert$	Degree			$\lvert E \rvert$
		Max.	**Mean**	**SD/Mean**	
G34	2,000	4	4.00	0.00	4,000
G39	2,000	210	11.78	1.17	11,778
de2010	24,115	45	4.81	0.62	58,028
shipsec8	114,919	131	56.90	0.25	3,269,240
kron_g500-17	131,072	29,935	94.78	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_g500-19	524,288	80,674	106.46	5.76	21,780,787
tx2010	914,231	121	4.87	0.63	2,228,136
kron_g500-20	2,097,152	213,904	117.92	7.47	91,040,932
M6	3,501,776	10	5.99	0.14	10,501,936
hugetric	6,592,765	3	2.99	0.01	9,885,854
rgg_n_2_23	8,388,608	40	15.14	0.26	63,501,393
hugetrace	12,057,441	3	2.99	0.01	18,082,179
nlpkkt200	16,240,000	27	26.60	0.09	215,992,816
hugebubbles	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
rmat-G500	48,877,747	2,407,313	85.28	15.48	2,084,251,521
rmat-SSCA	93,488,461	641,453	45.29	9.96	2,117,212,258
rmat-ER	134,217,728	241	32.00	0.29	2,147,483,625

hours for most graphs. nlpkkt200 using all weights, rmat-G500 and rmat-SSCA using vertex degrees weights cutoff time is set to two hundred hours. We were able to find the matchings on all graphs except rmat-G500 and rmat-SSCA using vertex degrees weights.

In general, the Direct-Increasing algorithm is the fastest and finished computing matchings on all graphs except rmat-G500 and rmat-SSCA when the weights are vertex degrees. Next, comes the Iterative algorithm, then the Direct-Augmenting algorithm. LEDA is the slowest algorithm, with LEDA3 being the fastest among LEDA variants.

For integer weights, we report the running time in Tables 6.2. The Direct-Increasing is 415, 366 and 37 times faster than LEDA1, LEDA2, and LEDA3, respectively, all on the geometric mean. Also, the Iterative algorithm is faster than all LEDA variants by factors of 182, 160, and 22. The Direct-Augmenting algorithm is 21, 19, and 1.2 times faster, respectively, again all on the geometric mean. The Direct-Increasing algorithm is around 5 times faster than the Direct-Augmenting algorithm on the rmat graphs, while the Iterative is faster than the Direct-Increasing on the rmat-ER graph by a factor of two.

Running time in seconds using real weights in range [1.0 1.3] is reported in Table 6.3. The Direct-Increasing is 1.7, 15 times faster than Iterative and Direct-Augmenting algorithms, respectively, all on geometric mean. The iterative algorithm is 9 times faster than the Direct-Augmenting algorithm on the geometric mean.

For vertex degrees weights, we report the running time in Table 6.4. Again, the Direct-Increasing is faster; it outperformed all LEDA variants by a factor of 138, 108, and 18 all on the geometric mean. LEDA3 runs 1.2 and 4 times faster than the Direct-Increasing algorithm on kron-g500-19 and kron-g500-20 graphs. The Iterative algorithm is 122, 96, and 19 times faster than LEDA1, LEDA2, and LEDA3, respectively. all on the geometric mean. The Iterative algorithm did not finish computing the matchings on four graphs. The Direct-Augmenting algorithm is 71, 56, and 10 times faster than LEDA implementations all on geometric mean. Note that, the Direct-Augmenting algorithm did not complete in 4 hours on rmat-ER while the Iterative finished in 34 minutes and the Direct-Increasing did so in one hour.

From the reported results, we can see that MVM exact algorithms outperform the MEM exact algorithms. While initializing the matching with a fractional one

Table 6.2.
The running time (seconds) of MVM and MEM exact algorithms. Random integer weights in [1 1000].

Graph	LEDA1	LEDA2	LEDA3	Direct Augmenting	Direct Increasing	Iterative
G34	0.1600	0.1700	0.0700	0.0102	0.0005	**0.0002**
G39	1.3200	1.3100	0.0200	0.0240	**0.0009**	0.0009
de2010	3.9000	4.0500	0.2800	0.4909	0.0150	**0.0112**
shipsec8	-	-	6.2100	27.119	**0.2206**	0.2607
kron_g500-17	124.31	119.26	8.9400	3.5373	**0.1660**	7.3513
mt2010	16.920	15.700	1.5300	1.3993	0.1091	**0.0934**
fe_ocean	613.51	611.760	753.69	142.28	0.6495	**0.6271**
tn2010	251.21	232.130	12.810	12.512	0.3329	**0.3007**
kron_g500-19	628.76	595.010	39.560	19.077	**0.8166**	68.104
tx2010	2140.9	2124.84	79.390	91.545	1.9591	**1.0547**
kron_g500-21	3361.9	3154.65	207.19	101.57	**3.8188**	683.89
M6	-	-	1080.4	1774.1	**9.3718**	9.9665
hugetric	-	-	369.94	870.48	**17.870**	21.250
rgg_n_2_23	-	-	5867.8	2118.7	41.564	**20.958**
hugetrace	-	-	599.95	1396.9	35.628	**32.087**
nlpkkt200	-	-	-	-	**582050**	-
hugebubbles	-	-	1581.0	3158.3	70.828	**66.059**
road_usa	1389.1	824.070	127.60	63.437	28.483	**14.806**
europe_osm	4076.4	1910.96	267.12	79.393	52.613	**31.710**
rmat-G500	-	-	-	1318.1	**314.67**	-
rmat-SSCA	-	-	-	5557.1	**953.91**	-
rmat-ER	-	-	-	-	3176.3	**1664.4**

boosts the performance of LEDA considerably, it is still much slower than the MVM algorithms; the Direct-Increasing and Iterative can be faster than LEDA3 by a factor of 1000. The Direct-Increasing algorithm is the only algorithm that successfully finished finding the matching on the nlpkkt200 graph in about 160 hours, while the

Table 6.3.
The running time (seconds) of MVM and MEM exact algorithms. Random real weights in [1.0   1.3].

Graph	Direct Augmenting	Direct Increasing	Iterative
G34	0.0087	0.0005	**0.0002**
G39	0.0174	0.0010	**0.0009**
de2010	0.2189	0.0136	**0.0112**
shipsec8	6.4723	**0.2404**	0.2520
kron_g500-17	3.2481	**0.1521**	6.6568
mt2010	0.7030	0.1093	**0.0926**
fe_ocean	62.541	**0.5749**	0.6404
tn2010	4.9192	0.3541	**0.2901**
kron_g500-19	17.244	**0.8180**	74.874
tx2010	32.857	1.6196	**1.0218**
kron_g500-21	102.36	**3.8639**	653.76
M6	644.99	**9.2931**	9.8297
hugetric	342.79	**17.987**	21.417
rgg_n_2_23	650.09	40.250	**21.420**
hugetrace	532.37	**31.366**	32.301
nlpkkt200	-	**627657**	-
hugebubbles	1112.0	**59.232**	66.299
road_usa	47.870	29.061	**14.946**
europe_osm	71.707	53.071	**31.266**
rmat-G500	1247.7	**174.27**	-
rmat-SSCA	4432.1	**466.13**	-
rmat-ER	-	**1731.1**	1860.0

other algorithms failed in 200 hours. Although the Iterative algorithm performed very well and ran faster than the Direct-Increasing algorithm on many graphs, it did not terminate in four hours on rmat-G500 and rmat-SSCA using all types of weights. This is because, after a maximum cardinality matching is computed, a high

Table 6.4.
The running time (seconds) of MVM and MEM exact algorithms.Vertex degrees weights.

Graph	LEDA1	LEDA2	LEDA3	Direct Augmenting	Direct Increasing	Iterative
G34	0.0200	0.0200	0.0100	0.0002	0.0003	**0.0002**
G39	0.5700	0.5700	0.0200	0.0028	0.0010	**0.0010**
de2010	2.8800	2.7300	0.2600	0.0482	0.0145	**0.0119**
shipsec8	58.480	48.060	5.2600	0.7518	**0.2329**	0.2421
kron_g500-17	3042.9	3147.6	75.920	**19.755**	20.565	151.26
mt2010	17.700	17.870	1.5800	0.3009	0.1011	**0.0973**
fe_ocean	51.070	48.960	543.25	2.7413	0.6048	**0.5901**
tn2010	245.88	237.21	9.8900	0.8807	0.3289	**0.3014**
kron_g500-19	-	-	540.89	**477.80**	647.21	5802.9
tx2010	1730.2	1751.59	60.900	4.5231	1.5755	**1.0388**
kron_g500-21	-	-	**2887.9**	6387.2	10758	-
M6	-	-	1125.5	27.417	10.788	**10.158**
hugetric	390.33	303.30	278.86	24.850	**15.983**	21.588
rgg_n_2_23	-	-	10671	268.30	36.692	**21.553**
hugetrace	2085.5	782.00	406.32	26.107	**22.778**	32.607
nlpkkt200	-	-	-	-	465623	-
hugebubbles	2009.6	1609.9	995.86	65.080	**54.361**	67.233
road_usa	1049.0	669.87	130.04	26.423	19.381	**15.516**
europe_osm	3255.5	947.78	229.28	42.849	**32.874**	35.534
rmat-G500	-	-	-	-	-	-
rmat-SSCA	-	-	-	-	-	-
rmat-ER	-	-	-	-	3535.4	**2032.1**

percentage around 51%, and 25%, respectively, of vertices are unmatched, which increases the number of searches and iterations. The Iterative algorithm runs slower than the Direct-Augmenting algorithm on graphs with high average degree graphs because of the same reason.

Table 6.5.
Exact matching weights and cardinalities.

Graph	Random integers	Random real	Vertex degrees	
		Weight		Cardinality
G34	9.8E5	2.3E3	8.0E3	1,000
G39	9.7E5	2.3E3	2.4E4	1,000
de2010	1.2E7	2.7E4	1.2E5	11,853
shipsec8	5.7E7	1.3E5	6.8E6	57,459
kron_g500-17	4.4E7	9.1E4	1.0E7	38,823
mt2010	6.5E7	1.5E5	6.3E5	63,685
fe_ocean	7.1E7	1.6E5	8.2E5	71,718
tn2010	1.2E8	2.7E5	1.2E6	117,989
kron_g500-19	1.6E8	3.2E5	4.3E7	136,770
tx2010	4.5E8	1.0E6	4.4E6	449,167
kron_g500-21	5.5E8	1.1E6	1.8E8	482,339
M6	1.7E9	4.0E6	2.1E7	1,750,888
hugetric	3.3E9	7.6E6	2.0E7	3,296,382
rgg_n_2_23	4.2E9	9.6E6	1.3E8	4,194,303
hugetrace	6.0E9	1.4E7	3.6E7	6,028,720
nlpkkt200	8.1E9	1.8E7	4.6E8	8,000,000
hugebubbles	9.7E9	2.2E7	5.8E7	9,729,043
road_usa	1.2E10	2.6E7	5.6E7	11,325,669
europe_osm	2.5E10	5.8E7	1.1E8	25,149,787
rmat-G500	1.4E10	2.8E7	-	11,783,556
rmat-SSCA	3.9E10	8.2E7	-	35,024,914
rmat-ER	6.7E10	1.5E8	4.3E9	67,108,864

6.2.2 Approximation Algorithms

Here, we compare the new approximation algorithms, the direct 1/2- and 2/3-approximation algorithms (1/2-Dir and 2/3-Dir), the iterative 1/2- and 2/3- approximation algorithms (1/2-Iter and 2/3-Iter), and the initialized version of the iterative approximation algorithms (1/2-Init-Iter and 2/3-Init-Iter) with several approximation algorithms for MEM. The MEM approximation algorithms we chose are the scaling approximation algorithm, ROMA initialized with a GPA matching, uninitialized ROMA, and Suitor. The scaling algorithm provides the best approximation ratio, and it has not been implemented before. GPA-ROMA and ROMA find high matching weights in practice. Suitor is one of the fastest 1/2-approximation algorithms for MEM. For the direct approximation algorithms and Suitor, we sorted the adjacency lists in non-increasing order of weights, since we saw significant improvements in the running time. The reported running time does not include the adjacency list sorting time.

We compare the relative performance with respect to the Direct-Increasing exact algorithm running time. In addition, We compare the number of scanned edges as a metric that is not influenced by machine specifications. Also, we present the percentage of the main steps performed by several approximation algorithms. For the scaling algorithm, we report the percentage of searching for augmenting paths, dual update, and handling blossoms. For GPA-ROMA, we report the percentages of sorting, searching for paths and cycles, the dynamic programming procedure, and ROMA algorithm time. We report percentages of sorting and searching for augmenting paths in the direct approximation algorithms, and for the iterative approximation algorithms, we report percentages of phase one (computing approximate cardinality matching) and phase two (searching for augmenting and increasing paths). The approximation algorithms compute nearly optimal weights, and in order to differentiate among them, we report the gap to optimality 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. Similarly we report the gap to a maximum cardinality as $100\,(1-|M_A|/|M_{opt}|)$. Since results using different weights types are similar with few exceptions, we report here integer weights results and briefly summarize the results of the other weight types. Results for real and vertex degree weights can be found in the appendix.

In general, 1/2-Iter and 1/2-Init-Iter are the fastest algorithms. 1/2-Dir comes second. Suitor comes third, followed by 2/3-Iter and 2/3-Init-Iter. Finally, there are 2/3-Dir, ROMA, GPA-ROMA, and lastly, the scaling algorithm.

For integer weights, the relative performance of the approximation algorithm with respect to the exact algorithm time is reported in Table 6.6. The 1/2-Init-Iter is the fastest algorithm, being 65.6 times faster than the exact algorithm on the geometric mean. The 1/2-Iter algorithm is about 63.2 times faster, followed by the 1/2-Dir, which is faster by a factor of 52.5. The Suitor algorithm is slower than the other 1/2-approximation algorithms, despite sorting the adjacency lists, and it is faster than the exact by a factor of 25.9. The 2/3-Iter and 2/3-Init-Iter are very close to the Suitor algorithm; they are 25.1 and 24.2 times faster, respectively. The 2/3-Dir and ROMA are 15.0 and 1.2 times faster, all on the geometric mean. The GPA-ROMA and scaling approximation algorithms are slower than the exact algorithm by a factor of 1.5 and 2.9.

For the real weights, the previous ordering of algorithms in terms of running time can be seen, except that the scaling algorithm outperforms GPA-ROMA and ROMA by a factor of 2.4 and 1.2. This is due to the small ranges of real weights. Also, the gap between 1/2-Dir and the fastest algorithm becomes bigger due to the cost of sorting real numbers.

For the degree weights, the fastest algorithm is the 1/2-Dir. It is 103 times faster than the exact algorithm on the geometric mean. The 1/2-Iter and 1/2-Init-Iter algorithm are about as fast as the 1/2-Dir. The 2/3-Iter and 2/3-Init-Iter are faster than the exact by a factor of 46 and they are 1.3 time faster than Suitor. The rest of the algorithms come in the same integer weight order in terms of running time.

We observed a few factors that impact the running time:

1- Graph structure. First, the number of edges clearly impacts the running time for all approximation algorithms. For this reason, rmat-ER and rmat-G500, with about two billion edges, are the problems for which most approximation algorithms needed the highest time. The connectivity of a graph also impacts the running time.

For example, nlpkkt200 with a high average degree will cause a large search tree to be generated even for for relatively short augmenting paths.

2- The number of scanned edges. In Figures 6.1 and 6.2, we show the run times of the 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 primarily determined by the number of edges scanned by the algorithms. There are cases in which the running time is low even when the number of scanned edges is high, and vice versa. For the iterative algorithms, some graphs are easy to match, since more than 99.9% of vertices are matched in the cardinality matching initialization phase. For the Direct-Augmenting algorithms, the time needed for sorting could make the runtime higher. In addition, updating duals and handling blossoms in the scaling algorithm can also increase runtimes. The ratio of the number of scanned edges to the number of edges is reported in Tables 6.8, A.3 and A.8. The scaling algorithm has the highest number of scanned edges, which is due to the high number of iterations. As shown in Table 6.7 the majority of running time is spent in searching for a set of vertex-disjoint augmenting paths. The GPA-ROMA algorithm also has very high number of scanned edges because it searches for the highest gain 2-augmentations among all possible 2-augmentation paths and cycles, from every matched or unmatched vertex. The GPA initialization also searches a large number of edges. It can be seen from Table 6.7 that these two type of searches constitute around 80 % of the running time.

3- Range of weights. This impacts the scaling algorithm, since the number of scales increases with a larger range. For real weights in range [1 1.3], the scaling algorithm outperformed the GPA-ROMA and ROMA algorithms, while for integer weights in range [1 1000], it became the slowest algorithm. For degree weights, the range of the difference between the maximum degree and the minimum degree determines the runtime. The kron g500-logn21 graph has a maximum degree of 213,904 and minimum degree of 1, and the scaling algorithm completes in 844 seconds.

4- Type of weights. This factor impacts algorithms that sort vertices or edges, namely GPA-ROMA, 1/2-Dir and 2/3-Dir. This can be seen in the percentage of time

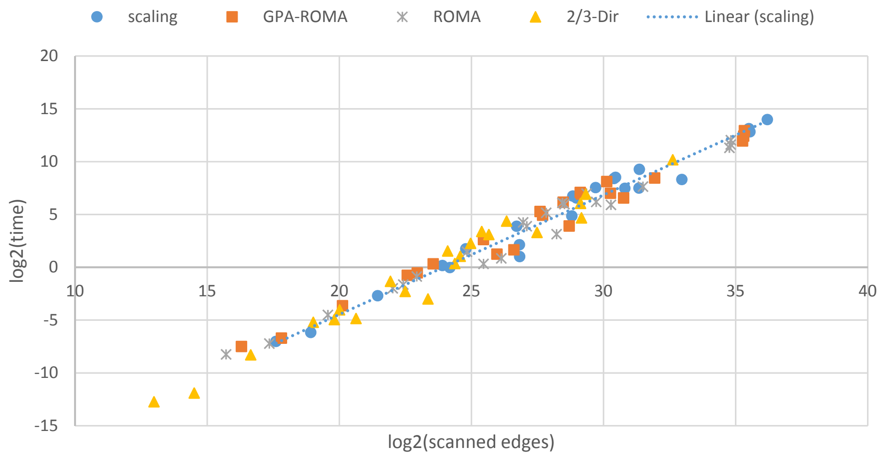

Figure 6.1. Running time plotted against the number of edges scanned by the scaling, GPA-ROMA, ROMA and 2/3-Dir algorithms (plotted on a log-log scale).

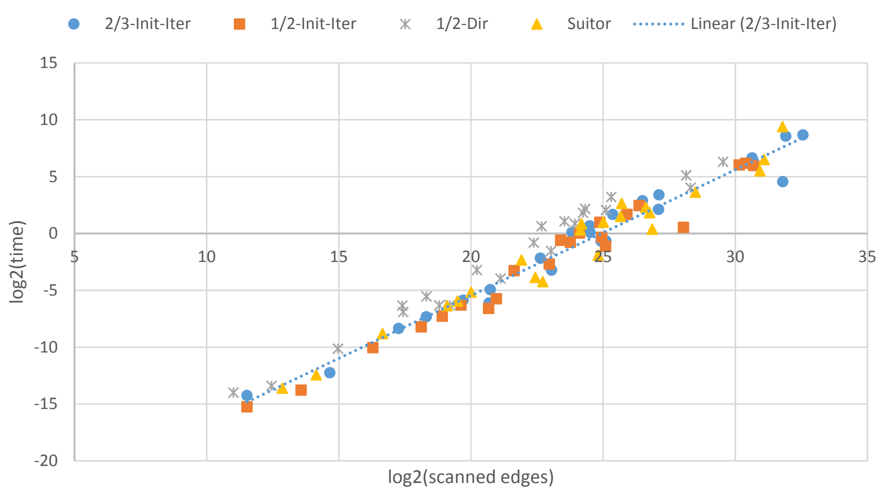

Figure 6.2. Running time plotted against the number of edges scanned by the 2/3-Init-Iter, 1/2-Init-Iter, 1/2-Dir and Suitor algorithms (plotted on a log-log scale).

spent on sorting for these algorithms. For the GPA-ROMA algorithm, sorting real weights increases the percentage to around 5.2 % from 3.5 %. The 1/2-Dir algorithm spent 7.6 % of the time sorting integer weights, while for real weights it used 30% of the running time. The 2/3-Dir algorithm spent 2.7 % of the time sorting integer weights, while for real weights it used 15% of the running time.

5- Sorting adjacency lists. This affected the Suitor, 1/2-Dir and 2/3-Dir algorithms. As mentioned before, all reported results of these algorithms are obtained with the sorted adjacency list, which made a big difference in the running time. The reported times does not include the time to sort the adjacency lists. The Suitor, 1/2-Dir, and 2/3-Dir algorithms gain a speedup of 1.2, 1.2, and 1.1 using integer weights, 1.3, 1.2, and 1.2 using real weights, and 1.8, 1.6, and 2.8, respectively, using degree weights. Sorting adjacency lists in most cases exceeded the matching time (e.g., sorting the adjacency list of rmat-G500 costs around 570 seconds, but the matching is obtained by the Suitor algorithm in 45 seconds).

6- The ratio between phase one and phase two in the 1/2 and 2/3-Init-Iter algorithms. If the percentage of work in phase one is larger than that in phase two, then the algorithm finishes much faster because most of the matching is computed using a fast cardinality matching algorithm, while less work is done in phase two, which is more expensive. As shown in Table 6.7, on rmat-ER and rgg_n_2_23, the 2/3-Init-Iter spent around 80 % of the time in phase one and finished in 99 and 0.6 seconds, respectively. in spite of the large number of edges. On rmat-G500, it spent around 88 % of the time in phase two and finished in 405 seconds. Unfortunately, this cannot be controlled and is determined by both the graph structure and whether a graph admits perfect or close to perfect matching. If we consider the same problems, rmat-ER admits a perfect matching and rgg_n_2_23 admits a perfect matching minus two vertices, while the matching in rmat-G500 is off by around a million vertices from a perfect matching.

Now, we turn to comparing gaps to optimal weights and cardinalities. Overall, the 2/3-Iter and 2/3-Init-Iter algorithms obtain the best weights and cardinalities. GPA-ROMA comes third, then ROMA, followed by the 2/3-Dir algorithm. The 1/2-Iter and 1/2-Init-Iter algorithms obtained better weights and cardinalities than the scaling algorithm. Last are the 1/2-Dir and Suitor algorithms, which have the worst weights and cardinalities for all the weight types.

As shown in Tables 6.9 and 6.10, when the weights are integers the 2/3-Iter and 2/3-Init-Iter approximation algorithms obtain the best weight and cardinality with gaps to optimality of 0.4% and 1.2%, respectively, all on average. The quality of weights and cardinalities of GPA-ROMA comes third, with gaps of 0.7% and 2.3%, while for ROMA, the gaps are 0.8% and 2.5%. For the 2/3-Dir algorithm the gap to optimal weight is 0.9%, and the gap to maximum cardinality is 3.0%, on average. Unexpectedly, the 1/2-Iter and 1/2-Init-Iter algorithms obtained better matchings than the matchings of the scaling algorithm, on average. For both 1/2-Iter and 1/2-Init-Iter algorithms the gap to optimal weight is 2.4% and the gap to optimal cardinality is around 4.1%, while for the scaling algorithm weight gap is 2.5% and the cardinality gap is 6.0%. The Suitor and 1/2-Dir algorithms are the worst, with gaps of 6% and 11%. The same ordering in quality is seen with degree weights, except that the gaps between the algorithms become larger.

For the real weights, again the 2/3-Iter and 2/3-Init-Iter algorithms obtain the best weight and cardinality, with gaps of 0.3% and 1.2% on average, respectively. The scaling algorithm obtains the third best weight, with a gap of 1.0%. The 1/2-Iter and 1/2-Init-Iter algorithms come fourth and fifth, with a gap of 1.1%. The order of the algorithms based on quality of cardinalities follows the same order as that of the integer weights.

6.3 Results from Parallel Algorithms

We implemented the parallel code using C++, OpenMP 3.1, and the g++ compiler functions __sync_lock_test_and_set and __sync_lock_release for locking. We pinned threads to cores to reduce the overhead of thread migration between cores by setting the environment variable GOMP_CPU_AFFINITY="0-($t-1$)", where t is the number of threads. Using 20 threads with 20 cores, thread i is pinned to core i. We used static scheduling and experimented with chunk sizes of 256 and the default value equal to the loop count divided by the number of threads. We report the faster running time from these two options. The test set consists of problems from Table 6.1 where the number of vertices is greater than 2 million.

We compare the 2/3-Iter, 2/3-Init-Iter, 1/2-Iter and 1/2-Init-Iter approximation algorithms and the Suitor algorithm. Suitor is known to be the most concurrent approximation algorithm for edge-weighted matching since it processes vertices in arbitrary order, and vertices are free to offer proposals to their highest weight available neighbor.

We report running times (in seconds) and speedups in Tables 6.11 and 6.12. The speedup for a particular α-approximation algorithm is computed as the ratio of the time of the *fastest* serial algorithm among all α-approximation algorithms to the time needed on twenty threads by the parallel α-approximation algorithm under consideration, for $\alpha = 2/3, 1/2$. Thus, the baseline serial algorithm could be different for algorithms with differing approximation ratios.

When integer weights are used, clearly the 1/2-Iter and 1/2-Init-Iter algorithms are the fastest parallel algorithms on average. Both are faster than the Suitor algorithm on all but two problems. The uninitialized variant of the iterative algorithm is faster than the initialized variant for the 1/2-approximation algorithm, while the ordering is reversed for the 2/3-approximation iterative algorithms. The 1/2-Iter and 1/2-Init-Iter algorithms also achieved higher speedups, 8.9 and 7.4, respectively, in the geometric mean on these problems; the speedup of the Suitor algorithm was 3.5. The fastest 2/3-approximation algorithm is slower than the fastest 1/2-approximation algorithm in parallel as well, by a factor of 2.50 in the geometric mean. The speedup of 2/3-Iter and 2/3-Init-Iter are 9.7 and 10.7, respectively. There are four problems in which the fastest 2/3-approximation algorithm is faster than the 1/2-approximate Suitor algorithm, which is surprising. We obtained similar results using real weights. For degree weights, the 2/3-Iter and 1/2-Iter algorithms show the best speedups, and Suitor is the worst with speedup of 0.9.

We define scalability to be the ratio of the time of a serial approximation algorithm to the time taken by that algorithm in parallel on twenty threads. For integer weights, the 2/3-Init-Iter algorithm scales well on twenty threads and is slightly better than the Suitor algorithm in geometric mean (11.1 for the former and 10.2 for the latter). The 1/2-Init-Iter algorithm scales slightly worse than Suitor, again in the geometric mean. We obtained similar results for real weights, except that Suitor obtains the best ratio of 10.5. When degree weights are used, Suitor scales worse than all other algorithms, with a ratio of 5.1 on the geometric mean, while 1/2-Iter shows the best scalability with a ratio of 9.1. The parallel Suitor algorithm obtains the same weight as the single thread implementation, while the weights differ slightly in geometric mean by 0.02% for the parallel 2/3-Iter and 2/3-Init-Iter, and 0.2% for the parallel 1/2-Iter and 1/2-Init-Iter algorithms. The difference is due to different initial maximal matchings obtained in each run. However, the weight and cardinality of matchings from the parallel iterative approximation algorithms are always better than that of the parallel Suitor algorithm. The invariance of the matching obtained in serial and parallel is an advantage of the Suitor algorithm.

Table 6.6.
Relative performance w.r.t the Direct-Increasing MVM algorithm running time. Vertex weights are random integers in the range [1 1000].

Graph	$1-\epsilon$, $\epsilon=1/3$	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	Scal.	GPA-ROMA	ROMA	Dir	Iter	Init-Iter	Dir	Iter	Init-Iter	Suitor
G34	0.022	0.031	0.052	1.148	3.240	3.306	2.787	5.093	**6.635**	2.116
G39	0.069	0.099	0.141	3.617	5.393	4.633	10.22	11.34	**13.39**	5.263
de2010	0.072	0.140	0.254	3.510	5.233	3.661	**12.55**	12.02	11.81	5.028
shipsec8	0.130	0.111	0.209	2.077	15.83	18.12	21.11	19.74	**25.38**	3.811
kron_g500-17	1.684	2.347	4.081	213.0	80.80	69.28	**585.3**	394.5	397.2	139.6
mt2010	0.084	0.159	0.373	3.455	6.881	5.486	11.27	**15.75**	14.53	7.727
fe_ocean	0.645	0.910	1.953	19.49	101.5	101.5	51.13	175.1	**187.7**	38.67
tn2010	0.091	0.242	0.548	4.891	11.50	9.272	14.02	**25.68**	23.83	10.77
kron_g500-19	2.365	4.577	7.791	331.9	120.5	109.01	**1065**	465.1	441.0	256.7
tx2010	0.072	0.172	0.380	2.647	5.45	4.773	9.807	**11.21**	10.27	5.34
kron_g500-20	3.869	7.293	11.37	537.9	178.4	159.2	**2.0E3**	933.4	867.4	528.1
M6	0.107	0.329	0.674	4.871	8.63	9.458	**17.39**	17.11	17.16	5.474
hugetric	0.200	0.550	1.112	7.313	18.92	20.01	13.72	**32.99**	32.37	17.20
rgg_n_2_23	0.116	0.164	0.288	2.136	25.80	33.38	11.54	36.08	**46.14**	5.937
hugetrace	0.173	0.453	0.903	6.675	18.93	20.23	15.37	**34.46**	31.43	16.17
nlpkkt200	1.8E3	1.7E3	2.9E3	2.3E4	2.3E4	2.5E4	1.4E5	3.4E5	**4.1E5**	4.7E4
hugebubbles	0.198	0.541	1.049	7.720	19.69	20.84	18.78	31.70	**33.77**	10.61
road_usa	0.041	0.110	0.217	1.423	2.155	2.027	3.309	4.826	4.675	**5.266**
europe_osm	0.052	0.115	0.252	1.544	3.262	3.024	3.430	6.136	5.841	**6.456**
rmat-G500	8.268	15.10	23.45	903.7	156.9	146.9	**3.7E3**	999.5	954.9	1312
rmat-SSCA	3.767	6.151	10.63	276.5	100.6	89.08	**968.9**	533.1	515.1	367.6
rmat-ER	0.103	0.215	0.398	1.439	13.70	16.73	21.05	18.92	**23.22**	2.492
Geom. Mean	0.348	0.647	1.212	14.96	25.09	24.16	52.52	63.25	**65.57**	25.94

Table 6.7.
Percentage of time taken by the major steps in the approximation algorithms. Random integer weights in [1 1000]. The remaining time is spent in variable declarations and initializations.

Graph	$1-\epsilon$, $\epsilon = 1/3$ Scal.			$2/3-\epsilon$, $\epsilon = 0.01$ GPA-ROMA				2/3-Dir		2/3-Init-Iter		1/2-Dir		1/2-Init-Iter	
	Srch.	Blsm	Duals	Sort	Srch.	DP.	2-aug	Sort	Srch.	P. 1	P. 2	Sort	Srch.	P. 1	P. 2
G34	73.8	0.00	16.7	0.86	28.2	0.85	58.6	5.90	64.2	22.4	4.40	11.4	27.9	1.93	19.9
G39	83.8	0.15	10.6	2.96	15.4	0.51	73.4	3.56	77.3	38.6	36.5	8.76	44.9	28.0	24.3
de2010	76.7	0.67	14.1	1.73	31.5	1.60	58.1	4.58	65.2	29.6	65.6	17.9	58.7	60.2	28.5
shipsec8	90.7	0.60	3.7	9.60	28.5	0.67	59.3	0.66	98.3	92.5	3.58	5.56	87.0	2.3	93.2
kron_g500-17	94.0	0.00	2.70	10.3	26.4	0.31	60.6	2.57	93.2	20.7	78.7	6.35	82.5	62.0	34.5
mt2010	80.0	0.61	10.9	1.36	42.5	2.46	48.4	3.21	78.9	34.1	61.8	7.92	82.0	59.5	33.0
fe_ocean	82.8	0.00	9.29	1.60	41.4	1.86	50.4	2.87	85.1	49.4	39.0	7.51	83.1	15.4	65.5
tn2010	85.7	0.41	6.58	1.81	45.7	1.89	46.1	2.59	76.5	31.2	65.7	6.79	84.2	57.4	35.0
kron_g500-19	95.1	0.00	1.85	9.06	27.3	0.27	60.9	1.72	94.8	17.0	82.5	5.57	83.8	80.3	17.7
tx2010	81.8	0.58	7.43	2.34	44.3	1.65	46.2	1.61	68.2	18.1	80.5	5.56	87.0	78.5	18.3
kron_g500-21	96.7	0.00	1.16	6.02	22.3	0.18	69.6	2.01	95.7	12.8	86.9	7.35	84.1	83.1	15.6
M6	81.1	1.01	9.48	2.18	40.7	1.63	50.7	2.39	78.4	37.1	61.4	8.98	83.8	74.5	23.0
hugetric	84.5	0.40	4.81	1.61	38.3	1.98	49.9	3.93	87.0	37.0	59.7	8.39	82.3	72.9	22.3
rgg_n_2_23	83.3	1.79	6.84	3.04	30.9	0.66	60.4	1.38	84.6	78.9	14.9	6.75	87.5	16.1	75.0
hugetrace	84.2	0.38	4.95	1.59	39.1	2.10	49.0	3.72	88.0	37.6	58.1	8.43	83.2	56.0	40.5
nlpkkt200	89.3	0.00	3.68	4.12	28.4	0.68	62.2	0.99	98.2	3.45	96.2	5.34	90.6	60.6	33.8
hugebubbles	84.9	0.39	4.50	1.51	38.7	2.21	52.7	3.47	88.7	35.2	61.8	7.95	86.4	70.2	25.7
road_usa	84.9	0.39	5.77	1.26	37.4	1.86	47.5	3.51	84.6	17.5	80.8	9.44	83.2	82.6	13.8
europe_osm	83.7	0.32	6.10	1.24	46.2	1.69	41.4	3.70	86.4	14.2	83.5	8.53	84.6	80.0	14.9
rmat-G500	96.9	0.00	0.62	5.25	27.7	0.34	64.2	2.39	94.4	11.5	88.4	5.08	87.5	90.6	9.03
rmat-SSCA	96.0	0.00	0.91	5.05	29.8	0.61	63.6	1.78	95.5	15.5	84.4	4.65	90.1	83.5	15.7
rmat-ER	96.8	0.03	0.75	3.14	35.4	1.17	60.9	0.19	99.6	77.3	22.0	3.92	89.1	32.2	66.9
Arith. Mean	86.7	0.35	6.06	3.53	33.9	1.23	56.1	2.67	85.6	33.3	59.8	7.64	79.7	56.7	33.0

Table 6.8.
The ratios of the number of scanned edges by approximation algorithms to $|E|$. Random integer weights in [1 1000].

Graph	$1-\epsilon$, $\epsilon = 1/3$ Scal.	$2/3-\epsilon$, $\epsilon = 0.01$ GPA-RO.	RO.	2/3-approx Dir	Iter	Init-Iter	1/2-approx Dir	Iter	Init-Iter	Suitor
G34	50.0	20.2	13.5	2.03	0.74	0.74	0.51	0.74	0.74	1.85
G39	42.3	19.5	14.2	1.98	1.64	2.20	0.48	0.89	1.03	1.53
de2010	49.7	19.7	13.4	1.77	2.32	2.72	0.55	1.28	1.38	1.77
shipsec8	36.6	20.1	14.1	3.29	0.51	0.51	0.14	0.51	0.51	1.73
kron_g500-17	23.2	20.1	14.4	0.32	1.27	1.70	0.12	0.33	0.40	1.34
mt2010	49.3	19.6	13.3	1.66	2.25	2.68	0.55	1.32	1.55	1.75
fe_ocean	46.7	19.9	13.7	2.25	0.78	0.79	0.42	0.69	0.70	1.80
tn2010	48.7	20.7	13.4	1.77	2.54	2.92	0.55	1.13	1.35	1.76
kron_g500-19	21.6	20.1	14.4	0.27	1.23	1.46	0.10	0.36	0.38	1.35
tx2010	49.5	20.7	13.4	1.80	2.52	2.91	0.55	1.25	1.46	1.77
kron_g500-20	20.7	20.1	14.3	0.24	1.19	1.58	0.09	0.30	0.36	1.33
M6	50.4	20.8	13.7	2.39	1.96	2.28	0.52	1.19	1.35	1.80
hugetric	48.6	20.6	13.2	1.82	1.36	1.50	0.67	1.02	1.12	1.87
rgg_n_2_23	43.1	20.3	14.0	2.96	0.57	0.57	0.25	0.56	0.57	1.79
hugetrace	48.5	20.7	13.2	1.82	1.20	1.30	0.67	0.93	1.00	1.87
nlpkkt200	39.0	19.1	14.0	2.80	16.9	17.3	0.17	1.26	1.28	1.75
hugebubbles	48.6	20.7	13.2	1.82	1.36	1.48	0.68	0.93	1.06	1.87
road_usa	51.1	20.3	12.6	1.53	2.83	3.27	0.72	1.90	2.18	1.84
europe_osm	51.0	21.7	12.7	1.56	2.49	2.69	0.77	1.49	1.60	1.90
rmat-G500	24.1	19.9	14.0	0.28	2.57	3.05	0.16	0.75	0.84	0.98
rmat-SSCA	23.1	20.1	14.3	0.32	1.53	1.91	0.14	0.54	0.56	1.08
rmat-ER	36.9	20.2	14.1	3.08	0.71	0.78	0.36	0.61	0.66	1.73
Geo. Mean	39.2	20.2	13.7	1.32	1.59	1.80	0.34	0.81	0.89	1.63

Table 6.9.
The gap to optimality of the weights of the matching obtained from the approximation algorithms. Vertex weights are random integers in the range [1 1000].

Graph	$1-\epsilon$,	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	$\epsilon=1/3$	GPA-				Init-			Init-	
	Scal.	RO.	RO.	Dir	Iter	Iter	Dir	Iter	Iter	Su.
G34	1.89	0.31	0.30	0.47	**0.00**	**0.00**	2.88	**0.00**	**0.00**	2.88
G39	1.63	0.04	0.06	0.06	0.02	**0.02**	2.92	1.34	1.27	2.92
de2010	3.46	0.90	0.93	0.99	**0.52**	0.53	6.75	3.14	3.15	6.75
shipsec8	0.02	0.00	0.00	0.00	**0.00**	**0.00**	0.05	0.00	0.00	0.05
kron_g500-17	5.82	1.97	2.09	2.13	**1.00**	1.02	14.47	6.28	6.34	14.5
mt2010	3.86	1.10	1.14	1.21	**0.63**	0.64	7.61	3.48	3.51	7.61
fe_ocean	1.06	0.14	0.15	0.21	**0.02**	0.02	2.26	0.10	0.10	2.26
tn2010	3.46	0.95	0.99	1.04	**0.54**	0.55	7.02	3.16	3.19	7.02
kron_g500-19	5.29	1.72	1.81	1.95	**0.82**	0.84	14.33	5.81	5.83	14.3
tx2010	2.25	0.83	0.87	0.94	**0.47**	0.48	6.40	2.86	2.88	6.40
kron_g500-20	5.05	1.52	1.61	1.79	**0.70**	0.72	14.09	5.41	5.44	14.1
M6	0.70	0.16	0.16	0.21	**0.08**	0.08	2.39	0.89	0.91	2.39
hugetric	1.57	0.50	0.64	0.81	**0.17**	0.17	4.43	0.79	0.86	4.43
rgg_n_2_23	0.20	0.03	0.03	0.03	**0.00**	0.00	0.56	0.00	0.00	0.56
hugetrace	1.56	0.49	0.62	0.79	**0.13**	0.13	4.39	0.59	0.65	4.39
nlpkkt200	0.30	0.14	0.15	**0.07**	0.19	0.19	0.08	0.49	0.49	0.08
hugebubbles	1.57	0.50	0.63	0.81	**0.15**	0.15	4.42	0.68	0.75	4.42
road_usa	3.35	1.16	1.50	1.74	**0.91**	0.92	7.81	4.13	4.17	7.81
europe_osm	3.08	**0.53**	1.81	2.00	0.77	0.77	6.77	1.80	1.83	6.77
rmat-G500	4.69	1.03	1.08	1.35	**0.48**	0.50	13.1	4.86	4.82	13.1
rmat-SSCA	5.16	1.50	1.59	1.84	**0.89**	0.90	13.3	5.95	5.96	13.3
rmat-ER	0.07	0.00	0.00	0.00	**0.00**	0.00	0.19	0.02	0.03	0.19
Arith. Mean	2.55	0.70	0.82	0.93	**0.39**	0.39	6.19	2.35	2.37	6.19

Table 6.10.
The gap to optimality of the cardinality of the matching obtained from the approximation algorithms. Vertex weights are random integers in the range [1 1000].

Graph	$1-\epsilon$, $\epsilon = 1/3$ Scal.	$2/3-\epsilon$, $\epsilon = 0.01$ GPA-RO.	RO.	2/3-approx Dir	Iter	Init-Iter	1/2-approx Dir	Iter	Init-Iter	Su.
G34	7.59	2.53	2.53	3.58	**0.00**	0.00	9.01	0.00	0.00	9.01
G39	7.29	1.03	1.13	1.22	**0.69**	0.70	9.74	5.32	5.16	9.74
de2010	9.86	3.98	4.05	4.42	**2.75**	2.79	14.07	7.91	7.94	14.07
shipsec8	0.91	0.14	0.15	0.17	**0.00**	0.00	1.13	0.00	0.00	1.13
kron_g500-17	6.82	2.90	3.06	2.85	**1.51**	1.56	16.93	7.36	7.49	16.93
mt2010	9.96	4.08	4.15	4.56	**2.75**	2.78	14.52	7.93	7.99	14.52
fe_ocean	6.10	1.75	1.77	2.41	**0.19**	0.19	8.19	0.29	0.30	8.19
tn2010	9.84	4.06	4.17	4.50	**2.72**	2.76	14.41	7.90	7.98	14.41
kron_g500-19	5.65	2.40	2.51	2.45	**1.16**	1.20	16.24	6.54	6.60	16.24
tx2010	7.35	3.73	3.81	4.26	**2.50**	2.53	13.55	7.34	7.42	13.55
kron_g500-20	5.16	2.10	2.21	2.21	**0.98**	1.01	15.69	5.95	6.01	15.69
M6	4.75	1.92	1.97	2.36	**1.28**	1.30	8.48	4.32	4.39	8.48
hugetric	6.96	3.34	3.54	4.63	**1.16**	1.18	11.61	2.50	2.70	11.61
rgg_n_2_23	2.57	0.77	0.79	0.93	**0.00**	0.00	3.85	0.02	0.04	3.85
hugetrace	6.88	3.26	3.45	4.55	**0.90**	0.92	11.51	1.87	2.04	11.51
nlpkkt200	1.53	0.17	0.17	0.13	**0.00**	0.00	2.26	0.00	0.00	2.26
hugebubbles	6.94	3.32	3.52	4.61	**1.02**	1.04	11.58	2.16	2.34	11.58
road_usa	7.09	3.34	3.81	4.59	**2.43**	2.46	13.37	7.23	7.32	13.37
europe_osm	8.22	**1.91**	5.23	6.29	1.96	1.98	13.57	3.13	3.24	13.57
rmat-G500	4.02	1.31	1.37	1.50	**0.60**	0.62	13.84	4.95	4.91	13.84
rmat-SSCA	5.85	2.31	2.42	2.55	**1.36**	1.39	15.69	6.90	6.92	15.69
rmat-ER	1.60	0.06	0.07	0.08	**0.03**	0.03	2.31	0.73	0.77	2.31
Arith. Mean	6.04	2.29	2.54	2.95	1.18	1.20	10.98	4.11	4.16	10.98

Table 6.11.
2/3-approximation algorithms run time (seconds) and speedup obtained with twenty threads. Vertex weights are random integers in the range [1 1000].

	2/3-Iter		2/3-Init-Iter	
Graph	Time	Speed-up	Time	Speed-up
kron_g500-21	0.27	**14.1**	0.29	13.4
M6	0.14	7.81	0.12	**9.10**
hugetric	0.12	9.14	0.11	10.1
rgg_n_2_23	0.07	**9.16**	0.07	8.70
hugetrace	0.18	8.79	0.17	**9.26**
nlpkkt200	3.22	7.24	1.81	**12.9**
hugebubbles	0.41	7.83	0.36	**8.70**
road_usa	0.66	10.5	0.62	**11.1**
europe_osm	0.84	11.5	0.74	**13.1**
rmat-G500	22.6	**16.8**	23.8	15.9
rmat-SSCA	25.6	13.0	24.5	**13.5**
rmat-ER	17.0	5.84	14.9	**6.70**
Geom. Mean		9.72		**10.7**

Table 6.12.
1/2-approximation algorithms run time (seconds) and speedup obtained with twenty threads. Vertex weights are random integers in the range [1 1000].

	1/2-Iter		1/2-Init-Iter		Suitor	
Graph	Time	Speed-up	Time	Speed-up	Time	Speed-up
kron_g500-21	0.06	**11.3**	0.10	7.65	0.18	4.18
M6	0.07	**8.57**	0.08	7.39	0.13	4.54
hugetric	0.06	**10.3**	0.08	8.04	0.14	4.63
rgg_n_2_23	0.07	6.19	0.06	**7.12**	0.30	1.52
hugetrace	0.11	**8.53**	0.14	6.72	0.24	3.91
nlpkkt200	0.19	**7.65**	0.24	6.01	1.50	0.96
hugebubbles	0.19	**10.5**	0.24	8.15	0.41	4.75
road_usa	0.31	5.66	0.45	3.90	0.22	**7.88**
europe_osm	0.45	**6.91**	0.65	4.80	0.47	6.65
rmat-G500	3.60	**16.5**	4.22	14.1	8.39	7.10
rmat-SSCA	5.10	**12.3**	5.69	11.0	14.0	4.49
rmat-ER	9.67	7.41	8.29	8.65	72.7	0.99
Geom. Mean		**8.91**		7.40		3.54

Table 6.13.
Scalability of parallel approximation algorithms using 20 threads. Vertex weights are random integers in the range [1 1000].

Graph	2/3-Iter	2/3-Init-Iter	1/2-Iter	1/2-Init-Iter	Suitor
kron_g500-21	14.1	**15.0**	11.3	8.23	14.6
M6	8.56	9.10	8.60	7.39	**10.6**
hugetric	9.66	10.1	**10.3**	8.19	6.7
rgg_n_2_23	11.9	8.70	7.92	7.12	**13.4**
hugetrace	**9.39**	9.26	8.53	7.37	6.2
nlpkkt200	7.75	**12.9**	9.13	6.01	12.1
hugebubbles	8.28	8.70	**11.2**	8.15	7.8
road_usa	10.5	**11.8**	10.0	7.09	7.9
europe_osm	11.5	**14.2**	11.5	8.41	6.7
rmat-G500	16.8	**17.0**	16.5	14.8	14.3
rmat-SSCA	13.0	**15.3**	12.3	11.4	14.5
rmat-ER	7.14	6.70	9.10	8.65	**13.8**
Geom. Mean	10.4	**11.1**	10.3	8.3	10.2

7 CONCLUSIONS

We have studied the vertex-weighted matching problem (MVM), characterized exact and approximate solutions, and designed a number of new exact, approximation, and parallel algorithms.

We designed two exact algorithms for MVM on non-bipartite graphs whose time complexities are $O(mn)$ and $O(\Delta mn)$. The new exact algorithms have been implemented and compared with maximum edge-weighted matching algorithms from LEDA, and an earlier exact algorithm, the Direct-Augmenting algorithm. The results show that the fastest new algorithm outperforms the fastest variant of LEDA implementations by a factor of 28 on average and the Direct-Augmenting algorithm by a factor of 15 on average on our test problems.

We established the approximation ratio of the 2/3-Dir algorithm. The proof is quite involved and is based on new techniques: we distinguish between two types of matched vertices, origins and terminuses, and then establish a relationship between origins, terminuses, and failed vertices, which are vertices matched by the exact algorithm but unmatched by the approximation algorithm. We show that for each failed vertex, there are two distinct matched vertices in the approximate matching that are at least as heavy as the failed vertex. These two vertices may not be on the same alternating path in the graph induced by the symmetric difference of the matchings computed by the exact and approximation algorithms.

We proved that if a graph does not admit an augmenting path of length $2k-1$ and weight-increasing path of length $2k$ with respect to a matching M, then the weight of M is at least a fraction $k/(k+1)$ of the maximum matching. We designed iterative approximation algorithms that satisfy such sufficient conditions and achieve the $k/(k+1)$-approximation ratio.

The new algorithms have been implemented and compared with several MEM approximation algorithms. The results show that the new 2/3-approximation algorithms obtained better matching quality in terms of weight and cardinality than all MEM algorithms on three different sets of weights: integer and real values, and degrees. The gap to optimal weight is around 0.1 %, and the gap to optimal cardinality is around 0.99 % on our test problems. In addition, the new 2/3-approximation algorithm rquired times comparable to the 1/2-approximation algorithms for MEM, and much faster than $2/3 - \epsilon$ and $1 - \epsilon$ MEM approximation algorithms designed earlier. The new 1/2-approximation algorithms run much faster than all MEM approximation algorithms. These results show that the MVM approximation algorithms perform better than the more general MEM approximation algorithms. This is because we exploit the structure of the vertex-weighted matching problem.

We designed new parallel 1/2- and 2/3-approximation algorithms based on the iterative approach. To our knowledge, this is the *first* parallel algorithm for approximating a weighted matching problem with *approximation ratio better than* 1/2. We designed a new method for locking augmenting and weight-increasing paths to ensure correctness of the parallel algorithm. We proved that the locking technique does not cause deadlock, livelock or starvation. We implemented the parallel approximation algorithms using OpenMP on a shared-memory multi-core machine and compared the results with the parallel Suitor algorithm. The results show the new 2/3-approximation algorithms scale very well on the 20 threads we have used. The runtimes are close to that of the Suitor algorithm, while the parallel MVM algorithm obtains greater matching weights. The new parallel 1/2-approximation algorithms run faster than the Suitor algorithm, and again obtain greater weight and cardinality. In practice Suitor is currently the fastest 1/2-approximation algorithm on both serial and parallel computers for the MEM problem.

Now we will discuss future work and directions arising from our results.

- A $k/(k+1)$-approximate cardinality matching can be found in $O(km)$ using k rounds. It would be interesting to investigate a similar approach to achieve

$O(km)$ running time for the MVM problem. It is not obvious if it is possible to find a set of vertex-disjoint augmenting and weight-increasing paths of length $2i-1$ and $2i$ in the i-th round in $O(m)$ time.

- We will explore the b-matching problem on vertex-weighted graphs. Can we further exploit the graph structure and find better approximations than the 1/2-approximate edge-weighted b-matching [83, 84]? Are there advantages in using our direct or iterative approach?

- One could consider designing and implementing a distributed-memory parallel approximation algorithm using the iterative approach considered here, and running experiments on hundreds to thousands of cores.

REFERENCES

[1] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[2] Aranyak Mehta. Online matching and ad allocation. *Foundations and Trends in Theoretical Computer Science*, 8(4):265–368, 2012.

[3] Vahid Tabatabaee, Leonidas Georgiadis, and Leandros Tassiulas. QoS provisioning and tracking fluid policies in input queueing switches. *IEEE/ACM Transactions on Networking*, 9(5):605–617, 2001.

[4] Gagan R. Gupta, Sujay Sanghavi, and Ness B. Shroff. Node weighted scheduling. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 97–108. ACM, 2009.

[5] Bo Ji and Yu Sang. Throughput characterization of node-based scheduling in multihop wireless networks: A novel application of the Gallai-Edmonds structure theorem. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 41–50. ACM, 2016.

[6] Gagan Raj Gupta, Sujay Sanghavi, and Ness Shroff. Is it enough to drain the heaviest bottlenecks? In *2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 483–490. IEEE, 2009.

[7] Bo Ji, Gagan R. Gupta, and Yu Sang. Node-based service-balanced scheduling for provably guaranteed throughput and evacuation time performance. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[8] Gagan Raj Gupta. *Delay Efficient Control Policies for Wireless Networks*. PhD thesis, Purdue University, 2009.

[9] Bo Ji and Jie Wu. Node-based scheduling with provable evacuation time. In *2015 49th Annual Conference on Information Sciences and Systems*, pages 1–6. IEEE, 2015.

[10] Yu Sang, Gagan R. Gupta, and Bo Ji. Node-based service-balanced scheduling for provably guaranteed throughput and evacuation time performance. *IEEE Transactions on Mobile Computing*, 17(8):1938–1951, 2017.

[11] Colin E. Bell. Weighted matching with vertex weights: An application to scheduling training sessions in NASA space shuttle cockpit simulators. *European Journal of Operational Research*, 73(3):443–449, March 1994.

[12] Thomas F. Coleman and Alex Pothen. The null space problem II. algorithms. *SIAM Journal on Algebraic Discrete Methods*, 8(4):544–563, 1987.

[13] Ali Pinar, Edmond Chow, and Alex Pothen. Combinatorial algorithms for computing column space bases that have sparse inverses. *Electronic Transactions on Numerical Analysis*, 22:122–145, 2006.

[14] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.

[15] Sergey Bereg, Alexander E. Holroyd, Lev Nachmanson, and Sergey Pupyrev. Drawing permutations with few corners. In *International Symposium on Graph Drawing*, pages 484–495. Springer, 2013.

[16] Ravindra K. Ahuja and James B. Orlin. A faster algorithm for the inverse spanning tree problem. *Journal of Algorithms*, 34(1):177–193, 2000.

[17] Radoslaw Cymer. Weighted matching as a generic pruning technique applied to optimization constraints. *Annals of Operations Research*, 217(1):165–211, 2014.

[18] Radoslaw Cymer. *Applications of Matching Theory in Constraint Programming.* PhD thesis, University of Hanover, 2013.

[19] Gagan Aggarwal, Gagan Goel, Chinmay Karande, and Aranyak Mehta. Online vertex-weighted bipartite matching and single-bid budgeted allocations. In *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1253–1264. SIAM, 2011.

[20] Zhiyi Huang, Zhihao Gavin Tang, Xiaowei Wu, and Yuhao Zhang. Online vertex-weighted bipartite matching: Beating $1-1/e$ with random arrivals. *ACM Transactions on Algorithms*, 15(3):38, 2019.

[21] Thomas H. Spencer and Ernst W. Mayr. Node weighted matching. In *International Colloquium on Automata, Languages, and Programming*, pages 454–464. Springer, 1984.

[22] Florin Dobrian, Mahantesh Halappanavar, Alex Pothen, and Ahmed Al-Herz. A 2/3-approximation algorithm for vertex weighted matching in bipartite graphs. *SIAM Journal on Scientific Computing*, 41(1):A566–A591, 2019.

[23] Mahantesh Halappanavar. *Algorithms for Vertex-Weighted Matching in Graphs.* PhD thesis, Old Dominion University, 2009.

[24] Kurt Mehlhorn, Stefan Naher, and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[25] Anonymous. LEDA 6.5 Description. Accessed: 10/22/2018.

[26] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *International Colloquium on Automata, Languages, and Programming*, pages 531–543. Springer, 2004.

[27] Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 491–500. ACM, 2019.

[28] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM*, 61(1):1, 2014.

[29] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE, 2014.

[30] Ahmed Al-Herz and Alex Pothen. A 2/3-approximation algorithm for vertex-weighted matching. *Discrete Applied Mathematics*, 2019. Published online Oct. 19, 2019. DOI 10.1016/j.dam.2019.09.013.

[31] Ahmed Al-Herz and Alex Pothen. A parallel 2/3-approximation algorithm for vertex-weighted matching. In *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 2020. To appear.

[32] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24. Springer, 2003.

[33] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[34] László Lovász and Michael D. Plummer. *Matching Theory*, volume 367. American Mathematical Society, 2009.

[35] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Courier Corporation, 2001.

[36] Michael O. Ball, Thomas L. Magnanti, Clyde L. Monma, and George L. Nemhauser. *Network Models*. Elsevier, 1995.

[37] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842, 1957.

[38] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for general graph matching problems. *Journal of the ACM*, 38(4):815–853, 1991.

[39] Nicholas J. A. Harvey. Algebraic algorithms for matching and matroid problems. *SIAM Journal on Computing*, 39(2):679–702, 2009.

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

[41] Piotr Sankowski. Maximum weight bipartite matching in matrix multiplication time. *Theoretical Computer Science*, 410(44):4480–4488, 2009.

[42] Marek Cygan, Harold N. Gabow, and Piotr Sankowski. Algorithmic applications of Baur-Strassens theorem: Shortest cycles, diameter, and matchings. *Journal of the ACM*, 62(4):28, 2015.

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

[44] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[45] Harold N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, 23(2):221–234, 1976.

[46] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[47] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science*, pages 17–27. IEEE, 1980.

[48] Vijay V. Vazirani. A proof of the MV matching algorithm. *Unpublished manuscript*, 2014.

[49] Harold N. Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs.* PhD thesis, Stanford University, Stanford, CA, USA, 1974.

[50] Michael O. Ball and Ulrich Derigs. An analysis of alternative strategies for implementing matching algorithms. *Networks*, 13(4):517–549, 1983.

[51] Zvi Galil, Silvio Micali, and Harold Gabow. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM Journal on Computing*, 15(1):120–130, 1986.

[52] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[53] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for weighted matching in general graphs. *ACM Transactions on Algorithms*, 14(1):8, 2018.

[54] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[55] Andrew L. Dulmage and Nathan S. Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10(4):516–534, 1958.

[56] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011.

[57] Ding-Zhu Du, Ker-I Ko, and Xiaodong Hu. *Design and Analysis of Approximation Algorithms.* Springer, 2011.

[58] Vijay V. Vazirani. *Approximation Algorithms.* Springer, 2013.

[59] Dorit S. Hochbaum. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Co., 1996.

[60] Alex Pothen, S. M. Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

[61] Stefan Hougardy. Linear time approximation algorithms for degree constrained subgraph problems. In William J. Cook, Laszlo Lovasz, and Jens Vygen, editors, *Research Trends in Combinatorial Optimization*, pages 185–200. Springer, 2009.

[62] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.

[63] Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 259–269. Springer, 1999.

[64] Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[65] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *International Workshop on Experimental and Efficient Algorithms*, pages 242–255. Springer, 2007.

[66] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[67] Seth Pettie and Peter Sanders. A simpler linear time 2/3-ϵ approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004.

[68] Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 673–682. IEEE, 2010.

[69] Sven Hanke and Stefan Hougardy. *New approximation algorithms for the weighted matching problem*. Forschungsinst. für Diskrete Mathematik, 2010.

[70] William T. Tutte. The factorization of linear graphs. *Journal of the London Mathematical Society*, 1(2):107–111, 1947.

[71] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[72] Ariful Azad, Aydin Buluc, and Alex Pothen. Computing maximum cardinality matchings in parallel on bipartite graphs via tree grafting. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):44–59, 2017.

[73] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1995.

[74] Burkhard Monien, Robert Preis, and Ralf Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609–1634, 2000.

[75] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[76] Iain Duff and Bora Uçar. Combinatorial problems in solving linear systems. *Combinatorial Scientific Computing*, page 21, 2012.

[77] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[78] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[79] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[80] D. Bader, Kamesh Madduri, John Gilbert, Viral Shah, Jeremy Kepner, Theresa Meuse, and Ashok Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2:1–10, 2006.

[81] Kurt Mehlhorn and Guido Schäfer. Implementation of $O(nm \log n)$ weighted matchings in general graphs: the power of data structures. *Journal of Experimental Algorithmics*, 7:4, 2002.

[82] David Applegate and William Cook. Solving large scale matching problems. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 12, pages 557–576. American Mathematical Society, 1993.

[83] Arif Khan, Alex Pothen, Md. Mostofa Patwary, Nadathur Satish, Narayanan Sunderam, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-Matching. *SIAM Journal on Scientific Computing*, 38:S593–S619, 2016.

[84] Arif Khan, Alex Pothen, Md. Mostofa Ali Patwary, Mahantesh Halappanavar, Nadathur Rajagopalan Satish, Narayanan Sundaram, and Pradeep Dubey. Designing scalable b-Matching algorithms on distributed memory multiprocessors by approximation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 773–783. IEEE, 2016.

A RESULTS USING REAL-VALUES AND VERTEX DEGREES AS WEIGHTS

Table A.1.
Relative performance w.r.t the Direct-Increasing MVM algorithm running time. Vertex weights are random real in the range [1.0 1.3].

Graph	$1-\epsilon$, $\epsilon=1/3$	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	Scal.	GPA-ROMA	ROMA	Dir	Iter	Init-Iter	Dir	Iter	Init-Iter	Suitor
G34	0.323	0.088	0.147	2.146	8.670	13.67	3.155	13.49	**15.26**	6.153
G39	0.354	0.097	0.142	3.025	5.649	5.532	6.336	9.667	**13.37**	6.171
de2010	0.362	0.166	0.318	3.120	6.436	5.525	6.629	13.49	**13.73**	6.843
shipsec8	0.443	0.103	0.192	1.782	14.01	16.68	11.40	18.65	**22.31**	3.510
kron_g500-17	0.151	0.049	0.085	3.570	1.659	1.499	7.448	**7.617**	7.591	2.856
mt2010	0.433	0.182	0.439	3.119	7.956	6.935	6.544	**17.71**	16.86	9.172
fe_ocean	3.465	0.827	1.785	14.01	85.33	93.35	25.22	154.7	**163.66**	37.43
tn2010	0.492	0.283	0.651	4.573	13.75	12.05	9.540	**29.47**	27.22	12.97
kron_g500-19	0.116	0.055	0.092	3.626	1.404	1.217	**8.302**	5.708	5.290	3.082
tx2010	0.488	0.266	0.576	3.417	7.578	6.962	8.380	**17.33**	16.15	8.644
kron_g500-20	0.087	0.040	0.067	2.659	0.955	0.859	**8.498**	4.896	4.856	2.812
M6	0.460	0.301	0.619	3.966	8.846	9.061	11.31	15.57	**16.38**	5.433
hugetric	0.814	0.459	0.927	5.490	17.03	18.51	8.347	29.36	**29.93**	14.70
rgg_n_2_23	0.899	0.306	0.541	3.627	56.94	63.95	13.92	58.36	**87.49**	10.93
hugetrace	0.840	0.432	0.880	4.981	18.06	20.96	9.185	**32.40**	30.66	15.60
nlpkkt200	1.0E4	1.7E3	3.1E3	2.2E4	2.9E4	2.9E4	1.0E5	3.8E5	**4.3E5**	5.1E4
hugebubbles	0.869	0.465	0.929	5.528	17.42	19.82	10.18	30.02	**30.21**	14.15
road_usa	0.327	0.204	0.429	2.142	4.315	4.022	3.124	9.003	8.785	**10.51**
europe_osm	0.354	0.179	0.425	1.927	5.402	5.147	3.119	10.49	9.831	**10.74**
rmat-G500	0.087	0.045	0.071	2.515	0.469	0.424	**5.437**	2.810	2.706	3.111
rmat-SSCA	0.171	0.083	0.143	4.817	1.373	1.228	**11.86**	7.454	7.346	5.129
rmat-ER	0.491	0.232	0.436	1.865	13.56	17.17	19.76	21.02	**25.08**	2.604
Geom. Mean	0.614	0.254	0.485	5.104	10.05	10.15	12.60	24.61	**25.75**	10.65

Table A.2.

Percentage of time taken by the major steps in the approximation algorithms. Random real weights in [1.0 1.3]. The remaining time is spent in variable declarations and initialization.

Graph	$1-\epsilon$, $\epsilon=1/3$ Scal.			$2/3-\epsilon$, $\epsilon=0.01$ GPA-ROMA				2/3- Dir		2/3- Init-Iter		1/2- Dir		1/2- Init-Iter	
	Srch.	Blsm	Duals	Sort	Srch.	DP.	2-aug	Sort	Srch.	P. 1	P. 2	Sort	Srch.	P. 1	P. 2
G34	58.3	0.00	17.2	4.03	28.0	0.87	60.1	35.7	43.0	48.2	25.8	48.2	25.8	18.0	1.90
G39	76.7	3.20	9.77	7.56	15.9	0.56	72.6	26.1	59.4	45.7	37.1	45.7	37.1	22.8	31.4
de2010	66.1	2.50	13.5	4.85	32.0	1.86	57.1	32.5	47.4	59.5	34.0	59.5	34.0	27.8	60.1
shipsec8	83.5	1.21	4.32	12.3	27.9	0.66	57.8	5.74	93.3	20.5	77.0	20.5	77.0	93.3	1.69
kron_g500-17	89.5	0.00	2.91	13.7	25.8	0.29	56.7	20.6	76.1	35.4	59.5	35.4	59.5	32.7	63.8
mt2010	70.7	2.05	10.5	3.79	45.3	2.18	43.7	25.6	60.3	47.1	48.1	47.1	48.1	34.4	57.7
fe_ocean	65.6	0.00	10.4	4.45	39.7	1.91	49.4	23.8	66.4	44.0	51.4	44.0	51.4	61.8	22.1
tn2010	73.7	2.39	7.57	3.70	44.4	1.81	45.7	21.9	60.8	45.1	50.7	45.1	50.7	35.4	57.6
kron_g500-19	92.8	0.00	1.78	12.7	28.1	0.24	56.3	17.4	79.5	34.1	61.5	34.1	61.5	15.9	82.3
tx2010	75.8	1.60	6.42	3.28	43.6	1.62	47.2	14.9	59.8	35.3	61.4	35.3	61.4	15.6	81.7
kron_g500-21	94.2	0.08	1.22	8.66	24.1	0.18	65.0	12.0	85.9	28.9	67.5	28.9	67.5	17.5	81.0
M6	78.6	2.16	6.06	3.12	40.4	1.66	50.3	12.3	71.3	32.6	64.0	32.6	64.0	23.6	73.8
hugetric	72.2	1.50	6.24	2.37	38.5	1.90	49.4	15.3	78.6	32.7	62.9	32.7	62.9	20.9	74.2
rgg_n_2_23	76.7	3.00	5.49	4.51	33.8	0.65	58.0	6.37	81.2	11.3	87.3	11.3	87.3	76.9	14.9
hugetrace	71.8	1.31	6.11	2.33	39.2	2.02	47.4	16.2	77.4	30.8	65.3	30.8	65.3	34.5	59.7
nlpkkt200	80.4	0.00	4.27	5.92	31.3	0.62	59.2	3.26	96.3	16.3	81.9	16.3	81.9	32.0	62.7
hugebubbles	72.8	1.46	6.00	2.24	40.7	1.94	47.7	15.7	78.2	30.5	66.3	30.5	66.3	22.4	72.5
road_usa	76.3	2.13	5.54	1.90	43.1	1.89	44.4	15.5	75.1	28.1	68.9	28.1	68.9	23.1	73.7
europe_osm	75.9	1.24	5.60	1.74	50.2	1.44	38.1	16.1	76.5	27.1	69.3	27.1	69.3	12.1	83.4
rmat-G500	92.9	0.00	0.53	4.79	27.7	0.32	66.5	3.85	93.4	7.94	90.4	7.94	90.4	23.9	75.8
rmat-SSCA	92.5	0.03	0.79	3.18	26.4	0.52	68.1	3.52	94.2	8.05	90.2	8.05	90.2	19.3	79.8
rmat-ER	94.6	0.00	0.43	2.75	34.7	1.17	60.3	0.81	98.9	3.87	95.4	3.87	95.4	68.5	30.7
Arith. Mean	78.7	1.17	6.03	5.18	34.6	1.20	54.6	15.7	75.1	30.6	64.4	30.6	64.4	33.3	56.5

Table A.3.
The ratios of the number of scanned edges by approximation algorithms to $|E|$. Random real weights in [1.0 1.3].

Graph	$1-\epsilon$, $\epsilon=1/3$	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	Scal.	GPA-RO.	RO.	Dir	Iter	Init-Iter	Dir	Iter	Init-Iter	Su.
G34	8.20	20.2	13.6	2.03	0.74	0.74	0.53	0.74	0.74	1.57
G39	10.9	19.5	14.2	1.98	1.72	2.02	0.48	0.91	1.08	1.34
de2010	10.7	19.7	13.4	1.78	2.32	2.72	0.55	1.14	1.36	1.53
shipsec8	7.78	20.1	14.1	3.29	0.52	0.52	0.14	0.51	0.51	1.50
kron_g500-17	5.60	20.1	14.4	0.32	1.28	1.52	0.12	0.34	0.40	1.33
mt2010	10.8	20.6	13.3	1.67	2.24	2.66	0.55	1.19	1.42	1.52
fe_ocean	7.66	19.9	13.7	2.25	0.78	0.80	0.42	0.70	0.71	1.52
tn2010	10.6	20.7	13.4	1.77	2.05	2.66	0.55	1.13	1.35	1.53
kron_g500-19	5.50	20.1	14.4	0.27	1.23	1.45	0.10	0.36	0.42	1.32
tx2010	10.6	20.7	13.4	1.80	2.29	2.68	0.55	1.25	1.46	1.53
kron_g500-20	5.30	20.1	14.4	0.24	1.36	1.58	0.09	0.30	0.35	1.31
M6	10.3	20.8	13.7	2.39	1.96	2.27	0.52	1.19	1.35	1.52
hugetric	9.77	20.7	13.2	1.82	1.36	1.50	0.67	1.02	1.12	1.59
rgg_n_2_23	10.0	20.3	14.0	2.96	0.57	0.57	0.25	0.56	0.57	1.51
hugetrace	9.62	20.7	13.2	1.82	1.20	1.30	0.67	0.89	0.96	1.59
nlpkkt200	7.58	19.1	14.0	2.80	17.6	18.0	0.17	1.28	1.30	1.51
hugebubbles	9.76	20.7	13.2	1.82	1.28	1.40	0.68	0.97	1.06	1.59
road_usa	12.2	20.3	12.6	1.53	3.15	3.59	0.72	1.89	2.18	1.59
europe_osm	11.8	21.7	12.7	1.56	2.32	2.52	0.77	1.57	1.67	1.64
rmat-G500	6.34	19.9	14.0	0.28	2.90	3.05	0.16	0.68	0.84	1.00
rmat-SSCA	5.96	20.1	14.3	0.32	1.53	1.91	0.14	0.50	0.56	1.07
rmat-ER	6.18	20.2	14.1	3.09	0.72	0.76	0.36	0.60	0.66	1.49
Geo. Mean	8.49	20.3	13.7	1.32	1.59	1.77	0.34	0.80	0.89	1.45

Table A.4.
The gap to optimality of the weights of the matching obtained from the approximation algorithms. Vertex weights are random real in the range [1.0 1.3].

Graph	$1-\epsilon$,	$2/3-\epsilon$, $\epsilon = 0.01$		2/3-approx			1/2-approx			
	$\epsilon = 1/3$	GPA-				Init-			Init-	
	Scal.	RO.	RO.	Dir	Iter	Iter	Dir	Iter	Iter	Su.
G34	1.89	1.95	2.23	3.15	**0.00**	0.00	8.43	0.00	0.00	8.43
G39	1.90	0.94	1.22	1.04	0.69	**0.64**	8.86	4.70	4.77	8.86
de2010	6.88	3.50	3.65	4.06	**2.46**	2.48	13.0	7.21	7.25	13.0
shipsec8	0.00	0.13	0.13	0.14	**0.00**	0.00	1.01	0.00	0.00	1.01
kron_g500-17	5.71	2.71	2.87	2.73	**1.43**	1.46	16.6	7.20	7.33	16.6
mt2010	6.29	3.66	3.77	4.09	**2.48**	2.51	13.6	7.40	7.46	13.6
fe_ocean	1.44	1.41	1.56	2.13	**0.17**	0.17	7.43	0.26	0.28	7.43
tn2010	6.43	3.62	3.74	4.04	**2.42**	2.45	13.4	7.29	7.36	13.4
kron_g500-19	5.92	2.27	2.43	2.39	**1.11**	1.14	16.0	6.46	6.51	16.0
tx2010	3.71	3.32	3.44	3.82	**2.25**	2.27	12.6	6.78	6.85	12.6
kron_g500-20	5.51	2.00	2.13	2.15	**0.94**	0.97	15.4	5.88	5.93	15.4
M6	**0.82**	1.69	1.74	2.08	1.13	1.14	7.69	3.87	3.94	7.69
hugetric	2.73	2.84	3.17	4.13	**1.03**	1.05	10.7	2.27	2.46	10.7
rgg_n_2_23	0.07	0.68	0.70	0.81	**0.00**	0.00	3.42	0.02	0.03	3.42
hugetrace	2.65	2.71	3.09	4.06	**0.80**	0.81	10.6	1.71	1.86	10.6
nlpkkt200	0.07	0.16	0.18	0.13	**0.03**	0.03	2.03	0.07	0.07	2.03
hugebubbles	2.76	2.82	3.15	4.11	**0.91**	0.92	10.6	1.97	2.14	10.6
road_usa	6.62	3.02	3.51	4.21	**2.23**	2.26	12.6	6.83	6.91	12.6
europe_osm	6.33	**1.73**	4.79	5.73	1.81	1.83	12.7	2.96	3.06	12.7
rmat-G500	6.14	1.28	1.33	1.48	**0.58**	0.60	13.7	4.96	4.91	13.7
rmat-SSCA	6.54	2.20	2.31	2.45	**1.29**	1.32	15.4	6.78	6.80	15.4
rmat-ER	**0.00**	0.06	0.06	0.07	0.03	0.03	2.03	0.65	0.68	2.03
Arith. Mean	1.00	1.42	1.60	1.78	**0.30**	0.30	8.53	1.04	1.10	8.53

Table A.5.
The gap to optimality of the cardinality of the matching obtained from the approximation algorithms. Vertex weights are random integers in the range [1.0 1.3].

Graph	$1-\epsilon$, $\epsilon=1/3$	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	Scal.	GPA-RO.	RO.	Dir	Iter	Init-Iter	Dir	Iter	Init-Iter	Su.
G34	2.08	2.20	2.52	3.55	**0.00**	0.00	9.25	0.00	0.00	9.25
G39	2.04	1.07	1.39	1.19	0.79	**0.73**	9.74	5.22	5.29	9.74
de2010	7.26	3.90	4.06	4.52	**2.75**	2.78	14.0	7.84	7.88	14.0
shipsec8	0.00	0.15	0.15	0.17	**0.00**	0.00	1.15	0.00	0.00	1.15
kron_g500-17	4.99	2.85	3.01	2.83	**1.51**	1.53	16.9	7.36	7.50	16.9
mt2010	6.54	4.06	4.18	4.53	**2.76**	2.80	14.6	7.99	8.06	14.6
fe_ocean	1.58	1.60	1.77	2.42	**0.19**	0.19	8.20	0.29	0.31	8.20
tn2010	6.75	4.03	4.16	4.50	**2.71**	2.74	14.4	7.91	7.99	14.4
kron_g500-19	5.17	2.37	2.53	2.46	**1.15**	1.19	16.2	6.56	6.63	16.2
tx2010	3.84	3.70	3.82	4.26	**2.52**	2.54	13.6	7.37	7.45	13.6
kron_g500-20	4.70	2.08	2.22	2.21	**0.98**	1.01	15.7	5.96	6.01	15.7
M6	**0.90**	1.92	1.98	2.36	1.28	1.30	8.49	4.32	4.40	8.49
hugetric	2.91	3.19	3.55	4.63	**1.16**	1.18	11.6	2.50	2.70	11.6
rgg_n_2_23	0.08	0.78	0.80	0.93	**0.00**	0.00	3.85	0.02	0.04	3.85
hugetrace	2.83	3.05	3.46	4.55	**0.90**	0.92	11.5	1.88	2.04	11.5
nlpkkt200	0.02	0.16	0.18	0.13	**0.00**	0.00	2.26	0.00	0.00	2.26
hugebubbles	2.95	3.18	3.52	4.61	**1.02**	1.04	11.6	2.17	2.35	11.6
road_usa	6.73	3.30	3.82	4.59	**2.44**	2.47	13.4	7.24	7.33	13.4
europe_osm	6.67	**1.91**	5.24	6.29	1.97	1.99	13.6	3.14	3.24	13.6
rmat-G500	5.15	1.32	1.37	1.50	**0.60**	0.62	13.9	4.97	4.92	13.9
rmat-SSCA	5.89	2.32	2.42	2.56	**1.36**	1.39	15.7	6.92	6.94	15.7
rmat-ER	**0.00**	0.07	0.07	0.08	0.03	0.03	2.31	0.74	0.78	2.31
Arith. Mean	3.60	2.24	2.56	2.95	**1.19**	1.20	11.0	4.11	4.18	11.0

Table A.6.
Relative performance w.r.t the Direct-Increasing MVM algorithm running time. Vertex weights are the vertex degrees.

Graph	$1-\epsilon$, $\epsilon=1/3$	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	Scal.	GPA-ROMA	ROMA	Dir	Iter	Init-Iter	Dir	Iter	Init-Iter	Suitor
G34	0.763	0.103	0.119	2.803	6.162	5.933	4.493	8.795	**9.194**	7.833
G39	0.050	0.122	0.166	3.060	6.714	5.554	15.76	**16.77**	16.76	12.16
de2010	0.094	0.193	0.359	4.071	9.048	7.999	**21.25**	19.25	19.65	7.231
shipsec8	0.217	0.134	0.225	3.017	16.24	17.85	**26.06**	18.82	23.93	6.012
kron_g500-17	1.280	5.549	9.333	6.915	277.8	280.8	**1500**	1411	1472	145.0
mt2010	0.077	0.189	0.431	4.267	9.043	8.989	15.38	**20.35**	19.03	8.430
fe_ocean	6.519	1.321	2.482	51.86	133.3	131.8	144.2	195.5	**210.4**	80.70
tn2010	0.103	0.304	0.643	6.108	15.62	14.16	23.05	**32.55**	30.58	11.84
kron_g500-19	5.894	36.79	58.82	23.35	1308	1354	**1.0E04**	6271	5870	801.6
tx2010	0.133	0.307	0.614	4.776	11.50	9.859	22.99	**24.37**	21.63	7.937
kron_g500-20	12.74	101.2	150.20	31.13	2988	2346	**3.4E4**	1.7E04	1.7E04	1878
M6	0.207	0.409	0.789	5.890	11.50	12.07	35.27	24.15	25.09	8.699
hugetric	1.391	0.491	0.976	12.05	21.85	22.87	**48.45**	41.93	41.74	27.05
rgg_n_2_23	0.298	0.323	0.510	5.785	56.45	64.43	31.73	71.81	**90.54**	10.57
hugetrace	1.333	0.381	0.739	10.46	20.38	22.93	**41.54**	37.34	39.45	24.32
nlpkkt200	2.4E4	2.1E3	3.0E3	3.0E5	3.0E5	2.8E5	6.3E5	6.3E05	6.5E5	2.4E5
hugebubbles	1.656	0.520	0.990	12.71	23.40	24.80	**57.35**	43.89	47.92	28.56
road_usa	0.105	0.158	0.293	3.640	4.482	4.195	**13.61**	11.41	9.892	7.318
europe_osm	0.107	0.134	0.279	4.084	6.054	5.376	**13.45**	11.59	10.96	7.940
rmat-G500	-	-	-	-	-	-	-	-	-	-
rmat-SSCA	-	-	-	-	-	-	-	-	-	-
rmat-ER	0.191	0.513	0.911	3.725	26.96	36.34	**55.79**	38.00	44.23	5.803
Geom. Mean	0.766	0.857	1.505	11.87	46.35	45.77	**103.4**	99.82	102.2	35.09

Table A.7.
Percentage of time taken by the major steps in the approximation algorithms. Degree weights are used. The remaining time is spent in variable declarations and initialization.

Graph	$1-\epsilon$, $\epsilon=1/3$ Scal.			$2/3-\epsilon$, $\epsilon=0.01$ GPA-ROMA				2/3- Dir		2/3- Init-Iter		1/2- Dir		1/2- Init-Iter	
	Srch.	Blsm	Duals	Sort	Srch.	DP.	2-aug	Sort	Srch.	P. 1	P. 2	Sort	Srch.	P. 1	P. 2
G34	20.2	0.00	6.71	1.63	50.1	1.71	40.6	9.35	34.6	20.5	5.55	15.7	13.4	17.8	1.84
G39	87.1	0.43	5.69	3.07	18.9	0.87	74.9	2.94	78.1	46.8	26.8	9.24	41.3	24.2	23.5
de2010	83.4	0.00	7.96	1.42	32.1	1.46	59.1	3.12	67.9	42.0	51.2	13.9	53.9	40.8	43.2
shipsec8	92.1	0.36	3.55	3.88	31.4	1.41	59.7	0.88	97.0	93.7	2.84	6.11	81.4	93.1	1.69
kron_g500-logn17	98.1	0.01	0.61	8.89	24.8	0.23	63.8	0.59	98.6	31.3	67.8	8.01	80.6	46.4	49.0
mt2010	85.1	0.00	6.24	1.04	42.4	2.53	48.6	2.19	76.3	43.2	52.1	7.21	75.2	39.5	52.2
fe_ocean	44.8	0.00	25.8	1.66	39.8	2.35	49.5	5.85	59.3	68.6	18.8	14.8	57.6	74.0	7.55
tn2010	87.3	0.00	5.27	1.05	40.0	2.31	51.1	1.74	73.6	42.7	53.4	5.40	80.8	39.8	52.9
kron_g500-logn19	98.5	0.01	0.42	8.22	26.8	0.19	64.5	0.48	98.9	22.8	76.7	8.05	82.0	22.6	75.1
tx2010	81.7	1.12	5.54	1.04	39.7	1.77	52.8	1.15	63.3	27.2	70.8	5.60	82.0	24.6	71.0
kron_g500-logn21	99.1	0.00	0.26	5.47	21.0	0.14	71.8	0.50	99.2	13.0	86.8	12.8	79.3	18.1	80.4
M6	88.4	0.77	4.03	0.80	38.4	1.98	53.7	1.03	77.6	42.4	55.8	6.65	80.0	31.8	64.7
hugetric-00010	67.3	0.00	11.3	0.62	38.8	2.32	47.6	3.36	79.8	51.2	44.4	12.2	63.2	38.8	52.9
rgg_n_2_23_s0	85.1	1.53	4.82	1.31	27.4	0.98	66.4	1.14	75.5	81.0	11.8	5.40	85.2	78.2	11.2
hugetrace-00010	61.7	0.04	13.9	0.62	39.5	2.27	48.4	3.95	76.8	50.0	44.0	15.0	52.2	39.3	50.3
nlpkkt200	48.3	0.00	28.6	2.05	10.8	0.34	81.8	10.3	71.9	48.7	46.3	13.8	59.1	67.9	20.8
hugebubbles-00010	65.2	0.14	12.2	0.58	38.5	2.30	48.5	3.51	80.7	51.9	43.0	14.1	61.5	38.2	53.2
road_usa	83.8	1.11	4.94	0.46	37.0	1.84	50.8	2.86	73.0	23.7	73.9	10.5	68.2	35.8	59.6
europe_osm	85.3	0.00	5.14	0.46	42.4	1.64	47.9	4.53	67.8	26.3	69.8	14.1	60.0	25.7	66.4
rmat-G500	99.4	0.00	0.10	3.92	26.7	0.21	68.2	0.49	98.7	9.4	90.5	1.75	95.8	31.1	68.4
rmat-SSCA	99.4	0.00	0.13	4.22	33.2	0.59	62.7	0.34	99.0	25.3	74.4	4.07	87.8	23.0	75.8
rmat-ER	99.0	0.01	0.22	3.14	23.0	1.36	68.6	0.15	99.6	77.8	21.5	1.41	96.6	62.0	37.1
Arith. Mean	80.0	0.25	6.97	2.53	32.9	1.40	58.2	2.75	79.4	42.7	49.5	9.36	69.9	41.5	46.3

Table A.8.
The ratios of the number of scanned edges by approximation algorithms to $|E|$. Vertex Degrees are used for vertex weights.

Graph	$1-\epsilon$, $\epsilon=1/3$ Scal.	$2/3-\epsilon$, $\epsilon=0.01$ GPA-RO.	RO.	2/3-approx Dir	Iter	Init-Iter	1/2-approx Dir	Iter	Init-Iter	Su.
G34	0.74	11.8	13.4	2.68	0.74	0.74	0.74	0.74	0.74	1.74
G39	92.2	19.5	14.3	2.68	1.80	2.12	0.65	1.02	1.20	1.57
de2010	79.5	19.7	13.5	2.40	1.57	1.95	0.62	0.92	1.13	1.78
shipsec8	31.4	21.2	14.1	3.39	0.51	0.51	0.25	0.51	0.51	1.67
kron_g500-17	124	21.6	15.5	3.21	1.31	1.32	0.20	0.22	0.27	1.64
mt2010	106	19.5	13.5	2.38	1.95	2.07	0.62	1.11	1.33	1.77
fe_ocean	3.21	19.2	13.7	2.69	0.71	0.72	0.61	0.68	0.68	1.71
tn2010	97.4	19.6	13.5	2.43	1.75	2.11	0.61	1.05	1.26	1.79
kron_g500-19	144	21.8	15.7	3.22	1.35	1.34	0.17	0.23	0.28	1.63
tx2010	98.0	19.7	13.5	2.38	1.78	2.14	0.61	1.06	1.27	1.77
kron_g500-20	164	21.8	15.9	3.23	1.40	1.82	0.15	0.24	0.29	1.62
M6	27.1	20.7	13.6	2.51	1.72	2.03	0.61	1.02	1.17	1.71
hugetric	3.64	20.9	13.1	1.74	1.13	1.26	0.71	0.85	0.92	1.68
rgg_n_2_23	39.9	20.4	14.0	3.02	0.57	0.57	0.32	0.56	0.56	1.81
hugetrace	2.77	21.1	13.1	1.71	1.01	1.11	0.70	0.80	0.85	1.67
nlpkkt200	1.42	18.1	14.0	1.09	2.22	2.63	0.06	1.03	1.04	1.07
hugebubbles	3.24	20.9	13.1	1.73	1.08	1.19	0.70	0.83	0.89	1.67
road_usa	44.9	20.2	12.8	1.98	2.17	2.59	0.81	1.43	1.70	1.80
europe_osm	52.1	21.8	12.7	1.66	1.64	1.82	0.93	1.20	1.28	1.92
rmat-G500	274	21.7	15.9	3.09	4.44	4.84	0.36	0.47	0.56	1.57
rmat-SSCA	268	22.7	15.8	3.19	1.18	1.38	0.31	0.37	0.42	1.61
rmat-ER	57.7	19.2	14.2	3.19	0.72	0.77	0.37	0.61	0.66	1.73
Geo. Mean	31.3	20.0	14.0	2.43	1.31	1.46	0.43	0.68	0.76	1.67

Table A.9.
The gap to optimality of the weights of the matching obtained from the approximation algorithms. Vertex weights are vertex degrees.

Graph	$1-\epsilon$,	$2/3-\epsilon$, $\epsilon=0.01$		2/3-approx			1/2-approx			
	$\epsilon=1/3$	GPA-				Init-			Init-	
	Scal.	RO.	RO.	Dir	Iter	Iter	Dir	Iter	Iter	Su.
G34	0.00	0.00	3.00	0.00	0.00	**0.00**	0.00	0.00	0.00	0.00
G39	4.47	**0.71**	1.21	1.48	1.13	1.13	10.0	8.82	8.82	10.0
de2010	10.8	3.01	3.62	4.06	**1.89**	1.89	13.4	6.87	6.90	13.4
shipsec8	0.04	0.15	0.10	0.01	**0.00**	0.00	0.02	0.00	0.00	0.02
kron_g500-17	3.19	0.38	0.55	0.68	**0.14**	0.14	3.26	0.82	0.83	3.26
mt2010	11.4	3.28	3.90	4.40	**1.96**	1.96	14.1	7.04	7.06	14.1
fe_ocean	0.34	0.38	1.70	0.56	**0.09**	0.09	0.85	0.16	0.16	0.85
tn2010	11.2	3.17	3.84	4.22	**1.80**	1.81	13.9	6.74	6.78	13.9
kron_g500-19	4.31	0.33	0.48	0.60	**0.12**	0.12	2.87	0.70	0.71	2.87
tx2010	2.58	3.04	3.60	3.95	**1.83**	1.84	12.9	6.77	6.81	12.9
kron_g500-20	3.99	0.27	0.39	0.47	**0.10**	0.10	2.50	0.60	0.60	2.50
M6	7.80	2.21	2.44	2.30	**1.58**	1.58	10.3	5.80	5.86	10.3
hugetric	3.19	2.11	4.51	1.61	**1.60**	1.60	3.56	3.54	3.54	3.56
rgg_n_2_23	0.75	0.43	0.44	0.49	**0.00**	0.00	2.22	0.02	0.02	2.22
hugetrace	2.21	1.29	4.53	1.18	**1.17**	1.17	2.53	2.51	2.51	2.53
nlpkkt200	0.20	**0.15**	0.56	0.22	0.22	0.22	0.22	0.23	0.23	0.22
hugebubbles	2.86	2.22	4.46	1.41	**1.40**	1.40	3.09	3.07	3.07	3.09
road_usa	8.63	4.46	4.84	5.45	**2.64**	2.65	14.6	8.44	8.46	14.6
europe_osm	5.88	2.34	6.44	2.92	**2.17**	2.17	7.07	3.40	3.40	7.07
rmat-G500	-	-	-	-	-	-	-	-	-	-
rmat-SSCA	-	-	-	-	-	-	-	-	-	-
rmat-ER	0.09	0.11	0.12	0.13	**0.02**	0.02	2.84	0.74	0.77	2.84
Arith. Mean	4.20	1.50	2.54	1.81	**0.99**	0.99	6.02	3.31	3.33	6.02

Table A.10.
The gap to optimality of the cardinality of the matching obtained from the approximation algorithms. Vertex weights are vertex degrees.

Graph	$1-\epsilon$, $\epsilon = 1/3$	$2/3-\epsilon$, $\epsilon = 0.01$		2/3-approx			1/2-approx			
	Scal.	GPA-RO.	RO.	Dir	Iter	Init-Iter	Dir	Iter	Init-Iter	Su.
G34	0.00	0.00	3.00	0.00	**0.00**	0.00	0.00	0.00	0.00	0.00
G39	11.3	**1.60**	2.60	3.10	2.40	2.40	19.5	17.4	17.4	19.5
de2010	22.7	6.95	8.15	8.59	**4.67**	4.67	22.7	12.8	12.8	22.7
shipsec8	0.01	0.22	0.16	0.01	**0.00**	0.00	0.04	0.00	0.00	0.04
kron_g500-17	34.4	9.76	13.0	12.6	**3.91**	3.91	36.3	16.1	16.2	36.3
mt2010	24.7	7.59	8.77	9.40	**4.71**	4.71	24.7	13.5	13.5	24.7
fe_ocean	0.63	0.75	2.19	1.08	**0.22**	0.22	1.50	0.35	0.35	1.50
tn2010	24.2	7.47	8.72	9.18	**4.53**	4.55	24.2	13.1	13.1	24.2
kron_g500-19	34.3	8.90	12.1	12.1	**3.34**	3.34	36.0	15.3	15.4	36.0
tx2010	5.24	6.60	7.60	8.04	**4.14**	4.16	21.4	12.1	12.2	21.4
kron_g500-20	35.0	8.43	11.5	11.0	**2.94**	2.95	35.8	14.8	14.9	35.8
M6	11.8	2.81	3.08	2.92	**2.02**	2.03	12.4	7.06	7.12	12.4
hugetric	3.56	2.12	4.52	1.62	**1.60**	1.60	3.58	3.54	3.55	3.58
rgg_n_2_23	1.37	0.85	0.88	0.96	**0.00**	0.00	3.37	0.03	0.04	3.37
hugetrace	2.52	1.29	4.53	1.19	**1.18**	1.18	2.54	2.52	2.52	2.54
nlpkkt200	**0.00**	0.00	0.58	0.00	0.00	0.00	0.00	0.00	0.00	0.00
hugebubbles	3.08	2.22	4.47	1.41	**1.40**	1.40	3.10	3.07	3.07	3.10
road_usa	17.3	8.55	8.88	9.89	**4.85**	4.86	22.8	13.3	13.4	22.8
europe_osm	8.36	3.07	7.50	3.81	**2.68**	2.68	8.36	4.13	4.13	8.36
rmat-G500	-	-	-	-	-	-	-	-	-	-
rmat-SSCA	-	-	-	-	-	-	-	-	-	-
rmat-ER	0.20	0.24	0.26	0.28	**0.06**	0.06	4.50	1.30	1.36	4.50
Arith. Mean	12.0	3.97	5.62	4.86	**2.23**	2.24	14.1	7.52	7.55	14.1

Table A.11.
2/3-approximation algorithms run time (seconds) and speedup obtained with twenty threads. Vertex weights are random reals in the range [1.0 1.3].

	2/3-Iter		2/3-Init-Iter	
Graph	Time	Speed-up	Time	Speed-up
kron_g500-21	0.29	**13.8**	0.30	13.7
M6	0.13	7.90	0.13	**7.96**
hugetric	0.10	**9.37**	0.11	8.70
rgg_n_2_23	0.08	**7.67**	0.08	7.73
hugetrace	0.19	7.74	0.18	**8.15**
nlpkkt200	3.03	7.17	2.25	**9.65**
hugebubbles	0.43	6.96	0.42	**7.09**
road_usa	0.57	**11.8**	0.65	10.3
europe_osm	0.81	**12.2**	0.84	11.7
rmat-G500	22.6	**16.5**	23.2	16.1
rmat-SSCA	25.8	**13.2**	26.1	13.0
rmat-ER	19.4	5.20	19.1	**5.28**
Geom. Mean		9.42		**9.51**

Table A.12.
1/2-approximation algorithms run time (seconds) and speedup obtained with twenty threads. Vertex weights are random reals in the range [1.0 1.3].

	1/2-Iter		1/2-Init-Iter		Suitor	
Graph	Time	Speed -up	Time	Speed -up	Time	Speed -up
kron_g500-21	0.06	**12.7**	0.07	11.3	0.23	3.45
M6	0.07	**8.70**	0.07	7.97	0.14	4.11
hugetric	0.07	**8.76**	0.07	8.49	0.14	4.33
rgg_n_2_23	0.07	6.37	0.07	**6.94**	0.39	1.18
hugetrace	0.15	6.64	0.14	**6.70**	0.23	4.29
nlpkkt200	0.20	**7.27**	0.20	7.15	1.38	1.06
hugebubbles	0.26	**7.40**	0.27	7.34	0.45	4.35
road_usa	0.36	4.99	0.39	4.58	0.24	**7.41**
europe_osm	0.48	**7.21**	0.53	6.61	0.50	7.03
rmat-G500	3.81	**16.3**	4.31	14.4	6.56	9.46
rmat-SSCA	5.35	**11.7**	6.37	9.82	10.9	5.74
rmat-ER	9.58	7.20	9.10	**7.58**	65.5	1.05
Geom. Mean	0.36	**8.31**	0.38	7.92	0.85	3.57

Table A.13.
Scalability of parallel approximation algorithms using 20 threads. Vertex weights are random reals in the range [1.0 1.3].

Graph	2/3-Iter	2/3-Init-Iter	1/2-Iter	1/2-Init-Iter	Suitor
kron_g500-21	13.8	**15.2**	12.7	11.4	10.3
M6	8.09	7.96	9.16	7.97	**10.1**
hugetric	**10.2**	8.70	8.93	8.49	7.03
rgg_n_2_23	8.62	7.73	9.55	6.94	**10.3**
hugetrace	8.98	8.15	6.64	7.08	**10.3**
nlpkkt200	7.17	9.78	8.23	7.15	**13.3**
hugebubbles	**7.91**	7.09	7.45	7.34	7.14
road_usa	**11.8**	11.1	8.95	8.43	7.41
europe_osm	12.2	**12.3**	10.4	10.2	7.03
rmat-G500	16.5	**17.7**	16.3	14.9	17.3
rmat-SSCA	13.2	14.5	11.7	10.0	**18.2**
rmat-ER	6.59	5.28	8.59	7.58	**14.9**
Geom. Mean	10.0	9.87	9.60	8.72	**10.5**

Table A.14.
2/3-approximation algorithms run time (seconds) and speedup obtained with twenty threads. Vertex weights are vertex degrees.

	2/3-Iter		2/3-Init-Iter	
Graph	Time	Speed-up	Time	Speed-up
kron_g500-21	0.25	14.5	0.23	**15.6**
M6	0.08	**10.9**	0.11	7.86
hugetric	0.10	**7.23**	0.10	6.99
rgg_n_2_23	0.08	**7.34**	0.10	5.71
hugetrace	0.15	6.49	0.13	**7.53**
nlpkkt200	0.61	**2.59**	0.71	2.23
hugebubbles	0.28	**7.79**	0.29	7.49
road_usa	0.46	9.31	0.44	**9.73**
europe_osm	0.54	10.1	0.51	**10.6**
rmat-G500	30.8	**17.7**	31.4	17.3
rmat-SSCA	19.6	11.1	18.5	**11.8**
rmat-ER	19.4	5.03	17.6	**5.53**
Geom. Mean		**8.27**		8.0

Table A.15.
1/2-approximation algorithms run time (seconds) and speedup obtained with twenty threads. Vertex weights are vertex degrees.

	1/2-Iter		1/2-Init-Iter		Suitor	
Graph	Time	Speed-up	Time	Speed-up	Time	Speed-up
kron_g500-21	0.04	**14.1**	0.05	13.3	8.99	0.07
M6	0.06	**6.85**	0.07	6.18	0.18	2.39
hugetric	0.05	8.08	0.05	**8.10**	0.31	1.24
rgg_n_2_23	0.07	6.01	0.06	**6.30**	0.46	0.87
hugetrace	0.08	**7.52**	0.08	7.42	0.42	1.38
nlpkkt200	0.13	**5.52**	0.15	4.71	0.71	1.01
hugebubbles	0.14	7.96	0.14	**8.22**	1.05	1.08
road_usa	0.22	**7.04**	0.27	5.77	0.25	6.31
europe_osm	0.31	**7.22**	0.34	6.61	0.52	4.27
rmat-G500	2.58	**15.6**	2.88	14.0	229	0.18
rmat-SSCA	3.89	**11.6**	4.04	11.1	151	0.30
rmat-ER	9.66	8.27	8.99	**8.89**	69.1	1.16
Geom. Mean		**8.37**		7.95		0.93

Table A.16.
Scalability of parallel approximation algorithms using 20 threads. Vertex weights are vertex degrees.

Graph	2/3-Iter	2/3-Init-Iter	1/2-Iter	1/2-Init-Iter	Suitor
kron_g500-21	14.5	**19.8**	14.1	13.6	16.3
M6	**11.4**	7.86	7.12	6.18	5.18
hugetric	7.57	6.99	8.08	**8.14**	1.34
rgg_n_2_23	8.38	5.71	7.58	6.30	**9.75**
hugetrace	7.30	7.53	**7.95**	7.42	1.49
nlpkkt200	2.59	2.32	**5.69**	4.71	2.06
hugebubbles	8.26	7.49	**8.70**	8.22	1.24
road_usa	9.31	**10.4**	7.69	7.27	6.31
europe_osm	10.1	**11.9**	9.22	8.94	4.27
rmat-G500	17.7	**18.1**	15.6	14.4	14.3
rmat-SSCA	11.1	**13.8**	13.2	11.1	13.5
rmat-ER	6.78	5.53	9.63	8.89	**14.3**
Geom. Mean	8.76	8.48	**9.14**	8.35	5.10