ALGORITHMS FOR
VERTEX-WEIGHTED MATCHING IN GRAPHS

by

Mahantesh Halappanavar
B.S. August 1996, Karnataka University
M.S. December 2003, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2009

Approved by:

__________________________
Alex Pothen (Director)

__________________________
Jessica Crouch

__________________________
Bruce Hendrickson

__________________________
Stephan Olariu

__________________________
Mohammad Zubair
ABSTRACT

ALGORITHMS FOR VERTEX-WEIGHTED MATCHING IN GRAPHS

Mahantesh Halappanavar

Old Dominion University, 2009
Director: Dr. Alex Pothen

A matching M in a graph is a subset of edges such that no two edges in M are incident on the same vertex. Matching is a fundamental combinatorial problem that has applications in many contexts: high-performance computing, bioinformatics, network switch design, web technologies, etc. Examples in the first context include sparse linear systems of equations, where matchings are used to place large matrix elements on or close to the diagonal, to compute the block triangular decomposition of sparse matrices, to construct sparse bases for the null space or column space of under-determined matrices, and to coarsen graphs in multi-level graph partitioning algorithms. In the first part of this thesis, we develop exact and approximation algorithms for vertex weighted matchings, an under-studied variant of the weighted matching problem. We propose three exact algorithms, three half approximation algorithms, and a two-third approximation algorithm. We exploit inherent properties of this problem such as lexicographical orders, decomposition into sub-problems, and the reachability property, not only to design efficient algorithms, but also to provide simple proofs of correctness of the proposed algorithms. In the second part of this thesis, we describe work on a new parallel half-approximation algorithm for weighted matching. Algorithms for computing optimal matchings are not amenable to parallelism, and hence we consider approximation algorithms here. We extend the existing work on a parallel half approximation algorithm for weighted matching and provide an analysis of its time complexity. We support the theoretical observations with experimental results obtained with MatchBoxP, toolkit designed and implemented in C++ and MPI using modern software engineering techniques. The work in this thesis has resulted in better understanding of matching theory, a functional public-domain software toolkit, and modeling of the sparsest basis problem as a vertex-weighted matching problem.
ACKNOWLEDGEMENTS

“One can pay back the loan of gold, but one dies forever in debt to those who are kind.” - Malayan Proverb

First and foremost, I would like to thank my advisor Alex Pothen, without him this work would have been impossible. He not only introduced me to the subject, but has also been a constant inspiration throughout. His support and encouragement has been invaluable both personally and professionally, for which I will remain forever indebted.

This work has evolved in collaboration with Florin Dobrian, a friend, guide and mentor who has irreversibly changed my thinking. I will also remain indebted to Assefaw Gebremedhin for his friendship and generousness in improving my presentation on numerous occasions.

I remain thankful to my committee members Jessica Crouch, Bruce Hendrickson, Stephan Olariu and Mohammad Zubair. Their comments have been thought provoking, and their suggestions invaluable. I also want to thank Erik Boman for his time and efforts in helping my research.

I will remain indebted to my supervisor Mike Sachon and coworkers Amit Kumar and Ruben Igloria, for providing flexibility, support and a productive work environment. Special thanks are due to Amit Kumar for his friendship that has only grown over the years.

I was introduced to academic research in my Masters program by Ravi Mukkamala. I will remain forever indebted for his mentorship - academic as well as spiritual.

I would like to thank the following departments at Old Dominion University - the Office of Graduate Studies for the University Graduate Fellowship during 2005 to 2006; the Office of Research and the Department of Computer Science for teaching and research assistanceships during 2001 to 2005; and the Office of Study Abroad for travel assistance in 2005.

With long hours away from home, the last five years have been especially hard on my wife Savitha and daughter Anika. They have accepted it in stride and I cannot thank them enough for it. I will remain thankful to my parents who have always emphasized education above everything else, my sister for being my inspiration, my in-laws for their support, and my very large extended family where everyone has made a special impression on me.
This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>xvii</td>
</tr>
</tbody>
</table>

## CHAPTERS

### I Introduction

I.1 Outline ........................................... 3
I.2 Combinatorial Scientific Computing .................. 3
I.3 Motivation ........................................ 4
I.4 Contributions .................................... 9
I.5 Chapter Summary .................................. 10

### II Background and Related Work

II.1 Introduction ..................................... 11
II.2 Foundations ...................................... 15
II.3 Maximum Cardinality Matching ..................... 25
II.4 Maximum Edge-Weight Matching ..................... 28
II.5 Approximation Algorithms .......................... 33
II.6 Chapter Summary .................................. 39

### III Exact Algorithms

III.1 Introduction and Related Work ..................... 40
III.2 Foundations ...................................... 44
III.3 New Algorithms for Maximum Vertex-weight Matching ........................................ 47
   III.3.1 Algorithm GlobalOptimal ...................... 48
   III.3.2 Algorithm LocalOptimal ....................... 50
   III.3.3 Algorithm HybridOptimal ...................... 51
   III.3.4 Negative Weights ............................. 53
III.4 Proof of Correctness .............................. 55
III.5 A Reachability-Based Algorithm ................... 61
III.6 Chapter Summary .................................. 62

### IV Approximation Algorithms

IV.1 Introduction ..................................... 64
IV.2 New $\frac{1}{2}$-approx Algorithms .................. 64
IV.3 Proof of Correctness ................................ 71
IV.4 Global $\frac{2}{3}$-approx Algorithm ................. 78
   IV.4.1 Proof of Correctness ......................... 78
IV.5 Potential Local $\frac{2}{3}$-approx Algorithm ........ 88
   IV.5.1 Correctness of Algorithm LOCALTWOThird .......... 89
IV.6 Experimental Results ................................ 91
IV.7 Chapter Summary .................................. 95

### V Parallel Approximate Algorithms

V.1 Introduction ..................................... 97
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.1.1 Complexity Analysis</td>
<td>100</td>
</tr>
<tr>
<td>V.2 Distributed Algorithm of Hoepman</td>
<td>104</td>
</tr>
<tr>
<td>V.2.1 Complexity Analysis</td>
<td>107</td>
</tr>
<tr>
<td>V.3 Parallel $\frac{1}{2}$-approx Algorithm</td>
<td>108</td>
</tr>
<tr>
<td>V.3.1 Complexity Analysis</td>
<td>119</td>
</tr>
<tr>
<td>V.4 Experimental Results</td>
<td>122</td>
</tr>
<tr>
<td>V.4.1 Data Set for Experiments</td>
<td>122</td>
</tr>
<tr>
<td>V.4.2 Performance of Serial Matching Algorithms</td>
<td>127</td>
</tr>
<tr>
<td>V.4.3 Performance of Parallel Matching Algorithm</td>
<td>130</td>
</tr>
<tr>
<td>V.4.4 Performance of Parallel Matching on Graphs from Applications</td>
<td>144</td>
</tr>
<tr>
<td>V.4.5 Analysis of Communication</td>
<td>147</td>
</tr>
<tr>
<td>V.5 Chapter Summary</td>
<td>150</td>
</tr>
<tr>
<td>VI Conclusions and Future Work</td>
<td>152</td>
</tr>
<tr>
<td>VI.1 Future Work</td>
<td>153</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms for maximum cardinality matching [66]. For a graph $G = (V,E)$, $n =</td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Power of data structures. For a graph $G = (V,E)$, $n =</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Algorithms for maximum edge-weight matching [66]. For a graph $G = (V,E)$ with weight function $w : E \rightarrow \mathbb{R}^+$, $n =</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Algorithms for approximate weighted matching. For a graph $G = (V,E)$, $n =</td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A survey of algorithms for maximum vertex-weight matching. For a given graph $G = (V,E)$, $n =</td>
</tr>
<tr>
<td>46</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A summary of algorithms proposed for vertex weighted matchings. Bipartite and general graphs are represented with B and G respectively. For a bipartite graph $G = (S,T,E)$, $n = (</td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A summary of algorithms proposed for vertex weighted matchings. Bipartite and general graphs are represented with B and G respectively. For a bipartite graph $G = (S,T,E)$, $n = (</td>
</tr>
<tr>
<td>65</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Matrix Instances. Downloaded from University of Florida Matrix Collection.</td>
</tr>
<tr>
<td>92</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Performance of Global-based Algorithms. The numbers represent compute time in seconds.</td>
</tr>
<tr>
<td>92</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Relative Performance of Global and Local-based Algorithms. The numbers represent compute time in seconds.</td>
</tr>
<tr>
<td>93</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Matrix Instances downloaded from University of Florida Matrix Collection. Unsymm represents unsymmetric matrices and Symm represents symmetric matrices.</td>
</tr>
<tr>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>
12 Synthetic and Model Graphs. SSCA#2 graphs are generated using GT-Graph generator. The number of vertices in the original graph are doubled to convert it into a bipartite graph to eliminate self-loops; duplicate edges, if any, are also eliminated. RGGs and grid graphs are generated with MatchBox-P and have random edge weights. 125

13 Performance of serial approx algorithm. The second column represents the ratio of weights of approximate and exact matchings. Similarly, the third column represents the ratio of cardinality of the two matchings. Fourth and fifth columns show the time in seconds to compute approximate and exact matchings respectively. 127

14 Grid graphs for weak scalability studies. Columns three and four represent the number of processors used to solve the grid graphs of a given size. 135
LIST OF FIGURES

1 Landscape of the matching problems. The vertex-weighted matching problem can be formulated as an edge-weighted matching problem. The weighted matching algorithms utilize techniques developed for the cardinality matching problem. The arrows indicate these relationships. 2

2 Representation of a sparsest column-space basis problem. A matrix $A$ with $k$ rows and $n$ columns, and a basis $B$ with $k$ rows and $k$ linearly independent columns. 7

3 A greedy algorithm for computing a sparsest column-space basis. (a) State before augmenting a basis $B$ with a column of current heaviest weight $w_{max}$ from $C$; (b) state after augmenting a basis with a sparsest linearly independent column from $C$. 7

4 Computation of a sparsest column-space basis with a maximum vertex-weight matching. (a) A matrix $A$; (b) A bipartite graph $(G)$ representation of $A$. Numbers on the right indicate the weight of each $S$ vertex. Bold lines represent the matched edges, and matched vertices are colored black; (c) A candidate basis as computed by a maximum vertex-weight matching in $G$. 9

5 An example of matching. (a) A bipartite graph $G$, (b) a matching $M$ in $G$. Bold lines represent matched edges, and matched vertices are colored black. 12

6 Types of matchings. Matched edges are represented with bold lines and matched vertices are filled with black color. (a) A maximal matching, (b) a maximum matching, and (c) a perfect matching. 13

7 Types of paths. Matched edges are represented with bold lines and matched vertices are colored black. (a) An alternating path starting with an unmatched vertex, (b) an alternating path starting with a matched vertex, and (c) an augmenting path. 15

8 Augmentation by symmetric difference. The matched edges are represented with bold lines and matched vertices are colored black. (a) Before augmentation, (b) after augmentation. 16

9 The symmetric difference of two matchings $M_S \oplus M_T$. Dashed lines represent edges in $M_S$ and Solid lines represent edges in $M_T$. (a) A cycle; (b)-(e) Augmenting or alternating paths. 17

10 Effect of $M \oplus P$. Bold lines represent matched edges and matched vertices are colored black. (a) Paths $P$ and $Q$ do not intersect; (b) paths $P$ and $Q$ intersect. This figure has been adapted from [57]. 18
Breadth-first search. The vertex being processed at a given step is colored purple, and also marked by an arrow. The shaded lines represent the processed edges. The vertices that have already been processed are colored black. The adjacency list for each vertex is maintained in an increasing order of the indices of vertices. (a) The input graph before execution, (b)-(f) the intermediate states of execution. State of the pseudo-queue at each step: (b) [2, 3, 4] (c) [3, 4, 5], dequeue 2, enqueue 5; (d) [4, 5, 6] dequeue 3, enqueue 6; (e) [5, 6] dequeue 4; (f) [6] dequeue 5.

Depth-first search. The vertex being processed at a given step is colored purple, and also marked by an arrow. The shaded lines represent the processed edges. The vertices that have already been processed are colored black. The adjacency list for each vertex is maintained in an increasing order of the indices of vertices. (a) The input graph before execution, (b)-(f) the intermediate states of execution. State of the pseudo-stack at each step: (b) [2, 3, 4] (c) [2, 3, 5] pop 4, move 2, move 3, push 5; (d) [3, 2, 6] pop 5, move 2, push 6; (e) [2, 3] pop 6, move 3; (f) [2].

Single-source single-path technique. The vertex being processed at a given step is colored purple, and also pointed by an arrow. The shaded lines represent potential augmenting paths. Bold lines represent matched edges and matched vertices are colored black. (a) The input graph before execution, (b)-(d) the intermediate states of execution, and (e) the final state.

Multiple-source single-path technique. The vertices being processed at a given step are colored purple. The shaded lines represent potential augmenting paths. Bold lines represent matched edges and matched vertices are colored black. (a) The input graph before execution, (b)-(d) the intermediate states of execution, and (e) the final state.

Multiple-source multiple-path technique. The vertices processed at a given step are colored purple. The shaded lines represent potential augmenting paths, bold lines represent matched edges and matched vertices are colored black. (a) The input graph before execution, (b) the intermediate state of execution, and (c) the final state.

Execution of Algorithm GlobalHeavy. The weights are associated with the edges. Bold lines represent matched edges, and matched vertices are colored black. Vertices processed at a given step are colored purple. Dashed lines represent the edges that are removed from the graph. (a) The input graph before execution, (b)-(c) the intermediate states of execution, and (d) the final state.
17 Execution of Algorithm LAM. The weights are associated with the edges. Bold lines represent matched edges. Matched vertices are colored black, and the vertices being processed at a given step are colored purple. The shaded edges represent dominating edges at a current step, and dashed lines represent the edges that are removed from the graph. (a) The input graph before execution, (b)-(e) the intermediate states of execution, and (f) the final state. 36

18 Execution of Algorithm PathGrow. The weights are associated with the edges. The solid bold-lines represent edges matched in $M_1$, and the dashed bold-lines represent the edges matched in $M_2$. The matched vertices are colored black, and the vertices processed at a given step are colored purple. The shaded edges highlight the edges that are being processed for matching at a given step. (a) The input graph before execution, (b)-(f) the intermediate states of execution. 38

19 Decomposition of the maximum vertex-weight matching problem. 41

20 The symmetric difference of two matchings $M_S \oplus M_T$. Dashed lines represent edges in $M_S$ and Solid lines represent edges in $M_T$. (a) A cycle; (b)-(e) Augmenting or alternating paths. 42

21 Execution of Algorithm GlobalOptimal. (a) The input graph $G = (S, T, E)$ before execution, weights are associated only with the $S$ vertices. (b)-(e) The intermediate states of execution. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight the shortest augmenting path from a given $S$ vertex. Vertices colored Violet represent the vertex processed at a given step, and the end-point of an augmenting path if one exists. The arrows indicate the $S$ vertex that is being processed at a given step. 50

22 Execution of Algorithm LocalOptimal. (a) The input graph $G = (S, T, E)$ before execution, weights are associated only with the $S$ vertices. (b)-(d) The intermediate states of execution, (e) the final state. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight all the augmenting paths that exist from a given $T$ vertex. The arrows indicate the $T$ vertex that is being processed at a given step. 52

23 Transformation of graphs with negative weights. (a) The input graph $G = (S, T, E)$ with some negative weights associated with the vertices, (b) the new graph $G'(S', T', E')$ with zero or positive weights. The new vertices are filled with Black color. 54

24 Illustration of the reachability property. Bold lines represent the matched edges and matched vertices are colored black. 56

25 Illustrates that reachability property holds for Algorithm GlobalOptimal. Bold lines represent the matched edges and matched vertices are colored black. (a) State before $(k+1)$-th augmentation, (b) state after $(k+1)$-th augmentation. 58
26 **Greedy initialization.** Bold lines represent matched edges, and matched vertices are colored black. (a) The input graph $G = (S, T, E)$, weights are associated only with the $T$ vertices, (b) a greedy initialization that picks best augmenting paths of length one, and (c) an optimal matching. 63

27 **Execution of Algorithm** GLOBALHALF. (a) The input graph $G = (S, T, E)$ with weights associated only with the $S$ vertices, (b)-(e) the intermediate states of execution. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges mark the augmenting paths of length one (an unmatched edge) from a given $S$ vertex, (f) the final state. 67

28 **Execution of Algorithm** LOCALHALF. (a) The input graph $G = (S, T, E)$ with weights associated only with the $S$ vertices, (b)-(d) the intermediate states of execution, (e) the final state. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges mark all the augmenting paths of length one (unmatched edges) that exist from a given $T$ vertex. 69

29 **Execution of Algorithm** GLOBALTWO-third. (a) The input graph $G = (S, T, E)$ before the execution, weights are associated only with $S$ vertices, (b)-(e) the intermediate states of execution. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight the shortest augmenting path from a given $S$ vertex, and (f) the final state. 80

30 **Symmetric difference.** (a) Input graph, weights are associated only with the $S$ vertices such that $s_1 \succ s_2 \succ s_3 \succ s_4$; (b) an optimal matching $M_*$ computed by Algorithm GLOBALOPTIMAL. Bold lines represent matched edges. At step one, edge $e(s_1, t_3)$ is matched; at step two, edge $e(s_2, t_2)$ is matched; at step three, the matching is augmented via path $[s_3, t_2, s_2, t_3, s_1, t_1]$; no path exists at step four; (c) a $\frac{2}{3}$-approx matching $M_3$ computed by Algorithm GLOBALTWO-third. Wavy lines represent matched edges; At step one, edge $e(s_1, t_3)$ is matched; at step two, edge $e(s_2, t_2)$ is matched; at step three, no augmenting path of length three exists; at step four, the matching is augmented via path $[s_4, t_3, s_1, t_1]$; and (d) the symmetric difference $M_* \oplus M_3$. The bold lines denote edges matched in $M_*$, and wavy lines denote edges matched in $M_3$. 81

31 **Intuition for proof of** $\frac{2}{3}$-approx algorithm GLOBALTWO-third. For each failed $S$ vertex, Algorithm GLOBALTWO-third will match two $S$ vertices that are at least as heavy as the failed vertex. Note that the association of matched vertices with failed vertices is dynamic. The figure is representative of a state at a particular step of execution. 82

32 **New augmenting paths.** Bold lines represent the matched edges and matched vertices are colored black. The two kinds of paths in Lemma IV.4.1 are shown as $P_1$ and $P_2$. 83
33 **Execution of Algorithm LOCALTWOTHIRD.** (a) The input graph \( G = (S, T, E) \) before the execution, weights are associated only with \( S \) vertices, (b)-(d) the intermediate states of execution, and (e) the final state. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight all the augmenting paths that exist from a given \( T \) vertex. 89

34 **Performance of Approximation Algorithms.** Cardinality of matchings of the approximation algorithms as a ratio of the cardinality of the exact algorithm. 94

35 **Performance of Approximation Algorithms.** Weight of matchings of the approximation algorithms as a ratio of the weight of the exact algorithm. 94

36 **New augmenting paths.** (a) No augmenting path of length less than or equal to five exist starting at vertex \( s_1 \) in graph \( G \) at step \( k \); (b) an augmenting path of length five is available from \( s_1 \) at a step after \( k \). 95

37 **Execution of Algorithm 22.** (a) The input graph \( G = (V, E) \) with weights associated with the edges; (b) an intermediate step of execution where the pointers are set for each vertex in the graph; (c) an intermediate step where vertices that are pointing to each other are matched. Bold lines represent matched edges. Dashed lines represent the edges removed from the graph; (d) reset pointers for vertices 4 and 6; (e) edge \( (4, 5) \) is matched; (d) the final state. Matched vertices are colored black. 102

38 **Complexity analysis.** A sample graph \( G \) with weights associated with the edges such that \( w(e_1) > w(e_2) > \cdots > w(e_8) \). 103

39 **Execution of Hoepman’s Algorithm.** (a) The input graph \( G = (V, E) \) with weights associated with the edges, vertices \( \{1, 2, 3\} \) are assigned to processors \( \{P_1, P_2, P_3\} \) respectively; (b) an intermediate step of execution when \texttt{REQUEST} messages are sent by each processor to their neighbors of choice; (c) an intermediate step when edge \( (2, 3) \) is matched. (d) A possible intermediate step when processors \( P_2 \) and \( P_3 \) send \texttt{UNAVAILABLE} messages to \( P_1 \) in that order, (d') an alternative situation when \( P_1 \) gets an \texttt{UNAVAILABLE} message from \( P_3 \), and sends a \texttt{REQUEST} to \( P_2 \). Eventually, \( P_1 \) will also receive an \texttt{UNAVAILABLE} message from \( P_2 \). (e) The final state. Matched vertices are colored black. 106

40 **Data distribution among processors.** (a) The input graph \( G = (V, E) \) with weights associated with the edges; (b) The vertex set \( V \) is partitioned among two processors \( P_0 \) and \( P_1 \). Processor \( P_0 \) owns vertices \( \{0, 3, 4\} \) and Processor \( P_1 \) owns vertices \( \{1, 2, 6\} \). (c) Data storage on the processors. Along with internal edges, each processor will also store the endpoints of the edges that get cut (cross-edges). These vertices are called the \textit{ghost} vertices and are colored purple in the figure. 109
Possible communication patterns. Message types are denoted by \textbf{R} for \textbf{REQUEST}, \textbf{U} for \textbf{UNAVAILABLE}, and \textbf{F} for \textbf{FAILURE}. (a) When two requests match, it results in a matched edge. An \textbf{UNAVAILABLE} message from \(P_1\) to \(P_0\) can be responded by an \textbf{UNAVAILABLE} message (b), or a \textbf{FAILURE} message (c) from \(P_0\) to \(P_1\). (d) An \textbf{UNAVAILABLE} message from \(P_0\) can either be responded with an \textbf{UNAVAILABLE} or a \textbf{FAILURE} message by \(P_1\).

Execution of parallel approximation algorithm. (a) The input graph \(G = (V, E)\) with weights associated with the edges, vertices \(\{0, 3, 4\}\) are assigned to processor \(\{P_0\}\), and vertices \(\{1, 2, 6\}\) are assigned to processor \(\{P_1\}\). (b) an intermediate step of execution when local computations are done. \textbf{REQUEST}(4, 1) message is sent from \(P_0\) to \(P_1\); (c) Processor \(P_0\) matches edge (0, 3) and sends messages: \textbf{UNAVAILABLE}(0, 6) and \textbf{REQUEST}(4, 6) to \(P_1\). Processor \(P_1\) matches edge (1, 2) and sends messages: \textbf{UNAVAILABLE}(1, 4) and \textbf{REQUEST}(6, 4) to \(P_0\). (d) Processor \(P_0\) matches edge (4, 6) and sends message \textbf{UNAVAILABLE}(4, 1) to \(P_1\). Processor \(P_1\) matches edge (6, 4) and sends message \textbf{UNAVAILABLE}(6, 0) to \(P_0\).

Illustration of different imbalance factors on Processor \(P_i\). 

Visualization of matrix structures.

Random geometric graph. A random geometric graph with 1,000 vertices as visualized with Pajek.

SSCA\#2 graph. An SSCA\#2 graph with 1,024 vertices as visualized with Pajek.

Five-point grid graph. A 10 X 10 five-point grid graph visualized with Pajek.

Nine-point grid graph. A 10 X 10 nine-point grid graph visualized with Pajek.

Performance of Serial Approximation Algorithms: Weight. The path growing algorithms are represented by PG1, PG2, and PG3.

Performance of Serial Approximation Algorithms: Cardinality.

Performance of Serial Approximation Algorithms: Compute Time.

4k grid graph: Edgecut as a function of number of vertices. Actual edgecut for different number of partitions using multi-level K-way partitioning algorithm in Metis, and ideal edgecut given by \((2\sqrt{|V|}((\sqrt{P} - 1))\), where \(V\) is the number of vertices and \(P\) is the number of partitions.

4k grid graph: Compute time (maximum). Maximum time is the time in seconds of the slowest processor in the group of processors used to solve the problem.

4k grid graph: Compute time (average). Average time is the sum of compute time on each processor in the group divided by the number of processors in that group.

Speedup for 4k x 4k grid graph.
4k grid graph: Cardinality after Phase-1. ............................. 134
Weak scaling for grid graphs: Series-1 uses the graph size and processor combinations as shown in Table 14. ......................... 136
Weak scaling for grid graphs: Series-2 uses the graph size and processor combinations as shown in Table 14. ......................... 136
Edgecut and number of messages for different grid graphs: The graph size and processor combinations are shown in Table 14. ................. 137
320k RGG: Edgecut as a function of number of vertices. Actual edgecut for different number of partitions using multi-level K-way partitioning algorithm in Metis. ........................................ 137
320k RGG: Compute time (maximum). Maximum time is the time in seconds of the slowest processor in the group of processors used to solve the problem. ........................................ 138
320k RGG: Compute time (average). Average time is the sum of compute time on each processor in the group divided by the number of processors in that group. ........................................ 138
320k RGG: Speedup. .......................................................... 139
320k RGG: Cardinality after Phase-1. ..................................... 139
524k SSCA#2: Edgecut as a function of number of vertices. Actual edgecut for different number of partitions using K-way partitioning algorithm in Metis. ........................................ 140
524k SSCA#2: Compute time (maximum). Maximum time is the time in seconds of the slowest processor in the group of processors used to solve the problem. ........................................ 141
524k SSCA#2: Compute time (average). Average time is the sum of compute time on each processor in the group divided by the number of processors in that group. ........................................ 142
524k SSCA#2: Speedup. .......................................................... 142
524k SSCA#2: Cardinality after Phase-1. ..................................... 143
Edgecut for graphs from applications. Percentage of edges cut is a ratio of edgecut to the number of edges in the graph. ................. 144
Graphs from Applications: Compute time for different matrices with different number of processors. Compute time in seconds (log$_2$ scale) is plotted on the Y-axis, and the number of processors is plotted on the X-axis. Max is the maximum time on any given processor in the set, and Avg is the average time for a given set of processors. ........... 145
Graphs from Applications: Compute time for different matrices with different number of processors. Compute time in seconds (logarithmic scale with base two) is plotted on the Y-axis, and the number of processors is plotted on the X-axis. Max is the maximum time on any given processor in the set, and Avg is the average time for a given number of processors. The Figure also has results for two instances of SSCA#2 graphs. ................................. 146
Communication. Total number of messages sent are bounded between twice and thrice the edge cut.  

Message Bundling. Percentage bundled represents the number of messages that could be bundled in Phase 1, higher the better. Percentage sent represents the actual number of messages that get sent due to bundling, lower the better.  

Limitations of the pointer-based approach. (a) The input graph $G = (V, E)$ with weights associated with the edges; (b) an intermediate step of execution where the pointers are set for each vertex in the graph; (c) an intermediate step where vertices that are pointing to each other are matched. Bold lines represent matched edges. Dashed lines represent the edges removed from the graph; (d) the final state. Matched vertices are colored black.
CHAPTER I

INTRODUCTION

“Pioneered by the work of Jack Edmonds, polyhedral combinatorics has proved to be a most powerful, coherent, and unifying tool throughout combinatorial optimization.” - Alexander Schrijver [66]

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 $M$ are incident on the same vertex. A graph can additionally have weights associated with the edges, or the vertices, or both. The objective of the matching problem can be to maximize the number of edges in $M$ (a maximum cardinality matching); or to maximize the total weight of matched edges (a maximum edge-weight matching problem); or to maximize the total weight of matched vertices (a maximum vertex-weight matching). Thus, we have three basic variations of the matching problem:

1. Maximum cardinality matching (MCM),
2. Maximum edge-weight matching (MEM), and

Figure 1 sketches a landscape of the matching problems. While the three problems are closely related, they also have unique features that distinguish them from each other. The cardinality and the edge-weighted matching problems have been studied extensively. However, the vertex-weighted matching problem has not received as much attention. The main focus of our work, therefore, is on the vertex-weighted matching problem.

An underlying combinatorial problem in many scientific computing applications is finding matchings in graphs. For example, the problem of coarsening a graph without losing the characteristics of the original graph in multi-level partitioning algorithms can be solved by computing a matching problem. The matching problem can be solved in polynomial time, and we will provide a detailed discussion of some of these algorithms in Chapter II. However, for many of the large-scale scientific computing applications, polynomial-time solutions are not always sufficient. Thus, there is a need for faster approximation algorithms for the matching problem. The weighted
FIG. 1: Landscape of the matching problems. The vertex-weighted matching problem can be formulated as an edge-weighted matching problem. The weighted matching algorithms utilize techniques developed for the cardinality matching problem. The arrows indicate these relationships.

The vertex-weighted matching problem in particular has numerous applications and therefore many linear-time approximation algorithms have been proposed for the same [24, 64]. The best known approximation for the edge-weighted matching problem is a $\left(\frac{2}{3} - \epsilon\right)$-approx algorithm with a run time of $O(|E| \log \frac{1}{\epsilon})$, where $|E|$ represents the number of edges and $\epsilon$ is a positive real number [59]. In this work we propose a $\frac{2}{3}$-approx algorithm for vertex-weighted matching with linear-time performance for a class of graphs with some restrictions.

Along with the development of new algorithms, there is a need for good open source implementation of the matching algorithms. Driven by these needs, we propose to accomplish the following with this dissertation:

- development of new exact and approximation MVM-algorithms,
- development of open source implementation of these algorithms, and
- development of use-case models for the vertex weighted matching problem.

We will now provide a brief outline of this thesis.
I.1 OUTLINE

The thesis is organized into six chapters. In this chapter we present an overview and motivation for this work. The second chapter provides an introduction to the matching theory, and discusses background and related work. Third and fourth chapters discuss the exact and approximation algorithms for the maximum vertex-weight matching problem (MVM) respectively. In chapter five we provide details of a parallel half-approximation algorithm and experimental results on a distributed memory parallel computer. The sixth chapter provides conclusions and plans for the future work.

In order to motivate our work, we will now provide a brief introduction to a field of study known as combinatorial scientific computing (CSC), where this dissertation belongs to. CSC encompasses three broad fields - computer science, applied mathematics, and operations research.

I.2 COMBINATORIAL SCIENTIFIC COMPUTING

Combinatorial scientific computing is the development, analysis and application of discrete algorithms for applications in scientific computing [33, 34]. The three components that characterize CSC are (i) identifying a scientific computing problem, and building an appropriate combinatorial model for this problem; (ii) developing an efficient solution for the combinatorial problem; and (iii) developing required software tools and evaluating the performance on representative test instances.

Computational simulation of a physical phenomenon is a better alternative to experiments in many situations, and in some cases the only alternative. However, realistic simulations of physical phenomena are extremely difficult. Computational challenges and massive resource requirements for numerous applications in science and engineering have been extensively documented by hundreds of field experts in the SCALeS (A Science-Based Case for Large-Scale Simulation) reports [42]. Combinatorial algorithms play a critical role in computational science by enhancing the efficiency of numerical algorithms, and in many cases enables a computation which would be infeasible otherwise. The role of combinatorial algorithms in scientific computing have been discussed in detail elsewhere, and we refer the readers to a paper by Hendrickson and Pothen [34] for one such discussion.

Approximation algorithms are generally developed for intractable problems [35].
However, approximation algorithms for problems that have known polynomial-time solutions are increasingly becoming popular. The motivation for this comes from the fact that many polynomial-time algorithms can be computationally very expensive for large-scale problems. A further need for approximation algorithms can come from resource limitations. One example is a scheduling problem in high-speed network switches, where the algorithms not only need to be fast, but should also be easy to implement in hardware [52].

As one of the fundamental combinatorial problems, matching is important both theoretically and practically. Theoretically, it is interesting because of its similarity to many NP-complete problems like the Integer Programming Problem, while at the same time lending itself to a polynomial time solution [57]. Such solutions have been made possible due to ingenious techniques like augmenting paths, and the identification and shrinking of blossoms [8, 48]. We believe that further study of these tools and techniques will promote good solutions for other combinatorial problems. The matching problem is also important from a practical perspective because of its use in many applications in diverse fields of science and engineering. Some of these applications are discussed in [1, 24, 25, 26, 8, 40, 46, 53, 62, 63, 64, 65]. In this thesis, we will discuss two such applications in order to motivate this study.

I.3 MOTIVATION

Vertex-weighted matching has many applications. Some of the problems that use maximum vertex-weighted matching (MVM) are:

- Sparsest column-space basis problem [60],
- Facility scheduling problem [11], and
- Reverse spanning tree problem [2].

In order to illustrate the process of modeling an application as a vertex-weighted matching problem, we will discuss two specific examples. The first problem is a specialized version of the dating problem provided as an exercise in [9] that we call a mercenary dating problem, and the second is the computation of a sparsest column-space basis of a matrix [60].
Mercenary Dating Problem

A dating service is provided with data from $m$ men and $n$ women sufficient to determine which pairs of men and women are compatible. The data also includes the price that each person will pay for getting matched; assume unique positive prices. The total revenue for the dating service is proportional to the total number of dates that it can arrange, and on the individual price that it receives from the matched people. The objective is to maximize the total revenue for the dating service (mercenary). Note that the with the assumption of positive prices revenue can always be increased by increasing the number of people that will get matched. We will prove this later. Some people might remain unmatched (a perfect matching may not exist).

Let us model the problem as a bipartite graph $G(S, T, E)$ with weight functions $w_S : S \rightarrow \mathbb{R}^+$ and $w_T : T \rightarrow \mathbb{R}^+$. The vertex set $S$ represents men and the vertex set $T$ represents women. A vertex in $S$ (and $T$) represents a single person. The compatibility of a man $s$ with a woman $t$ is represented as an edge $e_{st} \in E$. The weight function on the vertices represents the commission that each person is willing to pay if matched. The objective function of the mercenary dating problem can be accomplished by computing an MVM in $G$.

We will now provide an intuition for solving the problem by computing a maximum vertex-weight matching in the graph. The details of the algorithm will be discussed in Chapter III. First, ignore the weights associated with the $T$ vertices. Try to maximize the revenue that can be generated by matching as many men as possible based on the weights associated with the $S$ vertices. This simply reduces to computation of a maximum cardinality matching in $G$ with a particular order for processing the vertices (decreasing order of weights). Similarly, repeat the process by ignoring the weights associated with the $S$ vertices and by trying to maximize the revenue by matching as many women as possible. Thus, we now have two different matchings from two separate computations. We can no merge these two matchings together by retaining all the $S$ vertices matched in the first matching as well as all the $T$ vertices matched in the second matching. This results in an optimal solution to the mercenary dating problem. The details are provided in Chapter III.
Sparsest column-space Basis Problem

Another application of vertex weighted matching arises in the computation of a sparsest column-space basis (SCB) of a matrix. The sparsest column-space basis problem is an instance of the nice-basis problem that has numerous applications in scientific computing, including models of deforming structures, circuit and device modeling, equality constrained optimization, etc. We refer the readers to [60] for details. We will now briefly discuss the role of vertex weighted matching in the solution of SCB. This is a novel method for computing a SCB and has not been published elsewhere.

Consider a matrix $A$ with $k$ rows and $n$ columns, $n > k$, and rank $k$. A set of columns $C = \{c_1, c_2, \cdots, c_l\}$ is linearly independent if none of the columns in $C$ can be expressed as a linear combination of the others. The maximal number of linearly independent columns of $A$ is called the column rank of $A$. The row rank of $A$ is defined similarly. Since the row and column ranks are equal, they are called the rank of $A$. A generalized diagonal of $A$ is a subset of nonzeros with at most one chosen from each row and each column. The maximum number of nonzeros in a generalized diagonal is called the structural rank of $A$. The numerical rank of a matrix (we have called this the rank) is less than or equal to the structural rank of $A$. In the following discussions we will make a simplifying assumption that the numerical and the structural ranks of a matrix are equal.

A basis for the column-space of $A$ is a linearly independent set of columns with maximum rank (by the assumption on $A$, this is $k$). A sparsest basis for the column-space of $A$ is a basis with the fewest nonzeros in it. Formally, the sparsest column-space basis problem (SCB) can be defined as:

**Definition I.3.1.** Given a sparse matrix $A$ of rank $k$, with $k$ rows and $n > k$ columns, find a sparsest basis $B$ for its column-space.

The sparsest column-space basis selects $k$ out of $n$ sparse columns of $A$. A graphical representation of SCB is given by Figure 2. For a matrix with $k$ rows and $n$ columns there could be $\binom{n}{k}$ potential column-space bases. However, a simple greedy algorithm, as follows, works: Start with an empty set (of columns) $B$. Find the sparsest column based on the number of non-zeros in the column and represented with a weight function $w_i$. Add this column to $B$. Until $k$ columns have been added to $B$, add new (sparsest) columns such that they are linearly independent of the
FIG. 2: Representation of a sparsest column-space basis problem. A matrix $A$ with $k$ rows and $n$ columns, and a basis $B$ with $k$ rows and $k$ linearly independent columns. The set $B$ now represents the sparsest set over all choices of sparsest column-space bases. One step of this algorithm is illustrated in Figure 3. A sparsest column-space basis can be computed in $O(n^2)$ time and a $\frac{1}{2}$-approx solution in $O(nnz(A) + k^2)$ time, where $nnz(A)$ denotes the number of nonzero elements in $A$ [60].

FIG. 3: A greedy algorithm for computing a sparsest column-space basis. (a) State before augmenting a basis $B_i$ with a column of current heaviest weight $w_{max}$ from $C$; (b) state after augmenting a basis with a sparsest linearly independent column from $C$.

The proof that such a greedy algorithm will solve the sparsest column-space basis problem is given by a theory about greedy algorithms: combinatorial structures known as matroids, as named by Hessler Whitney [19, 45].

Definition 1.3.2. A matroid $M = (E, \mathcal{I})$ is defined as a set of elements $E$, and a nonempty collection of subsets, $\mathcal{I}$, of $E$ defined to be independent. The three properties that an independent set $I \in \mathcal{I}$ needs to satisfy are:

1. The empty set is independent;
2. **Subsets of an independent set are independent;**

3. **Given two independent sets with unequal cardinalities, the smaller set can be augmented with some element from the larger set to form a larger independent set (this is called the exchange property).**

Based on this background, we will now discuss how computing a sparsest column-space basis can be transformed into a maximum vertex-weight matching problem. A matrix $A$ with $k$ rows and $n$ columns can be represented as a bipartite graph $G = (S, T, E)$ with weight function $w : S \rightarrow \mathbb{R}^+$, where set $S$ represents the columns, set $T$ represents the rows, and each nonzero element in $A$ is represented by an edge $e_{st} \in E$. The weight of a column vertex is given by $w(s) = k + 1 - \text{deg}(s)$, where $\text{deg}(s)$ represents the number of nonzeros in column $s$. A matrix and its bipartite graph representation are shown in Figures 4.(a) and 4.(b).

A matching $M$ in $G$ corresponds to a subset of nonzeros in $A$, with at most one from each column and each row (see Figure 4.(a) for an example). By permuting the rows and columns of $A$, we can put the nonzeros corresponding to a matching on the diagonal of $A$. This is illustrated in Figure 4.(c). As discussed earlier, the maximum number of nonzeros in a matching is the structural rank of a matrix. If we make a simplifying assumption that the numerical rank of $A$ is equal to the structural rank of $A$, then a maximum matching in $G$ will result in a candidate basis with full structural rank. While the assumption that the numerical rank of a matrix is equal to the structural rank is true for many scientific computing applications, it is not always a correct assumption. However, the correctness of a candidate basis with full structural rank can be checked by numerical factorization.

Thus, the greedy algorithm for computing a sparsest basis, discussed earlier, can now be replaced by an algorithm for computing a matching. Specifically, a maximum vertex-weight matching, since it will compute a maximum matching that is as sparse as possible. The weights on the $S$ vertices are formulated such that maximizing the total weight of the matched vertices will minimize the number of nonzeros in the submatrix induced by this matching (basis $B$).

Spencer and Mayr provide a $O(\sqrt{nm \log n})$ time algorithm [69] for computing a maximum vertex-weight matching, where $n$ denotes the number of vertices and $m$ denotes the number of edges in a graph. Exact algorithms tend to be expensive for large-scale problems, and therefore, there is a need for approximation algorithms. We
FIG. 4: Computation of a sparsest column-space basis with a maximum vertex-weight matching. (a) A matrix $A$; (b) A bipartite graph $(G)$ representation of $A$. Numbers on the right indicate the weight of each $S$ vertex. Bold lines represent the matched edges, and matched vertices are colored black; (c) A candidate basis as computed by a maximum vertex-weight matching in $G$.

provide detailed discussions on exact and approximate MVM-algorithms in Chapters III and IV.

In summary the motivation for this work comes from:

- **Theory**: the need for a systematic study of vertex-weighted matching problem,

- **Implementation**: the need for public-domain tools that implement matchings, and

- **Applications**: the need for solutions of applications of vertex-weighted matching.

### I.4 CONTRIBUTIONS

The contributions of this thesis are:

1. Theory:
   - New framework for developing proof of correctness for vertex weighted matchings;
   - New $\frac{1}{2}$-approx algorithms for vertex weighted matchings;
   - New $\frac{2}{3}$-approx algorithm for bipartite vertex weighted matchings;

2. Experiments:
• Open-source library of C++ routines to compute various kinds of matchings;

• Open-source library of C++ and MPI routines to compute approximate matchings in parallel.

• Extensive experimental study of various (serial) matching algorithms, and scalability study of $\frac{1}{2}$-approx parallel algorithm with up to 8,192 processors.

3. Applications:

• Study of applicability of vertex weighted matchings in solving the sparsest basis problem.

• Study of approximation algorithms in sparse matrix computations.

I.5 CHAPTER SUMMARY

In this chapter we provided the motivation and rationale for this dissertation. We also introduced two specific application of the vertex weighted matching problem. We show how the sparsest-basis problem can be efficiently solved by modeling it as a maximum vertex-weight matching problem and concluded the chapter by listing some of the contributions of this work.
CHAPTER II
BACKGROUND AND RELATED WORK

“It (matching) is included in (class) P, thanks to the ingenious introduction of nontrivial combinatorial tools such as alternating paths and blossoms.” - Marek Karpinski and Wojciech Rytter [39]

Matching theory has been studied in great detail [8, 45, 48, 57, 66]. In this chapter, we will provide a brief introduction to matchings in graphs. We will also introduce the basic tools and techniques to compute a matching. We will discuss both exact and approximation algorithms for the maximum cardinality and the maximum edge-weight matchings in bipartite graphs. The approximation algorithms are also applicable to nonbipartite graphs. We will keep the discussion on the exact algorithms brief. Our goal is to provide sufficient background for a better understanding of the proposed algorithms. Since the approximation algorithms have been more recently developed, we will discuss them at a relatively greater detail. We refer the reader to above cited references for a thorough discussion on matching theory and algorithms.

II.1 INTRODUCTION

A graph \( G \) is a pair \((V, E)\), where \( V \) is a set of vertices and \( E \) is a set of edges that represent a binary relation on \( V \). A simple instance of a graph is shown in Figure 5. The vertices are represented with small circles, and the lines that connect two vertices represent the edges. In a graph, weights can be associated with edges, vertices, or both. In this proposal, we will only consider weights with real positive numbers. Graphs with negative weights will have to be considered separately. The association of weights in a graph \( G = (V, E) \) can be represented as \( w : E \rightarrow \mathbb{R}^+ \) for a weight function on edges, and \( w : V \rightarrow \mathbb{R}^+ \) for a weight function on vertices.

A bipartite graph \( G = (S, T, E) \) is a graph in which the vertex set \( V = S \cup T \) can be partitioned into two sets \( S \) and \( T \), \( S \cap T = \phi \), such that no two vertices in \( S \), or in \( T \), are joined by an edge. An example of a bipartite graph is shown in Figure 5. Since edges in a bipartite graph always join an \( S \) vertex to a \( T \) vertex, cycles of odd length cannot exist. Absence of odd-length cycles is a distinguishing characteristic.
of bipartite graphs, that is important and well exploited in the context of matching algorithms.

We use the following notations. Given a graph \( G = (V, E) \), an edge \( e \) belong to Set \( E \). We can further specify the two endpoints \( (u, v) \) of an edge as \( e_{uv} \). The weight assigned with an edge is denoted as \( w(e) \), and the weight of a vertex \( v \) is denoted as \( w(v) \). Given a vertex \( v \in V \), the set of edges incident on it is called the adjacency set, and denoted as \( \text{adj}(v) \). We will introduce other symbols and notations where appropriate.

A matching in a graph can be defined as follows:

**Definition II.1.1.** 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 \( M \) are incident on the same vertex.

A matching can also be seen as a pairing of two objects in the set. Using the example of mercenary dating problem that we introduced in Chapter 1, the set of men is denoted by \( \{S_1, S_2, S_3\} \), and the set of women is denoted by \( \{T_1, T_2, T_3\} \). A matching is pairing of a man with a woman such that no man is paired with more than one woman, and no woman is paired with more than one man. This is illustrated in Figure 5.

**FIG. 5:** An example of matching. (a) A bipartite graph \( G \), (b) a matching \( M \) in \( G \). Bold lines represent matched edges, and matched vertices are colored black.

**CLASSIFICATION**

Based on different criteria the matching problem can be classified as follows:

- **Input graph:** Bipartite and Nonbipartite,
- **Objective function:** Cardinality and Weighted,
• Placement of weights in the graph: Edge-weighted and Vertex-weighted,

• Optimality: Exact and Approximate.

A given matching problem can thus be specified as an exact maximum edge-weight matching problem, or as a $\frac{1}{2}$-approx vertex-weighted matching problem. The landscape of matching algorithms is provided in Figure 1.

The odd-length cycles that exist in nonbipartite graphs need special consideration and will significantly increase the conceptual complexity of a matching algorithm for nonbipartite graphs. However, the computational complexity might remain the same as that for bipartite graphs.

The cardinality of a matching is the number of edges in it and is denoted by $|M|$. Based on the cardinality there can be three types of matchings. A maximal matching is a matching that cannot be augmented by adding a new edge to it. However, it might be possible to increase the cardinality of a maximal matching by changing the set of matched edges. A maximum matching in a graph is a matching of maximum cardinality among all possible matchings. When all the vertices are matched, the matching is called a perfect matching. While a maximum matching is also a maximal matching, a maximal matching is not always a maximum matching. However, a perfect matching necessarily has maximum cardinality. These three types of matchings are illustrated in Figure 6.

![Figure 6: Types of matchings. Matched edges are represented with bold lines and matched vertices are filled with black color. (a) A maximal matching, (b) a maximum matching, and (c) a perfect matching.](image)

In a graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}^+$, the edge-weight of a matching $M$ is the sum of weights of the matched edges $\sum_{e \in M} w(e)$. For a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$, the vertex-weight of a matching is the sum of weights of matched vertices $\sum_{v \in V(M)} w(v)$, where $V(M)$ represents the set of matched vertices. We will denote the edge-weight and the vertex-weight as weight, and depend on the context for specific reference as to whether the weights
are associated with the edges or the vertices. For the current discussion we will only consider positive weights. We will later show that the same algorithms can be extended to include negative weights. A *maximum edge-weight matching*, also known as a maximum weighted matching, is a matching of maximum edge-weight among all possible matchings in a graph. A maximum edge-weight matching can be of maximal, maximum or perfect cardinality. A *maximum vertex-weight matching* is a matching of maximum vertex-weight among all possible matchings in a graph. When the weights are positive, a maximum vertex-weight matching is also a matching of maximum cardinality, which will proved in Chapter III.

An $\alpha$-approx algorithm computes a solution that is within a factor of $\alpha$ of the optimal value. For example, a $\frac{1}{2}$-approx algorithm for a maximum edge-weight matching problem guarantees that the weight of an approximate matching computed by the algorithm is at least half of the weight of an optimal matching. If $M_2$ denotes a matching computed by a $\frac{1}{2}$-approx algorithm, and $M_*$ denotes an optimal matching, then

$$\sum_{e \in M_2} w(e) \geq \frac{1}{2} \sum_{e \in M_*} w(e)$$  \hspace{1cm} (1)

Approximation algorithms for maximum cardinality matching are relatively easier than approximation algorithms for weighted matchings. While computing a linear time $\frac{1}{2}$-approx to maximum cardinality matching (maximal) is trivial, computing the same for weighted matching is not. We will discuss these approximation algorithms in Section II.5.
II.2 FOUNDATIONS

One of the most fundamental techniques in matching is the technique of augmentation. Given a graph $G = (V, E)$ and a matching $M$ in $G$, a path is said to be alternating if it alternates between an edge in $M$ (matched) and an edge not in $M$ (unmatched). An alternating path that starts and ends with edges that are not in $M$ (unmatched) is called an augmenting path. Note that an augmenting path will always have an odd number of edges and an even number of vertices. A few examples of paths are illustrated in Figure 7.

![Figure 7: Types of paths. Matched edges are represented with bold lines and matched vertices are colored black.](image)

(a) An alternating path starting with an unmatched vertex, (b) an alternating path starting with a matched vertex, and (c) an augmenting path.

The symmetric difference of two sets, denoted by the symbol $\oplus$, is computed by choosing the elements that are present in either of the sets, but not in both. Mathematically, the symmetric difference of two sets $M$ and $P$ is shown in Equation 2. The operator $\setminus$ represents the set resulting from retaining only those elements in the set on the left hand side of the operator that do not also exist in the set on the right hand side of the operator (the set minus operator).

$$M \oplus P = (M \setminus P) \cup (P \setminus M) \quad (2)$$

In the context of matching, the symmetric difference operation is important due to Lemma II.2.1, which states that the cardinality of a current matching can always be increased by performing a symmetric difference with an augmenting path. The process of symmetric difference is illustrated in Figure 8. Note that although the matched edges change, the matched vertices will always remain matched.

**Lemma II.2.1.** Consider a graph $G = (V, E)$ and a matching $M$. Let $P$ be an augmenting path in $G$ with respect to $M$. The symmetric difference, $M' = M \oplus P$, is a matching of cardinality $(|M| + 1)$.
FIG. 8: *Augmentation by symmetric difference*. The matched edges are represented with bold lines and matched vertices are colored black. (a) Before augmentation, (b) after augmentation.

*Proof.* There are two parts to the proof. First we will prove that the symmetric difference $M \oplus P$ will result in a matching, and then we will prove that the symmetric difference will result in a matching that increases the cardinality by one.

*(i)* An augmenting path $P$ is of the form $[e_1, e_2, e_3, \cdots, e_n]$, where all odd-indexed edges $\{e_1, e_3, \cdots, e_n\}$ are unmatched, and all even-indexed edges $\{e_2, e_4, \cdots, e_{n-1}\}$ are matched. Also, edges $e_1$ and $e_n$ are unmatched, and $n$ is an odd number. The symmetric difference is given by $M \oplus P = (M \setminus P) \cup (P \setminus M)$. The edges obtained by the operation $(M \setminus P)$ contain those edges that are in $M$, but are not part of the path $P$, and therefore a set of independent edges (it retains the matched edges independent of $P$). The edges obtained by the operation $(P \setminus M)$ contain those edges that are on the path $P$, but are not in $M$ (the unmatched edges in $P$). By definition, an augmenting path $P$ connects two distinct unmatched vertices, and therefore, edges $e_1$ and $e_n$ are independent edges. All the intermediate edges in $\{P \setminus M\}$ are also independent edges because they share vertices with matched edges. Therefore, the symmetric difference $M \oplus P$ results in a matching.

*(ii)* An augmenting path $P$ starts and ends with an unmatched edge, therefore, the number of unmatched edges in $P$ is exactly one larger than the number of matched edges in $P$. Thus, symmetric difference $M \oplus P$ results in a matching of cardinality of $(|M| + 1)$. 

The concept of symmetric difference immediately gives us a basic technique to compute a matching: find an augmenting path, and perform the symmetric difference. The proof of correctness for such an algorithm is given by Theorems II.2.1 and II.2.2.

**Theorem II.2.1** (Berge [1957]). A matching $M$ in a graph $G$ is a maximum matching if and only if there is no $M$-augmenting path in $G$. 

Proof. There are two aspects to the proof. 

(i) Suppose there exists an $M$-augmenting path in $G$, then the cardinality of $M$ can be increased by one, and therefore, $M$ is not a maximum matching and contradicts the assumption (follows from Lemma II.2.1). Therefore, if $M$ is a maximum matching, then there exist no $M$-augmenting paths in $G$.

(ii) Suppose that there exist no $M$-augmenting paths in $G$, and yet, $M$ is not a maximum matching. Let $M^*$ be a maximum matching in $G$. The symmetric difference $M \oplus M^*$ will result in a collection of alternating paths and cycles as illustrated in Figure 9. If one of these alternating paths is $M$-augmenting, then there also exists an $M$-augmenting path in $G$, and therefore, contradicts the assumption (follows from part (i)). Also, by assumption there are no $M^*$ augmenting paths in $M \oplus M^*$. Thus, the symmetric difference $M \oplus M^*$ will consist of alternating paths that are not augmenting paths, and cycles, and therefore, an equal number of edges from $M$ and $M^*$. Alternatively, $|M| = |M^*|$, and the theorem holds.

\[ \square \]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig9.png}
\caption{The symmetric difference of two matchings $M_S \oplus M_T$. Dashed lines represent edges in $M_S$ and Solid lines represent edges in $M_T$. (a) A cycle; (b)-(e) Augmenting or alternating paths.}
\end{figure}

**Theorem II.2.2.** Consider a graph $G = (V,E)$ and a matching $M$. Let $P$ be an augmenting path with two unmatched vertices $v$ and $w$ as endpoints. If there exists no augmenting path in $G$ starting from an unmatched vertex $u$ with respect to $M$, then there is no augmenting path from $u$ with respect to $M \oplus P$ either.

Proof. Let the augmenting path starting at $u$ be $Q$, and the augmenting path between $v$ and $w$ be $P$. This is illustrated in Figure 10. There are two possibilities:

(i) Paths $P$ and $Q$ do not intersect. This means that the two paths do not have any
vertices or edges in common. This is illustrated in Figure 10.(a). In such a case $P$
will not have any effect on the possibility of an augmenting path starting at $u$. If no
augmenting path exists from $u$ with respect to $M$, then no augmenting path exists
from $u$ with respect to $M \oplus P$ either. Therefore, the theorem holds.

(ii) Paths $P$ and $Q$ intersect each other. Path $Q$ is of the form $[u, u_1, \ldots, u_j, \ldots, u']$. Let $u_j$ be the first vertex on $Q$ that is also on $P$. This is illustrated in Figure 10.(b). The portion of $Q$ from $u$ up to $u_j$, along with the portion of $P$ that is incident on $u_j$ with a matched edge ($Q'$ in Figure 10.(b)), forms an augmenting path starting at $u$ with respect to $M$. This contradicts the assumption, and therefore, the theorem holds.

\[ \square \]

![Figure 10: Effect of $M \oplus P$. Bold lines represent matched edges and matched vertices are colored black. (a) Paths $P$ and $Q$ do not intersect; (b) paths $P$ and $Q$ intersect. This figure has been adapted from [57].](image)

**Corollary II.2.1.** If at some stage of an augmentation-based matching algorithm, there is no augmenting path starting at vertex $u$, then there will be no augmenting path from $u$ at any future step in the algorithm.

**Proof.** Inducting on the number of steps that remain after discovering that no augmenting path exists from a vertex $u$, we can use Theorem II.2.2 to show that there never will be an augmenting path from $u$, if none existed when $u$ was processed the first time. \[ \square \]

Thus, from Corollary II.2.1, it is enough if we process a given vertex only once. We will now discuss techniques to perform the search for augmenting paths in a graph.
GRAPH SEARCH TECHNIQUES FOR MATCHING

Searching for an augmenting path in a graph with respect to a matching is one the basic steps in the computation of a matching. There are two basic approaches to find an augmenting path - a breadth-first search, and a depth-first search. The difference between a breadth-first and a depth-first search comes from the way the elements are queued during a search. We will define two data structures known as a pseudo-queue, and a pseudo-stack. A pseudo-queue is different from a regular queue data structure in that the former excludes duplicate elements. Note, that Algorithm 1 does not attempt to add duplicates, and therefore, does need this special data structure. Similarly, there are no duplicates in a pseudo-stack. An additional characteristic of a pseudo-stack is that if a new element that is being added to the pseudo-stack already exists, then it is moved to the top of the pseudo-stack. We need vectors to store information about the parent-child relationships (parent), distance from the source (depth), and state of processing (color). We initialize color with $\phi$ for all vertices, and update it to Processable or Processed.

A breadth-first search is illustrated in Algorithm 1, and works as follows. Initialize the data structures by setting the color, parent and depth values to zeros. Start with a vertex $u$ and add it to the pseudo-queue data structure and mark it as Processable. Enqueue the vertices adjacent to $u$ and mark them as Processable. Add $u$ as the parent of all the enqueued vertices and set the depth values for these elements one greater than the depth value of the parent. Repeat the steps by dequeuing the front of the queue each time, until all the vertices have been processed. A breadth-first search on a small graph is illustrated in Figure 11.

A depth-first search is illustrated in Algorithm 2. The algorithm functions as follows. Start with a vertex $u$ and mark it as Processed. Enqueue the vertices adjacent to $u$ in a pseudo-stack data structure, and mark them as Processable. Add $u$ as the parent of all the enqueued vertices, and a depth value one greater than the depth of the parent. Dequeue the top of the pseudo-stack, and repeat the steps until all the vertices have been processed. A depth-first search on a small graph is illustrated in Figure 12.

The search for an augmenting path can be breadth-first, depth-first or a combination of these. The search could either start from one vertex (single-source), or simultaneously from a set of unmatched vertices (multiple-source). The general strategy is to find a shortest-augmenting path. Therefore, breadth-first search is generally
Algorithm 1 \textbf{Input:} A graph $G$ and a vertex source $u$. \textbf{Output:} A breadth-first tree. \textbf{Associated data structures:} $Q$ is a queue data structure. \textbf{Effect:} perform a breadth-first search.

1: \textbf{procedure} BreadthFirstSearch$(G = (V, E), u)$

2: \hspace{1em} \textbf{for all } $v \in V$ \textbf{do}

3: \hspace{2em} $\text{color}[v] = \phi$;

4: \hspace{2em} $\text{parent}[v] = 0$;

5: \hspace{2em} $\text{depth}[v] = 0$;

6: \hspace{1em} \textbf{end for}

7: $Q \leftarrow \{u\}$;

8: $\text{color}[u] \leftarrow \text{PROCESSABLE}$;

9: \textbf{while} $Q \neq \phi$ \textbf{do}

10: \hspace{2em} pick $v$ from $Q$;  \hspace{4em} \triangleright$Initialization$

11: \hspace{2em} Q \leftarrow Q \setminus \{v\}$;

12: \hspace{2em} $\text{color}[v] \leftarrow \text{PROCESSED}$;

13: \hspace{2em} \textbf{for all } $w \in \text{adj}[v]$ \textbf{do}

14: \hspace{3em} \textbf{if} $\text{color}[w] \neq \phi$ \textbf{then}

15: \hspace{4em} \text{continue};

16: \hspace{3em} \textbf{end if}

17: \hspace{3em} $\text{parent}[w] \leftarrow v$;

18: \hspace{3em} $\text{depth}[w] \leftarrow \text{depth}[v] + 1$;

19: \hspace{2em} $Q \leftarrow Q \cup \{w\}$;  \hspace{4em} \triangleright$Enqueue$

20: \hspace{2em} $\text{color}[w] \leftarrow \text{PROCESSABLE}$;

21: \hspace{2em} \textbf{end for}

22: \hspace{1em} \textbf{end while}

23: \textbf{end procedure}
Algorithm 2  

**Input:** A graph $G$ and a vertex source $u$.  
**Output:** A breadth-first (or depth-first) tree.  
**Associated data structures:** $S$ is a pseudo-stack data structure.  
**Effect:** perform a depth-first search.

1: procedure DEPTH-FIRST-SEARCH($G = (V, E), u$)  
2: for all $v \in V$ do  
3:  $\text{color}[v] = \phi$;  
4:  $\text{parent}[v] = 0$;  
5:  $\text{depth}[v] = 0$;  
6: end for  
7: $S \leftarrow \{u\}$;  
8: $\text{color}[u] \leftarrow \text{PROCESSABLE}$;  
9: while $Q \neq \phi$ do  
10:  pick $v$ from $S$;  
11:  $S \leftarrow S \setminus v$;  
12:  $\text{color}[v] \leftarrow \text{PROCESSED}$;  
13:  for all $w \in \text{adj}[v]$ do  
14:      if $\text{color}[w] \neq \phi$ then  
15:         move $w$ to the top of $S$;  
16:      continue;  
17:  end if  
18:  $\text{parent}[w] \leftarrow v$;  
19:  $\text{depth}[w] \leftarrow \text{depth}[v] + 1$;  
20:  $S \leftarrow S \cup \{w\}$;  
21:  $\text{color}[w] \leftarrow \text{PROCESSABLE}$;  
22: end for  
23: end while  
24: end procedure
FIG. 11: *Breadth-first search.* The vertex being processed at a given step is colored purple, and also marked by an arrow. The shaded lines represent the processed edges. The vertices that have already been processed are colored black. The adjacency list for each vertex is maintained in an increasing order of the indices of vertices. (a) The input graph before execution, (b)-(f) the intermediate states of execution. State of the pseudo-queue at each step: (b) \([2, 3, 4]\) (c) \([3, 4, 5]\), dequeue 2, enqueue 5; (d) \([4, 5, 6]\) dequeue 3, enqueue 6; (e) \([5, 6]\) dequeue 4; (f) \([6]\) dequeue 5.

used. Once an augmenting path is discovered, augmentation can be performed by either along a single path, or simultaneously along a set of vertex-disjoint augmenting paths. Thus the three strategies are:

1. Single-source single-path, illustrated in Figure 13, uses a breadth-first search.
2. Multiple-source single-path, illustrated in Figure 14, uses a breadth-first search.
3. Multiple-source multiple-path, illustrated in Figure 15, uses a combined breadth-first and depth-first search.

We will provide more details about these approaches in the following discussions on maximum cardinality and maximum edge-weight matching algorithms.
FIG. 12: *Depth-first search*. The vertex being processed at a given step is colored purple, and also marked by an arrow. The shaded lines represent the processed edges. The vertices that have already been processed are colored black. The adjacency list for each vertex is maintained in an increasing order of the indices of vertices. 

(a) The input graph before execution. (b)-(f) the intermediate states of execution. State of the pseudo-stack at each step: (b) [2, 3, 4] (c) [2, 3, 5] pop 4, move 2, move 3, push 5; (d) [3, 2, 6] pop 5, move 2, push 6; (e) [2, 3] pop 6, move 3; (f) [2].

FIG. 13: *Single-source single-path technique*. The vertex being processed at a given step is colored purple, and also pointed by an arrow. The shaded lines represent potential augmenting paths. Bold lines represent matched edges and matched vertices are colored black. 

(a) The input graph before execution, (b)-(d) the intermediate states of execution, and (e) the final state.
FIG. 14: *Multiple-source single-path technique*. The vertices being processed at a given step are colored purple. The shaded lines represent potential augmenting paths. Bold lines represent matched edges and matched vertices are colored black. (a) The input graph before execution, (b)-(d) the intermediate states of execution, and (e) the final state.

FIG. 15: *Multiple-source multiple-path technique*. The vertices processed at a given step are colored purple. The shaded lines represent potential augmenting paths, bold lines represent matched edges and matched vertices are colored black. (a) The input graph before execution, (b) the intermediate state of execution, and (c) the final state.
II.3 MAXIMUM CARDINALITY MATCHING

Maximum cardinality matching (MCM) algorithms for bipartite graphs are conceptually easier than those for nonbipartite graphs. In this section, we will discuss MCM algorithms for bipartite graphs, and refer the readers to [28, 29, 8, 45, 48, 57, 66, 73] for discussions on algorithms for nonbipartite graphs. We will provide two algorithms for MCM, a simple algorithm based on the single-source single-path approach, and an advanced algorithm based on the multiple-source multiple-path approach for searching an augmenting path.

The simple version of MCM is given in Algorithm 3. The algorithm functions as follows. Let $G = (S, T, E)$ be a bipartite graph, and $M$ an empty matching. Find an $M$-augmenting path $P$ in $G$, and perform the symmetric difference $M \oplus P$ to increase the cardinality of the current matching. Repeat the process until no $M$-augmenting paths exist in $G$. A breadth-first or depth-first search, as described in Algorithms 1 and 2, can be used to find an augmenting path starting at a given vertex. However, the former is preferred because it retrieves the shortest augmenting path from a given source, if such a path exists. This graph search operation is bounded by $O(m)$, where $m = |E|$ is the number of edges in $G$. Since $G$ is a bipartite graph, edges will always connect an $S$ vertex to a $T$ vertex. Therefore, it is sufficient to loop either over the $S$ vertices, or the $T$ vertices. A vertex needs to be processed only once, this follows from Corollary II.2.1. Thus, Algorithm MAX-CARD1 can be computed in $O(nm)$ time, where $n$ is either the number of $S$ vertices or $T$ vertices, depending on the vertex set used. Execution of Algorithm MAX-CARD1 based on a single-source single-path approach is illustrated in Figure 13, and that for a multiple-source multiple-path is illustrated in Figure 14.

**Algorithm 3**

**Input:** A bipartite graph $G$. **Output:** a matching $M$. **Effect:** computes a maximum cardinality matching using a single-source single-path approach.

1: procedure MAX-CARD1($G = (S, T, E)$, $M$)
2: $M \leftarrow \emptyset$;
3: for all $s \in S$ do $\triangleright$ Can also loop over $T$ vertices
4: Find an augmenting path $P$ starting at $s$;
5: if $P$ found then
6: $M \leftarrow M \oplus P$;
7: end if
8: end for
9: end procedure
In the previous section we briefly mentioned about the multiple-source multiple-path approach for finding augmenting paths in a graph and illustrated it in Figure 15. Hopcroft and Karp [37] use a similar technique and show that the worst-case bounds for such an approach in bipartite graphs is $O(\sqrt{nm})$, where $n$ is the number of vertices and $m$ the number of edges. From a simple observation of Figure 15, possibly many vertex-disjoint augmenting paths can be found with each pass, and therefore, drastically reduces the total number of steps that need to be performed. In fact, the number of steps is bounded by $O(\sqrt{n})$. We refer the reader to [37] for a proof.

A multiple-source multiple-path search approach works by finding a set of vertex-disjoint $M$-augmenting paths per iteration; specifically, a maximal set of shortest length vertex-disjoint $M$-augmenting paths. A breadth-first search is first performed to compute the length of the shortest augmenting path. Then, depth-first searches are done simultaneously from each unmatched vertex to find a maximal set of vertex-disjoint paths. Thus, the cardinality of a matching advances by $|M'| = |M| + d$, where $d$ is the number of vertex-disjoint augmenting paths, instead of $|M'| = |M| + 1$ for single-path approach. Algorithm 4 sketches a multiple-path technique for computing a maximum cardinality matching in a bipartite graph.


```
1: procedure MAX-CARD2(G = (S, T, E), M)
2:     M ← φ;
3:     repeat
4:         $\mathcal{P} ← \{P_1, P_2, \ldots, P_k\}$; \triangleright a maximal set of vertex-disjoint paths of shortest length
5:         M ← M ⊕ $\mathcal{P}$
6:     until $\mathcal{P} = φ$;
7: end procedure
```

We conclude our discussion on the maximum cardinality matching algorithms with Table 1 that summarizes the development of MCM algorithms in bipartite and nonbipartite graphs.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Graph Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1931</td>
<td>Konig</td>
<td>B</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>1955</td>
<td>Kuhn</td>
<td>B</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>1965</td>
<td>Edmonds</td>
<td>G</td>
<td>$O(n^2m)$</td>
</tr>
<tr>
<td>1972</td>
<td>Gabow</td>
<td>G</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>1973</td>
<td>Hopcroft and Karp</td>
<td>B</td>
<td>$O(\sqrt{nm})$</td>
</tr>
<tr>
<td>1974</td>
<td>Kameda and Munro</td>
<td>G</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>1974</td>
<td>Even and Kariv</td>
<td>G</td>
<td>$O(n^{2.5})$</td>
</tr>
<tr>
<td>1976</td>
<td>Kariv</td>
<td>G</td>
<td>$O(\sqrt{nm \log \log n})$</td>
</tr>
<tr>
<td>1980</td>
<td>Micali and Vazirani</td>
<td>G</td>
<td>$O(\sqrt{nm})$</td>
</tr>
<tr>
<td>1991</td>
<td>Alt, Blum, Melhorn and Paul</td>
<td>B</td>
<td>$O(n^{1.5} \sqrt{\frac{m}{\log n}})$</td>
</tr>
<tr>
<td>1991</td>
<td>Feder and Motwani</td>
<td>B</td>
<td>$O(\sqrt{nm \log n (\frac{n^2}{m})})$</td>
</tr>
<tr>
<td>1995</td>
<td>Goldberg and Karzanov</td>
<td>G</td>
<td>$O(\sqrt{nm \log n (\frac{n^2}{m})})$</td>
</tr>
</tbody>
</table>

TABLE 1: Algorithms for maximum cardinality matching [66]. For a graph $G = (V, E)$, $n = |V|$ represents the number of vertices, and $m = |E|$ the number of edges. For graph types, B denotes bipartite graphs, and G denotes nonbipartite graphs.
II.4 MAXIMUM EDGE-WEIGHT MATCHING

Given a graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}^+$, and a matching $M$, the weight of a matching is the sum of weight of matched edges $\sum_{e \in M} w(e)$. A matching $M$ in $G$ is a maximum edge-weight matching (MEM) if it has the largest weight of all matchings in the graph. Conceptually, an algorithm for computing a MEM is similar to an algorithm to compute a maximum cardinality matching (MCM). In both the cases, the general technique is to find augmenting paths and perform symmetric differences to increase the current size of the matching. However, for a MEM one also has to consider the weights associated with the edges. This will add complexity to the MEM algorithms. Traditionally, the MEM problem has been formulated as a linear programming problem, and is an example of the theory of duality. The intuition for such a formulation is given by Theorem II.4.1. The theorem highlights relationships between maximization and minimization, and between the weights on the edges and the weights on the vertices. We refer the reader to [66] for a proof of the theorem.

**Theorem II.4.1** (Egerváry [1931]). Consider a bipartite graph $G = (S, T, E)$ with weight function $w : E \to \mathbb{R}^+$. Let $V = \{S \cup T\}$ represent the set of vertices. The maximum weight of a matching $M$ in $G$ is equal to the minimum weight of $y(V)$, where $y : V \to \mathbb{R}^+$ is a set of dual weights on $V$ such that, for each edge $e_{st} \in E$,

$$y_s + y_t \geq w(e_{st}).$$

Linear programming (LP) problems are optimization (minimization or maximization) problems with linear objective function subject to linear inequality constraints. Linear programming problems are usually formulated as primal problems. Every primal formulation can also be recast as a dual LP problem (this primal-dual formulation for the MEM problem will be described shortly). The dual of a dual is the primal problem. The dual of a primal problem can be obtained by changing the objective function and the constraints. If one is a maximization problem, then other is a minimization problem. A solution to the objective function that satisfies all the constraints is known as a feasible solution. By design, every feasible solution to the dual program gives an upper bound on the optimal value of the primal feasible
solution, and vice versa. The solution is optimal when the primal and dual solutions are equal.

The primal-dual solution for the MEM problem in bipartite graphs is known as the Hungarian method for the assignment problem as proposed by Harold W. Kuhn [43]. Consider a bipartite graph $G = (S, T, E)$ with weight function $w : E \to \mathbb{R}^+$. Let $n_S = |S|$ and $n_T = |T|$ represent the number of $S$ and $T$ vertices respectively, and $m = |E|$ represents the number of edges. Let $n$ denote the total number of $S$ and $T$ vertices, $n = n_S + n_T$. If a vertex pair $(s_i, t_i)$ does not exist in the edge set $E$, then the weight $w_{st}$ is set to zero. The primal-dual formulation for the MEM problem is given by:

**Primal problem:**

$$z = \text{maximize} \sum_{s=1}^{n_S} \sum_{t=1}^{n_T} w_{st} x_{st},$$

subject to constraints:

$$\sum_{s=1}^{n_S} x_{st} = 1 \quad \text{for } t = 1, \ldots, n_T,$$

$$\sum_{t=1}^{n_T} x_{st} = 1 \quad \text{for } s = 1, \ldots, n_S,$$

$$x_{st} \in \{0, 1\} \quad \text{for } s = 1, \ldots, n_S; \quad t = 1, \ldots, n_T.$$

**Dual problem:**

$$w = \text{minimize} \sum_{s=1}^{n_S} u_s + \sum_{t=1}^{n_T} v_t,$$

subject to constraints:

$$u_s + v_t \geq w_{st} \quad \text{for } s = 1, \ldots, n_S; \quad t = 1, \ldots, n_T,$$

$$u_s, v_t \geq 0.$$

The primal variable $x_{st}$ is assigned to the edges, and can take a value of 1 if matched, and 0 if not. The dual variables $u_s$ and $v_t$ are assigned to the vertices, and help guide the graph search procedures. The optimality of the primal-dual solution is given by Lemma II.4.1. We refer the reader to [76] for a proof.

**Lemma II.4.1 (Complementary slackness condition).** If there exist vectors $u, v \in \mathbb{R}^n$ and a matching $X \in \{0, 1\}^m$ with the following properties:

1. $w_{st} = (w(e_{st}) - u_s - v_t) \leq 0$ for all $s, t$, and
2. \( X_{st} = 1 \) only when \( \overline{w}_{st} = 0 \),

then the matching \( X \) is optimal and has a value \( (\sum_{s=1}^{n_S} u_s + \sum_{t=1}^{n_T} v_t) \).

Based on the complimentary slackness condition, the key idea for the primal-dual algorithm is to maintain dual feasibility at all times (Condition 1 from Lemma II.4.1), and form a subgraph of these edges, known as the tight edges, for which \( \overline{w}_{st} = 0 \). From a vertex, a search for an augmenting path is made in this subgraph. If an augmenting path exists, then the current matching is augmented with this path and proceed to the next vertex. If no such path can be found in the tight subgraph, the duals are adjusted such that an augmenting path might become possible. The process repeats until the current vertex is matched. The process of updating the duals is nontrivial and assumes the presence of a perfect matching in the graph. Note that the required number of edges with zero weights can be trivially added to the initial bipartite graph in order to facilitate a perfect matching. When the number of \( S \) and \( T \) vertices differ \( (n_S \neq n_T) \), a perfect matching is either an \( S \)-perfect or a \( T \)-perfect matching based on the cardinalities. A skeleton for computing an \( S \)-perfect matching is described in Algorithm 5.

The search strategy in Algorithm MAX-WT is based on the single-source single-path approach, and iterations are made through the \( S \) vertices. The complexity of the graph search procedure is bounded by \( O(m) \), where \( m = |E| \) denotes the number of edges in \( G \). However, there is an additional task of updating the dual variables when a search for an augmenting path fails. From a given source, shortest augmenting paths to all possible unmatched vertices are built. The typical approach at this step is to use a Dijkstra-like search [19] to compute the smallest change in dual variables that is required to create a new augmenting path. This step is critical in determining the overall complexity of the algorithm. Updating the dual variables requires manipulation of priority queues, and therefore, the complexity of the algorithm is influenced by the choice of the priority queue implementation. The complexities as determined by some of the common data structures is summarized in Table 2.

We will conclude the discussion on MEM algorithms with a summary of historical development of MEM algorithms for bipartite and nonbipartite graphs as listed in Table 3.

1: procedure MAX-WT($G = (S,T,E), w : E \rightarrow \mathbb{R}^+$, $M$)
2: \hspace{1em} $M \leftarrow \emptyset$; \hspace{1em} \triangleright Initialization
3: \hspace{1em} $\forall s \in S$, $dual[s] = \max(w(e_{st}))$, for $t \in \text{adj}(s)$;
4: \hspace{1em} $\forall t \in T$, $dual[t] = \max((w(e_{st}) - dual[s]))$, for $s \in \text{adj}(t)$;
5: \hspace{1em} for all $s \in S$ do \hspace{1em} \triangleright Compute matching
6: \hspace{2em} while (true) do \hspace{1em} \triangleright Repeat until $s$ gets matched.
7: \hspace{3em} $w(e_{st}) = (w(e_{st}) - dual[s] - dual[t])$
8: \hspace{3em} $G = (S,T,E)$, where $E \subset E$ such that $\forall e_{st} \in E$, $w(e_{st}) = 0$;
9: \hspace{3em} Find an augmenting path $P_{s \rightarrow t}$ in $G$ with respect to $M$;
10: \hspace{3em} if $P$ found then
11: \hspace{4em} $M \leftarrow M \oplus P$;
12: \hspace{4em} break;
13: \hspace{3em} else
14: \hspace{4em} $\delta \leftarrow \text{minimum change required to update duals}$; \hspace{1em} \triangleright Dijkstra-like search
15: \hspace{4em} $dual[s] \leftarrow dual[s] - \delta$;
16: \hspace{4em} $dual[t] \leftarrow dual[t] + \delta$;
17: \hspace{3em} end if
18: \hspace{3em} end while
19: \hspace{1em} end for
20: end procedure

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Time to update duals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple vectors</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Binary heaps</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Fibonacci heaps</td>
<td>$O(m + n \log n)$</td>
</tr>
</tbody>
</table>

TABLE 2: Power of data structures. For a graph $G = (V,E)$, $n = |V|$ represents the number of vertices, and $m = |E|$ the number of edges.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Graph Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1957</td>
<td>Berge (theoretical)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1955</td>
<td>Kuhn, Munkres</td>
<td>B</td>
<td>$O(n^4)$</td>
</tr>
<tr>
<td>1960</td>
<td>Iri</td>
<td>B</td>
<td>$O(n^2m)$</td>
</tr>
<tr>
<td>1965</td>
<td>Edmonds</td>
<td>G</td>
<td>$O(n^4)$</td>
</tr>
<tr>
<td>1969</td>
<td>Dinitis and Kronrod</td>
<td>B</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>1973</td>
<td>Gabow</td>
<td>G</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>1976</td>
<td>Lawler</td>
<td>G</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>1982</td>
<td>Galil, Micali and Gabow</td>
<td>G</td>
<td>$O(nm \log n)$</td>
</tr>
<tr>
<td>1983</td>
<td>Ball and Derigs</td>
<td>G</td>
<td>$O(nm \log n)$</td>
</tr>
<tr>
<td>1988</td>
<td>Gabow and Tarjan</td>
<td>B</td>
<td>$O(\sqrt{nm \log(nW)})$</td>
</tr>
<tr>
<td>1989</td>
<td>Gabow, Galil, and Spencer</td>
<td>G</td>
<td>$O(n(m \log \log \max(\frac{m}{n}, 2) n + n \log n))$</td>
</tr>
<tr>
<td>1990</td>
<td>Gabow</td>
<td>G</td>
<td>$O(n(m + n \log n))$</td>
</tr>
<tr>
<td>1991</td>
<td>Gabow and Tarjan</td>
<td>B</td>
<td>$O(m \log(nW) \sqrt{n \alpha(n, m) \log n})$</td>
</tr>
<tr>
<td>1992</td>
<td>Orlin and Ahuja</td>
<td>B</td>
<td>$O(\sqrt{nm \log(nW)})$</td>
</tr>
<tr>
<td>2001</td>
<td>Kao, Lam, Sung, and Ting</td>
<td>B</td>
<td>$O(\sqrt{nmW \log_m(n^2/m)})$</td>
</tr>
</tbody>
</table>

TABLE 3: Algorithms for maximum edge-weight matching [66]. For a graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}^+$, $n = |V|$ represents the number of vertices, $m = |E|$ the number of edges, and $W$ is the largest absolute value of an integer weight. For graph types, B represents bipartite, and G the nonbipartite graphs.
II.5 APPROXIMATION ALGORITHMS

Approximation algorithms are generally developed for intractable problems [35]. Given that the matching algorithms are polynomial, approximation techniques for matchings were initially developed for greedy initialization in exact algorithms [25]. However, recent developments in approximation algorithms for matching have been motivated by scientific computing applications [24, 64]. For some applications matchings need to be computed on very large graphs, while for other applications, matchings need be computed a large number of times, although for small or medium sized graphs. The optimality of the matching is not critical for many of these applications, and therefore, motivate the development of fast approximation algorithms. Yet another motivation for the development of approximation algorithms for matchings is the simplicity in parallel implementations. In this section we will discuss some of the recent developments in approximation theory for matching algorithms as summarized in Table 4.

<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Strategy</th>
<th>Approx</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td>Avis</td>
<td>Global maximum</td>
<td>(\frac{1}{2})</td>
<td>(O(m \log n))</td>
</tr>
<tr>
<td>1999</td>
<td>Preis</td>
<td>Local maximum</td>
<td>(\frac{1}{2})</td>
<td>(O(m))</td>
</tr>
<tr>
<td>2003</td>
<td>Drake and Hougardy</td>
<td>Path-growing (PG)</td>
<td>(\frac{1}{2})</td>
<td>(O(m))</td>
</tr>
<tr>
<td>2003</td>
<td>Drake and Hougardy</td>
<td>PG with short augmentations</td>
<td>(\frac{1}{2} - \epsilon)</td>
<td>(O(m^{1/4}))</td>
</tr>
<tr>
<td>2004</td>
<td>Pettie and Sanders</td>
<td>Randomized, Deterministic</td>
<td>(\frac{1}{3} - \epsilon)</td>
<td>(O(m \log \frac{1}{\epsilon}))</td>
</tr>
</tbody>
</table>

TABLE 4: Algorithms for approximate weighted matching. For a graph \(G = (V, E)\), \(n = |V|\) represents the number of vertices, \(m = |E|\) the number of edges in \(G\), and \(\epsilon \rightarrow \mathbb{R}^+\) is a positive real number.

Avis proposed a simple heuristic algorithm for computing approximate matching [4]. The algorithm is as follows. Given a graph \(G = (V, E)\) with weight function \(w : E \rightarrow \mathbb{R}^+\), consider the edges in decreasing order of weights. Pick a heaviest unmatched edge and add it to the matching \(M\) (initially empty). From \(G\), remove all the edges that are incident on the endpoints of the current matched edge. Repeat the process until all the edges have been processed. This is illustrated in Algorithm 6.

It is easy to see that Algorithm GLOBALHEAVY computes a maximal matching in \(G\). Given the fact the cardinality of a maximal matching is at least half of a maximum cardinality, the weight of the matching computed by GLOBALHEAVY guarantees a \(\frac{1}{2}\)-approx to a maximum edge-weight matching in \(G\). Since the edges need to be
considered in sorted order, the time complexity for Algorithm GlobalHeavy is $O(m \log m + m)$, where $m = |E|$ is the number of edges in $G$. Execution of Algorithm GlobalHeavy on a simple graph is illustrated in Figure 16.

**Algorithm 6**


1: procedure GlobalHeavy($G = (V, E), w : E \rightarrow \mathbb{R}^+, M$)
2: $M \leftarrow \emptyset$;
3: repeat
4: Pick a globally heaviest edge $e_{uv} \in E$;
5: $M \leftarrow M \cup e_{uv}$;
6: Delete all edges incident on $u$ and $v$ from $E$;
7: until $E = \emptyset$;
8: end procedure

**FIG. 16:** Execution of Algorithm GlobalHeavy. The weights are associated with the edges. Bold lines represent matched edges, and matched vertices are colored black. Vertices processed at a given step are colored purple. Dashed lines represent the edges that are removed from the graph. (a) The input graph before execution, (b)-(c) the intermediate states of execution, and (d) the final state.

The locally-heaviest approximation algorithm (LAM) proposed by Robert Preis guarantees a $\frac{1}{2}$-approx for both cardinality and weight, and runs in linear time [54, 64]. The basic strategy for LAM is conceptually similar to a Tabu Search [31], in that local decisions made greedily will result in global optimization. The general structure of the algorithm is as follows. Given a graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}^+$, arbitrarily pick an unmatched edge $e_{uv} \in E$. Scan the edges that are incident on the vertices $u$ and $v$. If an edge $e_{ux}$ (or $e_{vy}$) is found such that $w(e_{ux}) > w(e_{uv})$, 
then proceed to the edge $e_{ux}$. Repeat this process recursively. An edge $e_{xy}$ is said to be a *locally-heaviest* or locally-dominating if it is heavier than all the edges incident on the vertices $x$ and $y$. Stop the recursive search when a locally-heaviest edge is found, and add it to the matching set. Remove all the edges that are incident on the matched edge, and repeat the process until all the edges have been processed.

A simple overview of the process is given in Algorithm 7. It is involved to show that the algorithm runs in linear time $O(m)$. We refer the readers to [64] for details.

Execution of LAM on a simple graph is shown in Figure 17.

**Algorithm 7**

**Input:** A graph $G$.  **Output:** a matching $M$.  **Effect:** computes a $\frac{1}{2}$-approx matching $M$ in $G$.

```
1: procedure LAM($G = (V,E), w : E \rightarrow \mathbb{R}^+$, $M$)
2:    $M \leftarrow \emptyset$;
3:    repeat
4:      Pick a locally-heaviest edge $e_{uv} \in E$;
5:      $M \leftarrow M \cup e_{uv}$;
6:      Delete all edges incident on $u$ and $v$ from $E$;
7:    until $E = \emptyset$;
8: end procedure
```

While LAM is conceptually simple, its implementation is nontrivial. Drake and Hougardy propose a simpler algorithm [24] based on the concept of growing a path in a given graph. The algorithm is sketched in Algorithm 8. The path-growing algorithm guarantees a $\frac{1}{2}$-approx for both cardinality and weight. The path-growing algorithm works as follows. Given a graph $G = (V,E)$ with weight function $w : E \rightarrow \mathbb{R}^+$, two empty matching sets $M_1$ and $M_2$, start with an arbitrary unmatched vertex $u$. Search for the heaviest edge $e_{uw} \in E$ incident on $u$, and add it to the matching set $M_1$. Remove $u$ and all the edges incident on $u$ from $G$. Now proceed to $v$ and perform the same steps. This time add the heaviest edge $e_{vw} \in E$ incident on $v$ to the matching set $M_2$. Repeat the process adding new edges alternatively to sets $M_1$ and $M_2$.

There are many schemes to select the final matching from path-growing approach. One can maintain the temporary matchings $M_1$ and $M_2$ locally or globally. In the global approach, as illustrated in Algorithm 8, the two sets $M_1$ and $M_2$ are compared only at the end of the execution. The final matching is the heavier of $M_1$ and $M_2$. For a local approach, $M_1$ and $M_2$ can be compared at the beginning of each new path during the execution, and the heavier of $M_1$ and $M_2$ is added to the final matching.
FIG. 17: Execution of Algorithm LAM. The weights are associated with the edges. Bold lines represent matched edges. Matched vertices are colored black, and the vertices being processed at a given step are colored purple. The shaded edges represent dominating edges at a current step, and dashed lines represent the edges that are removed from the graph. (a) The input graph before execution, (b)-(e) the intermediate states of execution, and (f) the final state.

at the end of each step. Alternatively, dynamic programming can also be used to compute the final matching. Dynamic programming will yield the best matching, and local selection will yield better results than global selection. For a given graph, an edge will be processed only once by Algorithm PathGrow, thus resulting in a linear time algorithm. We refer the reader to [24] for details.

In more recent work [74, 59], advances have been made to improve the approximation ratio from half to $(\frac{2}{3} - \epsilon)$. The basic technique is to iteratively improve the weight and the cardinality by performing short-augmentations that meet a certain threshold for improvement. An augmenting path of certain length, usually of length three or five edges, is called a short-augmenting path. One such simple scheme that looks for augmenting paths of length three in a graph with an initial maximal matching $M$ is shown in Algorithm 9. Augmenting with short paths will not always increase the weight of the final matching. Therefore, a greedy decision is made based on a threshold $\beta$ that represents the ratio of weight of the existing matching, and the weight of the matching after augmentation. For example, if the value of $\beta$ is one, then augmentation will be performed only if the weight of the final matching
Algorithm 8 Input: A graph $G$. Output: a matching $M$. Effect: computes a $\frac{1}{2}$-approx matching $M$ in $G$.

1: procedure PathGrow($G = (V, E), w : E \rightarrow \mathbb{R}^+, M$)
2: $M \leftarrow \phi; M_1 \leftarrow \phi; M_2 \leftarrow \phi$ \hspace{1em} $\triangleright$ Initialization
3: $i \leftarrow 1$
4: while $E \neq \phi$ do \hspace{1em} $\triangleright$ Compute $M_1$ and $M_2$
5: $M_1 \leftarrow \phi; M_2 \leftarrow \phi$
6: $i \leftarrow 1$
7: Arbitrarily pick a vertex $u \in V$ of degree $\geq 1$
8: while deg($v$) $\geq 1$ do $\triangleright$ deg($v$) represents the number of edges incident on a vertex $v$
9: Pick the heaviest edge $e_{uv} \in E$ incident on $u$;
10: $M_i \leftarrow M_i \cup \{e_{uv}\}$
11: $i \leftarrow (3 - i)$ \hspace{1em} $\triangleright$ Alternate between $M_1$ and $M_2$
12: Delete $u$ and all edges incident on $u$ from $G$;
13: $u \leftarrow v$
14: end while
15: end while
16: if $w(M_1) > w(M_2)$ then \hspace{1em} $\triangleright$ Compute $M$
17: $M \leftarrow M_1$
18: else
19: $M \leftarrow M_2$
20: end if
21: end procedure
FIG. 18: *Execution of Algorithm PathGrow*. The weights are associated with the edges. The solid bold-lines represent edges matched in $M_1$, and the dashed bold-lines represent the edges matched in $M_2$. The matched vertices are colored black, and the vertices processed at a given step are colored purple. The shaded edges highlight the edges that are being processed for matching at a given step. (a) The input graph before execution, (b)-(f) the intermediate states of execution.

At least remains the same (while the cardinality will increase). A $\frac{1}{2}$-approx matching computed with one of the algorithms discussed before, for example GLOBALHEAVY, can be used to compute the initial maximal matching $M$. 

1: procedure IMPROVE-MATCHING($G = (V, E), w : E \rightarrow \mathbb{R}^+, M, M'$)
2: $M' \leftarrow M$;
3: repeat $k$ times
4: for all $e \in M'$ do
5:   Find $\beta$-augmenting path $P$ centered at $e$; $\triangleright \beta$ is the threshold value
6:   if $P$ found then
7:     $M' \leftarrow M' \oplus P$;
8:   end if
9: end for
10: until
11: end procedure

II.6 CHAPTER SUMMARY

In this chapter, we gave a brief introduction to matching and discussed exact and approximation algorithms for matching in graphs. The scope of the exact algorithms was restricted to bipartite graphs. Some of the recent developments in approximation techniques for matchings were also discussed. One of the goals for this chapter has been to build the necessary background for presenting our work in the following chapters.
CHAPTER III

EXACT ALGORITHMS

“The complexity of the vertex-weighted matching problem is close to that of the unweighted matching problem.” - Thomas Spencer and Ernst Mayr [69]

The maximum vertex-weight matching (MVM) problem is simple as well as challenging, the complexity lies between that of the unweighted and the edge-weighted versions of the matching problem. Unlike the maximum edge-weight matching, the maximum vertex-weight matching problem has received little attention by researchers. After extensive search, we could locate only a handful of publications dedicated to the vertex-weighted matching problem. In this chapter we will provide an introduction, discuss related work and provide three new algorithms for the exact vertex-weighted matching problem. The approximation algorithms for vertex-weighted matching will be discussed in Chapter IV.

III.1 INTRODUCTION AND RELATED WORK

A maximum vertex-weight matching (MVM) can be defined as:

Definition III.1.1. Given a graph \( G = (V, E) \) with weight function \( w: V \to \mathbb{R}^+ \), a maximum vertex-weight matching \( M \) in \( G \) is a matching that maximizes the sum of weights of the matched vertices, denoted by \( V(M) \):

\[
\text{Maximize } \sum_{v \in V(M)} w(v) \tag{3}
\]

Note that an MVM problem can also be formulated as a maximum edge-weight matching problem by defining the weight of an edge as the sum of the weights of its incident vertices. However, we will show that an MVM is conceptually as well as computationally easier than an MEM problem. We will also show that the MVM problem is conceptually similar to the MCM problem.

The maximum vertex-weight matching problem was studied by Thomas Spencer and Ernst Mayr [69]. A brief mention of maximum vertex-weight matching is also made by Ketan Mulmuley, Umesh Vazirani and Vijay [55]. With specific application in Input Queueing Switches, Tabatabaee, Georgiadis and Tassiulas [71] also propose
an MVM algorithm. In this chapter we will provide relevant concepts from these
two papers and use them in our subsequent work. Detailed descriptions of the new
algorithms and the proof sketch of correctness will also be provided.

Spencer and Mayr show that the MVM problem in a nonbipartite graph can be
reduced to the MVM problem in a bipartite graph. Further, the bipartite MVM
problem itself can be simplified into two subproblems of computing the MVM in
special bipartite graphs called the restricted bipartite graphs. Spencer and Mayr also
show how to transform the MVM problem in a graph with negative weights to the
MVM problem in a graph with positive weights. Thus, computing the MVM in a
restricted bipartite graph will lead to a solution in general graphs. This relationship
is illustrated in Figure 19.

FIG. 19: Decomposition of the maximum vertex-weight matching problem.

Given a bipartite graph \( G = (S,T,E) \) and weight functions \( w_S : S \to \mathbb{R}^+ \) and
\( w_T : T \to \mathbb{R}^+ \), the two restricted bipartite graphs can be defined as: (i) \( G = (S,T,E) \)
and weight function \( w_S : S \to \mathbb{R}^+ \), and (ii) \( G = (S,T,E) \) and weight function
\( w_T : T \to \mathbb{R}^+ \). In the first restricted bipartite graph the weights on \( T \) vertices are set
to zero and in the second the weights on \( S \) vertices are set to zero, while everything
else remains the same. The fact that the matching problem in a bipartite graph can
be simplified into two subproblems of computing matchings in the restricted bipartite
graphs is given by Theorem III.1.1.

**Theorem III.1.1** (Mendelsohn-Dulmage). *Given two matchings \( M_S \) and \( M_T \) in a
bipartite graph \( G = (S,T,E) \), a new matching \( M \subseteq M_S \cup M_T \) can be computed in
linear time such that \( M \) matches all the \( S \) vertices matched by \( M_S \) and all the \( T \)
vertices matched by \( M_T \).*

**Proof.** Compute the symmetric difference \( M_S \oplus M_T \), this will contain a set of cycles
and paths as enumerated in Figure 20. In each case it is possible to pick edges for
such that it covers all the vertices of $S$ matched by $M_S$ and all the $T$ vertices matched by $M_T$. The edges that are matched by both $M_S$ and $M_T$ should also be added to $M$. All the above operations are bounded by $O(|E|)$. All these operations can be summarized as follows:

(a) A cycle: arbitrarily choose $M_S$ or $M_T$ edges,

(b) $M_S$-augmenting path: choose $M_T$ edges,

(c) $M_T$-augmenting path: choose $M_S$ edges,

(d) $M_S$-alternating path: choose $M_S$ edges,

(e) $M_T$-alternating path: choose $M_T$ edges, and

(f) $M_S \cap M_T$: choose $M_S$ or $M_T$ edges.

FIG. 20: The symmetric difference of two matchings $M_S \oplus M_T$. Dashed lines represent edges in $M_S$ and Solid lines represent edges in $M_T$. (a) A cycle; (b)-(e) Augmenting or alternating paths.

An implementation of the Mendelsohn-Dulmage technique is illustrated in Algorithm 10. The algorithm has three stages. In Stage 1, Lines 8-17, we will pick the relevant $M_S$ edges shown as Cases (c) and (d) in Figure 20. These edges can be detected by looking for $S$ vertices that are matched by $M_S$ and unmatched by $M_T$. In Stage 2, Lines 19-29, we pick the relevant $M_T$ edges shown as Cases (b) and (e) in Figure 20). These can be detected by looking for $T$ vertices that are matched by $M_T$ and unmatched by $M_S$. In Stage 3, Lines 30-36, we will pick the edges that will be matched by both $M_S$ and $M_T$, as well as the cycles.
Algorithm 10 Input: A bipartite graph $G$ and matchings $M_S$ and $M_T$. Output: a matching $M$. Effect: using Mendelsohn-Dulmage technique, computes a matching $M$ that matches all the $S$ vertices matched by $M_S$ and all the $T$ vertices matched by $M_T$.

1: procedure MENDELSOHN-DULMAGE($G = (S, T, E), M_S, M_T, M$)
2: for all $s \in S$ do \Comment{Initialize $M$}
3: \hspace{1em} $M[s] \leftarrow \phi$
4: end for
5: for all $t \in T$ do
6: \hspace{1em} $M[t] \leftarrow \phi$
7: end for
8: for all $s \in S$ do \Comment{Pick $M_S$ edges (Cases (c) and (d))}
9: \hspace{1em} if $M_s[s] \neq \phi$ and $M_t[s] = \phi$ then
10: \hspace{2em} $s' \leftarrow s$
11: \hspace{2em} repeat
12: \hspace{3em} $t' \leftarrow M_S[s']$
13: \hspace{3em} $M[s'] \leftarrow t'$
14: \hspace{3em} $M[t'] \leftarrow s'$
15: \hspace{3em} $s' \leftarrow M_T[t']$
16: \hspace{2em} until $s' = \phi$ or $M_S[s'] = \phi$
17: end if
18: end for
19: for all $t \in T$ do \Comment{Pick $M_T$ edges (Cases (b) and (e))}
20: \hspace{1em} if $M_t[t] \neq \phi$ and $M_s[t] = \phi$ then
21: \hspace{2em} $t' \leftarrow t$
22: \hspace{2em} repeat
23: \hspace{3em} $s' \leftarrow M_T[t']$
24: \hspace{3em} $M[s'] \leftarrow t'$
25: \hspace{3em} $M[t'] \leftarrow s'$
26: \hspace{3em} $t' \leftarrow M_S[s']$
27: \hspace{2em} until $t' = \phi$ or $M_T[t'] = \phi$
28: end if
29: end for
30: for all $s \in S$ do \Comment{Pick $M_S$ edges (Cases (a) and (f))}
31: \hspace{1em} if $M_s[s] \neq \phi$ and $M[s] = \phi$ then
32: \hspace{2em} $t \leftarrow M_S[s]$
33: \hspace{2em} $M[s] \leftarrow t$
34: \hspace{2em} $M[t] \leftarrow s$
35: end if
36: end for
37: end procedure
III.2 FOUNDATIONS

We will now discuss two theorems that provide necessary and sufficient conditions to prove the optimality of an MVM. An important observation is the fact that any maximum vertex-weight matching is also a maximum cardinality matching. This provides the necessary condition and is stated by Theorem III.2.1.

**Theorem III.2.1.** Given a graph $G = (V, E)$ and weight function $w : V \to \mathbb{R}^+$, a maximum vertex-weight matching $M$ in $G$ is also a maximum cardinality matching.

**Proof.** Let $M$ be a maximum vertex-weight matching that is not of maximum cardinality. Since $M$ is not of maximum cardinality, there is at least one augmenting path $P$ with respect to $M$. The symmetric difference $M \oplus P$ will increase the cardinality of $M$ by one edge and matches two new vertices while retaining all the vertices that were already matched by $M$. Since positive weights are associated with the vertices, the total weight of $M$ increases when its cardinality is increased. Therefore a maximum vertex-weight matching is also a maximum cardinality matching. \qed

If a graph has a perfect matching, then all the vertices will be matched by any maximum cardinality matching in this graph. Therefore any maximum cardinality matching will also be a maximum vertex-weight matching for this graph. However, when a maximum cardinality matching in a graph is not a perfect matching, computing a maximum vertex-weight matching will be conceptually harder than computing a maximum cardinality matching. Since only a subset of vertices need to be matched, we will have to explicitly consider the weights associated with the vertices. An important concept in vertex-weighted matching is the lexicographical ordering of vertex sets.

We will need the definition of a lexicographical order to differentiate vertices with duplicate weights. For a graph $G = (V, E)$ with weight function $w : V \to \mathbb{R}^+$, let each vertex be assigned a distinct integer label between 1 and $|V|$. A relationship between two vertices, and sets of vertices, can be established by using both the weights and the labels associated with the vertices. A precedence operator $\prec$ can be defined as follows: given two vertices $v_1$ and $v_2$, $v_1 \prec v_2$ if and only if $w(v_1) < w(v_2)$, or $w(v_1) = w(v_2)$ and $l(v_1) < l(v_2)$, where $l(v_1)$ and $l(v_2)$, the labels of vertices $v_1$ and $v_2$ are considered as integers. Conversely, $v_2$ succeeds $v_1$, denoted as $v_2 \succ v_1$.

The precedence relationship can be used to compare two matchings. Given two matchings $M_1$ and $M_2$ in a graph $G = (V, E)$, let $V_1 = V(M_1)$ and $V_2 = V(M_2)$
denote the set of vertices matched by $M_1$ and $M_2$ respectively. Assuming that the cardinality of the two matchings is the same $|V_1| = |V_2|$, we will say that $V_1$ is \textit{lexicographically smaller} than $V_2$, denoted as $V(M_1) < V(M_2)$, if the first difference between the two sets, $v_1 \in V_1$ and $v_2 \in V_2$, is such that $v_1 < v_2$. Conversely, $V_2$ succeeds $V_1$, denoted as $V_2 \succ V_1$. Given a set of maximum cardinality matchings in a graph $\{V_1, V_2, \ldots, V_k\}$, a \textit{lexicographically largest} matching $V_j$ is a matching such that it succeeds all other matchings, $V_j \succ V_i$ for any $i$ in $1 \leq i \leq k$ and $i \neq j$.

We have seen that any MVM is a maximum cardinality matching. The lexicographical order of a vertex set can be used to prove that some maximum cardinality matching is also a maximum vertex-weight matching in a graph and is given by Theorem III.2.2:

\textbf{Theorem III.2.2} (Mulmuley, Vazirani, Vazirani). \textit{Given a graph $G = (V, E)$ and weight function $w : V \rightarrow \mathbb{R}^+$, a lexicographically largest matching of maximum cardinality is also a maximum vertex-weight matching in $G$.}

\textit{Proof.} Let $M_L$ represent a lexicographically largest matching and $M_*$ represent a maximum vertex-weight matching. Also, let $M_L$ and $M_*$ be different, with respect to matched vertices, from each other. From Theorem III.2.1, $M_*$ 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 decreasing order of weights. Let $v_1 \in V$ be the first vertex where the two matched sets differ. The symmetric difference $M_L \oplus M_*$ will result in an alternating path $P$ starting at $v_1$, matched only by $M_L$ and ending with $v_2 \in V$, matched only by $M_*$. Since $v_1$ is the first vertex in the decreasing order that is different, it is larger than $v_2$ ($w(v_1) > w(v_2)$). The matching obtained by the symmetric difference $P \oplus M_*$ will have a weight larger than $M_*$, and therefore, contradicts the assumption that $M_*$ is a maximum vertex-weight matching.

If $w(v_1) = w(v_2)$, then by performing $M_* \leftarrow P \oplus M_*$ we have brought the two matchings $M_L$ and $M_*$ closer to each other. Continue considering the vertices in the decreasing order of weights until they are different. When such a vertex is found, it will contradict the assumption. If no such vertex is found, then both $M_L$ and $M_*$ will have the same weights. Thus, $w(M_L) = w(M_*)$. \hfill $\square$

The lexicographic order of matched vertices is an important observation that
assisted in the design of the first proposed algorithm, which sorts the vertices in decreasing order of their weights and process them in that order. The algorithm proposed by Spencer and Mayr [69] also uses a sorting-based approach to compute an MVM. Their *divide and conquer* strategy is successful because the choice of the heaviest vertices that should be matched can be determined independently from the choice of the lightest vertices that should be matched. Given a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$, there can be at most $O(\log_2 n)$ divisions, where $n$ is the number of vertices. Computing a maximum cardinality matching at each step will dominate the run time. Since any given problem can be reduced to computing an MVM in a bipartite graph, a maximum cardinality can be computed in $O(\sqrt{nm})$ time complexity [37], thus providing an overall time complexity of $O(\sqrt{nm} \log n)$ to compute an MVM in a graph. In their algorithm Tabatabae, Georgiadis and Tassiulas, first compute a maximum cardinality matching and then sort the unmatched vertices in decreasing order of weights. From each unmatched vertex processed in that order, an attempt to increase the weight of the matching is made. A maximum cardinality matching, as well as the subsequent computation can be bounded by $O(nm)$. Related work is summarized in Table 5.

<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>Spencer and Mayr</td>
<td>$O(\sqrt{nm} \log n)$</td>
</tr>
<tr>
<td>1987</td>
<td>Mulmuley, Vazirani and Vazirani</td>
<td>Theoretical</td>
</tr>
<tr>
<td>2001</td>
<td>Tabatabae, Georgiadis and Tassiulas</td>
<td>$O(nm)$</td>
</tr>
</tbody>
</table>

TABLE 5: A survey of algorithms for maximum vertex-weight matching. For a given graph $G = (V, E)$, $n = |V|$ represents the number of vertices, and $m = |E|$ the number of edges.
III.3 NEW ALGORITHMS FOR MAXIMUM VERTEX-WEIGHT MATCHING

In this section we provide three algorithms to compute maximum vertex-weight matchings (MVM). We will build on the work of Spencer and Mayr [69], and Mulmuley, Vazirani and Vazirani [55] for the exact algorithms. We also propose three algorithms for $\frac{1}{2}$-approx matchings and a $\frac{2}{3}$-approx algorithm. The approximation algorithms are discussed Chapter 4. The proposed algorithms are summarized in Table 6.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exact Algorithms</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLOBALOptimal</td>
<td>B</td>
<td>Sort-based</td>
<td>$O(n \log n + nm)$</td>
</tr>
<tr>
<td>LOCALOptimal</td>
<td>B</td>
<td>Search-based</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>HYBRIDOptimal</td>
<td>G</td>
<td>Sort and search-based</td>
<td>$O(n \log n + nm)$</td>
</tr>
<tr>
<td><strong>Approximation Algorithms</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLOBALHalf</td>
<td>B</td>
<td>$\frac{1}{2}$-approx; Sort-based</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>LOCALHalf</td>
<td>B</td>
<td>$\frac{1}{2}$-approx; Search-based</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>HYBRIDHalf</td>
<td>G</td>
<td>$\frac{1}{2}$-approx; Sort and search-based</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>GLOBALTwoThird</td>
<td>B</td>
<td>$\frac{2}{3}$-approx; Sort-based</td>
<td>$O(n \log n + nd_k)$</td>
</tr>
</tbody>
</table>

TABLE 6: A summary of algorithms proposed for vertex weighted matchings. Bipartite and general graphs are represented with B and G respectively. For a bipartite graph $G = (S, T, E)$, $n = (|S| + |T|)$ represents the number of vertices, $m = |E|$ the number the edges, and $d_k$ is a generalization of the vertex degree that denotes the average number of distinct alternating paths of length at most $k$ edges starting at a vertex in $G$.

The fundamental technique to compute an MVM is to find an augmenting path and augment the matching via symmetric difference of the augmenting path and the current matching. The algorithms for MVM are conceptually similar to algorithms for computing a maximum cardinality matching. The proposed algorithms use the single-source single-path approach (discussed in Chapter 2). In a single-source single-path approach, the search for an augmenting path starts from an unmatched vertex, and if found, augmentation can be performed along only one such path. For the proposed algorithms we will not be able to use the multiple-path approach proposed by Hopcroft and Karp [37], as discussed later in this chapter.

For the bipartite graph algorithms, we propose two basic approaches - global and local. The two approaches differ in the way the vertices are selected for processing.
While \texttt{GLOBALOPTIMAL} uses a global-order in selecting the vertices as sources for augmenting paths, \texttt{LOCALOPTIMAL} selects the sources arbitrarily (but, considers all the potential augmenting paths from this source in an order). From the perspective of computational complexity, both the techniques have similar worst-case bounds. However, there can be significant differences in performance. The primary motivation for developing two different approaches is to provide an algorithm for computing maximum vertex-weight matchings in nonbipartite graphs. This is achieved in the hybrid approach, \texttt{HYBRIDOPTIMAL}, where the source-vertices are processed in a global-order, as well as, ordering all the potential augmenting paths like the local approach. We will now discuss the three proposed algorithms in detail.

\section*{III.3.1 Algorithm \texttt{GlobalOptimal}}

The first proposed algorithm, shown in Algorithm 11, is based on processing the vertices according to a global order. We first decompose the given bipartite graph $G = (S,T,E)$, with weights associated with both $S$ and $T$ vertices, into two subgraphs, the restricted bipartite graphs, by ignoring the weights on the $T$ vertices and then on the $S$ vertices. Construction of the restricted bipartite graphs is represented in Algorithm 11 by Lines 5 and 6 for $S$ vertices, and Lines 15 and 16 for $T$ vertices.

For the first matching subproblem, we will compute the matching $M_S$ by finding shortest augmenting paths starting from unmatched $S$ vertices, considered in decreasing order of weights. Lines 7–14 represent the computation of $M_S$. A similar approach is used to compute the matching $M_T$ where weights are associated only with the $T$ vertices is represented by Lines 17–24 in \texttt{GLOBALOPTIMAL}. The final matching will be obtained by merging the two matchings $M_S$ and $M_T$ using Mendelsohn-Dulmage technique. Execution of \texttt{GLOBALOPTIMAL} on a simple bipartite graph with weights associated with $S$ vertices is shown in Figure 21. For this discussion, we will only consider positive weights. We will later show how to compute an MVM in bipartite graphs with negative weights.
Algorithm 11  \textbf{Input:} a bipartite graph $G$. \textbf{Output:} a matching $M$. \textbf{Associated Data Structures:} sets $\tilde{S}$ and $\tilde{T}$ are stored as \emph{stack} data structures. The elements in the stack follow a precedence order $\prec$, with the top of the stack being the heaviest element at any given time. \textbf{Effect:} computes a maximum vertex-weight matching $M$ in $G$

1: \textbf{procedure} \textsc{GlobalOptimal}(\textit{G} = (S, T, E), \textit{w}_S : S \rightarrow \mathbb{R}^+, \textit{w}_T : T \rightarrow \mathbb{R}^+, \textit{M})
2: \hspace{2em} M \leftarrow \phi;
3: \hspace{2em} M_S \leftarrow \phi;
4: \hspace{2em} M_T \leftarrow \phi;
5: \hspace{2em} \tilde{S} \leftarrow S \text{ in increasing order of weights } \textit{w}_S;
6: \hspace{2em} \tilde{T} \leftarrow T \text{ with weights set } \textit{w}_T \text{ to zero};
7: \hspace{2em} \textbf{while } \tilde{S} \neq \phi \textbf{ do} \quad \triangleright \text{ Compute } M_S
8: \hspace{3em} s \leftarrow \text{top of } \tilde{S};
9: \hspace{3em} \tilde{S} \leftarrow \tilde{S} \setminus s;
10: \hspace{3em} \text{Find a shortest augmenting path } P \text{ starting at } s;
11: \hspace{3em} \textbf{if } P \text{ found } \textbf{ then}
12: \hspace{4em} M_S \leftarrow M_S \oplus P;
13: \hspace{3em} \textbf{end if}
14: \hspace{2em} \textbf{end while}
15: \hspace{2em} \tilde{T} \leftarrow T \text{ in increasing order of weights } \textit{w}_T;
16: \hspace{2em} \tilde{S} \leftarrow S \text{ with weights } \textit{w}_S \text{ set to zero ;}
17: \hspace{2em} \textbf{while } \tilde{T} \neq \phi \textbf{ do} \quad \triangleright \text{ Compute } M_T
18: \hspace{3em} t \leftarrow \text{top of } \tilde{T};
19: \hspace{3em} \tilde{T} \leftarrow \tilde{T} \setminus t;
20: \hspace{3em} \text{Find a shortest augmenting path } P \text{ starting at } t;
21: \hspace{3em} \textbf{if } P \text{ found } \textbf{ then}
22: \hspace{4em} M_T \leftarrow M_T \oplus P;
23: \hspace{3em} \textbf{end if}
24: \hspace{2em} \textbf{end while}
25: \hspace{2em} M \leftarrow \textsc{MendelsohnDulmage}(M_S, M_T, M); \quad \triangleright \text{ Compute } M
26: \textbf{end procedure}
FIG. 21: Execution of Algorithm GlobalOptimal. (a) The input graph $G = (S,T,E)$ before execution, weights are associated only with the $S$ vertices. (b)-(e) The intermediate states of execution. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight the shortest augmenting path from a given $S$ vertex. Vertices colored Violet represent the vertex processed at a given step, and the end-point of an augmenting path if one exists. The arrows indicate the $S$ vertex that is being processed at a given step.

III.3.2 Algorithm LocalOptimal

For the second algorithm we adopt a strategy based on search within a restricted neighborhood of the graph, and is shown in Algorithm 12. The vertices are arbitrarily chosen as sources for augmenting paths, but the paths themselves are chosen for augmentation in an order. We again decompose the bipartite graph $G = (S,T,E)$, with weights associated with both $S$ and $T$ vertices, into two restricted bipartite graphs (Lines 5 and 14).

In the first matching subproblem a matching $M_S$ is computed as follows: arbitrarily start from an unmatched $S$ vertex $s_i$ and enumerate all possible augmenting paths $P_i$ with respect to the current matching $M_i$. Then choose the best augmenting path from $s_i$ to augment the current matching. A best augmenting path is a path that maximizes the weight of $M_i \oplus P_i$, in other words the path ending with the heaviest vertex. Repeat the process until all the $S$ vertices have been processed. Lines 6 – 13 represent the computation of $M_S$. A similar procedure can be used to compute the
matching $M_T$ on the second restricted bipartite graph. This is represented by Lines 17 – 24 in LocalOptimal. The final matching will be obtained by merging the two matchings $M_S$ and $M_T$ using the Mendelsohn-Dulmage technique. Execution of LocalOptimal on a simple bipartite graph with weights associated with $S$ vertices is shown in Figure 22.

**Algorithm 12**

**Input:** a bipartite graph $G$. **Output:** a matching $M$. **Effect:** computes a maximum vertex-weight matching $M$ in $G$.

```
1: procedure LocalOptimal($G = (S, T, E), w_S : S \rightarrow \mathbb{R}^+, w_T : T \rightarrow \mathbb{R}^+, M$)
2:     $M \leftarrow \phi$;
3:     $M_S \leftarrow \phi$;
4:     $M_T \leftarrow \phi$;
5:     $\tilde{T} \leftarrow T$ with weights $w_T$ set to zero ;
6:     while $\tilde{T} \neq \phi$ do \Comment{Compute $M_S$}
7:         $t \leftarrow$ any element of $\tilde{T}$;
8:         $\tilde{T} \leftarrow \tilde{T} \setminus t$;
9:         Find all augmenting paths $P_{t \rightarrow s} = \{P_1, P_2, ..\}$ starting at $t$;
10:        if $P_{t \rightarrow s} \neq \phi$ then
11:            $M_S \leftarrow M_S \oplus P_{\text{best}}$; \Comment{$P_{\text{best}}$ is the path with largest $s$ that will be matched}
12:        end if
13:     end while
14:     $\tilde{S} \leftarrow S$ with weights $w_S$ set to zero ;
15:     while $\tilde{S} \neq \phi$ do \Comment{Compute $M_T$}
16:         $s \leftarrow$ any element of $\tilde{S}$;
17:         $\tilde{S} \leftarrow \tilde{S} \setminus s$;
18:         Find all augmenting paths $P_{s \rightarrow t} = \{P_1, P_2, ..\}$ starting at $s$;
19:        if $P_{s \rightarrow t} \neq \phi$ then
20:            $M_T \leftarrow M_T \oplus P_{\text{best}}$; \Comment{$P_{\text{best}}$ is the path with largest $t$ that will be matched}
21:        end if
22:     end while
23:     $M \leftarrow$MendelsohnDulmage($M_S, M_T, M$); \Comment{Compute $M$}
24: end procedure
```

**III.3.3 Algorithm HybridOptimal**

While GlobalOptimal and LocalOptimal computed matchings in bipartite graphs, Algorithm 13 combines the two strategies to compute maximum vertex-weight matchings in general graphs. The given set of vertices are sorted in an increasing order of their weights and stored in a stack data structure, such that the
FIG. 22: *Execution of Algorithm LOCALOPTIMAL.* (a) The input graph $G = (S, T, E)$ before execution, weights are associated only with the $S$ vertices. (b)-(d) The intermediate states of execution, (e) the final state. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight all the augmenting paths that exist from a given $T$ vertex. The arrows indicate the $T$ vertex that is being processed at a given step.
top element is the current heaviest vertex. The vertices are then retrieved from the
stack one at a time. All possible augmenting paths starting from this vertex are
discovered and ordered based on the weight of the last vertex, which is also the only
unmatched vertex in the path. The current matching is augmented with the path
with the heaviest weight of the last vertex. The algorithm processes each vertex
only once and terminates when it processes every vertex in the graph. Note that
the implementation of this algorithm should be capable of processing cycles of odd
length (Blossoms).

Algorithm 13 Input: a graph \( G \). Output: a matching \( M \). Associated Data
Structures: set \( \tilde{V} \) is a stack data structure. The elements in the stack follow a
precedence order \( \prec \), with the top of the stack being the heaviest element at any
given time. Effect: computes a maximum vertex-weight matching \( M \) in \( G \).

1: procedure HYBRIDOPTIMAL\((G = (V, E), w : V \rightarrow \mathbb{R}^+)\)
2: \( M \leftarrow \phi \);
3: \( \tilde{V} \leftarrow V \) in increasing order of weights;
4: while \( \tilde{V} \neq \phi \) do \( \triangleright \) Compute \( M \)
5: \( v \leftarrow \text{top of } \tilde{V} \);
6: \( \tilde{V} \leftarrow \tilde{V} \setminus v \);
7: Find all augmenting paths \( P_{v \leadsto w} = \{P_1, P_2, \ldots\} \) starting at \( v \);
8: if \( P_{v \leadsto w} \neq \phi \) then
9: \( M \leftarrow M \oplus P_{\text{best}}; \quad \triangleright P_{\text{best}} \) is the path with largest \( w \) that will be
matched
10: \( \tilde{V} \leftarrow \tilde{V} \setminus w \);
11: end if
12: end while
13: end procedure

III.3.4 Negative Weights

Spencer and Mayr provide a method to handle negative weights. Given a graph
\( G = (V, E) \) and weight function \( w : V \rightarrow \mathbb{R} \), for each vertex \( v_i \in V \) that has a
negative weight, add a new vertex \( v'_i \) and an edge \( e(v_i, v'_i) \). Also set \( w(v_i) = 0 \) and
\( w(v'_i) = \text{abs}(w(v_i)) \), the absolute value of the original weight. This will result in a
new graph \( G' = (V', E') \) and weight function \( w : \rightarrow (\mathbb{R}^+ \cup \{0\}) \). An MVM \( M \) in \( G' \) will
also be an MVM in \( G \). While \( M \) will also be a maximum cardinality matching in \( G' \),
the same is not necessarily true in \( G \).

For the proposed algorithms GLOBALOPTIMAL and LOCALOPTIMAL, we can
adopt a similar technique. Given a bipartite graph $G = (S, T, E)$ and weight functions $w_S : S \rightarrow \mathbb{R}$, $w_T : T \rightarrow \mathbb{R}$, for each $s_i \in S$ that has a negative weight, add a new $T$ vertex $t'_i$ and an edge $e(s_i, t'_i)$. Also set $w_S(s_i) = 0$ and $w_T(t'_i) = \text{abs}(w_S(s_i))$. Perform similar transformations for all the $T$ vertices with negative weights. This will result in a new graph $G'(S', T', E')$ with weight functions $w_S : S \rightarrow (\mathbb{R}^+ \cup \{0\})$, $w_T : T \rightarrow (\mathbb{R}^+ \cup \{0\})$. The transformation is illustrated in Figure 23. Both the algorithm $\text{GLOBALOPTIMAL}$ and $\text{LOCALOPTIMAL}$ will compute an MVM in the new graph $G'$.

FIG. 23: Transformation of graphs with negative weights. (a) The input graph $G = (S, T, E)$ with some negative weights associated with the vertices, (b) the new graph $G'(S', T', E')$ with zero or positive weights. The new vertices are filled with Black color.
III.4 PROOF OF CORRECTNESS

In this section we provide the proof of correctness for the proposed algorithms. We will first discuss the proof for the two bipartite graph algorithms and then extend it to the algorithm for the general graph. In Section III.2 we provided the necessary and sufficient condition to prove the optimality of an MVM in a graph. In this section, we will provide an alternative method to prove the correctness of the proposed algorithms. The bipartite algorithms decompose the given problem into two matching problems on the restricted bipartite graphs. We will prove the correctness for an MVM computed in the first restricted bipartite graph, which can then be trivially extended to the second subgraph. The correctness of an MVM in the original graph can be proved subsequently using the Mendelsohn-Dulmage technique as stated in Theorem III.1.1. However, there is no such decomposition in the case of general graphs.

We will adapt the definitions of the lexicographical sets for the restricted bipartite cases. We will generally consider the first restricted bipartite case: a bipartite graph $G = (S, T, E)$ and weight function $w : S \rightarrow \mathbb{R}^+$ (we would have specifically set the weights on the $T$ vertices to zero). In all lexicographic comparisons, we will consider only the $S$ vertices. Recall that $M_S$ is a matching in this restricted bipartite graph that has weights only on the $S$ vertices. The $S$-vertex set of $M_S$, the $S$-vertex set of $M_S$, will be represented as $S(M_S)$.

The proof for the correctness for GlobalOptimal is straight-forward, however, the proof for the correctness of Algorithm LocalOptimal is nontrivial. In order to provide a uniform method of proof for both these algorithms, we introduce the concept of reachability property, which can be defined as:

**Definition III.4.1** (Reachability Property). Consider a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$, and any matching $M$ in $G$. The matching $M$ satisfies the reachability property if for any $M$-unmatched vertex $v$, and any $M$-matched vertex $v'$ reachable by an $M$-alternating path from $v$, the condition that $v \prec v'$ holds.

As illustrated in Figure 25, the alternating path for a reachability test starts with an unmatched vertex and ends with a matched vertex. This path is always of even length with an equal number of matched and unmatched edges, and has only one unmatched vertex. We use the concept of reachability to prove of correctness of all the proposed algorithms. Existence of the reachability property is a sufficient
condition for optimality, this is stated in Theorem III.4.1.

**Theorem III.4.1.** Consider a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$, and a maximum cardinality matching $M$ in $G$. If $M$ satisfies the reachability property, then it is also a maximum vertex-weight matching in $G$.

*Proof.* Let $M_L$ represent a lexicographically largest matching of maximum cardinality in $G$, and therefore, a maximum vertex-weight matching (MVM) as follows from Theorem III.2.2. In order to prove that $M$ is an MVM in $G$, we only need to prove that $w(V(M)) = w(V(M_L))$. Assume, by contradiction, that $w(V(M)) \leq w(V(M_L))$.

We will make an argument similar to the one provided in the proof of Theorem III.2.2. Consider the matched vertices in $M_L$ and $M$ in decreasing order of weights. Let $v_i \in V$ be the first vertex where the two matched sets differ. The symmetric difference $M_L \oplus M$ will result in an alternating path $P$ starting at $v_i$, matched only by $M_L$. The alternating path $P$ must contain the same number of edges from $(M_L \setminus M)$ and $(M \setminus M_L)$, if not, we would have an augmenting path for one of the matchings (which, we know is not true). Hence the path $P$ ends with some vertex $v_j$, matched only by $M$. Note that the vertex $v_j$ is matched by $M$, but not by $M_L$, due to it being the last vertex on the alternating path $P$. Since $v_i$ is the first vertex in the decreasing order that is different, its weight is larger than the weight of $v_j$, $w(v_i) > w(v_j)$. However, from the reachability property for $M$, the weight $v_j$ cannot be smaller than the weight of $v_i$ and this contradicts the assumption that $w(V(M)) \leq w(V(M_L))$.

**FIG. 24:** *Illustration of the reachability property.* Bold lines represent the matched edges and matched vertices are colored black.
If \( w(v_i) = w(v_j) \), then replace \( M \) by \( M \oplus P \). This will not affect the weight of matching \( M \). Continue considering the vertices in the decreasing order of weights until the next differing vertex is found. We can repeat the above argument for such a vertex. When there are no more vertices to be considered, then both \( M_L \) and \( M \) have the same weights. Thus \( w(V(M)) = w(V(M_L)) \).

The reachability property provides a sufficient condition to prove the optimality of a maximum vertex-weight matching in a graph. We will now prove that the proposed algorithms will satisfy the reachability property, and thus compute optimal vertex-weight matchings. These are stated in Theorems III.4.2, III.4.3 and III.4.4.

**Theorem III.4.2.** Consider a graph \( G = (S, T, E) \) with weight function \( w : S \to \mathbb{R}^+ \), and a matching \( M_S^G \) computed by algorithm **GlobalOptimal**. The matching \( M_S^G \) satisfies the reachability property.

**Proof.** We will prove the theorem by using mathematical induction. We will consider those steps when Algorithm **GlobalOptimal** augments the current matching, called the augmenting steps. Let \( M_i^S \) correspond to a matching at some intermediate step in the algorithm. We will prove that the theorem holds true at each augmenting step, and therefore at the end of the execution of **GlobalOptimal**.

**Base case:** Let \( s_1 \in S \) be the first matched vertex. Since **GlobalOptimal** considers the \( S \) vertices for augmentation in the decreasing order of weights, \( s_1 \) will precede all other \( S \) vertices from which \( s_1 \) is reachable through an \( M_i^S \)-alternating path. Thus the base case holds true. For simplicity, assume that there are no isolated vertices in \( G \).

**Step k:** Assume that the reachability property holds true after the \( k \)-th augmentation.

**Step (k+1):** Let the \((k+1)\)-th augmentation be performed along the \( M_{k}^S \)-augmenting path \( P_{k+1} \) from \( s_{k+1} \in S \) to \( t_{k+1} \in T \). In order to prove the theorem, we need to show that for any \( M_{k}^S \)-unmatched vertex \( s_i \), and any \( M_{k}^S \)-matched vertex \( s'_i \) reachable through an \( M_{k}^S \)-alternating path, the condition that \( s_i \prec s'_i \) holds after the \((k+1)\)-th augmentation. Note that the vertices \( s'_i \) and \( s_{k+1} \) can be the same.

When the \((k+1)\)-th augmenting path \( P_{k+1} \) and any \( M_{k}^S \)-alternating path between \( s_i \) and \( s'_i \) are vertex disjoint, the \((k+1)\)-th augmentation has no affect on the reachability of \( s'_i \) from \( s_i \). However, if \( s'_i \) becomes reachable after the \((k+1)\)-th augmentation, then the alternating path between \( s_i \) and \( s'_i \), and the augmenting path
$P_{k+1}$ have at least one vertex (and one edge) in common. This is illustrated in Figure 25. Now, there are only two alternatives: (i) the two vertices $s'_i$ and $s_{k+1}$ are the same. In such a case, there was at least one augmenting path from $s_i$ to $t_{k+1}$, but $s_{k+1}$ was preferred, and the condition $s_i \prec s'_i$ holds; or (ii) the two vertices $s'_i$ and $s_{k+1}$ are different. In which case we know that all the matched $s \in S$ vertices succeed $s_{k+1}$, and the condition $s_i \prec s'_i$ holds.

\[\square\]

![Figure 25](image_url)

**FIG. 25:** Illustrates that reachability property holds for Algorithm **GLOBALOPTIMAL**. Bold lines represent the matched edges and matched vertices are colored black. (a) State before $(k + 1)$-th augmentation, (b) state after $(k + 1)$-th augmentation.

**Theorem III.4.3.** Given a graph $G = (S, T, E)$ with weight function $w : S \rightarrow \mathbb{R}^+$, and a matching $M^L_S$ computed by algorithm **LOCALOPTIMAL**. The matching $M^L_S$ satisfies the reachability property.

**Proof.** Similar to the proof of Theorem III.4.3, we will induct on the $M^L_S$ augmenting steps. We will show that at the end of any given iteration of the algorithm, $M^L_S$ will satisfy the reachability property such that for any $M^L_S$-matched vertex $s' \in S$ reachable via an $M^L_S$-alternating path $P$ originating at any $M^L_S$-unmatched vertex $s \in S$, $s \prec s'$.

**Base case:** **LOCALOPTIMAL** will arbitrarily start from a $T$ vertex, say $t_1$, and process all $S$ vertices adjacent to $t_1$, $S_1 = \text{adj}(t_1)$. It will then select the largest $s \in S_1$, say $s_1$. After matching $s_1$ to $t_1$, $s_1$ will be reachable via an $M^L_S$ alternating path only from the vertices in $S_1 \setminus \{s_1\}$. Since $s_1$ is heaviest vertex in $S_1$, the reachability property will hold true for the base case.
Step $k$: Assume that the reachability property holds true after the $k$-th augmentation.

Step $(k+1)$: Given that the reachability property holds true at step $k$, we will prove that it also holds true for step $(k + 1)$. Let the two vertices matched at step $(k + 1)$ be $t_{k+1} \in T$ and $s_{k+1} \in S$. **LOCALOPTIMAL** will consider all the unmatched $S$ vertices reachable via an $M^S_k$-augmenting path from $t_{k+1}$, let this set be $S_{k+1} \subset S$. Vertex $s_{k+1}$ is selected by Algorithm **LOCALOPTIMAL** because it is the largest among all the vertices in the set $S_{k+1}$.

Again, in order to have an impact any $M^S_k$-matched vertex $s' \in S$ reachable via an $M^S_k$-alternating path $P$, after the $(k + 1)$-th augmentation, originating at any $M^S_k$-unmatched vertex $s \in S$, should contain $t_{k+1}$ in the path (Figure 25). The two possibilities are: (i) $s \notin S_{k+1}$, in which case nothing changes with respect to $s$ from the augmentation at step $(k + 1)$. Therefore, from the assumption at step $k$, $s \prec s'$; or (ii) $s \in S_{k+1} \setminus \{s_{k+1}\}$: for these $S$ vertices there are two possibilities - an $M^L_k$-matched vertex $s'$ reachable via an alternating path was reachable either before the $(k + 1)$-th augmentation, and therefore $s \prec s'$, or becomes reachable after the $(k + 1)$-th augmentation. In the latter case, we know that **LOCALOPTIMAL** will select the largest vertex, and therefore, $s \prec s_{k+1}$. From step $k$, we also know that $s_{k+1} \prec s'$ (since $s_{k+1}$ will also be available for matching until step $k + 1$ for all $s'$ vertices reachable via $s_{k+1}$). Thus, $s \prec s'$, and the property holds true for step $(k + 1)$. \hfill \qed

We will now prove that the reachability property holds true for a matching computed by **HYBRIDOPTIMAL** in Theorem III.4.4.

**Theorem III.4.4.** Given a graph $G = (V, E)$ with weight function $w : V \to \mathbb{R}^+$, and a matching $M$ computed by algorithm **HYBRIDOPTIMAL**. The matching $M$ satisfies the reachability property.

**Proof.** Similar to two earlier proofs, we will again induct on the $M$-augmenting steps.

**Base case:** **HYBRIDOPTIMAL** will start from the heaviest vertex, say $v_1$, and process all vertices adjacent to $v_1$, $V_1 = \text{adj}(v_1)$. It will then select the largest vertex in $V_1$, say $w_1$, for matching. After matching the edge $v_1$ to $w_1$, there are two possibilities: (i) vertex $v_1$ will be reachable via an $M$-alternating path from vertices in $V_w \in \text{adj}(w_1)$. But, we already know that $v_1$ is the heaviest vertex, and therefore, reachability property holds; and (ii) vertex $w_1$ will be reachable via an $M$-alternating
path from vertices in $V_v \in \text{adj}(v_1)$. But, HybridOptimal has already processed all vertices in $V_v$ and, $w_1$ is the heaviest vertex in this set. Thus, the reachability property holds for the base case.

**Step k:** Assume that the reachability property holds true after the $k$-th augmentation.

**Step (k+1):** Let the two vertices matched at step $(k + 1)$ be $v_{k+1}$ and $w_{k+1}$. HybridOptimal will start with the current heaviest vertex $v_{k+1}$, and process all vertices reachable via an $M^i$-augmenting path from it, let this set be $P_{k+1} \subset V$. Vertex $w_{k+1}$ is selected by because it is the heaviest among all the vertices in $P_{k+1}$.

Again, we are only concerned with the vertices that become reachable via vertices $v_{k+1}$ and $w_{k+1}$. (Figure 25). However, we are not worried about vertex $v_{k+1}$ becoming reachable to any unmatched vertex after $(k + 1)$-th augmentation because we know that it is the current heaviest vertex. Therefore, we are only concerned about the vertex $w_{k+1}$, and other matched vertices becoming reachable through it. Let $v$ represent the unmatched vertices and $v'$ represent the matched vertices that are reachable via an $M^i$-alternating path from $v$.

The two possibilities are: (i) $v \notin P_{k+1}$, in which case nothing changes with respect to $v$ from the augmentation at step $(k + 1)$. Therefore, from the assumption at step $k$, $v < v'$; or (ii) $v \in P_{k+1} \setminus \{w_{k+1}\}$: for these vertices there are two possibilities - a matched vertex $v'$ reachable via an $M^i$-alternating path was reachable either before the $(k + 1)$-th augmentation, and therefore, $v < v'$, or becomes reachable after the $(k + 1)$-th augmentation. In the latter case, we know that HybridOptimal will select the largest vertex, and therefore, $v < v'$. From step $k$, we also know that $w_{k+1} < v'$ (since $w_{k+1}$ will also be available for matching before step $k + 1$ for all $v'$ vertices reachable via $w_{k+1}$). Thus, $v < v'$, and the property holds true for step $(k + 1)$. □

From Theorems III.4.1, III.4.2, III.4.3 and III.4.4, the optimality of GlobalOptimal, LocalOptimal and HybridOptimal immediately follows, and is stated in Corollary III.4.1.

**Corollary III.4.1.** Given a graph $G = (S, T, E)$ with weight function $w : S \rightarrow \mathbb{R}^+$, Algorithms GlobalOptimal and LocalOptimal will compute maximum vertex-weight matchings $M_S$ in $G$. 
III.5 A REACHABILITY-BASED ALGORITHM

A conceptually similar algorithm to compute maximum vertex weighted matching was proposed by Tabatabaee, Georgiadis and Tassiulas [71]. The authors use the reachability property not only to provide a proof of correctness, but also to design their algorithm. Our goal of this discussion is to demonstrate the power of expressing optimality of a matching using the existence of reachability property in the graph with respect to a matching. The algorithm is sketched in Algorithm 14.

Algorithm 14 Input: a graph $G$. Output: a matching $M$. Effect: computes a maximum vertex-weight matching $M$ in $G$. Associated Data Structures: set $U$ is a stack data structure. The elements in the stack follow a precedence order $\prec$, with the top of the stack being the heaviest element at any given time.

1: procedure ReachabilityBasedAlg($G = (V, E), w : V \rightarrow \mathbb{R}^+$)
2: \hspace{1em} $M \leftarrow M_\star$, a maximum (cardinality) matching;
3: \hspace{1em} $U \leftarrow V \setminus V(M)$ in decreasing order of weights;
4: \hspace{1em} while $U \neq \emptyset$ do
5: \hspace{2em} $u \leftarrow$ top element of $U$;
6: \hspace{2em} $U \leftarrow U \setminus \{u\}$;
7: \hspace{2em} Find an alternating path $P_{u \rightarrow w}$ starting at $u$, such that $w \prec v$;
8: \hspace{2em} if $P_{u \rightarrow w} \neq \emptyset$ then
9: \hspace{3em} $M \leftarrow M \oplus P_{u \rightarrow w}$;
10: \hspace{3em} $U \leftarrow U \cup \{w\}$;
11: \hspace{2em} end if
12: \hspace{1em} end while
13: end procedure

The first step is to compute a maximum (cardinality) matching by ignoring all the weights on the vertices. Let $U \leftarrow V \setminus V(M)$ represent the unmatched vertices in decreasing order of weights. Consider the current heaviest vertex $v \in U$. If there exist an alternating path $P$ between $v$ and any vertex $w$ such that $w \prec v$, then switch $(M \leftarrow M \oplus P)$the matched edges of this path to match vertex $v$ instead of $w$. Note that since $M$ is a maximum matching there cannot exist any augmenting paths in $G$ with respect to $M$. Add $w$ to the set $U$, and repeat. If no such path is found, then remove vertex $v$ from $U$ and continue with the next heaviest vertex in $U$. The algorithm terminates when set $U$ becomes empty. Note that for every unmatched vertex the algorithm attempts to satisfy the reachability property with respect to the current matching. The computational cost for satisfying the reachability property for a vertex can be bounded by $O(|E|)$ (a breadth-first search can be used). The number
of unmatched vertices can be bounded by \(O(\mid V\mid)\), and therefore, the complexity of Algorithm 14 is \(O(\mid V\mid\mid E\mid)\). It can be noted that the reachability-based algorithm is a less sophisticated than the algorithm proposed by Spencer and Mayr [69].

The proof of correctness can be easily shown by demonstrating that Algorithm 14 computes a matching that satisfies the reachability property, and therefore, computes a maximum vertex-weight matching as provided by Theorem III.4.1.

### III.6 CHAPTER SUMMARY

In this chapter we introduced three new algorithms, GlobalOptimal, LocalOptimal and HybridOptimal for computing maximum vertex-weight matchings. Proof of correctness for the proposed algorithms were also discussed. We developed the concept of reachability property as a necessary condition to establish optimality of an MVM.

The proposed algorithms are easy to understand and simple to implement. However, there are limitations to the proposed algorithms, in that we can neither perform greedy initializations nor grow multiple paths. Although the greedy initializations do not have a bound on the approximation ratio, for practical purposes greedy initialization is very important. In some of our preliminary experiments on matrices from applications downloaded from the University of Florida Sparse Matrix Collection, greedy initializations tend to match a substantial percentage of edges. Therefore, we consider that inability to perform greedy initializations for vertex-weighted matching algorithms is a limitation. In Figure 26, we show why greedy initialization fails. A vertex once matched cannot be unmatched in an augmentation-based algorithm. For example, vertex \(T_3\) gets matched in the greedy initialization phase, but should not be matched in a maximum vertex-weight matching. Note that this does not apply to Algorithm ReachabilityBasedAlg.

The multiple-path approach discussed in Chapter 2 has the best time complexity for maximum cardinality matching. However, for the proposed algorithms we will not be able to implement the multiple-path approach. We will encounter the same problem illustrated in Figure 26.
FIG. 26: *Greedy initialization*. Bold lines represent matched edges, and matched vertices are colored black. (a) The input graph $G = (S, T, E)$, weights are associated only with the $T$ vertices, (b) a greedy initialization that picks best augmenting paths of length one, and (c) an optimal matching.
CHAPTER IV

APPROXIMATION ALGORITHMS

"Although this may seem a paradox, all exact science is dominated by the idea of approximation." - Bertrand Russell [3]

IV.1 INTRODUCTION

Approximation algorithms are generally developed for computationally intractable problems [35]. For some applications such as the multi-level algorithm for graph partitioning, matchings are computed a large number of times within the algorithm [40]. For other applications such as algorithms used in the design of VLSI devices, matchings are computed on very large-scale graphs [74]. The need for fast approximation matching algorithms arise from both types of applications, especially when the need for speed overrides the need for accuracy. Some of these applications are discussed in [24, 64]. We provided an introduction to approximation algorithms for edge-weighted matching problem in Chapter II. In this chapter we propose new algorithms that guarantee approximation ratios of $\frac{1}{2}$ and $\frac{2}{3}$ for the maximum vertex-weight matching (MVM) problem. The $\frac{1}{2}$-approx algorithms have linear runtimes with respect to the number of edges in the graphs and log-linear in terms of the number of vertices. The log term arising due to sorting for the global approximation algorithm. The $\frac{1}{2}$-approx algorithm is log-linear with respect to the number of vertices for degree-bounded graphs. The proposed approximation algorithms are conceptually similar to the exact algorithms discussed in Chapter III, and can be classified into global and local approaches for computing the matchings. We refer the reader to Chapter III for a basic introduction on the vertex-weighted matching problem. Table 6 has been reproduced here for the ease of reading.

We begin the discussion with the $\frac{1}{2}$-approx algorithms, and proceed to the $\frac{2}{3}$-approx algorithm. The general structure of the approximation algorithms is similar to the exact algorithms discussed earlier.

IV.2 NEW $\frac{1}{2}$-APPROX ALGORITHMS

We propose three new algorithms for computing $\frac{1}{2}$-approx to the MVM problem.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exact Algorithms</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLOBALOPTIMAL</td>
<td>B</td>
<td>Sort-based</td>
<td>$O(n \log n + nm)$</td>
</tr>
<tr>
<td>LOCALOPTIMAL</td>
<td>B</td>
<td>Search-based</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>HYBRIDOPTIMAL</td>
<td>G</td>
<td>Sort and search-based</td>
<td>$O(n \log n + nm)$</td>
</tr>
<tr>
<td><strong>Approximation Algorithms</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLOBALHALF</td>
<td>B</td>
<td>$\frac{1}{2}$-approx; Sort-based</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>LOCALHALF</td>
<td>B</td>
<td>$\frac{1}{2}$-approx; Search-based</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>HYBRIDHALF</td>
<td>G</td>
<td>$\frac{1}{2}$-approx; Sort and search-based</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>GLOBALTWOThird</td>
<td>B</td>
<td>$\frac{1}{3}$-approx; Sort-based</td>
<td>$O(n \log n + nd_k)$</td>
</tr>
</tbody>
</table>

**TABLE 7:** A summary of algorithms proposed for vertex weighted matchings. Bipartite and general graphs are represented with B and G respectively. For a bipartite graph $G = (S, T, E)$, $n = (|S| + |T|)$ represents the number of vertices, $m = |E|$ the number the edges, and $d_k$ is a generalization of the vertex degree that denotes the average number of distinct alternating paths of length at most $k$ edges starting at a vertex in $G$.

The first proposed algorithm, GLOBALHALF, is based on processing the vertices in a global order of decreasing weights. Given a bipartite graph $G = (S, T, E)$ with weights associated with the $S$ and $T$ vertices, we will decompose it into two restricted bipartite graphs by first ignoring the weights on $T$ vertices, and then on $S$ vertices. The problem decomposition is represented in Algorithm 15 by Lines 5 and 6 for the $S$ vertices, and by Lines 15 and 16 for the $T$ vertices.

Consider the first restricted bipartite graph with weights on only $S$ vertices. Algorithm 15 processes the $S$ vertices in a succeeding order ($s_1 \succ s_2 \succ s_3 \ldots$). From a given vertex $s_i \in S$, search for any unmatched vertex $t_i \in T$ adjacent to $s_i$. If such a vertex is found, then add it to the current matching and proceed with the next unmatched $S$ vertex in succeeding order. Computation of $M_S$ in Algorithm 15 is represented by Lines 7 – 14. A similar approach to compute the matching $M_T$ for the second restricted graph is represented by Lines 17 – 24 in Algorithm 15. The final matching is obtained by merging the two matchings $M_S$ and $M_T$ using the Mendelsohn-Dulmage technique. Execution of Algorithm GLOBALHALF on a simple bipartite graph with weights associated only to the $S$ vertices is shown in Figure 27.

For the second $\frac{1}{2}$-approx algorithm, LOCALHALF, we adopt a strategy based on searching for an unmatched edge from the unweighted vertices in arbitrary order;
Algorithm 15  Input: A bipartite graph $G$.  Output: a matching $M$.  Associated Data Structures: sets $\tilde{S}$ and $\tilde{T}$ are stored as stack data structures. The elements in the stack follow a precedence order $\prec$, with the top of the stack being the heaviest element at any given time. Effect: computes a $\frac{1}{2}$-approx to maximum vertex-weight matching.

1: procedure GlobalHalf($G = (S, T, E), w_S : S \rightarrow \mathbb{R}^+, w_T : T \rightarrow \mathbb{R}^+, M$)  
2: $M \leftarrow \phi$; ▷ Initialization  
3: $M_S \leftarrow \phi$;  
4: $M_T \leftarrow \phi$;  
5: $\tilde{S} \leftarrow S$ in descending order of weights $w_S$;  
6: $\tilde{T} \leftarrow T$ with weights $w_T$ set to zero;  
7: while $\tilde{S} \neq \phi$ do ▷ Compute $M_S$  
8: $s \leftarrow$ top of $\tilde{S}$;  
9: $\tilde{S} \leftarrow \tilde{S} \setminus s$;  
10: Find an unmatched edge $e_{st}$ incident on $s$;  
11: if $e_{st}$ exists then  
12: $M_S \leftarrow M_S \cup \{e_{st}\}$;  
13: end if  
14: end while  
15: $\tilde{T} \leftarrow T$ in descending order of weights $w_T$;  
16: $\tilde{S} \leftarrow S$ with weights $w_S$ set to zero;  
17: while $\tilde{T} \neq \phi$ do ▷ Compute $M_T$  
18: $t \leftarrow$ top of $\tilde{T}$;  
19: $\tilde{T} \leftarrow \tilde{T} \setminus t$;  
20: Find an unmatched edge $e_{ts}$ incident on $t$;  
21: if $e_{ts}$ exists then  
22: $M_T \leftarrow M_T \cup \{e_{ts}\}$;  
23: end if  
24: end while  
25: $M \leftarrow$ MendelsohnDulmage($M_S, M_T, M$); ▷ Compute $M$  
26: end procedure
FIG. 27: *Execution of Algorithm* GLOBALHALF. (a) The input graph $G = (S, T, E)$ with weights associated only with the $S$ vertices, (b)-(e) the intermediate states of execution. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges mark the augmenting paths of length one (an unmatched edge) from a given $S$ vertex, (f) the final state.

however, we will need to find an edge that leads to the heaviest vertex on the weighted side. This approach does not depend on a global order but, on a local search. The input graph is divided into two restricted bipartite graphs by first ignoring the weights on the $T$ vertices and then on the $S$ vertices. The decomposition is represented in Algorithm 16 by Lines 5 and 14.

For the first restricted bipartite graph, a matching $M_S$ is computed as follows: arbitrarily start from an unmatched vertex $t_i \in T$, and enumerate all the unmatched edges incident on the vertex $t_i$. If such edges exist, then choose the *best edge* from this set and augment the current matching. We define the best edge as the edge that leads to a heaviest weighted vertex. Repeat the process until all the $T$ vertices have been processed. Lines 6 – 13 in Algorithm 16 represent the computation of $M_S$. A similar procedure can be used to compute the matching $M_T$ for the second restricted bipartite graph. This is represented by Lines 15 – 22 in Algorithm 16. The final matching will be obtained by merging the two matchings $M_S$ and $M_T$ using the Mendelsohn-Dulmage technique. The execution of Algorithm LOCALHALF on a
Algorithm 16 Input: a bipartite graph $G$. Output: a matching $M$. Associated Data Structures: sets $\tilde{S}$ and $\tilde{T}$ are stored as stack data structures. The elements in the stack can be in any arbitrary order. Effect: computes a $\frac{1}{2}$-approx MVM.

1: procedure LOCALHALF($G=(S,T,E), w_S: S \rightarrow \mathbb{R}^+, w_T: T \rightarrow \mathbb{R}^+, M$)
2: \hspace{1em} $M \leftarrow \phi$; \hspace{1em} $\triangleright$ Initialization
3: \hspace{1em} $M_S \leftarrow \phi$;
4: \hspace{1em} $M_T \leftarrow \phi$;
5: \hspace{1em} $\tilde{T} \leftarrow T$ with weights $w_T$ set to zero ;
6: \hspace{1em} while $\tilde{T} \neq \phi$ do $\triangleright$ Compute $M_S$
7: \hspace{2em} $t \leftarrow$ top of $\tilde{T}$;
8: \hspace{2em} $\tilde{T} \leftarrow \tilde{T} \setminus t$;
9: \hspace{2em} Find all unmatched edges $e_{ts}$ incident on $t$;
10: \hspace{2em} if unmatched edges exist then
11: \hspace{3em} $M_T \leftarrow M_T \cup \{e_{\text{best}}\}$; \hspace{1em} $\triangleright e_{\text{best}}$ is $e_{ts}$ with largest $w(s)$
12: \hspace{2em} end if
13: \hspace{1em} end while
14: $\tilde{S} \leftarrow S$ with weights $w_S$ set to zero ;
15: \hspace{1em} while $\tilde{S} \neq \phi$ do $\triangleright$ Compute $M_T$
16: \hspace{2em} $s \leftarrow$ top of $\tilde{S}$;
17: \hspace{2em} $\tilde{S} \leftarrow \tilde{S} \setminus s$;
18: \hspace{2em} Find all unmatched edges $e_{st}$ incident on $s$;
19: \hspace{2em} if unmatched edges exist then
20: \hspace{3em} $M_S \leftarrow M_S \cup \{e_{\text{best}}\}$; \hspace{1em} $\triangleright e_{\text{best}}$ is $e_{st}$ with the largest $w(t)$
21: \hspace{2em} end if
22: \hspace{1em} end while
23: $M \leftarrow$ Mendelsohn-Dulmage($M_S, M_T, M$); \hspace{1em} $\triangleright$ Compute $M$
24: end procedure
simple bipartite graph with weights associated with $S$ vertices is shown in Figure 28.

FIG. 28: *Execution of Algorithm LocalHalf*. (a) The input graph $G = (S, T, E)$ with weights associated only with the $S$ vertices, (b)-(d) the intermediate states of execution, (e) the final state. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges mark all the augmenting paths of length one (unmatched edges) that exist from a given $T$ vertex.

The third $\frac{1}{2}$-approx algorithm, HybridHalf, is designed to compute matchings in general graphs where the problem cannot be decomposed into two subgraphs. We combine the global and local strategies to form a hybrid approach, where the vertices are processed in a global order of decreasing weight. The search for an unmatched edge incident on the current heaviest vertex is made by processing all the adjacent edges, but picking the edge with the heaviest vertex incident on it. Algorithm 17 sketches the hybrid approach.

A $\frac{1}{2}$-approx matching $M$ is computed as follows: consider vertices in decreasing order of weights. Enumerate all the unmatched edges incident on the current heaviest vertex $v_i \in V$. If such edges exist, then choose the *best edge* from this set and augment the current matching. We define the best edge as the edge that leads to the heaviest neighboring vertex. Repeat the process until all the vertices have been processed.

We now discuss the correctness of the proposed $\frac{1}{2}$-approx algorithms.
Algorithm 17 Input: a graph $G$. Output: a matching $M$. Associated Data Structures: set $\tilde{V}$ is a stack data structure. The elements in the stack follow a precedence order $\prec$, with the top of the stack being the heaviest element at any given time. Effect: computes a $\frac{1}{2}$-approx MVM.

1: procedure HYBRIDHALF($G = (V, E), w : V \rightarrow \mathbb{R}^+$)
2: $M \leftarrow \phi$;
3: $\tilde{V} \leftarrow V$ in increasing order of weights;
4: while $\tilde{V} \neq \phi$ do $\triangleright$ Compute $M$
5: $v \leftarrow$ top of $\tilde{V}$;
6: $\tilde{V} \leftarrow \tilde{V} \setminus v$;
7: Find all unmatched edges $e_{vx}$ incident on $v$;
8: if unmatched edges exist then
9: $M \leftarrow M \cup \{e_{best}\}$ $\triangleright$ $e_{best}$ is $e_{vw}$ with largest $w(x)$
10: $\tilde{V} \leftarrow \tilde{V} \setminus w$;
11: end if
12: end while
13: end procedure
IV.3 PROOF OF CORRECTNESS

The proofs of correctness of the $\frac{1}{2}$-approx algorithms are fairly straightforward. The main idea is to establish a relationship between the matched and the unmatched vertices that reveal the correctness of the approximation ratio. In order to establish this relationship, we will introduce the concept of failed vertices. Consider a graph $G = (S, T, E)$ with weight function $w : S \rightarrow \mathbb{R}^+$, a matching $M_*$ computed by Algorithm $\text{GLOBALOPTIMAL}$, and a matching $M_2$ computed by Algorithm $\text{GLOBALHALF}$. A failed $S$-vertex is a vertex that is matched in $M_*$, but not in $M_2$. The same definition is extended to the $T$ vertices for the second restricted bipartite graph, can also be extended similarly for Algorithms $\text{LOCALOPTIMAL}$ and $\text{LOCALHALF}$, $\text{HYBRIDOPTIMAL}$ and $\text{HYBRIDHALF}$). No distinction between $S$ and $T$ vertices is made for general graphs.

The intuition for the proof of correctness comes from the fact that for every failed vertex, there will be at least one distinct vertex, at least as large as the failed vertex, that will be matched by the $\frac{1}{2}$-approx algorithm. Thus, resulting in half approximation to the optimal matching. This relationship is characterized by the restricted reachability property, which can be defined as follows.

**Definition IV.3.1** (Restricted Reachability Property). Consider a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$, and any matching $M$ in $G$. The matching $M$ satisfies the restricted reachability property if for any $M$-unmatched vertex $v$, and any $M$-matched vertex $v'$ reachable from $v$ by an $M$-alternating path of length two edges, the condition $v \prec v'$ holds.

We show that if a given maximal matching satisfies the restricted reachability property, then it is also a $\frac{1}{2}$-approx to the optimal matching. This is stated in Lemma IV.3.1.

**Lemma IV.3.1.** Consider a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$, and a maximal matching $M_2$ in $G$. If $M_2$ satisfies the restricted reachability property, then $M_2$ is a $\frac{1}{2}$-approximation to a maximum vertex-weight matching in $G$.

**Remark:** The reader should note the requirement for a maximal matching in this Lemma.

**Proof.** Let $M_*$ represent a maximum vertex-weight matching, and $M_2$ represent any
maximal matching in $G$ with the restricted reachability property. Consider the symmetric difference $M_\ast \oplus M_2$. This will result in a collection of paths and cycles. All possibilities for bipartite graphs are enumerated in Figure 20. Note that even for general graphs, there cannot be cycles of odd length, and therefore, Figure 20 also holds true for general graphs (without the distinction of $S$ and $T$ vertex sets). The edges that are matched in both the algorithms will not be represented in the symmetric difference, and these edges will not have a negative impact on the approximation ratio. The vertices in a cycle will be matched by both the matchings, and therefore, will not affect the approximation ratio. For this lemma, we only need to consider the paths, augmenting or alternating.

Consider the paths that start at failed vertices (matched in $M_\ast$, but not in $M_2$). Given that $M_2$ is a maximal matching, paths of length one in $M_\ast \oplus M_2$ cannot exist. Consider an alternating path of length two, of form $[v, w, v']$, in $M_\ast \oplus M_2$. Since, $M_2$ satisfies the restricted reachability property, $v \prec v'$. Such an alternating path would contradict the optimality of a maximum vertex-weight matching, and therefore, cannot exist.

Now let us consider paths of length greater than two in $M_\ast \oplus M_2$. Consider an $M_2$-augmenting path of length three, of form $[v_1, v_2, v_3, v_4]$, where $v_1$ and $v_4$ are matched only in $M_\ast$. From the restricted reachability property in $M_2$, $v_1 \prec v_3$ and $v_4 \prec v_2$. The same argument will hold for all augmenting paths of length five or more. Consider an $M_2$-alternating path of length four, of form $[v_1, v_2, v_3, v_4, v_5]$, where $v_1$ is matched only in $M_\ast$, and $v_5$ is matched only in $M_2$. From the restricted reachability property in $M_2$, $v_1 \prec v_3$ and all other vertices are matched in $M_2$. Thus, for alternating paths of any even-length $P_i$, except the very first vertex, $M_2$ matches all other vertices in $P_i$, irrespective of the length of $P_i$. Let $V(M)$ represent the vertices matched in $M$. From the restricted reachability property, on the path $P_i$, for a vertex $v_i'$ at a distance two edges from $v_i$, the following relation holds $v_i \prec v_i'$. Summing over all the failed vertices ($N$), we obtain:

$$\sum_{i=1}^{N} w(v_i) \leq \sum_{i=1}^{N} w(v_i').$$  \hspace{1cm} (4)

The weight of the maximum vertex-weight matching can be represented as follows.

$$\sum_{v \in V(M_\ast)} w(v) = \sum_{v_i \in V(M_\ast \setminus M_2)} w(v_i) + \sum_{v_j \in V(M_\ast \cap M_2)} w(v_j).$$ \hspace{1cm} (5)
The set $V(M_e \setminus M_2)$ represents the set of failed vertices. Therefore, we can rewrite the first term on the right-hand-side of Equation 5 with respect to the failed vertices as

$$\sum_{v \in V(M_e)} w(v) = \sum_{i=1}^{N} w(v_i) + \sum_{v_j \in V(M_e \cap M_2)} w(v_j).$$

(6)

Substituting from Equation 4 we get

$$\sum_{v \in V(M_e)} w(v) \leq \sum_{i=1}^{N} w(v_i') + \sum_{v_j \in V(M_e \cap M_2)} w(v_j).$$

(7)

Each of two sets on the right-hand-side of the equation above are subsets of the matched vertices in $M_2$, and thus we have

$$\sum_{v \in V(M_e)} w(v) \leq \sum_{v_i \in V(M_2)} w(v_i) + \sum_{v_j \in V(M_2)} w(v_j).$$

(8)

Rewriting the equation above, we have

$$\sum_{v \in V(M_2)} w(v) \geq \frac{1}{2} \sum_{v \in V(M_e)} w(v).$$

(9)

With the result from Lemma IV.3.1, we only need to show that a given $\frac{1}{2}$-approx algorithm satisfies the restricted reachability property. We will use mathematical induction and show that Algorithms GLOBALHALF, LOCALHALF and HYBRIDHALF satisfy the restricted reachability property. This is stated in Theorems IV.3.1, IV.3.2 and IV.3.3 respectively.

**Theorem IV.3.1.** Consider a graph $G = (S, T, E)$ with weight function $w : S \rightarrow R^+$. A maximal matching $M_2$ in $G$ computed by Algorithm GLOBALHALF satisfies the restricted reachability property.

**Proof.** A necessary condition for the theorem is that the matching $M_2$ be a maximal matching. We will first prove that GLOBALHALF will compute a maximal matching. Consider step $k$ during the execution of GLOBALHALF when vertex $s_k \in S$ is processed. If at step $k$ no augmenting path of length one, starting at $s_k$ exists, then it means that all the adjacent vertices $t_i \in \text{adj}(s_k)$ have already been matched before step $k$. In order to create a new augmenting path of length one from $s_k$ at a future step, one of the adjacent $T$ vertices must be unmatched. However, we also know that
a vertex (or edge) once matched will always remain matched during the course of this algorithm. Thus, if none exist at a given step, no new augmenting path of length one can become available at a future step. GLOBALHALF searches all the $S$ vertices for augmenting paths of length one. Thus, GLOBALHALF will compute a maximal matching in $G$.

Let $M^k_2$ represent a matching computed by GLOBALHALF at the end of step $k$. We will induct on the steps when $M_2$ matches a new $S$ vertex, and show that the theorem holds true at each augmenting step, and therefore, at the end of execution of Algorithm GLOBALHALF.

**Base case**: Let $s_1 \in S$ be the first matched vertex. Since Algorithm GLOBALHALF will consider the $S$ vertices for augmentation in decreasing order of weights, $s_1$ will succeed all other $S$ vertices from which $s_1$ is reachable through an $M^1_2$-alternating path. Thus, the base case holds true.

**Step k**: Assume that the restricted reachability property holds true after the $k$-th augmentation.

**Step $(k+1)$**: Let GLOBALHALF process vertex $s_{k+1} \in S$ at step $(k+1)$, and let $s_{k+1}$ be matched to $t_{k+1} \in T$. Consider an $M^{k+1}_2$-unmatched vertex $s \in S$, and an $M^{k+1}_2$-matched vertex $s' \in S$ reachable via an $M^{k+1}_2$-alternating path of length two edges from $s$. The two possibilities are: (i) $s'$ was reachable from $s$ before the $(k+1)$-th augmentation, in which case, $s < s'$ from step $k$, or (ii) $s'$ becomes reachable from $s$ after the $(k+1)$-th augmentation, which means that $s'$ and $s_{k+1}$ represent the same vertex. Also, $s$ is one of the unmatched $S$ vertices adjacent to $t_{k+1}$. However, we know that $s_{k+1}$ succeeds all the unmatched $S$ vertices adjacent to $t_{k+1}$. By the structure of GLOBALHALF, $s < s_{k+1}$. Thus, the theorem holds true. $\square$

**Theorem IV.3.2.** Consider a graph $G = (S, T, E)$ with weight function $w : S \rightarrow R^+$. A maximal matching $M_2$ in $G$ computed by Algorithm LOCALHALF satisfies the restricted reachability property.

**Proof.** This proof is similar to the proof of restricted reachability property for GLOBALHALF as discussed in Theorem IV.3.1.

A necessary condition for the theorem is that the matching $M_2$ is a maximal matching. We will first prove that LOCALHALF will compute a maximal matching. Consider step $k$ during the execution of LOCALHALF when vertex $t_k \in T$ is processed. If at step $k$ no augmenting path of length one starting at vertex $t_k$ exists, then all the adjacent vertices $s_i \in adj(t_k)$ have already been matched before this step. In order
to create a new augmenting path of length one from \( t_k \) at a future step, one of the adjacent \( S \) vertices must be unmatched. However, we also know that a vertex (and edge) once matched will always remain matched during the course of this algorithm. Thus, no new augmenting path of length one can become available at a future step, if none exists at a given step. \textsc{LocalHalf} searches all the \( T \) vertices for augmenting paths of length one. Thus, \textsc{LocalHalf} will compute a \textit{maximal} matching in \( G \).

Let \( M^k_2 \) represent a matching computed by \textsc{LocalHalf} at the end of step \( k \). We will induct on the steps when \( M_2 \) matches a new \( S \) vertex, and show that the theorem holds true at each augmenting step, and therefore, at the end of execution of Algorithm \textsc{LocalHalf}.

**Base case:** Let the first edge matched by \textsc{LocalHalf} be \((t_1, s_1) \in E\). We know that \textsc{LocalHalf} will consider all the \( s_i \in S \) adjacent to \( t_1 \), before matching it to \( s_1 \). Therefore, the restricted reachability property holds true for the base case.

**Step \( k \):** Assume that the restricted reachability property holds true after the \( k \)-th augmentation.

**Step \( (k+1) \):** Let \((t_{k+1}, s_{k+1}) \in E\) be the edge matched by Algorithm \textsc{LocalHalf} at step \((k + 1)\). Consider an \( M_2^{k+1}\)-unmatched vertex \( s \in S \), and an \( M_2^{k+1}\)-matched vertex \( s' \in S \) reachable via an \( M_2^{k+1}\)-alternating path of length two edges from \( s \). The two possibilities are: (i) \( s' \) was reachable from \( s \) before the \((k + 1)\)-th augmentation, in which case, \( s \prec s' \) from step \( k \), or (ii) \( s' \) becomes reachable from \( s \) after the \((k + 1)\)-th augmentation, which means that both \( s' \) and \( s_{k+1} \) represent the same vertex. Also, \( s \) will have to be one of the unmatched \( S \) vertices adjacent to \( t_{k+1} \). Since \textsc{LocalHalf} processes all the unmatched \( S \) vertices adjacent to \( t_{k+1} \) we have \( s \prec s_{k+1} \). Thus, the theorem holds. \( \square \)

**Theorem IV.3.3.** Consider a graph \( G = (V, E) \) with weight function \( w: V \to \mathbb{R}^+ \). A maximal matching \( M_2 \) in \( G \) computed by Algorithm \textsc{HybridHalf} satisfies the restricted reachability property.

**Proof.** This proof is similar to the proofs of restricted reachability property for the bipartite graphs.

Again, a necessary condition for the theorem is that the matching \( M_2 \) is a maximal matching. From the earlier proofs that argue that no new augmenting paths of length one can become available at a future step, if none exists at a given step, it can be easily shown that \textsc{HybridHalf} will compute a maximal matching.
Let $M^k_2$ represent a matching computed by HybridHalf at the end of step $k$. Again, we will induct on the steps when $M_2$ matches a new vertex and show that the theorem holds true at each augmenting step, and therefore, at the end of execution.

**Base case:** Let the first edge matched by HybridHalf be $(v_1, w_1) \in E$, while processing vertex $v_1$. For the restricted reachability property to hold, we need to show that all the $v_1$-adjacent vertices (reachable to $w_1$), and all the $w_1$-adjacent vertices (reachable to $v_1$) will satisfy the required property. We already know that $v_1$ is the heaviest vertex, and HybridHalf will consider all the $w_i \in V$ adjacent to $v_1$, before matching it to $w_1$. Therefore, the restricted reachability property holds true for the base case.

**Step $k$:** Assume that the restricted reachability property holds true after the $k$-th augmentation.

**Step $(k+1)$:** Let $(v_{k+1}, w_{k+1}) \in E$ be the edge matched by HybridHalf at step $(k + 1)$, while processing vertex $v_{k+1}$.

Consider an $M^{k+1}_2$-unmatched vertex $v$, and an $M^{k+1}_2$-matched vertex $v'$ reachable via an $M^{k+1}_2$-alternating path of length two edges from $v$. The two possibilities are: (i) $v'$ was reachable from $v$ before the $(k + 1)$-th augmentation, in which case, $v \prec v'$ from the assumption at induction-step $k$, or (ii) $v'$ becomes reachable from $v$ after the $(k + 1)$-th augmentation. This means that both $v'$ can either be $v_{k+1}$ or $w_{k+1}$. Therefore, we only need to consider the vertices adjacent to vertices $v_{k+1}$ or $w_{k+1}$.

We know that $v_{k+1}$ is the heaviest vertex among all the unmatched vertices at this stage, and therefore, all the unmatched $w_{k+1}$-adjacent vertices will be lighter that it. We also know that HybridHalf considered all the unmatched $w_i \in V$ adjacent to $v_{k+1}$, and $w_{k+1}$ was the heaviest of all. Thus, the theorem holds.

For the bipartite graphs where the final matching $M$ is computed by merging the matchings $M_S$ and $M_T$ using the Mendelsohn-Dulmage technique (Theorem III.1.1). It is not guaranteed that $M$ be maximal. However, the matching $M_S$ is already $\frac{1}{2}$-approx with respect to the $S$ vertices and this approximation ratio also holds for $S$-vertices matched in $M$. It can similarly be extended to the $T$ vertices.

From Lemma IV.3.1, and Theorems IV.3.1, IV.3.2 and IV.3.3, the optimality of Algorithms GlobalHalf and LocalHalf immediately follows, and is stated in Corollary IV.3.1.

**Corollary IV.3.1.** Given a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}^+$,
Algorithm HybridHalf will compute $\frac{1}{2}$-approx to a maximum vertex-weight matching in $G$. Given a bipartite graph $G = (S, T, E)$ with weight function $w : S, T \rightarrow \mathbb{R}^+$, Algorithms GlobalHalf and LocalHalf will compute $\frac{1}{2}$-approx to a maximum vertex-weight matching in $G$.

The time complexities for the $\frac{1}{2}$-approx algorithms are stated in Theorems IV.3.4 and IV.3.5.

**Theorem IV.3.4.** Given a graph $G = (S, T, E)$ with weight functions $w_S : S \rightarrow \mathbb{R}^+$ and $w_T : T \rightarrow \mathbb{R}^+$, let $n = (|S| + |T|)$ represent the number of vertices and $m = |E|$ represent the number of edges. A $\frac{1}{2}$-approx matching $M_2$ in $G$ can be computed in $O(n \log n + m)$ time by Algorithm GlobalHalf and in $O(m)$ time by Algorithm LocalHalf.

**Proof.** Algorithm GlobalHalf processes the vertices in a global order. The given set of vertices can be sorted in decreasing order of vertex weights in time $O(n \log n)$. From each set of vertices $S$ and $T$, GlobalHalf will consider the adjacent edges and will therefore compute a matching $M_S$ and $M_T$ bounded by $O(m)$, resulting in a total time complexity of $O(n \log n + m)$. The two matchings, $M_S$ and $M_T$ can be merged using the Mendelsohn-Dulmage technique in linear time, $O(m)$. Since Algorithm LocalHalf does not need to process the vertices in a global order, it is bounded by $O(m)$. 

**Theorem IV.3.5.** Given a graph $G = (V, E)$ with weight functions $w : V \rightarrow \mathbb{R}^+$ let $n = (|V|)$ represent the number of vertices and $m = |E|$ represent the number of edges. A $\frac{1}{2}$-approx matching $M_2$ in $G$ can be computed in $O(n \log n + m)$ time by Algorithm HybridHalf.

**Proof.** HybridHalf processes the vertices in a global order. The given set of vertices can be sorted in decreasing order of vertex weights in time $O(n \log n)$. For each vertex, HybridHalf will process all the adjacent edges and will therefore incur a cost of by $\Theta(m) = \sum_{v \in V} \delta(v)$, where $\delta(v)$ is the degree of vertex $v$. Thus, the total complexity is $O(n \log n + m)$. 

We will now proceed to the $\frac{2}{3}$-approx algorithms.
IV.4 GLOBAL $\frac{2}{3}$-APPROX ALGORITHM

The first proposed algorithm for $\frac{2}{3}$-approx GLOBAL TWO THIRDS is similar to the half-approximation Algorithm GLOBAL HALF. The main idea is to process the vertices according to a global order of weights associated with the vertices. For Algorithm GLOBAL TWO THIRDS, we first decompose the given bipartite graph $G = (S, T, E)$, with weights associated with both $S$ and $T$ vertices, into two restricted bipartite graphs by first ignoring the weights on $T$ vertices and then on $S$ vertices. This process is represented in Algorithm 18 by Lines 5 and 6 for the $S$ vertices, and by Lines 15 and 16 for the $T$ vertices.

Consider the first restricted bipartite graph $G = (S, T, E)$ with weight function $w : S \rightarrow \mathbb{R}^+$. A $\frac{2}{3}$-approx matching $M_S$ is computed by considering the $S$ vertices in succeeding order. From a given $S$ vertex $s_i$, find a shortest augmenting path $P$ of length $\leq 3$. If such an augmenting path is found, then augment the current matching with the symmetric difference $M_S \oplus P$ and continue with the next $S$ vertex in succeeding order. Computation of $M_S$ is represented by Lines 7 – 14. A similar approach to compute the matching $M_T$, when weights are associated only with the $T$ vertices, is the second subproblem and is represented by Lines 17 – 24 in GLOBAL TWO THIRDS. The final matching will be obtained by merging the two matchings $M_S$ and $M_T$ using the Mendelsohn-Dulmage technique. Execution of Algorithm GLOBAL TWO THIRDS on a simple bipartite graph with weights associated with the $S$ vertices is shown in Figure 29.

IV.4.1 Proof of Correctness

While the proof of correctness for the $\frac{1}{2}$-approx algorithms is straightforward, the proof of correctness for the $\frac{2}{3}$-approx algorithms is nontrivial. The concept of reachability that was used to build the proofs for exact and $\frac{1}{2}$-approx algorithms will not be sufficient for the current task. We will now show why the previous arguments fail for the $\frac{2}{3}$-approx algorithms. Consider the symmetric difference of the matchings computed by Algorithms GLOBAL OPTIMAL and GLOBAL TWO THIRDS, $M_4 \oplus M_3$. The result will be a set a distinct paths and cycles. Note that the paths will always start and end with vertices that are matched only by one of the algorithms, however, all the intermediate vertices will be matched by both. Let us consider those paths that start with an $S$ vertex matched only by Algorithm GLOBAL OPTIMAL; we call
Algorithm 18 Input: A bipartite graph $G$. Output: a matching $M$. Associated Data Structures: sets $\tilde{S}$ and $\tilde{T}$ are stored as stack data structures. The elements in the stack follow a precedence order $\prec$, with the top of the stack being the heaviest element at any given time. Effect: computes a $\frac{2}{3}$-approx to a maximum vertex-weight matching.

1: procedure \textsc{GlobalTwoThird}(G = (S, T, E), w_S : S → R^+, w_T : T → R^+, M)
2: \hspace{1em} M ← φ;
3: \hspace{1em} M_S ← φ;
4: \hspace{1em} M_T ← φ;
5: \hspace{1em} $\tilde{S}$ ← $S$ in descending order of weights $w_S$;
6: \hspace{1em} $\tilde{T}$ ← $T$ with weights $w_T$ set to zero ;
7: \hspace{1em} while $\tilde{S} \neq φ$ do
8: \hspace{2em} $s$ ← top of $\tilde{S}$;
9: \hspace{2em} $\tilde{S} ← \tilde{S} \setminus s$;
10: \hspace{2em} Find a shortest augmenting path $P$ of length $\leq 3$ starting at $s$;
11: \hspace{2em} \hspace{1em} if $P$ found then
12: \hspace{3em} $M_S ← M_S \oplus P$;
13: \hspace{2em} \hspace{1em} end if
14: \hspace{1em} end while
15: \hspace{1em} $\tilde{T}$ ← $T$ in descending order of weights $w_T$;
16: \hspace{1em} $\tilde{S}$ ← $S$ with weights $w_S$ set to zero ;
17: \hspace{1em} while $\tilde{T} \neq φ$ do
18: \hspace{2em} $t$ ← top of $\tilde{T}$;
19: \hspace{2em} $\tilde{T} ← \tilde{T} \setminus t$;
20: \hspace{2em} Find a shortest augmenting path $P$ of length $\leq 3$ starting at $t$;
21: \hspace{2em} \hspace{1em} if $P$ found then
22: \hspace{3em} $M_T ← M_T \oplus P$;
23: \hspace{2em} \hspace{1em} end if
24: \hspace{1em} end while
25: \hspace{1em} $M ← \textsc{MendelsohnDulmage}(M_S, M_T, M)$;
26: end procedure
such vertices as the failed vertices, since the approximation algorithm failed to match them. While it can be shown that the failed vertex is lighter than the $M_3$-matched $S$ vertex at a distance of two edges from it, such a relationship between the failed vertex and the $M_3$-matched $S$ vertex at a distance of four edges from it cannot be established. This failure is illustrated with a simple example in Figure 30.

Albeit this shortfall, the intuition for the proof is still to show that for each failed vertex there are at least two vertices, as heavy as the failed vertex, that will be matched by Algorithm GlobalTwoThird. This association will immediately result in the $\frac{2}{3}$-approximation. Figure 31 captures this association.

With this intuition, we will now build the proof. We will discuss the proof for the first restricted bipartite graph where the weights are associated only with the $S$ vertices. The same proof can be trivially extended to the second restricted bipartite graph with weights associated with the $T$ vertices. Consider the concurrent execution of Algorithms GlobalOptimal and GlobalTwoThird on the restricted bipartite graph $G = (S, T, E)$ with weight function $w : S \to \mathbb{R}^+$. Both the algorithms

FIG. 29: *Execution of Algorithm GlobalTwoThird.* (a) The input graph $G = (S, T, E)$ before the execution, weights are associated only with $S$ vertices, (b)-(e) the intermediate states of execution. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight the shortest augmenting path from a given $S$ vertex, and (f) the final state.
FIG. 30: Symmetric difference. (a) Input graph, weights are associated only with the $S$ vertices such that $s_1 \succ s_2 \succ s_3 \succ s_4$; (b) an optimal matching $M_*$ computed by Algorithm $\text{GlobalOptimal}$. Bold lines represent matched edges. At step one, edge $e(s_1,t_3)$ is matched; at step two, edge $e(s_2,t_2)$ is matched; at step three, the matching is augmented via path $[s_3,t_2,s_2,t_3,s_1,t_1]$; no path exists at step four; (c) a $2/3$-approx matching $M_3$ computed by Algorithm $\text{GlobalTwoThird}$, Wavy lines represent matched edges; At step one, edge $e(s_1,t_3)$ is matched; at step two, edge $e(s_2,t_2)$ is matched; at step three, no augmenting path of length three exists; at step four, the matching is augmented via path $[s_4,t_3,s_1,t_1]$; and (d) the symmetric difference $M_* \oplus M_3$. The bold lines denote edges matched in $M_*$, and wavy lines denote edges matched in $M_3$.

will consider the $S$ vertices in succeeding order of weights. While Algorithm $\text{GlobalOptimal}$ searches for a shortest augmenting path without any restrictions on the length of the path, the search in Algorithm $\text{GlobalTwoThird}$ is restricted to augmenting paths of at most three edges. However, at any step $k$, both the algorithms will consider the same vertex $s_k \in S$ for matching.

A failed vertex is a vertex that is matched in the optimal matching, but not in the approximation matching. For the current discussion, we will consider only the failed $S$ vertices, since the weights are associated only with them. The time step of execution is an important parameter for the proof. Therefore, to accommodate the time step, we will introduce a new notation. The failed vertex at step $k$ is represented as $s^{kk} \in S$. The other failed vertices in $S$ at this step are represented as $s^{ik}$, for $1 \leq i \leq k$. Our objective is to associate two unique $M_3$-matched $S$ vertices with each failed vertex $s^i$. We will use subscripts to represent such vertices, $s^{ik}_a$ and $s^{ik}_b$. The rationale to use two indices $(i, k)$ to represent the past and current steps is due to the
FIG. 31: *Intuition for proof of* $\frac{2}{3}$-approx algorithm *GLOBALTwoThird*. For each failed $S$ vertex, Algorithm *GLOBALTwoThird* will match two $S$ vertices that are at least as heavy as the failed vertex. Note that the association of matched vertices with failed vertices is *dynamic*. The figure is representative of a state at a particular step of execution.

The fact that the association of vertices could change during the execution. Let $M^k_s$ and $M^k_3$ represent the matchings at step $k$ as computed by Algorithms *GLOBALOptimal* and *GLOBALTwoThird*.

We will now proceed further. First, we will show that we need to process a vertex only once. This is stated in Lemma IV.4.1.

**Lemma IV.4.1.** Consider the execution of Algorithm *GLOBALTwoThird* on a restricted bipartite graph $G = (S, T, E)$ with weight function $w : S \rightarrow \mathbb{R}^+$. If at any step $k$, there exists no augmenting path of length $\leq 3$ starting at a vertex $s^k \in S$, then there will be no augmenting path of length $\leq 3$ from $s^k$ at a later stage of execution.

**Proof.** Consider the execution of Algorithm *GLOBALTwoThird* at the beginning of step $k$, let the $S$ vertex currently being processed be $s^k$. We will denote the $T$ vertices at a distance of one edge from $s^k$ as $t^i_1$, and those at a distance of three edges from $s^k$ as $t^i_3$. Since we are considering a bipartite graph, the $S$ vertices will be at an even distance from each other. Let the $S$ vertices at a distance of two edges from $s^k$ be denoted as $s^i_2$, and that at a distance of four edges be $s^i_4$.

Let us first consider the augmenting paths of length one edge. If there exists no augmenting path of length one from vertex $s^k$, then all the adjacent $T$ vertices, $t^i_1$, have already been matched. In order for a new augmenting path of length one to become available from $s^k$ at a later stage of execution, one of these $T$ vertices should
get unmatched. However, we know that during the execution of Algorithm \textsc{GlobalTwoThird}, once a vertex is matched it will always remain matched. Therefore, the lemma holds true for augmenting paths of length one.

Now we consider paths from the vertex \( s^k \). Since there is no augmenting path of length three from the vertex \( s^k \), these paths can be of two different kinds. The first kind of path has the form \([s^k, t_1^{i,k}, s_2^{i,k}, t_3^{i,k}, s_4^{i,k}, \ldots] \), where \( t_1^{i,k} \) is matched to \( s_2^{i,k} \), and \( t_3^{i,k} \) is matched to \( s_4^{i,k} \) and so on. The second kind of path has the form \([s^k, t_1^{i,k}, s_2^{j,k}, t_3^{j,k}] \), where the first two vertices are the same as the first two vertices from the path of the first kind, and the last two vertices, \( s_2^{j,k} \) and \( t_3^{j,k} \) are unmatched. These two kinds of paths are illustrated in Figure 32 as \( P_1 \) and \( P_2 \) respectively.

An augmenting path of length three beginning at \( s^k \) can exist at a later step because either (i) \( t_3^{i,k} \) becomes unmatched, or (ii) \( s_2^{i,k} \) becomes matched to \( t_1^{i,k} \). The first case cannot occur since a vertex once matched is always matched in a matching algorithm based on augmentations. In the second case, \( s_2^{j,k} \) becomes matched in a previous augmentation step (but after the \( k \)-th step) involving the augmenting path \([s_2^{j,k}, t_1^{i,k}, s_2^{i,k}, t_3^{i,k}] \), where the last vertex is an unmatched vertex. But such an augmenting path would imply an augmenting path at the \( k \)-th step from \( s^k \) consisting of \([s^k, t_1^{i,k}, s_2^{j,k}, t_3^{j,k}] \). This contradiction completes the proof.

FIG. 32: \textit{New augmenting paths.} Bold lines represent the matched edges and matched vertices are colored black. The two kinds of paths in Lemma IV.4.1 are shown as \( P_1 \) and \( P_2 \).

We will now argue for the correctness of the claimed approximation ratio of \( \frac{2}{3} \). Consider the concurrent execution of Algorithms \textsc{GlobalOptimal} and \textsc{GlobalTwoThird} on the first restricted bipartite graph with weights on the \( S \) vertices. We will consider the steps when a vertex \( s_k \in S \) gets matched by Algorithm \textsc{GlobalOptimal}, but not by Algorithm \textsc{GlobalTwoThird}. We define these vertices as the \textit{failed vertices}. An important relationship between the failed and the matched vertices is stated in Lemma IV.4.2.
Lemma IV.4.2. Consider the restricted bipartite graph \( G = (S, T, E) \) with weight function \( w : S \rightarrow \mathbb{R}^+ \). Let \( M^k_* \) represent the matching computed by Algorithm \textsc{GlobalOptimal} at the end of step \( k \), and let \( M^k_3 \) represent the matching computed by Algorithm \textsc{GlobalTwoThird} at the end of step \( k \). (i) For each failed vertex that exists at the end of step \( k \), \( s^{i,k} \), \( 1 \leq i \leq k \), there are two distinct vertices \( s^{i,k}_a \) and \( s^{i,k}_b \) that are matched in \( M^k_3 \). (ii) At the end of step \( k \), the following relation holds: \( s^{k,k} \prec s^{k,k}_a \) and \( s^{k,k} \prec s^{k,k}_b \).

Proof. We consider the proof of (i). Consider the symmetric difference \( M^k_* \oplus M^k_3 \). Each failed vertex \( s^{i,k} \) is matched in the first, but not in the second of these matchings, and hence begins an alternating path in the symmetric difference. This alternating path cannot have length two, of the form \( s^{i,k}, t^{i,k}_1, s^{i,k}_a \). If this is true, then only one of the vertices \( s^{i,k} \) and \( s^{i,k}_a \) can be matched to \( t^{i,k}_1 \). If \( s^{i,k} \prec s^{i,k}_a \), then the optimal algorithm made a wrong choice, and therefore, contradicts. If otherwise, then the approximate algorithm made a wrong choice and contradicts again.

If the alternating path is of length three, then it is an \( M^k_3 \)-augmenting path of length three and the approximation algorithm would have matched along this path. Therefore, the alternating path must be of length at least four. If the alternating path has even length (greater than or equal to four), then it ends with a terminal \( S \)-vertex that is matched in \( M^k_3 \) but not in \( M^k_* \). Hence this terminal vertex cannot be another failed \( S \)-vertex. If the alternating path is of odd length, then it terminates with a \( T \)-vertex. From these two cases, we conclude that every failed vertex begins a vertex-disjoint alternating path of length four or more, and has the form \( [s^{i,k}, t^{i,k}_1, s^{i,k}_a, t^{i,k}_3, s^{i,k}_b, \ldots] \).

Part (ii) follows from three observations:

1. Both the exact and approximation matching algorithms consider \( S \)-vertices in a succeeding order for matching;

2. \( s^{k,k} \) is the last failed vertex (which happens at step \( k \)); and

3. the vertices \( s^{k,k}_a \) and \( s^{k,k}_b \) have been matched in earlier steps.

\( \Box \)

We provided a conclusive relationship between the failed and the matched vertices in for a given step in Lemma IV.4.2. However, in order to provide an overall
approximation ratio of \(\frac{2}{3}\), we will induct on the failed steps. Theorem IV.4.1 provides this argument.

**Theorem IV.4.1** (Counting Technique). Consider a bipartite graph \(G = (S, T, E)\) with weight function \(w : S \rightarrow (R^+)^\), a matching \(M_s\) computed by Algorithm GlobalOptimal, and a matching \(M_3\) computed by Algorithm GlobalTwoThird. For every failed vertex \(s^i \in S\), there are two distinct vertices \(s^i_a\) and \(s^i_b\) that are matched by \(M_3\), such that \(s^i < s^i_a\) and \(s^i < s^i_b\).

**Proof.** The proof is based on induction. Consider the failed \(S\) vertices during the concurrent execution of Algorithms GlobalOptimal and GlobalTwoThird. We will reuse the notation from proof of Lemma IV.4.2.

**Base case:** Consider the step when the first failed vertex \(s^{1,1} \in S\) is encountered. We know from Lemma IV.4.2 that at the end of this step, there are two vertices \(s^{1,1}_a\) and \(s^{1,1}_b\), matched in \(M_3^1\), such that \(s^{1,1}_a < s^{1,1}_a\) and \(s^{1,1}_b < s^{1,1}_b\). Let \(s^{1}_a = s^{1,1}_a\) and \(s^{1}_b = s^{1,1}_b\).

**Step \(k\):** Assume true for the first \(k\) failures, \(1 \leq i \leq k\).

**Step \(k+1\):** At the end of the step when the \((k+1)\)-th failed vertex is encountered, from Lemma IV.4.2 we know that there are at least two distinct vertices matched in \(M_3^{k+1}\) such that \(s^{k+1,k+1}_a < s^{k+1,k+1}_a\) and \(s^{k+1,k+1}_b < s^{k+1,k+1}_b\).

A potential conflict arises when the \(M_3^{k+1}\)-matched vertices \(s^{k+1,k+1}_a\) and \(s^{k+1,k+1}_b\) had already been associated with a failed vertex \(s^i, i < k\), in a previous step. They are now being reused at step \((k+1)\). We will show how to address such a case.

From the inductive assumption at step \(k\), we know that for every failed vertex \(s^i, 1 \leq i \leq k\), there are two vertices \(s^i_a\) and \(s^i_b\), such that the relations \(s^i < s^i_a\) and \(s^i < s^i_b\) hold. Now consider two sets \(S_1 = \bigcup_{i=1}^{k+1} \{s^{i,k+1}_a, s^{i,k+1}_b\}\) and \(S_2 = \bigcup_{i=1}^{k} \{s^{i}_a, s^{i}_b\}\). The cardinalities are given by \(|S_1| \geq 2(j+1)|\) and \(|S_2| \geq 2(j)|\). This follows from Lemma IV.4.2. Thus, \(|S_1 \setminus S_2| \geq 2). Therefore, there are at least two distinct vertices in \(\{S_1 \setminus S_2\}\) that can be associated with \(s^{k+1}\) as \(s^{k+1}_a = s^{k+1,k+1}_a\) and \(s^{k+1}_b = s^{k+1,k+1}_b\). Since we know that \(s^{k+1}\) is the most current vertex processed, all the matched \(S\) vertices will be at least as large as this vertex. Thus the theorem holds. \(\square\)

From Theorem IV.4.1, the approximation follows immediately, and is stated in Corollary IV.4.1.

**Corollary IV.4.1.** Given a bipartite graph \(G = (S, T, E), w : S \rightarrow R^+, algorithm
GlobalTwoThird computes a $\frac{2}{3}$-approximation to maximum vertex-weight matching.

Proof. Let $M_*$ denote the optimal matching, and $M_3$ denote the matching computed by Algorithm GlobalTwoThird. We will consider the first restricted bipartite graph with weights associated only to $S$ vertices. Let $S(M)$ denote the $S$ vertices matched in $M$, and $N$ the number of failed vertices with respect to $M_3$. From Theorem IV.4.1, it immediately follows that for every failed vertex $s^i$ GlobalTwoThird matches at least two heavier vertices, $s^i_a$ and $s^i_b$. Therefore,

$$\sum_{i=1}^{N} w(s^i) \leq \frac{1}{2} \sum_{i=1}^{N} w(s^i_a) + w(s^i_b). \quad (10)$$

The weight of the optimal matching $M_*$ can be represented as

$$\sum_{s \in S(M_*)} w(s) = \sum_{s_i \in S(M_*) \setminus M_3} w(s_i) + \sum_{s_j \in S(M_* \cap M_3)} w(s_j). \quad (11)$$

We know that the set $S(M_* \setminus M_3)$ represents the set of failed vertices. We can rewrite the first term of right-hand-side in Equation 11 as

$$\sum_{s \in S(M_*)} w(s) \leq \sum_{i=1}^{N} w(s^i) + \sum_{s_j \in S(M_* \cap M_3)} w(s_j). \quad (12)$$

Using the results from Equation 10, we have

$$\sum_{s \in S(M_*)} w(s) \leq \frac{1}{2} \sum_{i=1}^{N} w(s^i_a) + w(s^i_b) + \sum_{s_j \in S(M_* \cap M_3)} w(s_j). \quad (13)$$

We can simplify the first term of R.H.S., in Equation 13 that results in

$$\sum_{s \in S(M_*)} w(s) \leq \frac{1}{2} \sum_{s_i \in S(M_3)} w(s_i) + \sum_{s_j \in S(M_* \cap M_3)} w(s_j). \quad (14)$$

The set $S(M_* \cap M_3)$ in the second term of R.H.S., can be simply replaced with a set $S(M_3)$. Therefore, we have

$$\sum_{s \in S(M_*)} w(s) \leq \frac{1}{2} \sum_{s_i \in S(M_3)} w(s_i) + \sum_{s_j \in S(M_3)} w(s_j). \quad (15)$$

Therefore,

$$\sum_{s \in S(M_*)} w(s) \leq \frac{3}{2} \sum_{s_i \in S(M_3)} w(s_i). \quad (16)$$
Rewriting the equation above, we have

$$\sum_{s \in S(M_3)} w(s) \geq \frac{2}{3} \sum_{s \in S(M_*)} w(s).$$

The time complexity for Algorithm \textsc{GlobalTwoThird} is stated in Theorem IV.4.2.

\textbf{Theorem IV.4.2.} Given a graph $G = (S, T, E)$ with weight functions $w_S : S \rightarrow \mathbb{R}^+$ and $w_T : T \rightarrow \mathbb{R}^+$, let $n = (|S| + |T|)$ represent the number of vertices and $m = |E|$ represent the number of edges. Algorithm \textsc{GlobalTwoThird} computes a matching $M_3$ in $G$ in $O(n \log n + nd_3)$, where $d_3$ is the vertex degree that denotes the average number of distinct alternating paths of length at most three edges starting at a vertex in $G$.

\textit{Proof.} Algorithm \textsc{GlobalTwoThird} processes the vertices in a global order. The given set of vertices can be sorted in decreasing order of vertex weights in time $O(n \log n)$. From each set of vertices $S$ and $T$, \textsc{GlobalTwoThird} will search for shortest augmenting paths of length at most three. In order to find augmenting paths of length one edge, we only need to process all the edges incident on the given vertex and is therefore bounded by $O(m)$. An augmenting path of length three edges is of the form $[s_1, t_1, s_2, t_2]$, where vertices $t_1$ and $s_2$ are matched by an edge $(t_1, s_2)$. In order to search augmenting paths of length up to three edges, Algorithm \textsc{GlobalTwoThird} will incur a cost of at most $(\text{deg}(s_1), \text{deg}(s_2))$. Due to the matched edge, vertex $s_2$ can be directly reached from vertex $t_1$. Let us represent this search operation as $d_3$, where $d_3$ is the vertex degree that denotes the average number of distinct paths of length at most three edges starting at a vertex. Thus, the run time of Algorithm \textsc{GlobalTwoThird} can be bounded by $O(n \log n + nd_3)$. The two matchings, $M_S$ and $M_T$ can be merged using the Mendelsohn-Dulmage technique in linear time $O(m)$. \hfill \Box

We will now proceed to the describe a potential local-approach to compute a $\frac{2}{3}$-approx VWM in a bipartite graph.
IV.5 POTENTIAL LOCAL $\frac{2}{3}$-APPROX ALGORITHM

For the second potential algorithm for computing a $\frac{2}{3}$-approx matching we adopt a strategy based on restricting the search to a limited length of augmenting path from a given vertex. Therefore, we name it as LOCALTwoThird$^1$. The vertices are chosen for matching based on a local order. We first decompose the given bipartite graph $G = (S, T, E)$, with weights associated with both $S$ and $T$ vertices, into two restricted bipartite graphs by ignoring the weights on the $S$ vertices and then on the $T$ vertices. This is represented in LOCALTwoThird by Lines 5 and 14.

Algorithm 19  

**Input:** a bipartite graph $G$.  
**Output:** a matching $M$.  
**Effect:** computes a $\frac{2}{3}$-approx to maximum vertex-weight matching.

```
1: procedure LOCALTwoThird($G = (S, T, E), w_S : S \rightarrow \mathbb{R}^+, w_T : T \rightarrow \mathbb{R}^+, M$)
2:    $M \leftarrow \phi$;
3:    $M_S \leftarrow \phi$;
4:    $M_T \leftarrow \phi$;
5:    $\tilde{S} \leftarrow S$ with weights $w_S$ set to zero ;
6:    while $\tilde{S} \neq \phi$ do
7:      $s \leftarrow$ top of $\tilde{S}$;
8:      $\tilde{S} \leftarrow \tilde{S} \setminus s$;
9:      Find all augmenting paths $P_{s \rightarrow t} = (P_1, P_2, ..)$ of length $\leq 3$ starting at $s$;
10:     if $P$ found then
11:        $M_S \leftarrow M_S \oplus P_{\text{best}}$; ▷ $P_{\text{best}}$ is the path with largest $t$ that will be matched
12:     end if
13:  end while
14:  $\tilde{T} \leftarrow T$ with weights $w_T$ set to zero ;
15:  while $\tilde{T} \neq \phi$ do
16:    $t \leftarrow$ top of $\tilde{T}$;
17:    $\tilde{T} \leftarrow \tilde{T} \setminus t$;
18:    Find all augmenting paths $P_{t \leftarrow s} = (P_1, P_2, ..)$ of length $\leq 3$ starting at $t$;
19:    if $P$ found then
20:      $M_T \leftarrow M_T \oplus P_{\text{best}}$; ▷ $P_{\text{best}}$ is the path with largest $s$ that will be matched
21:    end if
22:  end while
23:  $M \leftarrow$ Mendelsohn-Dulmage($M_S, M_T, M$); ▷ Compute $M$
24: end procedure
```

In the first matching subproblem a matching $M_S$ is computed as follows: arbitrarily start from an unmatched $S$ vertex $s_i$ and enumerate all alternating paths $P_t$,

$^1$The proof of correctness for Algorithm LOCALTwoThird has not been completed.
of length at most three, with respect to the current matching $M_i$. Pick the best augmenting path from $s_i$ and augment the current matching. A best augmenting path is a path that maximizes $M_i \oplus P_i$. Repeat the process until all the $S$ vertices have been processed. Lines 6 – 13 represent the computation of $M_S$. Similarly a matching $M_T$ can be computed when weights are associated only with the $S$ vertices. This is the second subproblem and is represented by Lines 15 – 22 in LOCALTWOSECOND. The final matching will be obtained by merging the two matchings $M_S$ and $M_T$ using the Mendelsohn-Dulmage technique. Execution of LOCALTWOSECOND on a simple bipartite graph with weights associated with $S$ vertices is shown in Figure 33.

FIG. 33: Execution of Algorithm LOCALTWOSECOND. (a) The input graph $G = (S, T, E)$ before the execution, weights are associated only with $S$ vertices, (b)-(d) the intermediate states of execution, and (e) the final state. Bold lines represent matched edges, and matched vertices are colored black. The shaded edges highlight all the augmenting paths that exist from a given $T$ vertex.

IV.5.1 Correctness of Algorithm LocalTwoThird

We have not been successful to prove the correctness of LOCALTWOSECOND. A critical part of the proof for GLOBALTWOSECOND was Theorem IV.4.1, where for the induction step $(k + 1)$ we could safely state that all the matched vertices at that step were heavier than the $(k + 1)$-th failed vertex. We could state such a fact because the vertices were considered in a decreasing order of weight. However, we will not
be able to state the same for a matching computed by \texttt{LOCALTWOThird} where vertices are processed in an arbitrary order. We will therefore leave the proof of \texttt{LOCALTWOThird} as an open problem.
IV.6 EXPERIMENTAL RESULTS

In this section we present experimental results from our implementation of matching algorithms in a toolkit called MatchBox. The two types of experiments done are serial and parallel. The goals for serial experiments are to demonstrate the efficiency of approximation algorithms in terms of execution time, cardinality and weight of matching as compared to those of the exact algorithms. The experiments are conducted on a system equipped with four 2.4 GHz Intel quad core processors and 32 GB RAM at Old Dominion University.

The graphs used for experiments are representations of regular sparse matrices downloaded from the University of Florida Sparse Matrix Collection. A matrix is stored as a bipartite graph, where rows and columns of the matrix represent vertices, and the nonzero elements represent edges. The absolute value of a nonzero element in the matrix is considered as the weight of the edge that connects the vertices representing the row and the column of the nonzero element. In the following experiments, the degrees of vertices are used as the weights of the vertices. A similar model is used to represent symmetric matrices. Since the files downloaded from the University of Florida Sparse Matrix Collection store only the lower triangle of the matrix, we explicitly add edges to represent both the upper and lower triangles of the matrix. The matrices used in the experiments are listed in Table 8.

The performance of global algorithms is presented in Table 9. It can be noted that the $\frac{1}{2}$-approx algorithms are very fast and the $\frac{1}{2}$-approx algorithms are relatively fast. As mentioned earlier, the degree of the vertices are used as weights for both the $S$ and $T$ vertex sets for a given graph. The two matchings, $M_S$ and $M_T$ are computed separately and the final matching is obtained by merging the two matchings using the Mendelsohn-Dulmage technique.

Comparision between the Global and Local-based algorithms is presented in Table 10. For the $\frac{1}{2}$-approx algorithms, it can be noted that Algorithm LOCALHALF is faster than the Algorithm GLOBALHALF in many cases, except for largest graph in the collection. However, for $\frac{2}{3}$-approx algorithms, the Global-based algorithm almost always beats the Local-based algorithm. Note that we did not prove the correctness of Algorithm LOCALTWOSECOND, and the data is provided here for comparison. The Local-based algorithms are forced to enumerate all possible paths of certain length and are therefore inefficient. For the same reason, we do not provide results for Algorithm LOCALOPTIMAL, which we believe is not a practical algorithm.
<table>
<thead>
<tr>
<th>Name</th>
<th>#S-Vertices</th>
<th>#T-Vertices</th>
<th>#Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>nemsemm</td>
<td>3,945</td>
<td>75,352</td>
<td>1,053,986</td>
</tr>
<tr>
<td>dbic1</td>
<td>43,200</td>
<td>226,317</td>
<td>1,081,843</td>
</tr>
<tr>
<td>pds-100</td>
<td>156,243</td>
<td>514,577</td>
<td>1,096,002</td>
</tr>
<tr>
<td>dbir2</td>
<td>18,906</td>
<td>45,877</td>
<td>1,158,159</td>
</tr>
<tr>
<td>lp-os60</td>
<td>10,280</td>
<td>243,246</td>
<td>1,408,073</td>
</tr>
<tr>
<td>lp-nug</td>
<td>52,260</td>
<td>379,350</td>
<td>1,567,800</td>
</tr>
<tr>
<td>karted</td>
<td>46,502</td>
<td>133,115</td>
<td>1,770,349</td>
</tr>
<tr>
<td>watson-2</td>
<td>352,013</td>
<td>677,224</td>
<td>1,846,391</td>
</tr>
<tr>
<td>stat96v2</td>
<td>29,089</td>
<td>957,432</td>
<td>2,852,184</td>
</tr>
<tr>
<td>stat96v3</td>
<td>33,841</td>
<td>1,113,780</td>
<td>3,317,736</td>
</tr>
<tr>
<td>stormG2</td>
<td>528,185</td>
<td>1,377,306</td>
<td>3,459,881</td>
</tr>
<tr>
<td>cont11</td>
<td>1,468,599</td>
<td>1,961,394</td>
<td>5,382,999</td>
</tr>
<tr>
<td>rail2586</td>
<td>2,586</td>
<td>923,269</td>
<td>8,011,362</td>
</tr>
<tr>
<td>degme</td>
<td>185,501</td>
<td>659,415</td>
<td>8,127,528</td>
</tr>
<tr>
<td>rail4284</td>
<td>4,284</td>
<td>1,096,894</td>
<td>11,284,032</td>
</tr>
<tr>
<td>tp-6</td>
<td>142,752</td>
<td>1,014,301</td>
<td>11,537,419</td>
</tr>
<tr>
<td>spal-004</td>
<td>10,203</td>
<td>321,696</td>
<td>46,168,124</td>
</tr>
</tbody>
</table>

**TABLE 8:** Matrix Instances. Downloaded from University of Florida Matrix Collection.

<table>
<thead>
<tr>
<th>Name</th>
<th>Exact</th>
<th>$\frac{1}{2}$-approx</th>
<th>$\frac{2}{3}$-approx</th>
</tr>
</thead>
<tbody>
<tr>
<td>nemsemm1</td>
<td>0.05</td>
<td>0.02</td>
<td>0.07</td>
</tr>
<tr>
<td>dbic1</td>
<td>0.04</td>
<td>0.04</td>
<td>0.05</td>
</tr>
<tr>
<td>pds-100</td>
<td>0.82</td>
<td>0.12</td>
<td>0.29</td>
</tr>
<tr>
<td>dbir2</td>
<td>0.07</td>
<td>0.01</td>
<td>0.09</td>
</tr>
<tr>
<td>lp-os60</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>lp-nug</td>
<td>25.48</td>
<td>0.03</td>
<td>0.26</td>
</tr>
<tr>
<td>karted</td>
<td>3.31</td>
<td>0.04</td>
<td>0.18</td>
</tr>
<tr>
<td>watson-2</td>
<td>0.62</td>
<td>0.26</td>
<td>0.66</td>
</tr>
<tr>
<td>stat96v2</td>
<td>0.33</td>
<td>0.12</td>
<td>0.4</td>
</tr>
<tr>
<td>stat96v3</td>
<td>0.37</td>
<td>0.15</td>
<td>0.45</td>
</tr>
<tr>
<td>stormG2</td>
<td>1.34</td>
<td>0.64</td>
<td>1.44</td>
</tr>
<tr>
<td>cont11</td>
<td>24.52</td>
<td>0.83</td>
<td>1.5</td>
</tr>
<tr>
<td>rail2586</td>
<td>0.04</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>degme</td>
<td>10.17</td>
<td>0.27</td>
<td>0.92</td>
</tr>
<tr>
<td>rail4284</td>
<td>0.06</td>
<td>0.06</td>
<td>0.07</td>
</tr>
<tr>
<td>tp-6</td>
<td>6.49</td>
<td>0.3</td>
<td>1.39</td>
</tr>
<tr>
<td>spal-004</td>
<td>733.36</td>
<td>0.14</td>
<td>26.69</td>
</tr>
</tbody>
</table>

**TABLE 9:** Performance of Global-based Algorithms. The numbers represent compute time in seconds.
<table>
<thead>
<tr>
<th>Name</th>
<th>$\text{GLOBAL}_\frac{1}{2}$</th>
<th>$\text{LOCAL}_\frac{1}{2}$</th>
<th>$\text{GLOBAL}_\frac{2}{3}$</th>
<th>$\text{LOCAL}_\frac{2}{3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>nemsemml</td>
<td>0.02</td>
<td>0.01</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>dbic1</td>
<td>0.04</td>
<td>0.01</td>
<td>0.05</td>
<td>0.16</td>
</tr>
<tr>
<td>pds-100</td>
<td>0.12</td>
<td>0.03</td>
<td>0.29</td>
<td>0.28</td>
</tr>
<tr>
<td>dbir2</td>
<td>0.01</td>
<td>0.01</td>
<td>0.09</td>
<td>0.41</td>
</tr>
<tr>
<td>lp-os-a-60</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.1</td>
</tr>
<tr>
<td>lp-nug30</td>
<td>0.03</td>
<td>0.02</td>
<td>0.26</td>
<td>1.32</td>
</tr>
<tr>
<td>karterd</td>
<td>0.04</td>
<td>0.02</td>
<td>0.18</td>
<td>3.03</td>
</tr>
<tr>
<td>watson-2</td>
<td>0.26</td>
<td>0.03</td>
<td>0.66</td>
<td>0.69</td>
</tr>
<tr>
<td>stat96v2</td>
<td>0.12</td>
<td>0.02</td>
<td>0.4</td>
<td>0.91</td>
</tr>
<tr>
<td>stat96v3</td>
<td>0.15</td>
<td>0.03</td>
<td>0.45</td>
<td>1.08</td>
</tr>
<tr>
<td>stormG2-1000</td>
<td>0.64</td>
<td>0.06</td>
<td>1.44</td>
<td>23.1</td>
</tr>
<tr>
<td>cont11-l</td>
<td>0.83</td>
<td>0.07</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>rail2586</td>
<td>0.05</td>
<td>0.07</td>
<td>0.05</td>
<td>0.76</td>
</tr>
<tr>
<td>degme</td>
<td>0.27</td>
<td>0.09</td>
<td>0.92</td>
<td>7.94</td>
</tr>
<tr>
<td>rail4284</td>
<td>0.06</td>
<td>0.11</td>
<td>0.07</td>
<td>1.14</td>
</tr>
<tr>
<td>tp-6</td>
<td>0.3</td>
<td>0.12</td>
<td>1.39</td>
<td>9.81</td>
</tr>
<tr>
<td>spal-004</td>
<td>0.14</td>
<td>0.29</td>
<td>26.69</td>
<td>1495.95</td>
</tr>
</tbody>
</table>

TABLE 10: Relative Performance of Global and Local-based Algorithms. The numbers represent compute time in seconds.

The quality of a matching can be measured in terms of the cardinality (the number of edges in the matching) and the weight (sum of weights of the matched edges) of the matching. We present the cardinality of the matchings computed by the different algorithms in Figure 34, and the weight of the matchings in Figure 35. The exact algorithm used in these comparison is Algorithm $\text{GLOBALOPTIMAL}$. It can be noted that the approximation algorithms compute matchings of high quality in terms of both cardinality and weight of the matchings.
FIG. 34: *Performance of Approximation Algorithms*. Cardinality of matchings of the approximation algorithms as a ratio of the cardinality of the exact algorithm.

FIG. 35: *Performance of Approximation Algorithms*. Weight of matchings of the approximation algorithms as a ratio of the weight of the exact algorithm.
IV.7 CHAPTER SUMMARY

In this chapter we introduced three new \(\frac{1}{2}\)-approx algorithms and two new \(\frac{2}{3}\)-approx algorithms to MVM problem. Proof of correctness for all the proposed \(\frac{1}{2}\)-approx and one of the \(\frac{2}{3}\)-approx algorithms were also discussed. We introduced the concept of the \textit{restricted reachability} property to provide the correctness of \(\frac{1}{2}\)-approx Algorithms \textsc{GlobalHalf}, \textsc{LocalHalf} and \textsc{HybridHalf}. We also introduced the concept of a \textit{counting technique} in order to provide the correctness of \(\frac{2}{3}\)-approx Algorithm \textsc{GlobalTwoThird}. While we did not succeed to prove the correctness of \textsc{LocalTwoThird} algorithm, this approach, if proved correct, will also provide us an algorithm to compute \(\frac{2}{3}\)-approx matchings in general graphs. We concluded the chapter by providing experimental results highlighting the effectiveness of the approximation algorithms, both in execution time and the quality of the matchings.

There are a few limitations to our current approach. The proposed techniques, global and local, fail to generalize for a \(\left(\frac{k}{k+1}\right)\)-approx, for \(k > 3\). As illustrated in Figure 36, an augmenting path of length five starting at a vertex \(s_1 \in S\) could appear at a later stage, while none existed when \(s_1\) was processed the first time. Therefore with the current approaches, we cannot guarantee an approximation ratio better than \(\left(\frac{4}{5}\right)\). However, we cannot say anything conclusively about a \(\frac{3}{4}\)-approx ratio for the proposed algorithms, and will study this in the follow-up work.

**FIG. 36:** New augmenting paths. (a) No augmenting path of length less than or equal to five exist starting at vertex \(s_1\) in graph \(G\) at step \(k\); (b) an augmenting path of length five is available from \(s_1\) at a step after \(k\).
Similar to the exact algorithms discussed in Chapter III, the proposed approximation algorithms also suffer from the same limitations: (i) absence of greedy initializations, and (ii) inability to grow multiple paths, both for $\frac{1}{2}$ and $\frac{2}{3}$-approx algorithms.
CHAPTER V

PARALLEL APPROXIMATE ALGORITHMS

“While petascale architectures certainly will be held as magnificent feats of engineering skill, the community anticipates an even harder challenge in scaling up algorithms and applications for these leadership-class supercomputing systems.” - David Bader [6]

V.1 INTRODUCTION

Parallelizing the augmentation-based algorithms for matching is nontrivial. While parallelizing the exact algorithms is hard, the approximation algorithms also pose a challenge. For example, consider a simple algorithm for computing half approximation to the maximum weighted matching problem. The algorithm proceeds by first sorting the edges based on their weights, and then matching them in a decreasing order of these weights. The edges are processed in a certain order, and therefore, the algorithm is serial in nature. In this chapter we will provide a parallel \( \frac{1}{2} \)-approx algorithm for edge-weighted matching, due to Hoepman [36], and Manne and Bisseling [50]. We will discuss implementation details and experimental analysis of this algorithm. Our contributions include a detailed description, efficient implementation for distributed memory architectures, and a thorough experimental analysis of the algorithm.

Existing literature on distributed algorithms for matching is predominantly based on the PRAM (Parallel Random Access Machine [44]) model. We refer the reader to a monograph on parallel algorithms for matching for a detailed discussion on the subject [39]. Some of the recent work has focussed on alternative models such as BSP (Bulk Synchronous Parallel [15, 17]) and CGM (Coarse Grained Multicomputer [20]), for example, [16, 50]. These approaches are different from the fine-grain approaches in the PRAM model [70], and are more suited for modern architectures with a cluster of computers with fast interconnects. Approximation algorithms have also been proposed [27, 38, 47, 68, 72]. Auction-based algorithms for computing matchings in bipartite graphs have been parallelized [7, 12, 13, 14, 18, 61, 75]. Parallel approximate algorithms have also been proposed in the context of application in high-speed network switches [30, 51, 56].
In the following discussions we will assume data structures for graph representations that store vertex-adjacency sets, and that the graphs are distributed among the processors via vertex partitioning. We will start the discussion by presenting a modified version of Preis's algorithm [64] that builds an intuition for the parallel algorithm. A distributed scheme of Preis's algorithm was developed by Hoepman. Manne and Bisseling show that this is a variant of Luby's parallel algorithm for computing maximal independent sets in a graph [49].

We introduced Preis's algorithm, Algorithm LAM, in Chapter II and refer the reader to [64] for details. The algorithm computes a half-approx matching by finding locally dominant edges and adding them to the set of matched edges. However, the search for locally dominant edges involves traversing through the graph. Thus, this algorithm is sequential in nature. Alternatively, the same matching can be computed by using a pointer-based technique that was proposed by Manne [50]. The pointer-based technique works as follows. Let each vertex set a pointer to the vertex that is the end point of a heaviest edge incident on it. If two vertices point to each other, then the edge connecting them is a locally dominant edge. Therefore, add this edge to the set of matched edges. Remove all edges that are incident on the matched vertices. Reset the pointers of those vertices that are affected by the changes and match the dominating edges. Repeat the process until all edges have been removed.

A basic step in the pointer-based algorithm is to set a pointer for a given vertex. A simple way of doing this is to traverse through the adjacency set $S(v)$ of a given vertex that contains unmatched neighboring vertices, find a heaviest neighbor and set the pointer to point this vertex. This is described in Algorithm 20. In case of ties, the indices of vertices are used to break ties (Line 5). The lowest numbered heaviest end-point of the edge incident on vertex $v$ is chosen. Note that since each vertex sets its pointer independent of other vertices, breaking the ties in a consistent manner is an important task. In the absence of a deterministic scheme to break ties, the algorithm may not function correctly when cycles of equal edge-weights exit. Also, note that the running time for this procedure can be improved by maintaining a sorted list of adjacent vertices so that the current heaviest vertex can be determined in constant time.

Once the pointer for a vertex $v$, represented by $\text{candidateMate}(v)$, has been set, the next step is to check if the vertex being pointed to by $v$ also points back to $v$. If so, we have successfully identified a locally dominant edge, and this edge can be
Algorithm 20 Compute Candidate Mate. Input: A vertex \( v \) and its adjacency set. Output: The end point, vertex, of the heaviest edge incident on \( v \). Associated data structures: A set \( S(v) \) of unmatched vertices adjacent to vertex \( v \). Effect: Find a candidate-mate for the given vertex \( v \).

1: \textbf{procedure} \textsc{ComputeCandidateMate}(v)  
2: \hspace{1em} w \leftarrow 0;  
3: \hspace{1em} \text{maxWt} \leftarrow -\infty;  
4: \hspace{1em} \textbf{for} \ z \in S(v) \ \textbf{do} \ \triangleright \text{The weight of an edge } (x, y) \text{ is denoted by } w(e_{xy}).  
5: \hspace{2em} \textbf{if} \ (\text{maxWt} < w(e_{vz})) \ \textbf{or} \ (\text{maxWt} = w(e_{vz}) \ \textbf{and} \ w < z) \ \textbf{then}  
6: \hspace{3em} w \leftarrow z;  
7: \hspace{3em} \text{maxWt} \leftarrow w(e_{vz});  
8: \hspace{2em} \textbf{end if}  
9: \hspace{1em} \textbf{end for}  
10: \hspace{1em} \textbf{return} \ w;  
11: \textbf{end procedure}

added to the set of matched edges. This process is shown in Algorithm 21. Once an edge is matched, all the edges incident on the matched vertices are removed. This is done by modifying the adjacency sets, \( S(v) \), of vertices as shown in Line 6 of the algorithm. However, only those vertices that are pointing to the matched vertices need to reset their pointers. Therefore, the matched vertices are added to set \( Q_M \), a set of matched vertices, for further processing (Line 8).

The complete pointer-based algorithm is shown in Algorithm 22. It can be observed that Algorithm 22 can be divided into three distinct phases: (i) initialization, (ii) processing all vertices independent of current matching, and (iii) processing specific vertices based on the current matched edge(s). Initialization of data structures is shown in Lines 2 through 7. The pointer for each vertex is set (Line 9) by a call to the function \textsc{ProcessExposedVertex}, which also tests if a dominant edge has been found that could be matched. Processing all the exposed vertices will result in at least one edge (the heaviest edge) being matched. All the matched vertices from this phase are added to the set \( Q_M \). Only those vertices that were pointing to the matched vertices will be processed. This is done in the \textbf{while} loop over set \( Q_M \) (Line 11 through Line 21). If a vertex is pointing to a matched vertex then the pointer for this vertex needs to be reset. This is done by a call to the function \textsc{ProcessExposedVertex}. The loop exits when all the matched vertices are processed. At this stage, no other edges can be matched, and therefore, the algorithm terminates. We
Algorithm 21 Process Exposed Vertex. Input: A vertex \( v \) and its adjacency set. Associated data structures: A set \( S(v) \) of unmatched vertices adjacent to vertex \( v \), a set \( Q_M \) of matched vertices, a vector \( \text{candidateMate} \) of pointers, and set \( M \) of matched edges. Effect: Processes an exposed vertex - find candidate-mate and match if possible.

1: procedure \( \text{PROCESSEXPOSEDVERTEX} (v) \)
2: \( \text{candidateMate}(v) \leftarrow \text{COMPUTECANDIDATEMATE}(v); \)
3: \( c \leftarrow \text{candidateMate}(v); \)
4: if \( c \neq 0 \) and \( \text{candidateMate}(c) = v \) then
5: \( M \leftarrow M \cup \{(v, c)\}; \)
6: \( S(v) \leftarrow S(v) \setminus \{c\}; \)
7: \( S(c) \leftarrow S(c) \setminus \{v\}; \)
8: \( Q_M \leftarrow Q_M \cup \{v, c\}; \)
9: end if
10: end procedure

will follow a similar three phase distinction to simplify the description and analysis of the parallel approximation algorithm. We refer the reader to [50] for a proof of correctness of the pointer-based algorithm.

Execution of Algorithm 22 on a simple graph is shown in Figure 37. It can be observed that in Step (c) of the figure, two edges, (1, 3) and (2, 5), concurrently become eligible for matching. This provides an intuition for the potential of parallelism in the pointer-based approach for computing approximate matchings.

V.1.1 Complexity Analysis

We will use the following notation for this analysis. Let the degree of a vertex \( v \) be denoted by \( d(v) \), which represents the number of edges incident on a vertex \( v \). The maximum degree of a graph \( G \) is represented by \( \Delta(G) \), or simply \( \Delta \), which is the maximum number of edges incident on any vertex in \( G \).

The complexity of Algorithm 22 is essentially determined by the complexity of finding a candidate-mate described in Algorithm 20, and the number of times this function will be called for a vertex. In a simple implementation, a linear search is performed to find the heaviest edge incident on a vertex \( v \), and therefore, the compute time is given by \( O(d(v)) \). The procedure to find a candidate-mate of a vertex \( v \) can be invoked at most the number of edges incident on \( v \). The total time can be obtained by the summation of work done for each vertex, \( O(\sum_{v \in V} d(v)^2) \).
Algorithm 22 Pointer-based Matching Algorithm.  **Input:** A graph $G(V,E)$ with weights associated with the edges.  **Output:** A $\frac{1}{2}$-approx matching $M$ in $G$.  **Associated data structures:** A set $S(v)$ of unmatched vertices adjacent to vertex $v$, a set $Q_M$ of matched vertices, a vector $\text{candidateMate}$ of pointers, and a set $M$ of matched edges.

1: procedure $\text{PointerBasedMatching}(G = (V,E), M)$
2:     for $v \in V$ do
3:         $\text{candidateMate}(v) \leftarrow 0$;
4:         $S(v) \leftarrow \text{adj}(v)$;
5:     end for
6:     $M \leftarrow \emptyset$;
7:     $Q_M \leftarrow \emptyset$;
8:     for $v \in V$ do
9:         $\text{PROCESSEXPOSEDVERTEX}(v)$;
10:     end for
11: while $Q_M \neq \emptyset$ do
12:     $u \leftarrow \text{pick from } Q_M$;
13:     $Q_M \leftarrow Q_M \setminus \{u\}$;
14:     for $v \in S(u)$ do
15:         $S(v) \leftarrow S(v) \setminus \{u\}$;
16:         if $\text{candidateMate}(v) = u$ then
17:             $\text{PROCESSEXPOSEDVERTEX}(v)$;
18:         end if
19:     end for
20:     $S(u) \leftarrow \emptyset$;
21: end while
22: return $M$;
23: end procedure
FIG. 37: Execution of Algorithm 22. (a) The input graph $G = (V, E)$ with weights associated with the edges; (b) an intermediate step of execution where the pointers are set for each vertex in the graph; (c) an intermediate step where vertices that are pointing to each other are matched. Bold lines represent matched edges. Dashed lines represent the edges removed from the graph; (d) reset pointers for vertices 4 and 6; (e) edge (4, 5) is matched; (d) the final state. Matched vertices are colored black.

Since $|E| = \sum_{v \in V} d(v)$, complexity can be expressed as $O(|E|\Delta)$.

The running time of Algorithm 22 can be improved by maintaining the adjacency set of each vertex in a decreasing order of weights. The status of a vertex, matched or not, can also maintained in constant time. With a sorted adjacency set, candidate-mate of a vertex can be computed in constant time. However, building the sorted adjacency set for a vertex $v$ will cost $O(d(v) \log d(v))$. The total time can be obtained by the summation of work done for each vertex, $O(\sum_{v \in V} d(v) \log d(v))$, which can be expressed as

$$O(|E| \log \Delta).$$

Note that if a vertex $v$ is matched, only those vertices that are at a distance two from $v$ and pointing to it need to reset their pointers. Therefore, a tighter bound can be expressed by $O(|V|\bar{d}_2)$, where $\bar{d}_k(G)$ is a generalization of the vertex degree that denotes the average number of distinct paths of length at most $k$ edges starting at a vertex in graph $G$. For example, consider the graph in Figure 38. It can be observed that there are eight distinct paths from vertex 9, for example, paths $\{(9, 1)\}$,
{(9, 1), (1, 5)}, etc. Therefore, \( \overline{d}(9) = 8 \). It can also be observed that for the internal vertices (1, 2, 3, 4), \( \overline{d}(v) = 5 \); and for external vertices (5, 6, 7, 8), \( \overline{d}(v) = 2 \). Thus, \( \overline{d}(G) = (8 + 20 + 8)/9 = 4 \).

![Diagram of a graph with labeled vertices and edges](image)

FIG. 38: *Complexity analysis.* A sample graph \( G \) with weights associated with the edges such that \((w(e_1) > w(e_2) > \cdots > w(e_8))\).

When the edge-weights are distributed uniformly randomly, the probability for any edge being removed from the adjacency set of a vertex is uniform. With this assumption, Manne and Bisseling show that the expected time can be bounded by \( O(|E|) \). We refer the reader to [50] for details.
V.2 DISTRIBUTED ALGORITHM OF HOEPMAN

The intuition for parallelism using pointers was provided in the previous section. In this section we will discuss how this scheme can be implemented in a distributed manner. Hoepman [36] provides a distributed algorithm that assigns one vertex per processor. A processor is capable of computing as well as communicating with other processors, and has independent memory that is not accessible by other other processors. Hoepman’s algorithm provides the necessary understanding for the parallel algorithm that we have implemented where each processor is assigned a set of vertices for processing.

Hoepman’s algorithm is described in Algorithm 23. The algorithm starts by assigning each processor a unique vertex and its adjacency set. In order to simplify, we will assign the same index to both the processor and the vertex. Thus, from the adjacency set, each processor will also know the identities of its neighboring processors. Each processor will maintain a set $S$ that is initialized with the adjacency set of the vertex it owns. Every time a processor receives a message from its neighbors, it removes the identity of that neighbor from set $S$. Similarly, each processor also maintains a set $Q_R$ to store the requests received from its neighbors. We will reuse Algorithm 20, introduced in Section 1, to compute the candidate-mate of a vertex.

The algorithm loops until set $S$ becomes empty (Lines 9 through 28). There are two possibilities for this to happen: (i) either a processor receives a message from all its neighbors, or (ii) it finds a mate. We will use two types of messages - REQUEST and UNAVAILABLE. A REQUEST message is sent when a processor wants to match with one of its neighboring processors. When a REQUEST message is matched with a corresponding REQUEST message, it means that a locally dominant edge has been identified (similar to two vertices pointing to each other). This will result in an edge being matched. An UNAVAILABLE message is sent when a processor successfully matches its vertex and is not interested in the matching process anymore (Lines 23 through 25). Execution of Algorithm 23 on a simple graph with three vertices is shown in Figure 39.
Algorithm 23 Distributed Algorithm of Hoepman. **Input:** A graph $G(V,E)$ with weights associated with the edges. **Output:** A $\frac{1}{2}$-approx matching $M$. **Data distribution:** Processor $P_i$ owns vertex $v_i$ and stores edges incident on $v_i$, $\text{adj}(v_i)$. **Associated data structures:** A set $S$ of processor identities that share an edge with $P_i$, a set $Q_R$ of requests received on $P_i$, a scalar $c$ that identifies the mate.

1: procedure DistributedMatchingAlgorithm($G = (V,E), M$) 
2: \hspace{1em} loop on each processor $P_i, i \in I = \{1, \ldots, |V|\}$ \triangleright One vertex per processor. 
3: \hspace{2em} $S \leftarrow \text{adj}(v_i)$; 
4: \hspace{2em} $Q_R \leftarrow \emptyset$; 
5: \hspace{2em} $c \leftarrow \text{ComputeCandidateMate}(v_i)$; 
6: \hspace{2em} if $c \neq \text{null}$ then 
7: \hspace{3em} send REQUEST to $c$; 
8: \hspace{2em} end if 
9: \hspace{2em} while $S \neq \emptyset$ do 
10: \hspace{3em} receive message from $u \in S$; 
11: \hspace{3em} if message = REQUEST then 
12: \hspace{4em} $Q_R \leftarrow Q_R \cup \{u\}$; 
13: \hspace{3em} else if message = UNAVAILABLE then 
14: \hspace{4em} $S \leftarrow S \setminus \{u\}$; \triangleright Processor $P_u$ has found a mate elsewhere 
15: \hspace{4em} if $c = u$ then 
16: \hspace{5em} $c \leftarrow \text{ComputeCandidateMate}(v_i)$; \triangleright Reset the pointer. 
17: \hspace{4em} if $c \neq 0$ then 
18: \hspace{5em} send REQUEST to $c$; 
19: \hspace{4em} end if 
20: \hspace{4em} end if 
21: \hspace{3em} end if 
22: \hspace{3em} if $c \neq \text{null}$ and $c \in Q_R$ then 
23: \hspace{4em} for all $w \in S \setminus \{c\}$ do 
24: \hspace{5em} send UNAVAILABLE to $w$; 
25: \hspace{4em} end for 
26: \hspace{3em} $S \leftarrow \emptyset$; 
27: \hspace{3em} end if 
28: \hspace{2em} end while 
29: \hspace{2em} return $c$; 
30: \hspace{1em} end loop 
31: \hspace{1em} Compute $M$ based on the $c$ values received from all processors; 
32: \hspace{1em} return $M$; 
33: end procedure
FIG. 39: Execution of Hoepman’s Algorithm. (a) The input graph $G = (V,E)$ with weights associated with the edges, vertices $\{1,2,3\}$ are assigned to processors $\{P_1, P_2, P_3\}$ respectively; (b) an intermediate step of execution when REQUEST messages are sent by each processor to their neighbors of choice; (c) an intermediate step when edge $(2,3)$ is matched. (d) A possible intermediate step when processors $P_2$ and $P_3$ send UNAVAILABLE messages to $P_1$ in that order, (d’) an alternative situation when $P_1$ gets an UNAVAILABLE message from $P_3$, and sends a REQUEST to $P_2$. Eventually, $P_1$ will also receive an UNAVAILABLE message from $P_2$. (e) The final state. Matched vertices are colored black.
V.2.1 Complexity Analysis

The number of messages a processor $P_i$ sends is $\Theta(d(v_i))$, where $d(v_i)$ is the degree of a vertex $v_i$ that $P_i$ owns. Let us define one time step as the time it takes for a processor to compute a candidate-mate and send a REQUEST message to the processor that owns the candidate-mate. If each processor can independently perform this task, then the computational time of Hoepman’s algorithm will be determined by the number of time steps it takes before every vertex either has a candidate-mate of its choice or has processed all the edges incident on the vertex it owns.

Similar to Algorithm 22 the complexity of finding a candidate-mate, as described in Algorithm 20, is given by $\Theta(|S(v)|)$ using a linear search for the heaviest edge, where $S(v)$ represents the set of unmatched vertices adjacent to vertex $v$. This can be bounded by $O(\Delta)$, where $\Delta$ is the maximum degree of any vertex in the graph.

In each step a processor either sends a REQUEST message to a particular processor or UNAVAILABLE messages to one or more processors. The number of messages sent by any given processor is the number of edges incident on the vertex it owns, $\Theta(|S(v)|)$. This can again be bounded by $O(\Delta)$.

Algorithm 23 has $(2|E|)$ messages communicated before completion. Manne and Bisseling [50] show that Hoepman’s algorithm can complete in $O(\log |E|)$ rounds when the weights of edges are random. Thus, the expected time for Hoepman’s algorithm with $|V|$ processors can be expressed as

$$O(\Delta \log |E|).$$  

(18)

We will now present a parallel $\frac{1}{2}$-approx algorithm where each processor gets a set of vertices and the associated edges. The main idea is to combine Algorithms 22 and 23 to develop an efficient algorithm for the given problem.
V.3 PARALLEL $\frac{1}{2}$-APPROX ALGORITHM

We now present a parallel $\frac{1}{2}$-approx algorithm for computing matchings in graphs. The main idea is to adapt the serial pointer-based algorithm into a distributed algorithm by using communication techniques from Hoepman's algorithm to match edges whose end-points are not owned by the same processor. We call these edges as cross-edges or cut-edges, and edgecut represents the number of cross-edges.

Note that the data structures for graph representation store vertex adjacencies and the graph is distributed via vertex partitioning. Given a graph $G(V, E)$ and $p$ processors, the vertex set $V$ is partitioned into $p$ subsets $V_1, \ldots, V_p$. Processor $P_i$ owns the vertex subset $V_i$. In addition to the vertices that the processor owns, it also stores some of the vertices that are owned by other processors. We will represent the subgraph on processor $P_i$ as $G'_i(V'_i, E'_i)$. The vertex set $V'_i = V_i \cup V^G_i$, where the set $V^G_i$ represents the vertices in $G'_i$ that are not owned by $P_i$ - the ghost vertices. The edge set $E'_i = E_i \cup E^G_i$, where $E_i$ represents the edges between two vertices in $V_i$ (the internal edges), and $E^G_i$ represents the edges with one end-point in $V_i$ and the other in $V^G_i$ (the cross-edges). This is shown in Figure 40. The ghost vertices are colored purple and the cross-edges are shown with dashed lines. Note that a processor $P_i$ will not store edges connecting two vertices in $V^G_i$. Processor $P_i$ will also store the identities of processors that own the ghost vertices. It can be observed that storing the ghost vertices will have implications on the memory usage and is suitable for sparse graphs that have partitions with a small number of edges cut.

We now present a framework for computing approximate weighted-matching in parallel. The framework is sketched in Algorithm 24. This framework can be easily extended to compute approximation matchings with different objectives such as maximizing the cardinality or vertex-weight of a matching. The parallel algorithm has three distinct phases - (i) initialization, (ii) independent computation, and (iii) shared computation. The algorithm follows the SPMD (Single Program Multiple Data) model targeted for implementation using MPI standards for distributed memory architectures.

The given graph is partitioned and distributed among $p$ processors as described earlier. The associated data structures used in the algorithm are as follows. A set $Q_G$, initialized with ghost vertices $V^G_i$, represents the set of ghost vertices that still need to be processed in some manner. A set $Q_M$, which is initially empty, stores the matched vertices as the algorithm proceeds. A vector $\text{counter}$, initialized with
FIG. 40: Data distribution among processors. (a) The input graph $G = (V, E)$ with weights associated with the edges; (b) The vertex set $V$ is partitioned among two processors $P_0$ and $P_1$. Processor $P_0$ owns vertices $\{0, 3, 4\}$ and Processor $P_1$ owns vertices $\{1, 2, 6\}$. (c) Data storage on the processors. Along with internal edges, each processor will also store the endpoints of the edges that get cut (cross-edges). These vertices are called the ghost vertices and are colored purple in the figure.

the number of edges in $E'_i$ incident on each ghost vertex, represents the number of messages that need to be sent (and received) with respect to a ghost vertex. A vector $\text{candidateMate}$ stores the desired mate (pointer) for each vertex in $V'_i$. The sets $S_l(v)$ and $S_g(v)$, initialized with the adjacency sets for local and ghost vertices respectively, represent the unmatched adjacent vertices of vertex $v$ in $V'_i$. All these data structures are initialized in the initialization phase represented by Lines 4 through 15 in Algorithm 24.

In Phase-1, each processor attempts to match as many edges as possible without having to depend on information from the neighboring processors. Therefore, we call this phase independent computation. The computation in Phase-1 is similar to the serial pointer-based algorithm. The two main tasks in Phase-1 are to process all the (unmatched or exposed) vertices once, and process the vertices that get matched in the first task. Calls to functions $\text{ProcessExposedVerticesParallel}$ and $\text{ProcessMatchedVerticesParallel}$ are made to complete these tasks. We will describe these functions soon. The calls to these two functions will initiate some communication among processors. There are three types of messages - $\text{REQUEST}$,


Algorithm 24 Framework for parallel approximate matching. **Input:** A graph \( G(V, E) \) with weights associated with the edges. **Output:** A \( \frac{1}{2} \)-approx matching \( M \). **Data distribution:** Given \( p \) processors, vertex set \( V \) is partitioned into \( p \) subsets \( V_1, \ldots, V_p \). Processor \( P_i \) owns \( V_i \); stores a set of ghost vertices \( V_i^G \) and the edges incident on these two vertex subsets. **Associated data structures:** Set \( Q_G \) represents the ghost vertices that need to be processed in some manner, a set \( Q_M \) of matched vertices, a vector \( \text{counter} \) represents the number of messages that need to be sent with respect to each ghost vertex, a vector \( \text{candidateMate} \) represents the desired mate for each vertex, sets \( S_l(v) \) and \( S_g(v) \) represent the unmatched local and global vertices adjacent to \( v \) resp., and a set of matched edges \( M_i \).

1: **procedure** PARALLEL MATCHING FRAMEWORK \((G = (V, E), M)\)
2: loop on each processor \( P_i, i \in I = \{1, \ldots, p\} \)
3:     *** Initialization ***
4:     for \( v \in V_i \cup V_i^G \) do
5:         \( \text{candidateMate}(v) \leftarrow 0; \)
6:     end for
7:     \( Q_G \leftarrow V_i^G; \) \quad \triangleright \text{Set of ghost vertices.}
8:     \( M_i \leftarrow \emptyset; \)
9:     for \( v \in V_i \) do
10:         \( S_l(v) \leftarrow \text{adj}(v) \cap V_i; \) \quad \triangleright \text{Set of adjacent local vertices.}
11:         \( S_g(v) \leftarrow \text{adj}(v) \cap V_i^G; \) \quad \triangleright \text{Set of adjacent ghost vertices.}
12:     end for
13:     for \( v \in V_i^G \) do
14:         \( \text{counter}(v) \leftarrow |\text{adj}(v) \cap V_i|; \) \quad \triangleright \text{Local degree of a ghost vertex}
15:     end for
16:     *** Phase 1: Independent Computation ***
17:     \( Q_M \leftarrow \emptyset; \)
18:     for \( v \in V_i \) do
19:         \( \text{PROCESS EXPOSED VERTEX PARALLEL}(v); \)
20:     end for
21:     \( \text{PROCESS MATCHED VERTEX PARALLEL}(); \)
22:     *** Phase 2: Shared Computation ***
23:     while \( Q_G \neq \emptyset \) do
24:         \( \text{PROCESS MESSAGE}(); \)
25:         \( \text{PROCESS MATCHED VERTEX PARALLEL}(); \)
26:     end while
27:     return \( M_i; \)
28: end loop
29: Compute \( M \) based on \( M_i \) from all processors;
30: return \( M; \)
31: end procedure
UNAVAILABLE and FAILURE, descriptions of which will soon follow. All the REQUEST and UNAVAILABLE messages originating in this phase can be queued (bundled or aggregated), and sent at the end of this phase. There cannot be any FAILURE messages originating in this phase.

In Phase 2, computation can only proceed based on the information received from the neighboring processors, and therefore, the name - shared computation. The basic tasks in this phase can be grouped as communication-based and computation-based. The computation begins when a message from a neighboring processor is received. Communication is handled in function PROCESSMESSAGE, which is called within a while loop (Line 24). Appropriate action, based on the type of message, is taken within this function. If the message results in edges being matched, then a call to function PROCESSMATCHEDVERTICESPARALLEL is made (Line 25). Detailed descriptions of these two functions will soon follow. The tasks are looped until the set \( Q_G \) becomes empty. As will be described soon, a ghost vertex \( g \) is removed from \( Q_G \) only when its counter \((g)\) becomes zero. This implies that all computations related to this vertex are complete. Matchings on each processor, \( M_i \), can be gathered on the master process, or consumed locally, depending on the needs of the applications. We will now present the details of different functions that are used in Algorithm 24.

All the communication involved in the algorithm is handled by three types of messages - REQUEST, UNAVAILABLE and FAILURE. Messages are asynchronous point-to-point messages sent by one processor to another. Each message contains identities of two vertices that represent a cross-edge. The meaning of a message is determined by the type of the message, as follows. A REQUEST message conveys a positive intent of matching a cross-edge sent by the owner-processor of one endpoint to the owner-processor of the other endpoint. An UNAVAILABLE message sent by a processor means that the local vertex identified in the message has already been matched, and therefore, a request to match this vertex by a neighboring processor cannot be satisfied. A FAILURE message sent by a processor means that the local vertex identified in the message could not be matched and that its owner-processor has finished all computation related to this vertex. Note the minor difference between the UNAVAILABLE and FAILURE types - the local vertex identified in the message is matched in the former and unmatched in the latter; although, both types imply a negative response to match a cross-edge as identified in the message.

We mentioned that computation in Phase-1 is similar to the serial pointer-based
algorithm. We will now present the modified versions of the algorithms that we discussed for the serial pointer-based algorithm in Section 1. Algorithm \textsc{ComputeCandidateMate}(v) will remain the same except for a small modification on Line 4 to reflect the local and ghost vertex sets on a processor. This is shown in Algorithm 25. Again, ties from duplicate weights are resolved based on the vertex indices.

\textbf{Algorithm 25} Compute candidate-mate in parallel. \textbf{Input:} A vertex \(v\) and its adjacency set. \textbf{Output:} The candidate-mate for a given vertex \(v\). \textbf{Associated data structures:} Sets \(S_l(v)\) and \(S_g(v)\) represent the unmatched local and global vertices adjacent to \(v\) resp.

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{ComputeCandidateMateParallel}{$v$}
\State \(w \leftarrow 0;\)
\State \(\text{maxWt} \leftarrow -\infty;\)
\For{\(z \in \{S_l(v) \cup S_g(v)\}\)} > Weight of an edge \((x,y)\) is denoted by \(w(e_{xy})\).
\If{(\(\text{maxWt} < w(e_{vz})\)) \textbf{or} (\(\text{maxWt} = w(e_{vz})\) \textbf{and} \(w < z\))} \State \(w \leftarrow z;\)
\State \(\text{maxWt} \leftarrow w(e_{vz});\)
\EndIf
\EndFor
\State \textbf{return} \(w;\)
\EndProcedure
\end{algorithmic}
\end{algorithm}

The other function used in the serial algorithm is Algorithm \textsc{ProcessExposedVertex}. A similar function for the parallel algorithm is described in Algorithm 26. Since the parallel algorithm needs the capability to process cross-edges, it should also be capable of communicating with its neighbors. Algorithm 26 shows the processing of an unmatched vertex. The first step in processing an unmatched vertex is to find the candidate-mate. If the candidate-mate is a ghost vertex, then a REQUEST message is sent to the owner of the ghost vertex. If the candidate-mate also points back to the exposed vertex, then a locally dominating edge has been discovered and can be matched. Note that the \textit{candidateMate}(g) of a ghost vertex will be set based on the REQUEST message from its owner. Once an edge is matched (Line 9), the endpoints are added to the set \(Q_M\) for further processing (Line 16). If an exposed vertex cannot be matched, FAILURE messages are sent to all the owner processors of cross-edges incident on this vertex (Lines 19 to 21). The adjacency sets \(S_l\) and \(S_g\) also need to modified. There are also additional computations that are done by a call to the function \textsc{ProcessCrossEdge} (Line 14) that will be described next.
Algorithm 26 Process an exposed vertex in parallel. **Input:** A vertex \( v \) and its adjacency set. **Associated data structures:** A set \( Q_M \) of matched vertices, a vector \( \text{candidateMate} \) represents the desired mate for each vertex, set \( S_i(v) \) represents the unmatched local vertices adjacent to \( v \), and a set of matched edges \( M_i \). **Effect:** Processes an exposed vertex - find candidate-mate, match if possible, update message counters and send messages if needed.

1: **procedure** \( \text{PROCESSEXPOSEDVERTEXPARALLEL}(v) \)
2: \( \text{candidateMate}(v) \leftarrow \text{COMPUTECANDIDATEMATEPARALLEL}(v); \)
3: \( c \leftarrow \text{candidateMate}(v); \)
4: **if** \( c \neq 0 \) **then**
5: \( \text{if} \ c \in V_i^G \text{ then} \quad \triangleright \ c \text{ is a ghost vertex.} \)
6: \( \text{send REQUEST}(v, c); \)
7: **end if**
8: **if** \( \text{candidateMate}(c) = v \) **then** \( \triangleright \text{Both vertices point to each other.} \)
9: \( M_i \leftarrow M_i \cup \{(v, c)\}; \)
10: **if** \( c \in V_i \) **then**
11: \( S_i(v) \leftarrow S_i(v) \setminus \{c\}; \)
12: \( S_i(c) \leftarrow S_i(c) \setminus \{v\}; \)
13: **else**
14: \( \text{PROCESSCROSSEDGE}(v, c); \quad \triangleright \ c \text{ is a ghost vertex.} \)
15: **end if**
16: \( Q_M \leftarrow Q_M \cup \{v, c\}; \)
17: **end if**
18: **else**
19: **for** \( w \in \text{adj}(v) \cap V_i^G \) **do** \( \triangleright \ w \text{ is a ghost vertex.} \)
20: \( \text{send FAILURE}(v, w); \)
21: **end for**
22: **end if**
23: **end procedure**
We observe that there is a shortcoming in Hoepman’s algorithm described in Algorithm 23. A processor $P_i$ will ignore all messages that it receives as soon as it successfully finds a mate and sends UNAVAILABLE messages to its remaining active neighbors. Once the UNAVAILABLE messages are received by these neighbors they will not send any message to $P_i$. However, there can be a situation when a processor $P_k$ sends a REQUEST message to $P_i$ before it receives an UNAVAILABLE message, but after $P_i$ has found a mate. This case is illustrated in Figure 39 in step (d’). Thus, the REQUEST message from $P_k$ will be lost, or not acknowledged, by Processor $P_i$ (Processor $P_1$ in Figure 39). The message passing interface MPI standard stipulates that every send be matched with a corresponding receive. Therefore, techniques that prevent message losses in the algorithm will facilitate implementation, especially using the MPI standards for distributed memory systems. We address this unacknowledged-message problem by providing two data structures to keep track of messages - (i) a set $Q_G$ of ghost vertices that need to be processed in some manner, and (ii) a vector counter that stores a number for each ghost vertex. The value for a ghost vertex in counter is initialized with the number of cross-edges incident on it (the local degree). The counting of messages can now be done by keeping track of each cross-edge and modifying the counters each time a communication happens. When all the cross-edges incident on a given ghost vertex $g$ are processed in some manner its $\text{counter}(g)$ becomes zero, it can then be removed from the set $Q_G$. This is shown in Algorithm 27.

**Algorithm 27** Process a cross-edge.  
**Input:** Two vertices that represent a cross-edge.  
**Associated data structures:** Set $Q_G$ represents the ghost vertices that need to be processed in some manner, a vector counter represents the number of messages that need to be sent with respect to each ghost vertex, and sets $S_g(v)$ represents the unmatched ghost vertices adjacent to $v$.  
**Effect:** Modifies the adjacency set of a ghost vertex, decrements its counter and modifies the set $Q_G$ if needed.

1: procedure **PROCESSCROSSEDGE**(l, g) $\triangleright$ g is ghost, and l is a local vertex.
2: \hspace{1em} $S_g(l) \leftarrow S_g(l) \setminus \{g\}$;
3: \hspace{1em} counter($g$) $\leftarrow$ counter($g$) $-$ 1;
4: \hspace{1em} if counter($g$) = 0 then
5: \hspace{2em} $Q_G \leftarrow Q_G \setminus \{g\}$; $\triangleright$ All computation for vertex $g$ is complete.
6: \hspace{1em} end if
7: end procedure

The call to function PROCESSEXPOSEDVERTEXPARALLEL will result in some edges (at least the heaviest edge) getting matched. The vertices that point to matched
vertices should reset their pointers to point to other potential mates. This is done in
function \textsc{ProcessMatchedVerticesParallel}, described by Algorithm 28. This
is similar to the processing in Algorithm 22 done in Lines 11 through 21. Again, the
function loops through the matched vertices in set $Q_M$. If the vertex being processed
is a ghost vertex, then it can simply be ignored (Lines 5 through 7). Adding ghost
vertices to $Q_M$ can be avoided, but is shown here for simplicity. Note that only those
vertices that were pointing to the matched vertices need to be processed, since these
vertices must find new candidate-mates (Lines 8 through 13). Ghost vertices that
are pointing to the matched vertices (via \textsc{REQUEST} messages) will be set to \textit{null} (Line
17) and an \textsc{UNAVAILABLE} message is sent to the owners of these ghost vertices (Line
19). Accordingly, those owners will have to find new candidate-mates.

\textbf{Algorithm 28} Process matched vertices in parallel. \textbf{Input:} A set of matched
vertices. \textbf{Associated data structures:} A set $Q_M$ of matched vertices, a vector
\textsc{candidateMate} represents the desired mate for each vertex, set $S_i(v)$ represents the
unmatched local vertices adjacent to $v$, and a set of matched edges $M_i$. \textbf{Effect:} Re-
sets the pointers of the vertices pointing to matched vertices. Modifies the adjacency
sets, and sends messages if needed.

\begin{verbatim}
1: procedure \textsc{ProcessMatchedVerticesParallel}
2:     while $Q_M \neq \emptyset$ do
3:         $u \leftarrow$ pick from $Q_M$;
4:         $Q_M \leftarrow Q_M \setminus \{u\}$;
5:         if $u \in V_i^G$ then $\triangleright$ Ignore ghost vertices.
6:             continue;
7:     end if
8:     for $v \in S_i(u)$ do $\triangleright$ Unmatched local vertices.
9:         $S_i(v) \leftarrow S_i(v) \setminus \{u\}$;
10:        if \textsc{candidateMate}(v) = $u$ then
11:            \textsc{ProcessExposedVertexParallel}(v);
12:        end if
13:     end for
14:     $S_i(u) \leftarrow \emptyset$;
15:     for $v \in (\text{adj}(u) \setminus V(M_i)) \cap V_i^G$ do $\triangleright$ Ghost vertices pointing to $v$; $V(M_i)$
16:         represents matched vertices.
17:         if \textsc{candidateMate}(v) = $u$ then $\triangleright$ Reset the pointer to \textit{null}.
18:             \textsc{candidateMate}(v) $\leftarrow$ 0;
19:         end if
20:     end for
21: end while
22: end procedure
\end{verbatim}
Computation in Phase-2 can start only when a message is received. This is done by calling function \texttt{ProcessMessage} until the set \(Q_G\) becomes empty. Function \texttt{ProcessMessage} is described in Algorithm 29. Since there are only three types of messages exchanged between processors, the actions that need to performed upon receiving messages can be organized based on the type of messages received.

When a \texttt{REQUEST}(g,l) message is received on Processor \(P_i\), it means that \(\text{candidateMate}(g)\) for the ghost vertex \(g\) can be set to the local vertex \(l\). If the \(\text{candidateMate}(l)\) equals \(g\), then a locally dominant edge has been found and can be matched (Line 9). If matched, the adjacency sets and counters are modified (Line 10), and the matched vertices are added to the set \(Q_M\).

An \texttt{UNAVAILABLE}(g,l) message conveys that the owner of the ghost vertex does not intend to match this cross-edge. Thus, a new candidate-mate for \(l\), if not already matched, has to be found (Line 19). If the local vertex has already been matched, then an \texttt{UNAVAILABLE} message would have already been sent (Algorithm 28, Line 19), and therefore, no further action needs to be taken and the function terminates (Line 16).

A \texttt{FAILURE}(g,l) message means that all computation related to this cross-edge is complete (the ghost vertex could not be matched). The counters are modified accordingly (Line 22). Note that a \texttt{FAILURE} message can be received only in response to an \texttt{UNAVAILABLE} message, and never as a response to a \texttt{REQUEST} message. Thus, nothing has to be done with respect to setting pointers for the local vertex \(l\).

In Algorithm 29, messages are processed one at a time. However, messages can be aggregated for better performance. If a bundled message is received, we can simply loop through the bundle, processing one message at a time.

In the given scheme, there are limited possibilities of message exchanges for a cross-edge. These are illustrated in Figure 41. Note that a \texttt{REQUEST} message will never be responded to with a \texttt{FAILURE} message (a request means that there is at least one eligible edge for matching). Also, a processor will send a \texttt{FAILURE} message only when it has received \texttt{UNAVAILABLE} messages from all its neighbors.

This completes the description of all the functions that are used in the parallel approximation algorithm. Execution of Algorithm 24 on a simple graph is shown in Figure 42.
Algorithm 29 Process a message. **Input:** A message that contains identities of two vertices. **Associated data structures:** A set $Q_M$ of matched vertices, a vector $\text{candidateMate}$ represents the desired mate for each vertex, and a set of matched edges $M_i$. **Effect:** Processes a message and act accordingly.

1: procedure PROCESSMESSAGE  
2:     receive message;  \(\triangleright g\) is a ghost, and \(l\) is a local vertex.  
3:     if message = REQUEST\((g, l)\) then  \(\triangleright\) CASE 1.  
4:         if \(l \in V(M_i)\) then  \(\triangleright V(M_i)\) is a set of matched vertices on \(P_i\).  
5:             return;  
6:         end if  
7:         candidateMate\((g)\) $\leftarrow l$;  
8:         if candidateMate\((l) = g\) then  \(\triangleright\) Add an edge to the matching.  
9:             $M_i \leftarrow M_i \cup \{(l, g)\}$;  
10:             PROCESSCROSSEDGE\((l, g)\);  
11:             $Q_M \leftarrow Q_M \cup \{l, g\}$;  
12:         end if  
13:     else if message = UNAVAILABLE\((g, l)\) then  \(\triangleright\) CASE 2.  
14:         PROCESSCROSSEDGE\((l, g)\);  
15:         if \(l \in V(M_i)\) then  
16:             return;  
17:         end if  
18:         if candidateMate\((l) = g\) then  
19:             PROCESSEXPOSEDVERTEXPARALLEL\((l)\);  
20:         end if  
21:     else if message = FAILURE\((g, l)\) then  \(\triangleright\) CASE 3.  
22:         PROCESSCROSSEDGE\((l, g)\);  
23:     end if  
24: end procedure

**FIG. 41:** Possible communication patterns. Message types are denoted by $R$ for REQUEST, $U$ for UNAVAILABLE, and $F$ for FAILURE. (a) When two requests match, it results in a matched edge. An UNAVAILABLE message from $P_1$ to $P_0$ can be responded by an UNAVAILABLE message (b), or a FAILURE message (c) from $P_0$ to $P_1$. (d) An UNAVAILABLE message from $P_0$ can either be responded with an UNAVAILABLE or a FAILURE message by $P_1$. 
FIG. 42: Execution of parallel approximation algorithm. (a) The input graph $G = (V, E)$ with weights associated with the edges, vertices $\{0, 3, 4\}$ are assigned to processor $\{P_0\}$, and vertices $\{1, 2, 6\}$ are assigned to processor $\{P_1\}$. (b) an intermediate step of execution when local computations are done. REQUEST(4,1) message is sent from $P_0$ to $P_1$; (c) Processor $P_0$ matches edge (0, 3) and sends messages: UNAVAILABLE(0, 6) and REQUEST(4, 6) to $P_1$. Processor $P_1$ matches edge (1, 2) and sends messages: UNAVAILABLE(1, 4) and REQUEST(6, 4) to $P_0$. (d) Processor $P_0$ matches edge (4, 6) and sends message UNAVAILABLE(4, 1) to $P_1$. Processor $P_1$ matches edge (6, 4) and sends message UNAVAILABLE(6, 0) to $P_0$. 
V.3.1 Complexity Analysis

Given a graph $G(V, E)$ with weight function $w : E \rightarrow \mathbb{R}^+$, and $p$ processors, let $n = |V|$ and $m = |E|$ be the number of vertices and edges respectively. Recall that $G$ is distributed on $p$ processors as follows. The vertex set $V$ is partitioned into $p$ subsets $V_1, \ldots, V_p$ and Processor $P_i$ owns the vertex subset $V_i$. Let $m' = |E_{cut}|$ represent the total edgecut, and $m'_{P_i}$ represent the number of cross-edges incident on the vertices owned by Processor $P_i$. Let $\Delta$ represent the maximum degree of any vertex in $G$, and $d(v)$ represent the degree of a vertex $v$.

We make the following assumptions in this analysis:

- The adjacency list of a vertex is maintained in a sorted order (we note that this does not increase the complexity of the algorithm);

- The weights of the edges are distributed uniformly randomly. Therefore, the expected number of rounds for completion is $O(\log m)$ [49, 50]; and

- The input graph has good separators resulting in well balanced partitions. Let $\alpha$ represent the load imbalance in the (interior) edges incident on the vertices owned by the same processor (ratio of the maximum number of interior-edges to the average number of interior-edges over all processors), $\beta$ represent a similar imbalance factor in the edges incident on the boundary vertices, and $\gamma$ represent the imbalance factor in the cut-edges. The imbalance factors are illustrated in Figure 43.

The compute time for Phase-1 on Processor $P_i$ is given by $O(\sum_{v \in V_i} d(v) \log d(v))$, where the log factor comes from sorting the adjacency sets. This can be relaxed to $O((\log \Delta) \sum_{v \in V_i} d(v))$. This can be generalized to any processor as

$$O(\frac{\alpha m \log \Delta}{p}).$$

(19)
The total communication cost, including that at the end of Phase-1 and during Phase-2, on Processor $P_i$ is at most $(3|m'_{P_i}|)$. This can be generalized to any processor as

$$O\left(\frac{\gamma m'}{p}\right).$$

(20)

The computation in Phase-2 is communication dependent. A cross-edge can only be matched after receiving a matching intent from the owner-processor of the other end-point of the cross-edge. Once a cross-edge is matched, or removed as a potential for matching, it can have a ripple effect on the interior edges or other cross-edges. However, the assumption of random edge-weights is critical in limiting the ripple effect and analyzing the expected complexity for Phase-2.

The computation in Phase-2 on Processor $P_i$ is given by

$$O\left(\sum_{v \in V(m'_{P_i})} d(v) \log d(v)\right),$$

where $V(m'_{P_i})$ represents the vertices in the set of cross-edges on $P_i$. The log factor arises from sorting the adjacency sets of vertices. The imbalances in the number of cross-edges ($\gamma$), as well as, imbalances in the internal edges incident on the boundary vertices ($\beta$) will affect the generalization of the computation cost in Phase-2. Thus, the computational complexity for Phase-2 for any processor is given by

$$O\left(\frac{\gamma \beta m' \log \Delta}{p}\right).$$

(21)

Thus, the total complexity for parallel $\frac{1}{2}$-approx algorithm is given by

$$O\left(\frac{\alpha m \log \Delta}{p} + \frac{\gamma m'}{p} + \frac{\gamma \beta m' \log \Delta}{p}\right).$$

(22)

The complexity analysis provides us an insight for expected speedup on a parallel architecture. Recall that the complexity for the serial algorithm is $O(m \log \Delta)$. Under the stated assumptions of random edge-weights and good separators, the speedup obtained can be expressed as:

$$\frac{p}{F(\alpha, \alpha \beta m' \log \Delta, G(\frac{\gamma m'}{p})}. $$

(23)

Where $G$ is a function on the communication cost depending the underlying architecture and $F$ is an overall function depending the graph structure, imbalance factors and the architecture of the parallel system. While the load balance factors ($\alpha, \beta, \gamma$) are important, also important is the edgecut, which directly influences the amount of communication that needs to be performed. On modern architectures such as
compute clusters with fast processors and relatively slow communication, edgecut is the most influential factor in determining performance. We also make an important assumption about the random distribution of edge-weights that directly influences the number of rounds of execution $O(\log m)$ instead of $O(m)$. We will now present experimental results on the parallel $\frac{1}{2}$-approx algorithm.
V.4 EXPERIMENTAL RESULTS

In this section we present experimental results from our implementation of matching algorithms in a toolkit called MatchBox-P. The two types of experiments done are serial and parallel. The goals for serial experiments are to demonstrate the efficiency of approximation algorithms in terms of execution time, cardinality and weight of matching as compared to those of exact algorithms. We will also demonstrate the efficiency of the pointer-based algorithm as compared to other approximation algorithms. For the parallel experiments we will try to identify classes of graphs for which the proposed algorithm, in its current implementation, is effective, and in the process expose the shortcomings and suggest improvements. The parallel experiments are conducted on a Cray XT4 system, Franklin, at NERSC with 9,660 compute nodes. Each compute node has a 2.3 GHz AMD Opteron quad core processor with 8 GB RAM. The nodes are interconnected using SeaStar2 router with a 3D torus topology. The details can be obtained from www.nersc.gov. The serial experiments are conducted on a system equipped with four 2.4 GHz Intel quad core processors and 32 GB RAM at Old Dominion University.

V.4.1 Data Set for Experiments

The graphs used for experiments can be broadly classified into two types: (i) graph representations of regular sparse matrices downloaded from the University of Florida Sparse Matrix Collection, and (ii) synthetic and model graphs. A matrix is stored as a general graph, where rows and columns of the matrix represent vertices, and the nonzero elements represent edges. The absolute value of a nonzero element in the matrix is considered as the weight of the edge that connects the vertices representing the row and the column of the nonzero element. A similar model is used to represent symmetric matrices. Since the files downloaded from the University of Florida Sparse Matrix Collection store only the lower triangle of the matrix, we explicitly add edges to represent both the upper and lower triangles of the matrix. Two types of synthetic graphs are used - random geometric graphs and scalable synthetic compact application (SSCA#2) graphs. Generation of random geometric graphs is implemented in MatchBox-P, and SSCA#2 graphs are generated with GT-Graph generator [5]. In order to eliminate self-loops, the SSCA#2 graphs are stored as bipartite graphs. Two-dimensional five-point and nine-point grid graphs
are used as the model graph problems. The matrices used in the experiments are listed in Table 11, and the associated structures are illustrated in Figure 44.

<table>
<thead>
<tr>
<th>Name</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC-680</td>
<td>1,365,424</td>
<td>1,693,767</td>
<td>Unsymm</td>
<td>Circuit simulation matrix</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>2,894,720</td>
<td>5,514,242</td>
<td>Unsymm</td>
<td>Circuit simulation matrix</td>
</tr>
<tr>
<td>Rajat31</td>
<td>9,380,004</td>
<td>20,316,253</td>
<td>Unsymm</td>
<td>Circuit simulation matrix</td>
</tr>
<tr>
<td>Cage14</td>
<td>3,011,570</td>
<td>27,130,349</td>
<td>Unsymm</td>
<td>DNA electrophoresis</td>
</tr>
<tr>
<td>Ldoor</td>
<td>1,904,406</td>
<td>84,035,431</td>
<td>Symmm</td>
<td>INDEED Test Matrix</td>
</tr>
<tr>
<td>Audikw-1</td>
<td>1,887,390</td>
<td>154,359,999</td>
<td>Symmm</td>
<td>Crankshaft model</td>
</tr>
</tbody>
</table>

TABLE 11: Matrix Instances downloaded from University of Florida Matrix Collection. Unsymm represents unsymmetric matrices and Symm represents symmetric matrices.

FIG. 44: Visualization of matrix structures.

A $d$-dimensional random geometric graph (RGG), represented as $G(n, r(n))$, is a graph generated by randomly placing $n$ vertices in a $d$-dimensional space and connecting pairs of vertices whose Euclidean distance is less than or equal to $r(n)$. In our experiments we only consider two-dimensional RGGs contained in a unit square, $[0,1]^2$, and the Euclidean distance between two vertices is used as the weight of the edge connecting them. Our primary objective is to generate RGGs that have good separators. Therefore, we generate RGGs that are as sparse as possible, but without generating too many isolated vertices or too many disconnected components. Connectivity, a monotonic property of RGG, in 2$d$ unit-square RGGs has a sharp threshold at $r_c = \sqrt{\frac{\ln n}{n \pi}}$ [21]. The connectivity threshold is also the longest edge length of the minimum spanning tree in $G$ [58]. The thermodynamic limit when a giant component appears with high probability is given by $r_t = \sqrt{\frac{\lambda_c}{n}}$ [21, 32]. Empirically, the value of $\lambda_c$ is given by 2.0736 for 2$d$ unit-square RGGs. The particular value of $r(n)$ that we have used in the experiments is $r_{ct} = (r_c + r_t)/2$. We refer the reader to [21, 23, 22, 32, 58] for details. A 2$d$ RGG with 1,000 vertices visualized with Pajek [10] is shown in Figure 45. Note that along with a few isolated vertices,
there are also a few disconnected components. The details of RGGs used in the
experiments are provided in Table 12.

FIG. 45: Random geometric graph. A random geometric graph with 1,000 vertices
as visualized with Pajek.

The SSCA#2 graphs were generated with the GTgraph generator [5]. For conve-
nience, we eliminate self-loops by considering the original graph as a bipartite graph
by simply representing every vertex in the original graph with two vertices (one in
each set) in the bipartite graph. We generated SSCA#2 graphs with the following
properties. For a particular value of $\lambda$, the graph has $2^\lambda$ vertices; the maximum size
of random-sized cliques is $2^{3\lambda}$; initial probability of interclique edges is set to 0.5; and
the weights of edges are uniformly randomly assigned with a maximum value of $2^\lambda$.
We refer the reader to [5] for details. Visualization of an SSCA#2 graph of 1,024
vertices with Pajek is shown in Figure 46. The details of SSCA#2 graphs used in
the experiments are provided in Table 12.

Model graphs used in the experiments are five-point and nine-point grid graphs.
The grid graphs are generated within MatchBox-P and the edge weights are assigned
uniformly randomly in the range 0 through RAND_MAX. Visualization of sample five-
point and nine-point graphs with Pajek are provided in Figures 47 and 48, and the
details of the grid graphs used in the experiments are provided in Table 12.
FIG. 46: *SSCA#2 graph*. An SSCA#2 graph with 1,024 vertices as visualized with Pajek.

<table>
<thead>
<tr>
<th>Name</th>
<th>#Vertices</th>
<th>#Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGG-1</td>
<td>320,000</td>
<td>63,148,387</td>
</tr>
<tr>
<td>RGG-2</td>
<td>8,388,608</td>
<td>404,249,646</td>
</tr>
<tr>
<td>SSCA#2-1</td>
<td>2,097,152</td>
<td>63,148,387</td>
</tr>
<tr>
<td>SSCA#2-2</td>
<td>8,388,608</td>
<td>404,249,646</td>
</tr>
<tr>
<td>FivePtGrid4k</td>
<td>16,000,000</td>
<td>31,992,000</td>
</tr>
</tbody>
</table>

TABLE 12: *Synthetic and Model Graphs*. SSCA#2 graphs are generated using GT-Graph generator. The number of vertices in the original graph are doubled to convert it into a bipartite graph to eliminate self-loops; duplicate edges, if any, are also eliminated. RGGs and grid graphs are generated with MatchBox-P and have random edge weights.
FIG. 47: *Five-point grid graph.* A 10 X 10 five-point grid graph visualized with Pajek.

FIG. 48: *Nine-point grid graph.* A 10 X 10 nine-point grid graph visualized with Pajek.
V.4.2 Performance of Serial Matching Algorithms

In this section we show experimental results from serial implementation of the matching algorithms. The goal for these experiments is to highlight the performance of approximation algorithms not only in the execution time but also for computing matching of good quality. We present the quality as a ratio of cardinality and weight of approximation matchings to those of exact matchings. Our implementation of exact matching algorithm is based on the primal-dual algorithm [57] using array data structures. For large graphs, we also observe empirically that the performance of binary-heap-based implementation is only incrementally better than the array-based implementation of the exact algorithm. The results are summarized in Table 13. It can be observed that the approximation algorithms generate matchings of high quality with huge gains in compute time.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Wgt-Ratio</th>
<th>Card-Ratio</th>
<th>Time-Approx(s)</th>
<th>Time-Exact(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC-680</td>
<td>1.00</td>
<td>0.99</td>
<td>0.13</td>
<td>46,639</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>0.99</td>
<td>0.81</td>
<td>0.28</td>
<td>170,059</td>
</tr>
<tr>
<td>Rajat31</td>
<td>1.00</td>
<td>1.00</td>
<td>0.58</td>
<td>1,361,146</td>
</tr>
<tr>
<td>Cage14</td>
<td>1.00</td>
<td>1.00</td>
<td>0.55</td>
<td>409,250</td>
</tr>
<tr>
<td>Ldoor</td>
<td>1.00</td>
<td>1.00</td>
<td>0.46</td>
<td>178,004</td>
</tr>
<tr>
<td>Audikw1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.72</td>
<td>242,591</td>
</tr>
</tbody>
</table>

TABLE 13: Performance of serial approx algorithm. The second column represents the ratio of weights of approximate and exact matchings. Similarly, the third column represents the ratio of cardinality of the two matchings. Fourth and fifth columns show the time in seconds to compute approximate and exact matchings respectively.

We will now present the relative performance of different half approximation algorithms. The two main categories of approximation algorithms are the sorting-based algorithms of Avis [4] and Preis [64], and path growing algorithms of Vinkemeier and Hougardy [24, 74]. The path growing algorithm finds simple paths of heaviest weight in a graph, alternatively adding edges to two sets of potential matchings. While in PG-1 the two sets of potential matching are compared at the very end, the two potential sets are compared for each distinct path in PG-2, and therefore, PG-2 is a better algorithm. PG-3 merges the two potential matching sets using Dynamic Programming techniques, and thus, has the best results, with respect to the weight of the matching, as compared to PG-1 and PG-2. Since the pointer-based algorithm is a version of Preis’s algorithm, which in turn is a version of Avis’s algorithm, we will only present the results for the pointer-based algorithm. Weight and cardinality
of the approximation matchings are shown in Figures 49 and 50 as a ratio to those of exact algorithm. The execution time for different algorithms is shown in Figure 51.

![Bar chart](image)

**FIG. 49:** Performance of Serial Approximation Algorithms: Weight. The path growing algorithms are represented by PG1, PG2, and PG3.

From the experimental results it can be observed that the pointer-based algorithm computes matchings of high quality at high speed. We will now present the performance results for the parallel half-approximation algorithm.
FIG. 50: Performance of Serial Approximation Algorithms: Cardinality.

FIG. 51: Performance of Serial Approximation Algorithms: Compute Time.
V.4.3 Performance of Parallel Matching Algorithm:

The parallel half-approximation algorithm has been implemented in C++ and uses Message Passing Interface (MPI) libraries for communication between processors. The implementation also uses the Standard Template Library (STL) data structures such as Vectors and Maps. We use multi-level K-way partitioning algorithm in Metis [41] for distributing input data among participating processors. As described in Algorithm PARALLEL_MATCHING_FRAMEWORK, the implementation has three distinct phases:

- **Initialization**: The actions performed in this phase are initialization of associated data structures such as the adjacency structures for the ghost vertices, mapping of ghost vertex indices to zero-based indices, allocation of memory for communication (based on the edgecut), etc.

- **Phase-1**: In this phase, candidate mates are set for all local vertices, and an attempt to match is performed. At the end of Phase-1, all the resulting communication is sent. Individual messages to a processor are aggregated and sent as one packet of information using MPI constructs for immediate messages (MPI_Isend()) [67].

- **Phase-2**: Computation in Phase-2 is communication dependent, and can only start once a message is received. It can be broadly classified into two supersteps - computation and communication. In our current implementation, we do not aggregate individual messages, but send (non-blocking) them immediately as needed. Given the fact that we have a bound on the number of messages that will be communicated, we have implemented asynchronous messaging using the MPI constructs for buffered messages (MPI_Bsend()) [67]. We note that the current implementation can be improved by performing message aggregation in Phase-2, while acknowledging that there will be a certain amount of overhead for message aggregation and potentially longer idle times as processors wait for messages.

We will now present details from parallel experiments for synthetic and model graphs for up to 8,192 processors on Franklin.
Five-Point Grid Graph of $4k \times 4k$ Size

The graph representing the $4k \times 4k$ grid has 16,000,000 vertices and 31,992,000 edges. Since the amount of communication is directly dependent on the edgecut, existence of good separators is important to obtain good performance for the parallel algorithm. For the following experiments we used multi-level K-way partitioning algorithm in Metis [41]. In Figure 52, we plot the edgecut as a function of number of vertices. An ideal partitioning of a square grid ($2D$ block distribution) will be proportional to $(2\sqrt{|V|}(\sqrt{P} - 1))$, where $|V|$ is the number of vertices and $P$ is the number of partitions. We observe a similar pattern in the partitions that were obtained from Metis giving us an expectation for good performance.

![Graph showing edgecut as a function of number of vertices](image)

**FIG. 52:** $4k$ grid graph: Edgecut as a function of number of vertices. Actual edgecut for different number of partitions using multi-level K-way partitioning algorithm in Metis, and ideal edgecut given by $(2\sqrt{|V|}(\sqrt{P} - 1))$, where $V$ is the number of vertices and $P$ is the number of partitions.

The maximum time is the longest time taken by any given processor in the group of processors used to compute a matching. Alternatively, it is the time taken by the slowest processor. The difference in the compute time of different processors can be due to various reasons including load imbalance, heterogeneous capacities, graph structure, and unusual behavior of different processors that is time dependent. This become an important factor when the number of processors used for a given job is very
large. Therefore, we also provide the average (mean) compute time for computing the matching. Ideally, the experiments should be repeated for a large number of times, but given limited resources we have not repeated similar experiments, especially for experiments with large number of processors. Maximum and average execution times for the $4^k$ grid graph are shown in Figures 53 and 54 respectively. For each type, the execution time of different phases of the computation are shown separately. The speedup obtained is shown in Figure 55.

![Graph showing compute time](image_url)

FIG. 53: $4^k$ grid graph: Compute time (maximum). Maximum time is the time in seconds of the slowest processor in the group of processors used to solve the problem.

It can be observed that while the execution time for Initialization and Phase-1 scale with the number of processors, the execution time for Phase-2 does not scale well, and drastically increases for 4,096 and 8,192 processors. It should be noted that the messages are aggregated only in Phase-1 of our current implementation. It should also be noted that for larger number of processors the amount of work done per processor is very small. In order to explore further, we plot the cardinality of the matching at the end of Phase-1 in Figure 56. It can be observed that close to 100 per cent cardinality is obtained at the end of Phase-1 in most cases. As the number of partitions are increased, the cardinality of matching at the end of Phase-1 also decreases resulting in more work during Phase-2. The edgecut as a function of the number of edges is also plotted in Figure 56. It can be observed that a very small
FIG. 54: 4k grid graph: Compute time (average). Average time is the sum of compute time on each processor in the group divided by the number of processors in that group.

FIG. 55: Speedup for 4k x 4k grid graph.
fraction of edges get cut.

FIG. 56: 4k grid graph: Cardinality after Phase-1.

Weak Scaling for Five Point Grid Graphs

We now present weak scaling studies on the five-point grid graphs. The largest graph is the graph with 16 million vertices, and we solve it on 2,048 and 1,024 processors as two separate series. For each subsequent data point, we will reduce the number of vertices and the number of processors by half. The test set is summarized in Table 14.

If the total compute time remains fairly constant for different graph size and number of processor combinations, then we demonstrate the weak scalability of the parallel $\frac{1}{2}$-approx algorithm. We plot the execution times for the two series in Figures 57 and 58. It can be observed that the total execution time remains fairly constant. In particular, initialization and Phase-1 show good scalability. However, Phase-2 does not scale proportionally, especially for smaller graph sizes. We plot edgecut and number of messages sent for each grid-size and number of processor combinations in Figure 59. The two curves are edgecut divided by the number of processors and messages sent divided by the number of processors. From this figure we can observe that the edgecut increases, and therefore, the total time for Phase-2
<table>
<thead>
<tr>
<th># Vertices</th>
<th>Grid Dimension</th>
<th>#P-Series1</th>
<th>#P-Series2</th>
</tr>
</thead>
<tbody>
<tr>
<td>16,000,000</td>
<td>4000 X 4000</td>
<td>2048</td>
<td>1024</td>
</tr>
<tr>
<td>8,000,000</td>
<td>2828 X 2828</td>
<td>1024</td>
<td>512</td>
</tr>
<tr>
<td>4,000,000</td>
<td>2000 X 2000</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>2,000,000</td>
<td>1414 X 1414</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1000 X 1000</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>500,000</td>
<td>707 X 707</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>250,000</td>
<td>500 X 500</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>125,000</td>
<td>354 X 354</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>62,500</td>
<td>250 X 250</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>31,250</td>
<td>177 X 177</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>15,625</td>
<td>125 X 125</td>
<td>2</td>
<td>-NA-</td>
</tr>
</tbody>
</table>

TABLE 14: Grid graphs for weak scalability studies. Columns three and four represent the number of processors used to solve the grid graphs of a given size.

also increases accordingly.

**Random Geometric Graph With 320k Vertices**

The 2d unit-square random geometric graph used for this experiment was generated with 320,000 vertices and an \( r(n) \) value of 0.003. The resulting graph has 1,490,855 edges with an average degree of 9.32, maximum degree of 24, and 28 isolated vertices. The graph was partitioned using the K-way partitioning algorithm in Metis. In Figure 60 we plot the edgecut as a function of the number of vertices. We observe that as the number of partitions increase the edgecut also increases, thus our expectation of good performance decreases for large number of partitions. Note that the given graph is rather small for large number of partitions. For example, with 8,192 processors, each processor will be responsible for only about 40 vertices. We restricted the size of the graph in order to preserve the computational time used on Franklin.

Maximum and average execution times for the 320k RGG are shown in Figures 61 and 62 respectively. For each type, the execution time of different phases of the computation are shown separately. The speedup obtained is shown in Figure 63.

It can be observed that while the execution time for initialization and Phase-1 scale with the numbers of processors, the execution time for Phase-2 does not scale well, and drastically increases for processors greater that 1,024. It should be noted that the messages are aggregated only in Phase-1 of our current implementation,
FIG. 57: *Weak scaling for grid graphs*: Series-1 uses the graph size and processor combinations as shown in Table 14.

FIG. 58: *Weak scaling for grid graphs*: Series-2 uses the graph size and processor combinations as shown in Table 14.
FIG. 59: *Edgecut and number of messages for different grid graphs:* The graph size and processor combinations are shown in Table 14.

FIG. 60: *320k RGG: Edgecut as a function of number of vertices.* Actual edgecut for different number of partitions using multi-level K-way partitioning algorithm in Metis.
FIG. 61: 320k RGG: Compute time (maximum). Maximum time is the time in seconds of the slowest processor in the group of processors used to solve the problem.

FIG. 62: 320k RGG: Compute time (average). Average time is the sum of compute time on each processor in the group divided by the number of processors in that group.
FIG. 63: 320k RGG: Speedup.

FIG. 64: 320k RGG: Cardinality after Phase-1.
and the amount of work done per processor becomes very small for larger number of processors. The cardinality of the matching at the end of Phase-1 is plotted in Figure 64. It can be observed that close to 100 per cent cardinality is obtained at the end of Phase-1 for up to 32 processors. As the number of partitions are increased, the cardinality of matching at the end of Phase-1 also decreases resulting in more work during Phase-2. The edgecut as a function of the number of edges is also plotted in Figure 64. It can be observed that the fraction of edges cut increase as the number of partitions increase indicating that amount of communication will grow at large number of partitions.

**SSCA#2 Graph With 524\textit{k} Vertices**

The SSCA#2 graph used for this experiment is generated with with a \( \lambda \) value of 19, and therefore, has \( 2^{19} = 524,288 \) vertices. The number of edges is 10,008,022. The graph is partitioned using the multi-level K-way partitioning algorithm in Metis. In Figure 65 we plot edgecut as a function of number of partitions. It can be observed that the edgecut drastically increases as the number of partitions increases, and therefore, good performance cannot be expected for larger number of partitions.

![Graph showing edgecut as a function of number of vertices](image)

**FIG. 65:** 524\textit{k} SSCA#2: Edgecut as a function of number of vertices. Actual edgecut for different number of partitions using K-way partitioning algorithm in Metis.

Maximum and average execution times for the 524\textit{k} SSCA#2 graph are shown
in Figures 66 and 67 respectively. For each type, the execution time of different phases of the computation are shown separately. The speedup obtained is shown in Figure 68.

The cardinality of the matching at the end of Phase-1 is plotted in Figure 69. It can be observed that close to 100 per cent cardinality is obtained at the end of Phase-1 for up to 512 processors, but it drastically decreases for partitions greater than 512, especially, for 8,192 partitions. The edgecut as a function of the number of edges is also plotted in Figure 69. It can be observed that the fraction of edges cut increase drastically for 4,096 and 8,192 partitions.

FIG. 66: 524k SSCA#2: Compute time (maximum). Maximum time is the time in seconds of the slowest processor in the group of processors used to solve the problem.
FIG. 67: 524k SSCA#2: Compute time (average). Average time is the sum of compute time on each processor in the group divided by the number of processors in that group.

FIG. 68: 524k SSCA#2: Speedup.
FIG. 69: 524k SSCA#2: Cardinality after Phase-1.
V.4.4 Performance of Parallel Matching on Graphs from Applications

We will now provide experimental results of the parallel approximation algorithm for the graphs representing matrices selected randomly from the University of Florida Matrix Collection. Communication in Algorithm PARALLEL-MATCHING-FRAMWORK is directly dependent on the edge-cut for a given number of partitions. Therefore, in order to predict the performance of the algorithm, we will present the edge-cut, for different numbers of partitions, as a percentage of the total number of edges for a graph in Figure 70. It can be observed that edge-cut for Rajat31 and Hamrle3 are under ten per cent, but are relatively high for ASIC-680k, Audikw1 and Cage14.

![Graph showing edge-cut for various applications](image)

FIG. 70: Edgecut for graphs from applications. Percentage of edges cut is a ratio of edgecut to the number of edges in the graph.

We will now present the execution time on Franklin for up to 4,096 processors (Figures 71 and 72). There are a few missing data points in the plots when a particular problem could not be solved for a particular number of processors. For example, Cage14 could not be solved for 512 processors. A major cause of failure has the restriction on the number of messages that a processor can send. Another cause of failure has been the limitation on memory usage, usually during the graph partitioning phase.
FIG. 71: *Graphs from Applications*: Compute time for different matrices with different number of processors. Compute time in seconds ($\log_2$ scale) is plotted on the Y-axis, and the number of processors is plotted on the X-axis. Max is the maximum time on any given processor in the set, and Avg is the average time for a given set of processors.
FIG. 72: Graphs from Applications: Compute time for different matrices with different number of processors. Compute time in seconds (logarithmic scale with base two) is plotted on the Y-axis, and the number of processors is plotted on the X-axis. Max is the maximum time on any given processor in the set, and Avg is the average time for a given number of processors. The Figure also has results for two instances of SSCA#2 graphs.
V.4.5 Analysis of Communication

In this section we will present details about the communication involved in computing the approximation matchings. The total number of messages is bounded between twice and thrice the edgecut. This is plotted in Figures 73 and 74.

FIG. 73: Communication. Total number of messages sent are bounded between twice and thrice the edge cut.

Message Bundling

Message bundling greatly influences performance. Here we show the performance of the message bundling that we have implemented only for Phase 1 of the algorithm. It can be observed that the number of messages that can be bundled in Phase 1, $M_B$ can be given by the relation $(|\text{Edgecut}| \leq M_B \leq 2|\text{Edgecut}|)$. We also know that a lower bound on the total number of messages sent is given by $(2|\text{EdgeCut}|)$. Thus, in a best possible scenario all the messages can be bundled resulting in at most $O(P^2)$ messages, where $P$ is the number of processors. The worst case results from a situation when every processor sends messages to every other processor. However, for graphs with good partitions the communication can be limited to a few processors resulting in a $O(P)$ bound on the number of messages. In Figures 75 and 76, we show the percentage of messages that could be bundled, and the actual number of
FIG. 74: Communication. Total number of messages sent are bounded between twice and thrice the edge cut.

messages sent (bundled, as well as unbundled) as a percentage of total messages that would have been sent if no bundling was performed. It should be noted that the communication time for bundled messages will be proportional to the number of messages bundled. Thus, bundled messages are sensitive to both latency and bandwidth of the underlying communication system. In the implementation, bundled messages are sent using the MPI construct MPI.Isend(), and unbundled messages are sent using MPI construct MPI.Bsend().
FIG. 75: Message Bundling. Percentage bundled represents the number of messages that could be bundled in Phase 1, higher the better. Percentage sent represents the actual number of messages that get sent due to bundling, lower the better.

FIG. 76: Message Bundling. Percentage bundled represents the number of messages that could be bundled in Phase 1, higher the better. Percentage sent represents the actual number of messages that get sent due to bundling, lower the better.
In this chapter we presented a parallel \( \frac{1}{2} \)-approx algorithm and discussed the experimental analysis on a distributed memory system. The proposed algorithm has several limitations. The limitations that directly affect performance are the structure of the graph and the edgecut resulting from partitioning the graph between multiple processors. A worst case for the number of rounds of execution can be illustrated by executing the pointer-based algorithm on a graph whose edges can be arranged on a straight line with edge-weights in a sorted order as shown in Figure 77. The number of rounds in this case is \( O(|E|) \). However, with the assumption of random edge-weights, the expected number of rounds is \( O(\log |E|) \), where \( E \) represents the set of edges in a graph.

![Diagram](image)

FIG. 77: Limitations of the pointer-based approach. (a) The input graph \( G = (V, E) \) with weights associated with the edges; (b) an intermediate step of execution where the pointers are set for each vertex in the graph; (c) an intermediate step where vertices that are pointing to each other are matched. Bold lines represent matched edges. Dashed lines represent the edges removed from the graph; (d) the final state. Matched vertices are colored black.

There are numerous challenges in implementing and executing the algorithm on current and future supercomputers with hundreds of thousands of processors. One specific challenge is the edgecut that dictates the execution time for Phase-2. Speculative algorithms can help minimize communication and we will explore this in our future work. We will also explore the benefits of alternative platforms with fast interconnects and slow processors. Better algorithms for unweighted and vertex-weighted
matching problems will also be explored in future work.

Acknowledgements: This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
CHAPTER VI
CONCLUSIONS AND FUTURE WORK

“Art is never finished, only abandoned.” - Leonardo da Vinci

The work completed in this thesis lays the groundwork for future research. The goals for this research were broadly organized into theory, implementation and applications. We were able to accomplish many of the goals we set for ourselves. The following list provides a summary of the contributions from this work:

1. Theory:
   - New framework for developing proof of correctness for vertex weighted matchings;
   - New $\frac{1}{2}$-approx algorithms for vertex weighted matchings;
   - New $\frac{2}{3}$-approx algorithm for bipartite vertex weighted matchings;

2. Experiments:
   - Open-source library of C++ routines to compute various kinds of matchings;
   - Open-source library of C++ and MPI routines to compute approximate matchings in parallel.
   - Extensive experimental study of various (serial) matching algorithms, and scalability study of $\frac{1}{2}$-approx parallel algorithm with up to 8,192 processors.

3. Applications:
   - Study of applicability of vertex weighted matchings in solving the sparsest basis problem.
   - Study of approximation algorithms in sparse matrix computations.
Constrained by time and priorities we have also left many questions unanswered. Some of the important open problems that will be addressed in the future work include

- How to provide a proof of correctness for $\frac{2}{3}$-approx algorithm LOCAL-TWO-THIRD?
- Is a $\frac{2}{3}$-approx algorithm possible for vertex weighted matching in general graphs?
- Is a $\frac{3}{4}$-approx algorithm possible for vertex weighted matching in bipartite and/or general graphs?

VI.1 FUTURE WORK

Preliminary work on a parallel approximate matching was completed as part of this research. The need for efficient parallel implementations has never been greater than now. As part of our future work we plan to continue to improve the current implementation, develop new algorithms - exact as well as approximate, and conduct scalability studies on different parallel architectures. Some specific goals for immediate future include:

- Conduct scalability studies on IBM Bluegene/P system at Argonne Leadership Computing Facility (ALCF), at the Argonne National Laboratory.
- Conduct scalability studies on SiCortex 5832 system Green at Rosen Center for Advanced Computing, Purdue University.
- Study impact on performance from different partitioning schemes.
BIBLIOGRAPHY


