Extending Search Phases in the Micali-Vazirani Algorithm*

Michael Huang[1] and Clifford Stein[2]

1 **Department of IEOR, Columbia University, New York, NY, USA**
`mh3166@columbia.edu`

2 **Department of IEOR, Columbia University, New York, NY, USA**
`cliff@ieor.columbia.edu`

Abstract

The Micali-Vazirani algorithm is an augmenting path algorithm that offers the best theoretical runtime of $O(n^{0.5}m)$ for solving the maximum cardinality matching problem for non-bipartite graphs. This paper builds upon the algorithm by focusing on the bottleneck caused by its search phase structure and proposes a new implementation that improves efficiency by extending the search phases in order to find more augmenting paths. Experiments on different types of randomly generated and real world graphs demonstrate this new implementation's effectiveness and limitations.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases matching, graph algorithm, experimental evaluation, non-bipartite

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.10

1 Introduction

Matching, a commonly studied problem in the field of algorithms, has a variety of applications in other fields such as bioinformatics, computer science, statistics, and operations research. The paper focuses on a new efficient implementation for finding the maximum cardinality matching for non-bipartite graphs. Additionally, our experiments on non-bipartite graphs generated from historical NYC taxi cab trip data, demonstrate non-bipartite matching in solving consumer matching problems like forming carpools.

Non-bipartite matching has been a well studied problem. Since first $O(n^4)$ algorithm provided by Edmonds' [3] in 1965 there have been contributions from Gabow [5], Kameda and Munro [9], Lawler [11], Even and Kariv [4], and Micali and Vazirani [15]. The Micali-Vazirani (MV) algorithm remains the most efficient algorithm for non-bipartite graphs with the same $O(\sqrt{n}m)$ Hopcroft and Karp runtime [8] for bipartite graphs. Implementation studies from the first DIMACS Implementation Challenge [2, 14] showed that the algorithm is efficient in practice as well by comparing it against Gabow's $O(n^3)$ implementation. In a more recent study by Kececioglu and Pecqueur [10], its performance was still better against Tarjan's $O(mn\,\alpha(m,n))$ implementation, but performed slightly worse compared to the modified $O(mn\,\alpha(m,n))$ implementation presented.

In this study, we improve the MV algorithm by testing various implementations on different graph types. Since the MV algorithm is an augmenting path algorithm, previous

* This work was partially supported by NSF grant CCF-1421161.

works have suggested improving the performance by reducing the number of phases. We also reduce the number of phases. Our key idea is that, in each phase, we find extra augmenting paths in addition to the maximal set of shortest edge-disjoint augmenting paths. This additional work only takes $O(m)$ time and, in many cases, significantly reduces the number of phases of the algorithm. We also consider several previously suggested improvements and find that one, greedy initialization, also helps reduce the number of phases. Using both of these improvements we obtain significant reduction in the number of phases. While the worst case running time remains, the observed running time is much better.

2 Algorithm

2.1 Basic Definitions

A *matching* for an undirected graph $G = (V, E)$ is a set of edges M such that no two edges meet at a vertex. We designated edges and the corresponding vertices in M and not in M as matched and unmatched edges/vertices, respectively. The maximum cardinality matching problem finds any matching such that the number of edges in the set is maximized. Many algorithms that solve this problem search for augmenting paths. An *alternating path* is a simple path that alternates between edges in M and in $E - M$. An *augmenting path* is an alternating path that starts and ends with unmatched vertices. We *augment* by changing unmatched edges to matched edges and vice versa for all edges on the augmenting path–increasing the cardinality of the matching increases by one. When there are no more augmenting paths, a maximum matching is found.

Matching in bipartite and non-bipartite graphs qualitatively differ. In the first type, all lengths of alternating paths to a matched vertex v have the same *parity* (odd or even), where as matched vertices in second type can be both–which lead to odd length alternating cycles. This makes matching more difficult for non-bipartite graph, but Edmonds addressed this challenge by introducing structures called *blossoms* to handle odd-length cycles. In the MV algorithm, *petals* are the equivalent structure and are essential for achieving the efficient runtime.

2.2 Micali-Vazirani Algorithm

We will give a brief description of the Micali-Vazirani Algorithm based on Vazirani's most recent paper on the topic [16]. This paper presented a complete proof of the runtime and defined some new terms that we will use to describe the finer details of the algorithm.

2.2.1 Key Concepts

First we provide concepts specific to the MV algorithm. Below are relevant definitions that are used in the high level description of the MV algorithm in Listing 1.

▶ **Definition 1** (Evenlevel and oddlevel of v). An evenlevel/oddlevel is the length of the shortest even/odd alternating path from an unmatched vertex to v, denoted as evenlevel(v) and oddlevel(v), respectively.

▶ **Definition 2** (Minlevel and maxlevel of v). The minlevel of v, denoted as minlevel(v), is the minimum of evenlevel(v) and oddlevel(v). Similarly, the maxlevel is the maximum of evenlevel(v) and oddlevel(v).

Listing 1 Micali-Vazirani Algorithm pseudocode.

```
1  Set initial greedy matching for G
2  Reset edge labels
3  Set minlevel = 0 and maxlevel = $\infty$ for each unmatched vertex
4  Set minlevel = $\infty$ and maxlevel = $\infty$ for each matched vertex
5  Set level = 0
6  If there exist u such that maxlevel(u) == level or minlevel(u) ==
       ↪ level then continue, else go to line 31
7  For each u such that maxlevel(u) == level or minlevel(u) == level:
8    For each unscanned (u,v) with appropriate edge parity:
9      If minlevel(v) $\geq$ level + 1 then,
10       Set minlevel(v) = level + 1
11       Add u to the list of predecessors of u
12       Label (u,v) as prop
13     Else,
14       label (u,v) as bridge
15       If tenacity((u,v)) != $\infty$ then
16         Add (u,v) to the list of bridges with the same tenacity
17 For each bridge of tenacity == 2*level + 1:
18   Find support using DDFS
19   If bottleneck found then
20     Augment alternating path
21     Delete the vertices in the augmented path and all vertices that
           ↪ are orphanned (no predcessors) as a result
22   Else,
23     For each v in the support:
24       Set maxlevel(v) = 2*level + 1 - minlevel(v)
25       If v is an inner vertex then
26         For all incident (v,u) which are not props:
27           If tenacity((u,v)) != $\infty$ then
28             Add (u,v) to the list of bridges with the same tenacity
29 Set level = level + 1
30 If augmentation occured then go back to line 2, else go to line 6
31 Return the current matching
```

▶ **Definition 3** (Inner and outer vertices)**.** A vertex is outer if oddlevel(v) > evenlevel(v) and inner otherwise.

▶ **Definition 4** (Tenacity)**.** Tenacity of vertex v is defined as, tenacity(v) = evenlevel(v) + oddlevel(v). Tenacity of edge (u, v) is defined as, tenacity$((u, v))$ = oddlevel(u) + oddlevel(v) + 1 for an matched edge and tenacity$((u, v))$ = evenlevel(u) + evenlevel(v) + 1 for an unmatched edge.

▶ **Definition 5** (Predecessor, prop, and bridge)**.** For any edge (u, v) such that minlevel(v) == minlevel(u) + 1, u is defined to be a predecessor of v. Any edge that joins a vertex and its predecessor is defined as a prop. If an edge is not a prop, then it is a bridge.

▶ **Definition 6** (Support of a bridge)**.** If (u, v) is a bridge of tenacity t, then the support is defined as $\{w|\text{tenacity}(w) = t \text{ and } \exists \text{ a maxlevel}(w) \text{ path containing } (u, v)\}$

The *double depth first search* (ddfs) algorithm is also specifically emphasized in Vazirani's paper, since it is an essential method for finding augmenting paths and forming petals. The ddfs finds disjoint paths to the root nodes from any pair of vertices in a level graph. In the MV algorithm, if such paths exist then an augmenting path is found, otherwise there is a *bottleneck* and we form a petal.

2.2.2 Algorithm Description

The MV algorithm is a non-bipartite matching algorithm that operates in phases. Each phase finds a maximal set of vertex disjoint shortest length augmenting paths. Like the Hopcroft-Karp algorithm for bipartite graphs, each phase synchronously constructs a level graph using breadth-first search from unmatched vertices to find alternating paths. Every time the level graph expands, the algorithm identifies bridges and performs double depth first searches on them to check for augmenting paths and to form petals. After performing the double depth first searches at the current level, the phase ends if a path was augmented. The algorithm terminates once we search the entire graph and do not find an augmenting path.

Below is a general overview of the MV algorithm in Listing 1:

- Lines 5–30: One complete phase
- Lines 6–16: The synchronous breadth-first search
- Lines 17–28: Using ddfs to find augmenting paths and forming petals
- Lines 30: Phase termination condition
- Lines 6: Algorithm termination condition

Since each phase ends when the maximum set of vertex disjoint minimum length alternating path is augmented, there are at most $\sqrt{n}$ phases. The algorithm also guarantees $O(m)$ [6] runtime per phase which leads to the overall runtime of $O(m\sqrt{n})$.

2.3 Past Work

In Kececioglu and Pecqueur [10] a new $O(mn\,\alpha(m,n))$ time implementation of Edmonds' algorithm demonstrated that including simple heuristics in the algorithm could significantly impact runtime. While the MV algorithm is theoretically the most efficient algorithm known for non-bipartite matching, experimental studies have identified potential inefficiencies in implementation [14]. Due to the heuristics' success in improving the Edmonds' algorithm, we considered three analogous heuristics and improvements to target these inefficiencies of the MV algorithm.

2.3.1 Greedy matching initialization

It is a natural idea to "initialize" a maximum matching algorithm with a greedy (maximal) matching. Kececioglu [10] considered initializing the modified Edmonds' algorithm with different greedy matching algorithms since starting with a larger matching would reduce the number of augmenting paths needed to be found. The function is the same in the MV algorithm, but its impact may be greater since we would start with fewer search trees which would reduce the amount of searching per phase in addition to reducing the total number of phases.

2.3.2 Order of bridge processing

Bridge processing potentially impacts the performance of the MV algorithm as well. Mattingly [14] provided examples demonstrating the importance of bridge processing order towards scenarios where a optimal matching could be found in a single phase. We can extend this observation and note that even if a maximum matching cannot be found in the phase, the order processing bridges can affect phase performance. By processing bridges that lead to augmentations first, we can avoid processing future bridges that may be topologically deleted. Changing the order of bridge processing leads to not only fewer phases as Mattingly suggested, but also shorter phases.

2.3.3 Blossom formation

In previous work, Crocker [2] noted that blossom formation limited the performance of Gabow's implementation of the Edmonds' blossom algorithm. Kececioglu [10] addressed the issue with heuristics to avoid or delay blossom related operations for Tarjan's implementation of Edmonds' blossom algorithm. While these heuristics failed to provide significant performance boosts, it may improve the performance of the MV algorithm. This is closely related the order of bridge processing since we may be able to avoid forming petals if the related bridges are deleted after an augmentation.

2.4 Preliminary Results

In initial trials, only greedy matching initialization significantly reduced the number of phases of the MV algorithm. For determining the best order for processing trees, difficulty arose from identifying which alternating paths should be augmented. Attempts were made by considering the number different free nodes and alternating paths that were associated with a bridge, however, tracking that information was complicated and priority was hard to determine since we could only compare bridges associated with alternating paths of the same length. Reducing blossom formation was unsuccessful as well since phases without augmentations need blossoms to be formed in order to construct the level graph properly, meaning the gains from reducing the number of blossoms formed is diminished in future phases.

2.5 Algorithmic contribution

The Micali-Vazirani algorithm can be improved either by reducing the number of phases or improving the efficiency of the phases themselves. Due to the bottlenecks analyzed and the previous work of others, this paper mainly focuses on the former rather than the latter and achieves this by proposing broader algorithmic changes rather than technical/computational changes. As a result, the improvements in the number of phases are unaffected by the computing environment and can be directly compared to the theoretical runtime.

2.5.1 Motivation

The primary inefficiency examined in this paper relates to the termination conditions of each phase. In previous works, it has been noted that the MV algorithm's performance is closely related to the number of phases due to the high overhead in initiating phases[2]. As the minimum alternating paths get longer, the time to grow each individual search tree from the unmatched vertices increases and the number of paths augmented per phase decreases. The initialization for each phase becomes inefficient if we continually reconstruct the same trees and do not find an augmenting path.

An avenue to attack this inefficiency is to preserve the search trees by continuing the search phase. Figure 1 provides motivation for why this works. The dashed edges denote matches while the solid edges are unmatched edges. As the graph continues to the right, the number of edges separating the triangles grows so that the edges between the kth and $k+1$th triangle is greater than the number of edges between the $k-1$th and kth triangle.

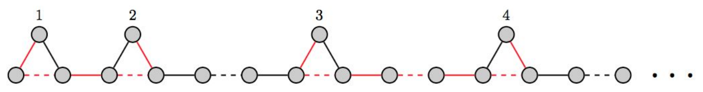

Figure 1 The red edges form the augmenting path between the free vertices at the top of each triangle.

In this scenario, the greedy initialization of the graph leaves vertices 1, 2, 3, and 4 unmatched. A phase of the base MV algorithm terminates after we augment the path between vertices 1 and 2 in the figure. Even though the search trees of vertices 3 and 4 remain unaffected by the augmentation, the phase resets and the progress towards finding the path between these two vertices is discarded. Extending the search phase would allow the augmenting path between 3 and 4 to be discovered in the next search level instead of the next phase. Assuming the numbering continues in the figure, nodes $2n - 1$ and $2n$ will be matched in the same phase as well. Thus, instead of augmenting the path between the $2n - 1$ and $2n$ in the nth phase, we augment the paths for all pairs in the first phase.

2.5.2 Termination conditions

Our primary contribution to the MV algorithm targets the termination bottleneck by not resetting phases even after an augmentation occurs. To describe the changes to the algorithm we reference the pseudocode in Listing 1 and the actual Python code. Below are the names of the key procedures in the Python code and the corresponding lines in the pseudocode.

- MIN – This is the search procedure that constructs the level graph and corresponds to lines 7–16 in Listing 1. The MIN procedure in the Python code is provided an array of vertices to search every phase. These vertices are set in the previous MIN procedures and in the previous MAX procedures. If the list is empty at the start of the procedure, the MV algorithm terminates.
- MAX – This is the bridge processing procedure that forms petals and augments alternating paths. It corresponds to lines 17–28 in Listing 1. When petals are formed, vertices are given their maxlabel and are added to the list of nodes for MIN to process. If augmentation occurs the current phase terminates.

In the MV algorithm, each phase alternates between running the MIN and MAX procedure until the termination condition in MIN or MAX is realized. Our modification removes the augmentation termination condition (line 30) in MAX and replaces it with the condition that the phase will terminate after running a designated number of MAX procedures that augmented at least one path. We can set this value so that each phase terminates only when the end condition is met by MIN. The extended phase MV algorithm pseudocode can be seen in Listing 2 in the appendix.

The modification allows the algorithm to preserve work put into constructing the level graph in the current phase. Additionally, the level graph that remains after augmentation preserves the search trees from unmatched nodes that were disjoint from the augmented paths since it topologically deletes the search trees of the matched nodes. This results in the selection of a maximal set of augmenting paths during each phase.

The three principles/observations that make the modification immediately attractive are:

1. It is guaranteed to not have more phases than the original algorithm.
2. In earlier phases, the alternating paths are shorter. That means the total number of alternating paths that intersect with augmented paths is lower and the maximal matching will find a larger fraction of the remaining matches.

3. In later phases, the number of alternating paths becomes more sparse as the number of free vertices decreases and are more likely to have different lengths. It also becomes less likely that augmenting one path will impact other search trees. Extending the phases saves in reconstructing unused search trees in a manner similar to the motivational example discussed earlier.

3 Experiments

In past experimental studies, the best algorithm in practice was Kececioglu's modification of Tarjan's $O(mn\,\alpha(m,n))$ algorithm, which ran roughly 4 times faster than Crocker's MV algorithm[10]. Our experiments will primarily focus on measuring the relative improvements that can be made on the MV algorithm by reducing the number of phases. The relative measure can provide insight on how it will perform against Kececioglu's algorithm.

Since the main goal focuses on methods reducing the number of search phases, less effort was put into the implementation and selection of data structures that could optimize the performance of each individual search phase. In order to remain consistent, we use the same implementation of the search phase for each variant of the algorithm.

The implementation was written in Python 2.7.10 and utilizes the time, csv, NumPy, and NetworkX modules. The majority of the experiments were conducted on Columbia Business School research grid. For larger graphs that required more memory to store the node and edge data, we used a Intel Xeon E5-2667 processor with 256 GB of main memory. The different implementations and experiments are available at:

`https://github.com/mh3166/Extended_MV_algorithm`

3.1 Variants/Implementations

For our experiment, we present two different modifications:

1. The first chooses a different initialization method. We construct the initial maximal matching using the heuristics and reductions discussed in Magun's experimental work with greedy matching algorithms [13]. Like our simple greedy algorithm, every time an edge is added to the the matching, the adjacent vertices and the adjacent edges are removed from the graph. We then find the maximal matching for the remaining graph. The new greedy algorithm performs 1 reductions which means that in the current graph, the edges of vertices of degree 1 must be included in the maximal matching. It also uses a heuristic (referred to as Heuristic 1), which states that each edge we add to the maximal matching must include a node with minimum degree in the current graph, and another heuristic, which states that the opposite node to the one in Heuristic 1 must have minimum degree among the neighboring vertices.
2. The second is our algorithmic contribution that replaces the termination condition for the MAX phase of the MV algorithm with a more relaxed version that only terminates a phase after it encounters a specified number of MAX phases that augmented a path. In the experiment, this number was 100 and was never reached in any trial.
   Let the following be notation for the different variants of the MV algorithm.
   a. **MV0** is the base algorithm
   b. **MV1** is the variant with modification **1.**
   c. **MV2** is the variant with modification **2.**
   d. **MV3** is the variant with modification **1.** and modification **2.**

3.2 Graphs

Building upon the previous MV algorithm work of Crocker [2] and Mattingly and Ritchey [14], we examined the graphs of previous experiments that had non-trivial processing times. Given that we know the structure of these graphs, the results can provide insight into the improved performance. We also tested the algorithm on real world graphs that provided significant challenge to the original algorithm in order to demonstrate the improved algorithm's robustness. Below are the selected graph:

1. **Random graphs** with n nodes and with expected degree d. This is a Erdős-Rényi graph, a binomial graph that was generated with the fast_gnp_ random_graph method in NetworkX [7]. It takes the inputs n and $p = \frac{d}{n}$. Following Crocker [2], we chose the average degrees that differed by factors of $2^{1/16}$ and chose a range of $0 \leq d \leq 8$.
2. **Grid graphs** with n^2 nodes with expected degree d. The grid graph is a $n \times n$ lattice graph that has some of the edges removed. Rather than having each node be adjacent to four edges, we construct it so that the average degree d of the graph is $2 \leq d \leq 4$. The graph is constructed by visiting each node and adding each of the four edges to the graph with probability $p = 1 - (1 - \frac{d}{4})^{1/2}$ which accounts for visiting each edge twice. We test sizes for $4 \leq n \leq 10$.
3. **One-connected triangles** with 2^k triangles. The graph has 2^k vertex disjoint triangles. These triangles are then interconnected by only one edge. To construct the graph, we add the following edges

$$(3i, 3i+1), (3i+1, 3i+2), (3i+2, 3i) \quad 0 \leq i \leq 2^k - 1,$$

$$(3i + (i \bmod 3), 3i + 3 + (i \bmod 3)) \quad 0 \leq i \leq 2^k - 2,$$

   where $\{0, \ldots, 3 \cdot 2^k - 1\}$ is the set of vertices. Since the algorithm will process the edges and nodes in numerical order, the graphs are randomized by switching numbering while maintaining the same structure. We test for $10 \leq k \leq 20$.
4. **Three-connected triangles** with 2^k triangles. The graph also has 2^k vertex disjoint triangles. The triangles are now interconnected by three edges instead of one. To construct the graph, we add the following edges

$$(3i, 3i+1), (3i+1, 3i+2), (3i+2, 3i) \quad 0 \leq i \leq 2^k - 1,$$

$$(3i, 3i+3), (3i+1, 3i+4), (3i+2, 3i+5) \quad 0 \leq i \leq 2^k - 2,$$

   where $\{0, \ldots, 3 \cdot 2^k - 1\}$ is the set of vertices. Again we randomize the graph by randomizing the labels of the nodes. We test for $10 \leq k \leq 14$. The lower range is because we run into recursion issues when opening petals in graphs with $k \geq 15$.
5. **Real World Graphs** are composed of selections from Stanford's Large Network Dataset Collection [12] as well as graphs constructed by the NYC taxi cab data from 2015 as shown in Table 1. From Stanford's data set we chose network graphs showing representing Amazon's product co-purchasing network as of certain dates. Most of these graphs were only selected for their size and average degree rather than for their practical application. From the NYC taxi cab data, we constructed a more realistic matching problem the graph by finding taxicab passengers who were close in departure time and location and were headed in a similar direction. See the github link for the code that generated these graphs. The maximum matching in this graph would provide the maximum number of carpooling opportunities in NYC in a day.

Table 1 Description of Real World Graphs.

Network Graph	$\|N\|$	$\|E\|$	Avg. Deg.
Amazon Co-Purchasing 3/2/03	262111	899792	3.43
Amazon Co-Purchasing 3/12/03	400727	2349869	5.86
Amazon Co-Purchasing 5/5/2003	410236	2439437	5.95
Amazon Co-Purchasing 6/1/2003	403394	2443408	6.06
2/1/15 Taxi Data	325109	952974	2.93
2/2/15 Taxi Data	569599	1487866	2.61
2/4/15 Taxi Data	1216990	3414986	2.81
2/5/15 Taxi Data	1578057	4564025	2.89
2/7/15 Taxi Data	2335680	6993447	2.99

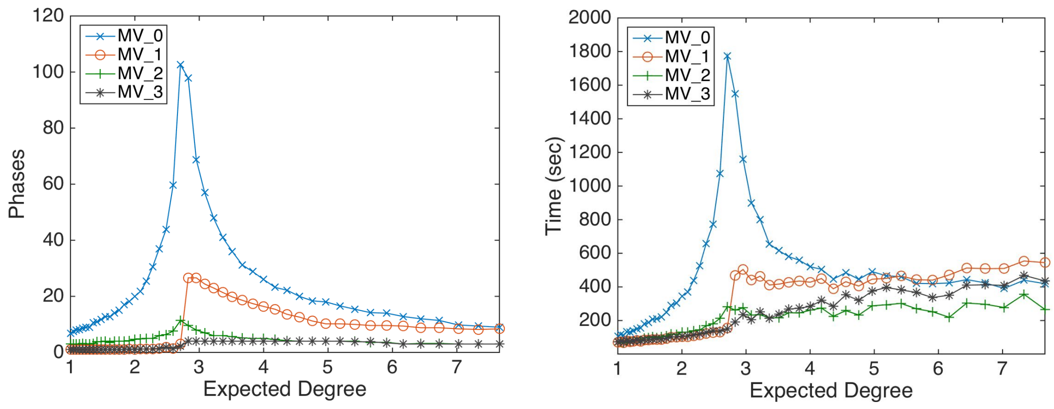

Figure 2 Average search phases and running time as the expected degree changes for random graphs with 2^{20} nodes.

4 Results

In this section, we observe the relationship between the structure of the graphs and the performance of the algorithms.

Overall, we see phase reduction and thus lower running time when utilizing our modification of extending phases. Changing the greedy initialization also has significant impact on phases (~60% reduction at best) and running time as well, but we get the most consistent performance in phase reduction when combining the two modifications as seen by MV3. In the few cases where MV3 performs worse than MV2, we are mainly hampered by overhead in our modified greedy initialization.

While we see overall improvement, we also see noticeable differences in performance on different graphs.

4.1 Random Graphs

The random graphs experiment provides a starting point to analyze what could reduce the number of phases in the MV algorithm. As seen in Figure 2, we replicated the observation from Crocker [2] that the number of phases peaks at degree ~3 and empirically observed the proposition by Bast[1] that with high probability the length of the maximum augmenting path–and thus the maximum number of phases–for any non-maximum matching is $O(\log n)$

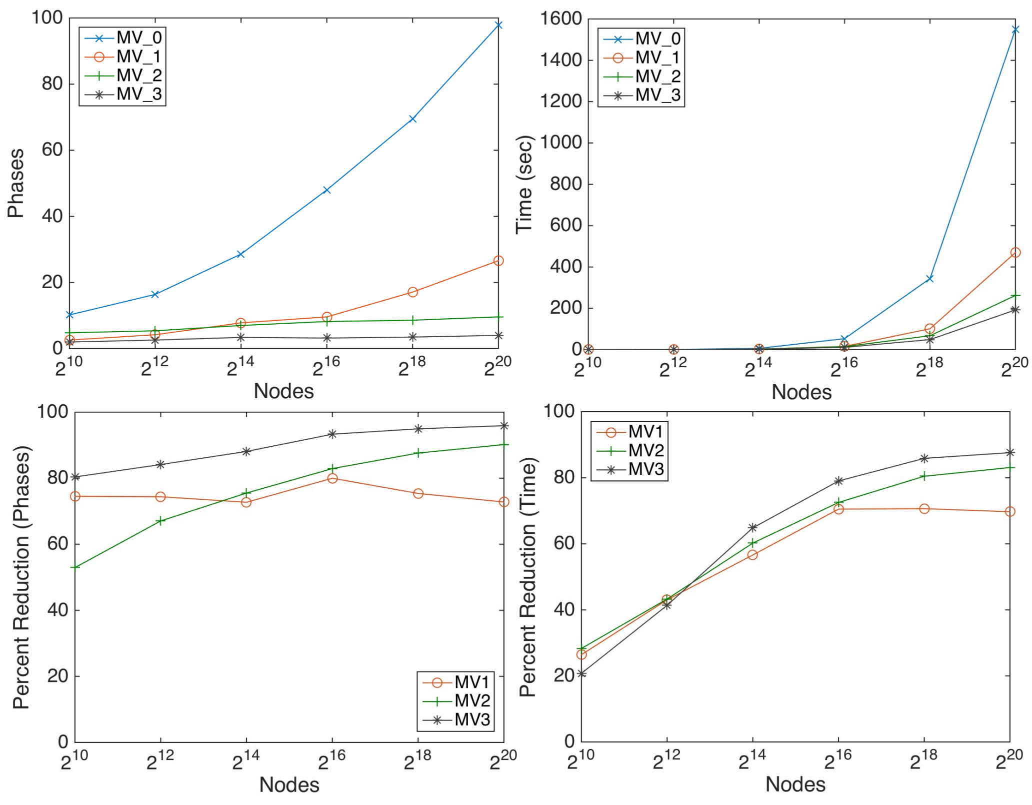

Figure 3 Average percent reduction of search phases and running time as the number of nodes increases for random graphs with an expected degree of $\sqrt{8}$.

for random graphs with average degree above some constant c. In this experiment, we also discovered that utilizing an initialization heuristic alone can reduce phases and that it can also improve the performance of the MV algorithm with extended search phases.

From Magun [13], we know that compared to initializing with a random maximal matching, both the starting number of matches and average degree of the free vertices are more likely to be greater. That implies the MV algorithm could be receiving a performance boost from the fewer matches to be found or the larger number of alternating paths per free vertex.

4.2 Grid graphs

To further investigate the effect of search phase extension and the initialization heuristic, we can compare the random graph experiment to the grid graph experiment by comparing Figure 4 to Figure 3. In the worst case examples, grid graphs require significantly more phases and thus more time to be solved compared to random graphs. Additionally, the initialization heuristic has little effect in improving performance as graph size grows. While we again see that the worst case performance occurs in randomly generated graphs with an average degree of 3, it is clear that the limited range of degrees for vertices has a negative effect on the performance of the MV algorithm.

4.3 One-connected Triangles

One-connected triangle graphs provide an even more extreme scenario where simple structures result in high phase costs for the MV algorithm. In Figure 5, we see that applying the

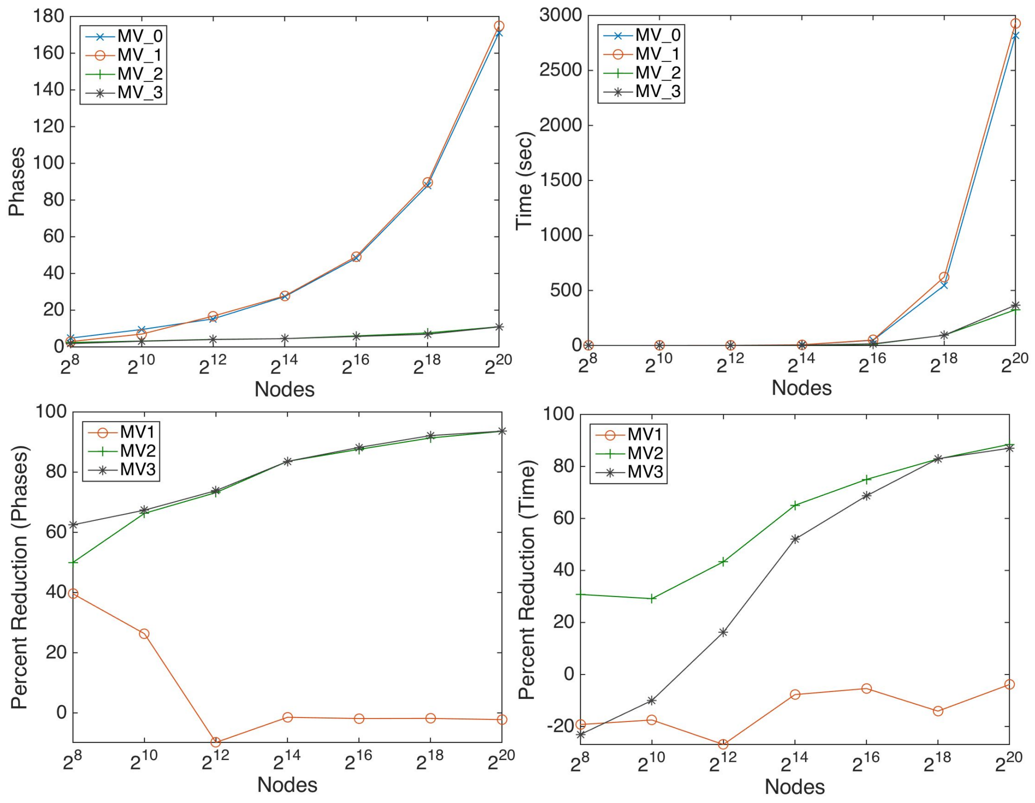

Figure 4 Average percent reduction of search phases and running time as the number of nodes increases for grid graphs with expected degree of 3.12.

initialization heuristic helps deal with that issue by consistently reducing the number of phases by 60%. However, by extending phases, the MV2 and MV3 algorithm perform significantly better by reducing the number of phases by over a factor of 50 as seen in Table 2. This type of graph closely resembles the motivational graph in Figure 1 and demonstrates the type of graphs and subgraphs that benefit from extending search phases. See the appendix for further discussion.

4.4 Three-connected Triangles

The Three-connected triangle graph experiment had similar results to the one-connected version in terms of the phase count. We include this experiment to demonstrate the effect of nested petals on performance. In the algorithm, the nested structure requires recursive calls to open petals in order to find augmenting paths. From a technical standpoint, in larger graphs this triggers the default maximum recursion depth safeguard for Python which terminates the algorithm early. This problem can be avoided by implementing a non-recursive method for finding augmenting paths as discussed by Mattingly [14].

4.5 Real World Graphs

In Table 3, we see that the benefits of the greedy initialization and phase extension apply to real world graphs. Our results primarily focused on solving the maximum cardinality matching problem for large graphs where each phase is computationally expensive. The

Table 2 Average runtime and phase for one connected triangle graphs as number of triangles increase.

	Average Runtime (sec)				Average Phases			
Triangles	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^{10}	0.923	0.374	0.34	0.327	24.1	10	4	3.7
2^{12}	7.734	2.627	2.019	2.04	50.5	18	4.8	4.2
2^{14}	89.298	24.088	11.175	11.002	103.1	35.6	5.4	5.1
2^{16}	829.863	195.934	53.978	52.98	200.8	70.2	6.3	5.8
2^{18}	6185.386	1668.073	238.776	221.504	410.2	140.1	8.4	6.5

Table 3 Search phases and running time for different real world graphs.

	Average Runtime (sec)				Average Phases			
Network Graph	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
Amazon Co-Purchasing 3/2/03	178.89	194.36	66.23	118.12	21	20	4	5
Amazon Co-Purchasing 3/12/03	398.23	466.23	227.72	248.53	18	19	5	3
Amazon Co-Purchasing 5/5/2003	380.5	422.09	191.73	256.7	18	17	5	4
Amazon Co-Purchasing 6/1/2003	400.7	469.1	260.45	287.07	17	18	5	4
2/1/15 Taxi Data	622.61	458.32	187.15	141.46	53	42	10	6
2/2/15 Taxi Data	972.02	946.72	315.67	257.57	56	48	11	7
2/4/15 Taxi Data	3142.8	2852.5	724.79	640.93	72	59	10	7
2/5/15 Taxi Data	5277.9	4552.8	1158.1	1035.1	75	63	10	7
2/7/15 Taxi Data	9480.2	8593.6	1868.8	1546.5	81	75	11	7

results of the real world graphs fall in line with the artificially created graphs. The Amazon Co-Purchasing graphs demonstrate the higher average degree correlates to fewer phases observation discovered in the random graph experiments. The modifications to the MV algorithm thus have a lesser impact towards improving performance. The taxi graph results provide an example of graphs whose performance is dictated by degree and structure. While the higher degree reduces the number of phases needed compared to that of the worst case random graphs, the number of phases needed does not decrease at the same rate. Thus, we see significant improvement similar to that of artificial graphs that have a specific structure. These real world graphs show that our modifications are effective against structure that hampers the MV algorithm.

5 Discussion

5.1 Runtime

Through our experiments we have seen that the removal of the termination condition in MAX greatly reduces the number of phases the algorithm must be processed. When we plot the growth of number of phases vs. the log size of the problem in Figure 6, we see that there is a linear relationship. Thus, it seems that the number of phases grows in $O(\log n)$.

Additionally, in our experiments, after obtaining the maximum matching, we also were able to calculate the fraction of remaining matches that were processed during each phase. Figure 7 plots the average fraction found per phase for one-triangle connected graphs as the graph size grows in number of triangles. We see that the algorithms with the extended phases can better preserve the $O(\log n)$ number of phases by consistently finding a high fraction of remaining matches per phase.

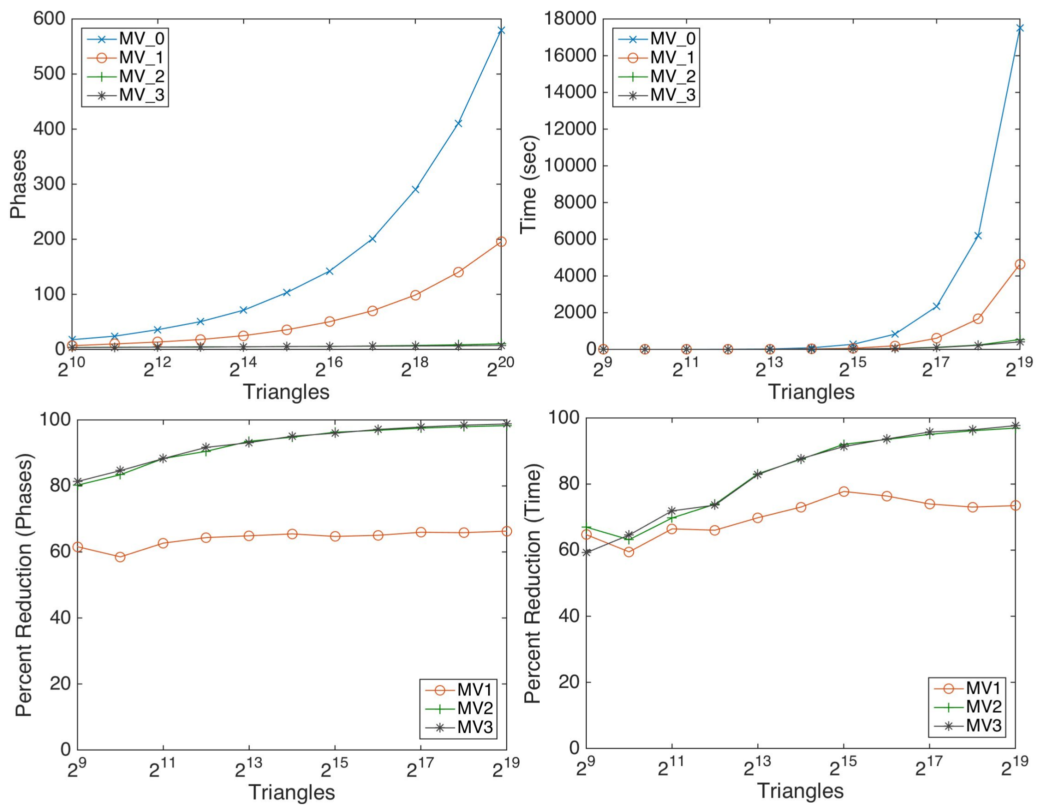

Figure 5 Average percent reduction of search phases and running time as the number of triangles increases for one-connected triangle graphs.

5.2 Worst Case Graph

From our experiments we have only seen graphs that perform well after making the algorithmic modification to the MV algorithm. Figure 8 is an example of a type of graph where the worst case runtime is $O(\sqrt{n}m)$.

With the greedy matching shown after the initial phase, both the base and the modified algorithm will have a $O(\sqrt{n}m)$ runtime if it first augments the path that intersects all other alternating paths. The resulting new graph again has a alternating path that intersects all other alternating paths. In each phase, there is the possibility that only one path will be augmented. Since we start with n matchings to be found and the starting graph is of size $O(n^2)$, we obtain the $O(\sqrt{n})$ number of phases. In this case, the modified algorithms have the same performance as the base MV algorithm.

6 Conclusion

We introduced a new implementation of the Micali-Vazirani algorithm that effectively reduced the number of search phases and demonstrated its effectiveness on a variety of graph types. Our primary contribution considers extending search phases which in our experiment reduces the number of phases by at least 40% in smaller graphs and up to 98% in larger graphs. The phase reduction translates well to runtime when the phase improvement is large. This saves in overhead cost from reseting the graph each phase. For cases where the graphs could already be solved by the base MV algorithm in a few phases, runtime improvement is

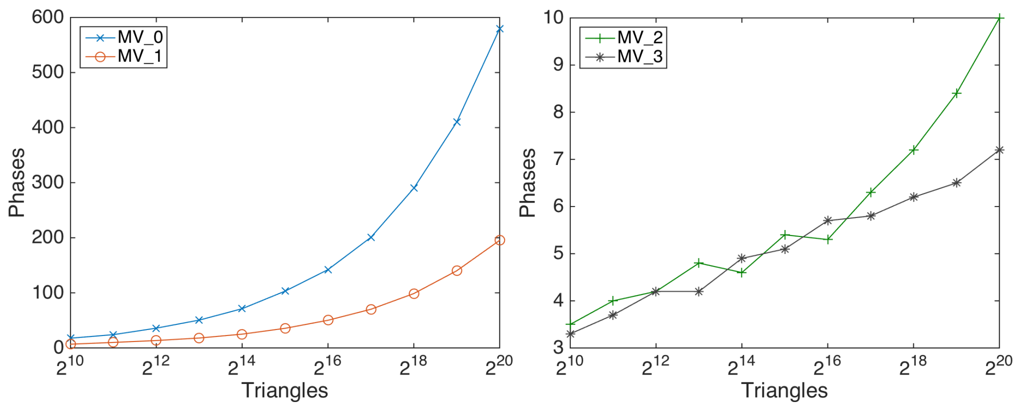

Figure 6 Comparison of phase growth with and without the termination condition in MAX phase.

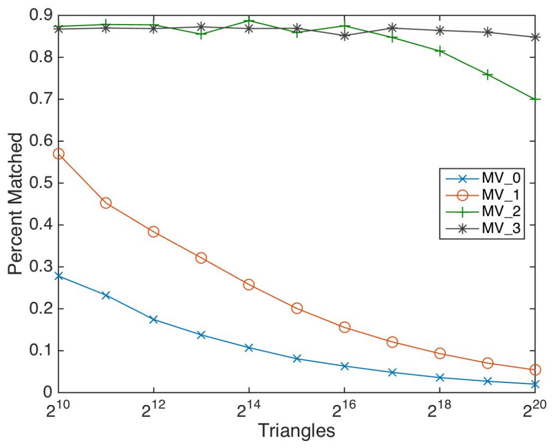

Figure 7 Comparison of average percent of remaining matches found per phase.

smaller since the algorithm is already using each phase efficiently. In the future, we hope to improve the more technical aspects of the implementation, such as choosing more efficient data structures and cleaning up any inefficiencies in the code. A fully optimized version would allow for better comparison with matching algorithms in previous works as well as matching algorithms exclusive to bipartite graphs. From a more theoretical standpoint, we would also like to study how the structure of the graphs impacts the performance of the modified MV algorithm. This may provide insight that could result in more significant contributions that could improve the performance of the MV algorithm.

References

1 Holger Bast, Kurt Mehlhorn, Guido Schafer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. *Theory of Computing Systems*, 39(1):3–14, 2006.

2 Steven T. Crocker. An experimental comparison of two maximum cardinality matching programs. In Catherine C. McGeoch David S. Johnson, editor, *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *Discrete Mathematics and Theoretical Computer Science*, pages 519–537, 1993.

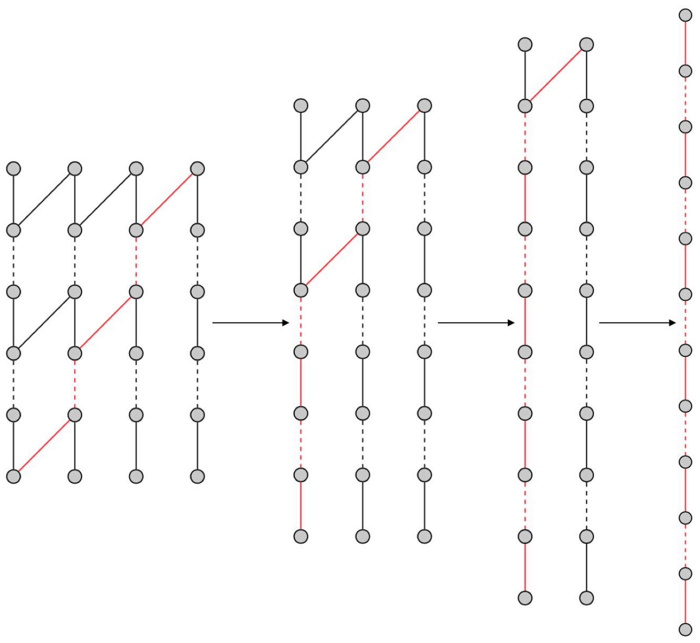

Figure 8 Example of $O(\sqrt{n}m)$ performance for the modified algorithm: Augmented paths are in red. In the example, the worst case maximal set of one is chosen in each phase.

3 Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

4 Shimon Even and Oded Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 100–112. IEEE, 1975.

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

6 Harold N. Gabow. Set-merging for the Matching Algorithm of Micali and Vazirani. *arXiv preprint arXiv:1501.00212*, 2014.

7 Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

8 John E. Hopcroft and Richard M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 122–125. IEEE, 1971.

9 T. Kameda and I. Munro. A $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing*, 12(1):91–98, 1974.

10 John D. Kececioglu and A. Justin Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Algorithm Engineering*, pages 121–132, 1998.

11 Eugene L. Lawler. Combinatorial optimization, 1976.

12 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, jun 2014. URL: `http://snap.stanford.edu/data`.

13 Jakob Magun. Greeding matching algorithms, an experimental study. *J. Exp. Algorithmics*, 3, sep 1998. `doi:10.1145/297096.297131`.

14 R. Bruce Mattingly and Nathan P. Ritchey. Implementing an $o(sqrtNm)$ cardinality matching algorithm. In Catherine C. McGeoch David S. Johnson, editor, *Network Flows and*

Matching: First DIMACS Implementation Challenge, volume 12 of *Discrete Mathematics and Theoretical Computer Science*, pages 539–556, 1993.

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

16 Vijay V. Vazirani. An improved definition of blossoms and a simpler proof of the MV matching algorithm. *CoRR*, abs/1210.4594, 2012. URL: `http://arxiv.org/abs/1210.4594`.

A Appendix

A.1 Additional Results: Graphs

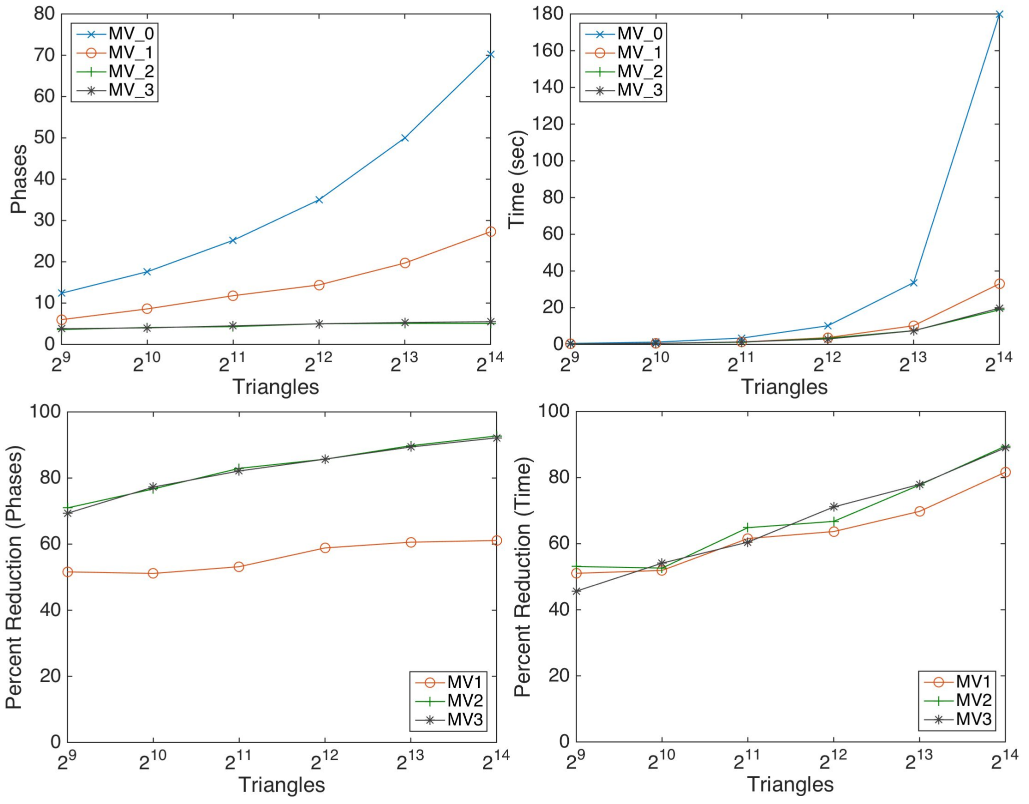

Figure 9 Average percent reduction of search phases and running time as the number of triangles increases for three-connected triangle graphs.

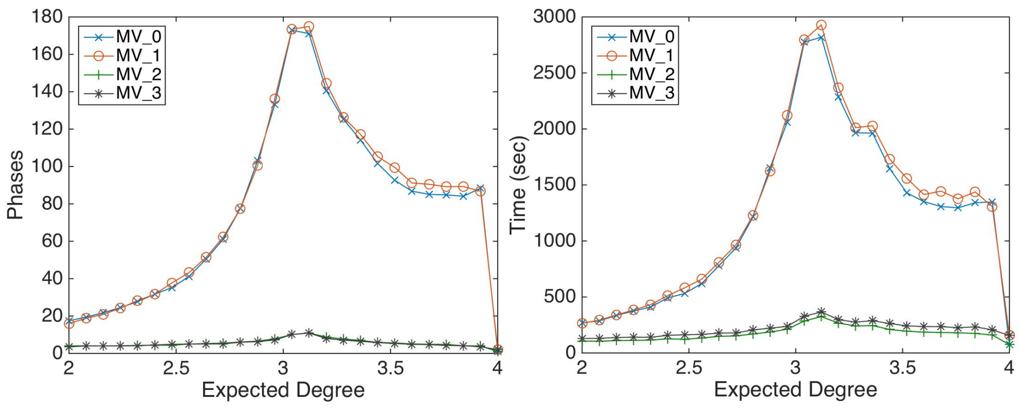

Figure 10 Average search phases and running time as the expected degree increases for grid graphs with 2^{20} nodes.

A.2 Additional Results: Tables

Table 4 Average search phases and running time as the number of nodes increases for random graphs with an expected degree of $\sqrt{8}$.

	Average Runtime (sec)				Average Phases			
N	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^{10}	0.106	0.078	0.076	0.084	10.2	2.6	4.8	2
2^{12}	0.846	0.482	0.48	0.496	16.4	4.2	5.4	2.6
2^{14}	6.208	2.694	2.47	2.184	28.6	7.8	7	3.4
2^{16}	53.052	15.654	14.58	11.13	48	9.6	8.2	3.2
2^{18}	343.739	100.947	67.186	48.578	69.5	17.1	8.6	3.5
2^{20}	1549.718	469.49	262.22	192.17	97.8	26.6	9.6	4

Table 5 Average runtime and phase for three connected triangle graphs as number of triangles increase.

	Average Runtime (sec)				Average Phases			
Triangles	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^9	0.535	0.262	0.251	0.291	12.4	6	3.6	3.8
2^{10}	1.266	0.609	0.6	0.581	17.6	8.6	4.1	4
2^{11}	3.464	1.332	1.218	1.37	25.5	11.8	4.3	4.5
2^{12}	10.144	3.686	3.374	2.931	35	14.4	5	5
2^{13}	33.639	10.169	7.482	7.423	50	19.7	5.1	5.3
2^{14}	179.793	33.029	18.856	19.772	70.2	27.3	5.1	5.5

Table 6 Average search phases and running time as the number of nodes increases for grid graphs with expected degree of 3.12.

	Average Runtime (sec)				Average Phases			
N	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^{10}	0.12	0.141	0.085	0.132	4.8	2.9	2.4	1.8
2^{12}	0.639	0.811	0.362	0.535	9.5	7	3.2	3.1
2^{14}	6.082	6.552	2.125	2.919	15.3	16.8	4.1	4
2^{16}	47.305	49.861	11.824	14.806	27.4	27.8	4.5	4.5
2^{18}	545.662	622.501	93.476	92.991	87.9	89.5	7.6	6.9
2^{20}	2819.185	2928.456	324.903	366.417	171.1	174.9	11	10.9

A.3 Micali-Vazirani Algorithm with Extended Phases

Listing 2 Micali-Vazirani Algorithm pseudocode.

```
1  Set initial greedy matching for G
2  Reset edge labels
3  Set minlevel = 0 and maxlevel = $\infty$ for each unmatched vertex
4  Set minlevel = $\infty$ and maxlevel = $\infty$ for each matched vertex
5  Set level = 0
6  Set augmentation_count = 0
7  If there exist u such that maxlevel(u) == level or minlevel(u) == level then continue, else go to line 31
8  For each u such that maxlevel(u) == level or minlevel(u) == level:
9    For each unscanned (u,v) with appropriate edge parity:
10     If minlevel(v) $\geq$ level + 1 then,
11       Set minlevel(v) = level + 1
12       Add u to the list of predecessors of u
13       Label (u,v) as prop
14     Else,
15       label (u,v) as bridge
16 If tenacity((u,v)) != $\infty$ then
17   Add (u,v) to the list of bridges with the same tenacity
18   For each bridge of tenacity == 2*level + 1:
19     Find support using DDFS
20     If bottleneck found then
21       Augment alternating path
22       Delete the vertices in the augmented path and all vertices that are orphanned (no predcessors) as a result
23     Else,
24       For each v in the support:
25         Set maxlevel(v) = 2*level + 1 - minlevel(v)
26         If v is an inner vertex then
27           For all incident (v,u) which are not props:
28             If tenacity((u,v)) != $\infty$ then
29               Add (u,v) to the list of bridges with the same tenacity
30 Set level = level + 1
31 If augmentation occured then augmentation_count += 1
32 If augmentation_count == 100 then go to line 2, else go to line 7
33 Return the current matching
```

A.4 Lemmas

▶ **Lemma 7.** *Augmenting a maximal set of vertex disjoint augmenting paths that includes a maximal set of vertex disjoint shortest augmenting paths increases the length of the shortest augmenting path in the new matching*

Proof. Let M be the initial matching, let M' be the subsequent matching after augmenting the maximal set of disjoint shortest augmenting paths P in M, and let M'' be the subsequent matching after augmenting the maximal set of disjoint augmenting paths P' in M. From Hopcroft and Karp [8] we know that if the length of the shortest augmenting path of a matching M is l, then the length of the shortest augmenting path in M' is strictly greater than l. Since $P \in P'$ and the paths in $P' - P$ are disjoint from P, augmenting the set $P' - P$ in matching M' gives us M''. The shortest augmenting path cannot get shorter, thus the shortest augmenting path in M'' is still strictly greater than l. ◀

A.5 Additional Discussion

We can show $O(m \log n)$ in the worst case for certain families of graphs while also proving the base algorithm operates in $O(\sqrt{n}m)$ time. The construction of such a graph is similar to that of the one-connected triangles graph in that it is constructed by joining vertex disjoint triangles. Instead of joining the triangles with one edge, it is replaced by a sequence of edges described below.

Let the number of edges between each triangle be determined by the following. Let us number the triangles from 1 to k. The number of edges joining triangle $2i - 1$ and $2i$ be $2(i - 1) + 1$ and let the number of edges joining triangle $2i$ and $2i + 1$ be $2i + 1$.

To demonstrate the worst case performance, let us assume that after a greedy matching, we have a graph such that the only free vertices are the nodes of each triangle i that is not adjacent to the set of edges connecting the triangle to triangle $i - 1$ or $i + 1$.

In the case of the modified algorithm MV2, we see that after the initial greedy matching, each phase selects a maximal set of augmenting paths. Since the unmatched vertices essentially lie on the same line, augmenting paths will always be guaranteed to be matched at most two potential free vertices. That means the algorithm also never encounters non vertex disjoint alternating paths. The problem can be reduced to selecting a maximal matching, where unmatched edges are the alternating paths. Rather than randomly choosing the matching, it is determined by the length of the alternating paths, but since it is a maximal selection, the equivalent problem is also a maximal matching. Since we know that a maximal matching is a 2-approximation of the maximum matching, it implies that the maximal set of alternating paths is also a 2-approximation of selecting the maximum set of augmenting paths. Thus, we are guaranteed to reduce the number of remaining matchings to be found in half each phase giving us the $O(m \log n)$ worse case run time.