Performance Studies of the Blossom V Algorithm

REU Site: Interdisciplinary Program in High Performance Computing

Changling Huang[1], Christopher C. Lowman[2], Brandon E. Osborne[3], Gabrielle M. Salib[4],
Graduate assistants: Ari Rapkin Blenkhorn[4], Jonathan S. Graf[5], Samuel Khuvis[5],
Faculty mentor: Matthias K. Gobbert[5], Clients: Tyler Simon[6] and David Mountain[6]

[1]Department of Computer Science, Rutgers University,
[2]Department of Mathematics, University of Maryland, College Park,
[3]Department of Physics and Astronomy, Austin Peay State University,
[4]Department of Computer Science and Electrical Engineering, UMBC,
[5]Department of Mathematics and Statistics, UMBC,
[6]Laboratory for Physical Sciences

Technical Report HPCF–2015–26, hpcf.umbc.edu > Publications

Abstract

The Blossom V algorithm is used in graph theory to compute a perfect matching of minimum cost. We conducted performance studies on the algorithm using the maya cluster in the UMBC High Performance Computing Facility to better understand the performance capabilities and emphasize potential approaches for improvement. In the performance studies, we varied the number of nodes, graph density, and weight range for numerous graphs. For each graph, we recorded execution times and memory usage. For all graphs used in the performance studies, we found that the majority of time is spent in initialization. We also found that as graph density increases, both execution time and memory usage increase. While we anticipated these conclusions, we reached other conclusions that were more surprising. We determined that as the weight range of a graph increases, initialization time and total execution time increase. We also found that scaling down integer weight ranges to real-valued weight ranges has a limited effect on initialization time and total execution time. Future studies should focus on speeding up the initialization process of the algorithm.

Key words. matching, perfect matching, augmenting path, blossom, blossom shrinking

AMS subject classifications (2010). 05C70, 68R10

1 Introduction

Blossom V is a powerful algorithm which finds a perfect matching of minimum cost in a graph [6]. Jack Edmonds (1965) developed the first installment of the Blossom algorithm with worst case complexity $O(n^2m)$, where n is the number of nodes and m is the number of edges in a graph. The current worst case complexity of the algorithm is $O(n(m + n\log n))$, as found by Gabow [4, 6].

Kolmogorov compares the speed of the Blossom V implementation with previous implementations in [6]. Blossom V outperforms its predecessors in most cases, including in practical applications such as Delaunay Triangulations and Planar Ising models [6].

Our group analyzed the overall performance of the Blossom V algorithm by collecting timing and memory data for graphs with various numbers of nodes, graph densities, and weight ranges. We used the analytical tool *gprof*, which reports the time spent in each function, to determine areas that would profit most from parallelization [5]. We used *Memcheck* to determine total memory usage of each graph and to determine the largest graph that we could test in terms of number of nodes and graph density.

Manipulating the weight ranges of graphs yielded unexpected timing results. Furthermore, varying graph density influenced execution times and total memory usage.

The remainder of this report is organized as follows: Section 2 provides information on graph theory and the Blossom V implementation. Section 3 details the methodology of our performance studies and analysis. Section 4 presents the results of our performance studies and analysis. Finally, Section 5 summarizes the results of these analyses and discusses future efforts for improvement of the algorithm.

2 Background

2.1 Graph Theory Terminology

The Blossom V algorithm operates on a weighted undirected graph, $G = (V, E, c)$, where V is the set of vertices (or nodes), E is the set of edges, and c is the set of costs (or weights) of the edges. The *order* of the graph, n, is defined as $|V(G)|$, the cardinality of the set V. The *size* of the graph, m is defined as $|E(G)|$, the cardinality of the set E.

A *matching* consists of a set of edges $M \subseteq E$, such that each node in V is incident with at most one edge in M. M is defined as *maximal* if the addition of any edge not in M will render M no longer a matching. A maximal matching with maximum cardinality is called a *maximum matching*. A maximum matching is a maximal matching, but a maximal matching is not always a maximum matching.

A *perfect matching* is a maximum matching where each node in V is incident with exactly

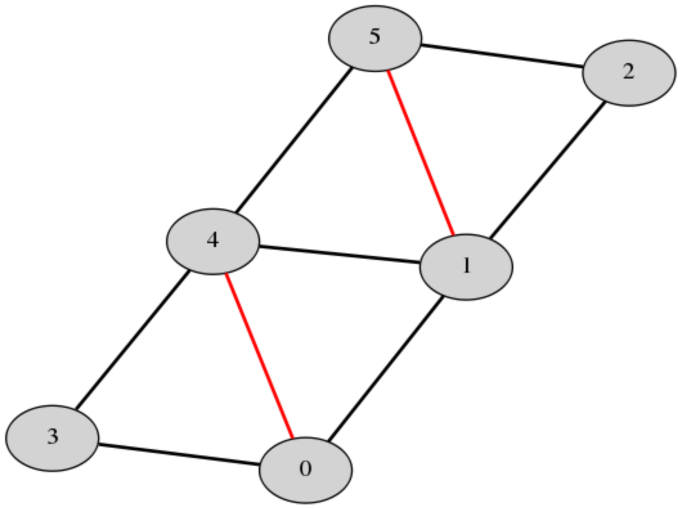


Figure 2.1: Edges 4-0 and 5-1 form a maximal, but not maximum, matching.

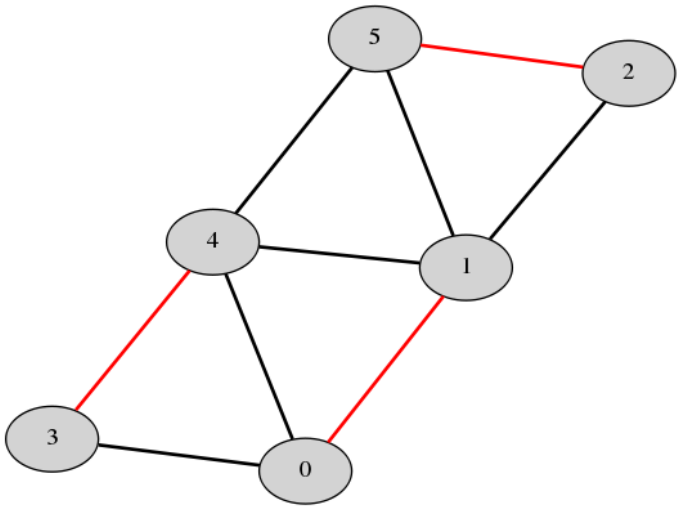


Figure 2.2: Edges 1-0, 4-3, and 5-2 form a perfect matching.

one edge in the matching; hence, a perfect matching can only be found in a graph where n is even. If n was odd, then every node in V would be incident with an edge in a perfect matching only if two edges shared a common node. However, a node incident with multiple edges contradicts the definition of a matching and thus a graph where n is odd cannot contain a perfect matching. Additionally, the cardinality of a perfect matching M must be $|M| = (n/2)$. If $|M| < (n/2)$, then at least 2 nodes are not incident with any edge in M and thus M is not perfect. If $|M| > (n/2)$, then $\exists$ a node in V that is incident with at least 2 edges in M, meaning M is not a matching at all. Thus, for a perfect matching M, $|M| = (n/2)$.

A graph can contain more than one perfect matching. The Blossom V algorithm aims to find a perfect matching of minimum cost in a weighted graph. The perfect matching is constructed by iteratively adding edges to an initially empty matching M along *augmenting paths* in the graph. A path P in G is *alternating* if edges within the path are alternately in and not in M. An *exposed* vertex is one that is not incident with any edge in M. P is augmenting if it starts and ends at two exposed vertices and is an alternating path. To perform a *matching augmentation* along an augmenting path P, M is replaced with a new matching $M_1 = M \oplus P$ as shown in Figure 2.3. It can be proven that a matching is maximum if and only if there is no M-augmenting path in G [7].

A critical component of the Blossom V algorithm involves the use of blossoms. Given a graph G, a *blossom* B is defined as a cycle consisting of $2k + 1$ edges, exactly k of which belong to a matching M. The blossoms can be *shrunk* and *expanded*, as shown in Figure 2.4. Shrinking the blossoms makes it possible to treat them as singular nodes during execution of the algorithm. Searches performed on the resulting reduced graph are more efficient.

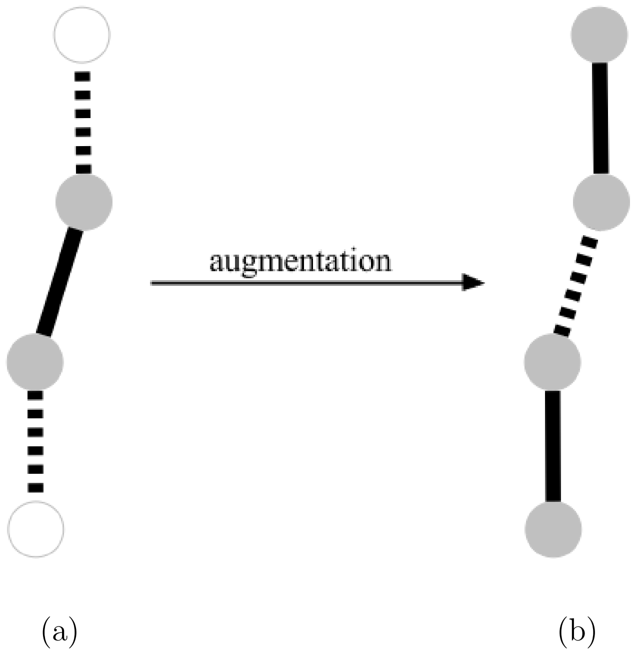


Figure 2.3: Figure (a) displays the augmenting path P prior to augmentation with the white nodes denoting exposed vertices and the solid line denoting the original matching M. In figure (b), a new matching M_1 is obtained by augmentation: the edge in M is dropped and the two edges connecting the incident nodes of M to the exposed nodes are added to M_1.

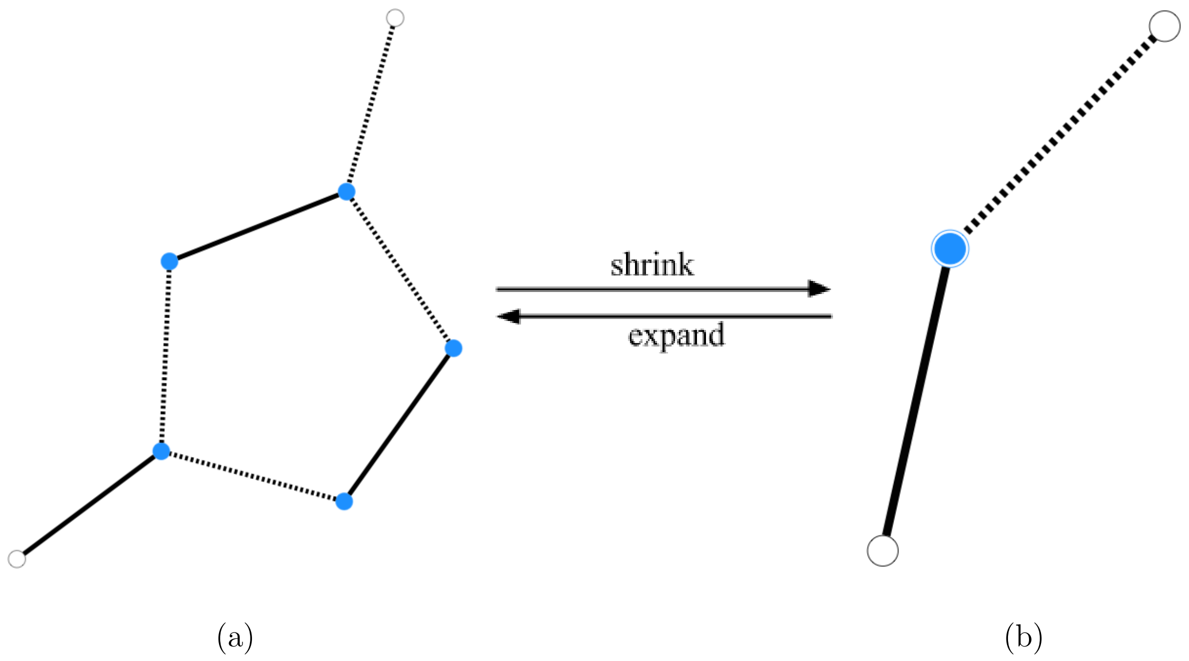


Figure 2.4: Blossom with five vertices in (a) being shrunk to a contracted node in (b).

2.2 Blossom V Overview

2.2.1 Optimization through Duality

To find a perfect matching of minimum cost, Blossom V uses the linear programming concept of duality. In duality theory, the desired optimization problem, or *primal problem*, can be solved by considering a second, closely-related *dual problem*. The primal problem is subject to specific constraints and the dual is used to ensure the final solution is optimal. The primal problem used to find the minimum cost matchings is expressed as

$$\min \sum_{e \in E} c_e x_e \tag{2.1}$$

with the constraints

$$\begin{aligned} x(\delta(v)) &= 1 \quad \forall\, v \in V, & (2.2)\\ x(\delta(S)) &\geq 1 \quad \forall\, S \in \mathcal{O}, & (2.3)\\ x_e &\geq 0 \quad \forall\, e \in E, & (2.4) \end{aligned}$$

where c_e represents the cost of edge $e = (u, v)$ and x_e denotes a vector in e holding values 1 or 0 (1 for a matched edge and 0 for an unmatched edge). In the list of constraints, $x(\delta(v))$ represents the sum of x_e vectors at each node, $\mathcal{O}$ denotes the set of all odd subsets (i.e. with odd cardinality) of V with at least three nodes, S is a subset of V, and $x(\delta(S))$ denotes the sum of all x_e vectors with one vertex in S. The corresponding dual problem is shown as

$$\max \sum_{v \in V} y_v + \sum_{S \in \mathcal{O}} y_S \tag{2.5}$$

subject to the constraints

$$\begin{aligned} \mathit{slack}(e) &\geq 0 \quad \forall\, e \in E, & (2.6)\\ y_S &\geq 1 \quad \forall\, S \in \mathcal{O}, & (2.7) \end{aligned}$$

where $\mathit{slack}(e)$ denotes the reduced cost of edge e, y_v represents the feasible dual vector at each vertex, and y_S represents the dual vector pertaining to each edge of $S \in \mathcal{O}$. If an edge has zero slack (i.e. the cost c_e of the edge is equal to the sum of y_u, y_v, and each y_S vector) it is known as *tight*. If $\mathit{slack}(e)$ is greater than zero only at unmatched edges and y_S is greater than zero only when $x(\delta(S)) = 1$, the complementary slackness conditions are satisfied and a perfect matching is reached [6].

2.2.2 Primal and Dual Updates

A *tree* T is defined as a connected graph without any cycles. The search for augmenting paths uses a *forest* data structure of individual trees that corresponds to different portions of a graph G. Every node v is given a label $l(v) \in \{+, -, \varnothing\}$. Nodes with label $\varnothing$ are considered *free vertices*; otherwise, they belong to an alternating tree. The root of the alternating tree is labeled + but it is considered unmatched. Consequently, the number of trees is the number of unmatched nodes. In addition, only + vertices may have more than one child; however, −

vertices may only have one. The algorithm finds matchings and dual solutions that satisfy the complementary slackness conditions and forms augmenting paths consisting of only tight edges. Without changing the dual variables, the following primal operations are performed using only tight edges:

Grow If $\exists$ $edge(u, v)$, $l(u) = +$ and $l(v) = \varnothing$, the tree of u can be *grown* by the addition of node u and the corresponding matched node.

Augment If $\exists$ $edge(u, v)$, $l(u) = l(v) = +$ and u, v belong to different trees, then the cardinality can be increased by flipping the matching along the path connecting the roots of the two trees. All vertices in the tree become free.

Shrink If $\exists$ $edge(u, v)$, $l(u) = l(v) = +$ and u, v are in the same tree, then there must exist a cycle of odd length that can be shrunk.

Expand If node v is a blossom with $y_v = 0$, $l(v) = -$, then it can be expanded to a blossom.

The primal updates cannot always be immediately applied; thus, the dual variables must be modified in order to create tight edges. A dual change δ_T is added to y_v for nodes labeled "+" and subtracted from y_v for nodes labeled "−" in each tree. Blossom V uses a variable δ approach with the constraints

$$\begin{aligned}
\delta_T &\le slack(u, v) && (u, v) \text{ is a } (+, \varnothing) \text{ edge}, u \in T, && (2.8)\\
\delta_T + \delta_{T'} &\le slack(u, v) && (u, v) \text{ is a } (+, +) \text{ edge}, u \in T, v \in T', T \ne T', && (2.9)\\
\delta_T &\le slack(u, v)/2 && (u, v) \text{ is a } (+, +) \text{ edge}, u, v \in T, && (2.10)\\
\delta_T &\le y_v && v \text{ is a "}-\text{" node and a blossom}, v \in T, && (2.11)\\
\delta_T + \delta_{T'} &\le slack(u, v) && (u, v) \text{ is a } (+, -) \text{ edge}, u \in T, v \in T', T \ne T'. && (2.12)
\end{aligned}$$

When one of the above constraints becomes tight, the primal operations can be applied. Constraint 2.8 corresponds to a GROW operation, 2.9 to an AUGMENT operation, 2.10 to a SHRINK operation, and 2.11 to an EXPAND operation. If constraint 2.12 becomes tight then no operations can be performed. The algorithm cycles between primal and dual updates until the optimal perfect matching is reached. Refer to [6] for a more detailed description of the implementation.

3 Methodology

3.1 Hardware

We conducted performance studies on the Blossom V algorithm using the 2013 portion of the maya cluster in the UMBC High Performance COmputing Facility. Figure 3.1 displays a schematic of one of the compute nodes on the cluster that was used to conduct the performance studies. Each compute node consists of two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs [2]. Each core of each CPU has 32 kB of L1 cache and 256 kB of L2 cache, and all cores of each CPU share 20 MB of L3 cache [2]. Each compute node has 64 GB of main memory [2].

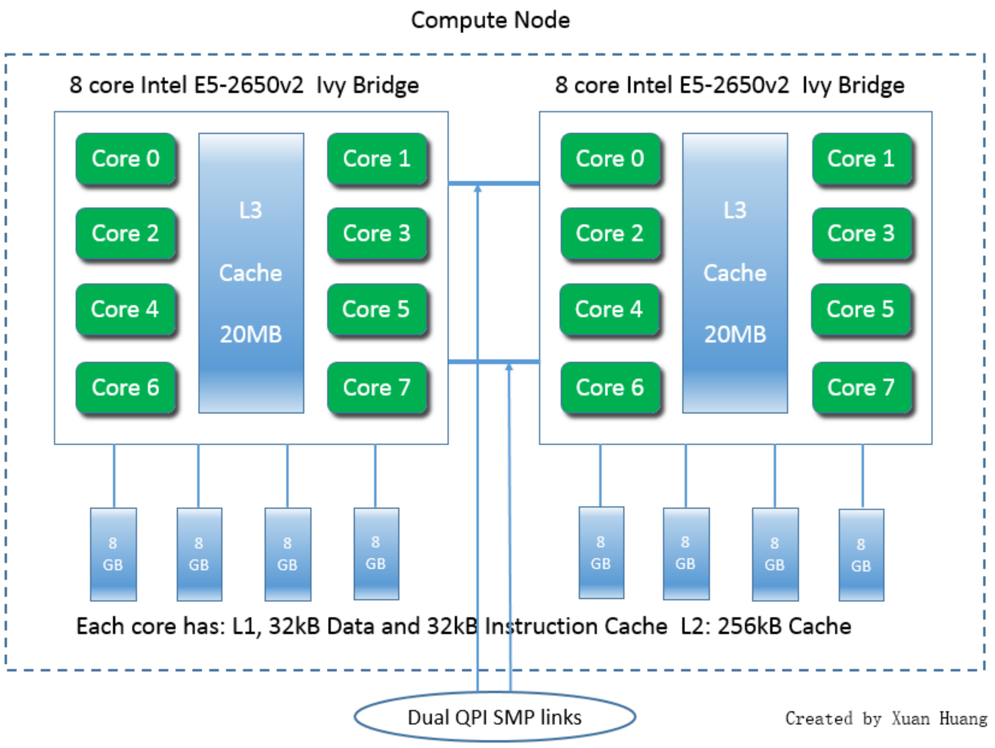

Compute Node
8 core Intel E5-2650v2 Ivy Bridge
8 core Intel E5-2650v2 Ivy Bridge
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
L3 Cache 20MB
8 GB
Each core has: L1, 32kB Data and 32kB Instruction Cache L2: 256kB Cache
Dual QPI SMP links
Created by Xuan Huang

Figure 3.1: Schematic of a maya 2013 compute node.

3.2 Generating Testable Data

We used the SSCA2 random graph generator in the GTgraphs suite to create testable graphs [1]. SSCA2 creates a directed graph with integer edge weights and writes it to a file in DIMACS format. We created graphs of up to 65536 vertices and 3913764 edges. A number of parameters could be specified, such as maximum parallel edges, minimum weight, and clique size; however, the graphs generated often did not contain a perfect matching and caused the algorithm to produce a segmentation fault. Because a reliable range of test data could not be produced by SSCA2, we wrote two additional generators to ensure that the graphs were connected.

The first generator creates *complete graphs*, or graphs in which every pair of distinct vertices is connected by exactly one edge. It accepts the desired scale for vertices and writes each matching and a randomly generated weight to each line of an output file in DIMACS format. The second generator creates connected graphs of a specified scale and density. *Graph density* d is defined as the number of edges in a graph divided by the total number of possible edges that a graph can have, shown as

$$d = \frac{2|E|}{|V|(|V|-1)}. \tag{3.1}$$

For simplicity, graph density will be referred to as "density" from this point on. The generator computes the number of edges $|E|$ and creates an adjacency matrix that stores random edge weights at each pertinent cell. Two vertices are matched by incrementing one parameter in the matrix, randomly generating the second, and applying a weight to the edge. A second pass is then performed to connect the remaining unconnected vertices and apply random weights to the edges. The data is then written to a file in DIMACS format.

3.3 Performance Study Methods

To determine which components of the Blossom V algorithm take the longest to complete, we used the GNU profiler gprof. gprof can generate a function call hierarchy, count the number of times each function is called, and determine the percentage of total time spent in each function [5]. In order to profile the code, the flag `-pg` must be added to the Makefile. Once compilation is complete, running the executable generates a `gmon.out` file containing the profiling information. The file can then be converted to a `dot` file using the `gprof2dot.py` script by [3]. The `dot` file can then be converted to a PNG file to view a tree diagram of function calls.

We conducted performance studies for graphs with $n = 1024$ (2^{10}), 2048 (2^{11}), 4096 (2^{12}), 8192 (2^{13}), 16384 (2^{14}), 32768 (2^{15}) nodes; graph densities $d = 0.125$, 0.25, 0.5, 1; and initial integer weight ranges r_i from 1 to 10^2, 1 to 10^4, 1 to 10^6, and 1 to 10^8. The complete graph generator was used to create graphs with a density of 1 and the connected graph generator was used for all other graphs. From this point on, the range of possible weights that can be assigned to an edge in a graph will be referred to as "weight range" for simplicity.

In our performance studies, we recorded initialization time and total execution time for graphs with various values of n, d, and r_i. Initialization time includes updating optimization variables and assigning matchings, while total execution time is how long the algorithm runs from start to finish. In order to determine the initialization time and total execution time of a graph, the program stores the wall clock time at the beginning and end of each major section and calculates the difference before displaying each of the times.

We first generated five random graphs, each with the same combination of n, d, and r_i. We then executed the algorithm on each of the five graphs in order to determine the initialization time and total execution time for each graph. Finally, we aggregated and averaged the timing values for the five graphs and recorded the average values. The process was repeated for each possible combination of n, d, and r_i.

In our performance studies, we also recorded total memory usage of the algorithm for each possible combination of n and d. In contrast to our timing experiments, we used only one graph for each (n, d) combination because graphs with the same number of nodes and edges require nearly identical amounts of memory, regardless of weight range. To determine memory usage, we used the Valgrind analysis tool Memcheck. Memcheck logs all memory reads/writes [8]. It also intercepts calls to malloc/new/free/delete [8]. The command `valgrind ./blossom5 -e <graph_file>` initiates the profiling with Memcheck; however, for graphs with $n = 32768$ nodes and densities $d = 0.5$ and $d = 1$, Memcheck did not work properly due to invalid write errors. For these graphs, we used Matlab to perform linear regression and recorded the predicted values with the associated R^2 values.

We were interested to see how scaling down the edge weights of a graph would affect initialization time, total execution time, and total memory usage for graphs with larger weight ranges. For each graph that was generated and used for the performance studies described above, we scaled down the weights and recorded initialization time, total execution time, and total memory usage and repeated the previous procedures. We scaled down the weights of a graph by dividing each edge's weight by the maximum value in the original graph's weight range. Consequently, the resulting scaled-down weight ranges r_s were 10^{-2} to 1, 10^{-4} to 1, 10^{-6} to 1, and 10^{-8} to 1.

Finally, we were interested to see how changing the initial weight ranges would affect initialization time and total execution time of graphs. We modified the initial weight ranges by maintaining the range and shifting the minimum and maximum values of the range. An example of a modified weight range r_m would be 301 to 400. For each initial weight range r_i, we chose five modified weight ranges. For each of these modified weight ranges, we generated five graphs, each with $n = 32768$ nodes and density $d = 1$. We then gathered initialization time and total execution time for each of the five graphs, averaged the times, and recorded them the same way we did for our initial performance studies. The variable r_m is used the represent a modified weight range from this point on.

3.4 Analysis Methods

We define $I_{(n,d,r_i)}$ and $T_{(n,d,r_i)}$ to represent the initialization time and total execution time, respectively, for a graph with n nodes, density d, and initial weight range r_i. Upon completion of the performance studies, we first analyzed the slowdown effect on initialization time and total execution time that results from increasing density.

$$S_d(I, n, d, r_i) = \frac{I_{(n,d,r_i)}}{I_{(n,0.125,r_i)}} \tag{3.2}$$

and

$$S_d(T, n, d, r_i) = \frac{T_{(n,d,r_i)}}{T_{(n,0.125,r_i)}} \tag{3.3}$$

represent the slowdown in initialization time and total execution time, respectively, when compared to the initialization time and total execution time for a graph with n nodes, density $d = 0.125$, and initial weight range r_i.

We then analyzed the slowdown effect on initialization time and total execution time that results from increasing the initial weight range r_i.

$$S_{r_i}(I, n, d, r_i) = \frac{I_{(n,d,r_i)}}{I_{(n,d,1-10^2)}} \tag{3.4}$$

and

$$S_{r_i}(T, n, d, r_i) = \frac{T_{(n,d,r_i)}}{T_{(n,d,1-10^2)}} \tag{3.5}$$

represent the slowdown in initialization time and total execution time, respectively, when compared to the initialization time and total execution time for a graph with n nodes, density d, and weight range 1 to 10^2.

For graphs with scaled-down weights, we define $I_{(n,d,r_s)}$ and $T_{(n,d,r_s)}$ as before, but with scaled-down real-valued weight ranges r_s from 10^{-2} to 1, 10^{-4} to 1, 10^{-6} to 1, and 10^{-8} to 1. Upon completion of the performance studies for graphs with scaled-down weights, we analyzed the speedup effect on initialization time and total execution time that results from scaling down the edge weights.

$$S_{r_s}(I, n, d, r_i, r_s) = \frac{I_{(n,d,r_i)}}{I_{(n,d,r_s)}} \tag{3.6}$$

and

$$S_{r_s}(T, n, d, r_i, r_s) = \frac{T_{(n,d,r_i)}}{T_{(n,d,r_s)}} \tag{3.7}$$

represent the speedup in initialization time and total execution time, respectively, when compared to the initialization time and total execution time for a graph with n nodes, density d, and initial weight range r_i.

Finally, for graphs with modified weight ranges r_m, we define $I_{(n,d,r_m)}$ and $T_{(n,d,r_m)}$ as before, but with modified integer weight ranges as described in Section 3.3. Upon completion of the performance studies with these modified weight ranges, we analyzed the slowdown effect on initialization time and total execution time for graphs with $n = 32768$ nodes and density $d = 1$ that results from using these modified weight ranges.

$$S_{r_m}(I, r_i, r_m) = \frac{I_{(32768,1,r_m)}}{I_{(32768,1,r_i)}} \tag{3.8}$$

and

$$S_{r_m}(T, r_i, r_m) = \frac{T_{(32768,1,r_m)}}{T_{(32768,1,r_i)}} \tag{3.9}$$

represent the slowdown in initialization time and total execution time, respectively, when compared to initialization time and total execution time for a graph with $n = 32768$ nodes, density $d = 1$, and initial weight range r_i.

4 Results

Table 4.1 demonstrates how initialization time and total execution time differ for graphs with various numbers of nodes, densities, and integer weight ranges. Tables 4.2 and 4.3 demonstrate the effect that density and weight range, respectively, have on initialization time and total execution time.

Table 4.4 demonstrates how initialization time and total execution time differ for graphs with various numbers of nodes, densities, and real-valued weight ranges. Table 4.5 demonstrates the effect that using real-valued weight ranges, as opposed to integer weight ranges, has on initialization time and total execution time.

Table 4.6 demonstrates how initialization time and total execution time differ for graphs with $n = 32768$ nodes, density $d = 1$, and modified weight ranges as described in Section 3.3. Also for Table 4.6, r_i indicates the original weight range, while r_m indicates the modified weight range. Table 4.7 demonstrates the effect that using modified weight ranges has on initialization time and total execution time of graphs with $n = 32768$ nodes and density $d = 1$.

Finally, Table 4.8 demonstrates the effect that density has on memory usage for graphs with integer weight ranges and graphs with real-valued weight ranges.

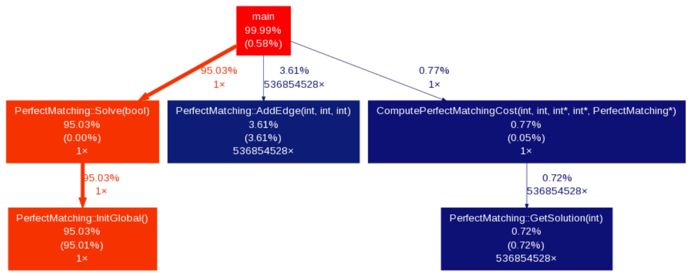

main
99.99%
(0.58%)
95.03%
1×
3.61%
536854528×
0.77%
1×
PerfectMatching::Solve(bool)
95.03%
(0.00%)
1×
PerfectMatching::AddEdge(int, int, int)
3.61%
(3.61%)
536854528×
ComputePerfectMatchingCost(int, int, int*, int*, PerfectMatching*)
0.77%
(0.05%)
1×
95.03%
1×
0.72%
536854528×
PerfectMatching::InitGlobal()
95.03%
(95.01%)
1×
PerfectMatching::GetSolution(int)
0.72%
(0.72%)
536854528×

Figure 4.1: Tree diagram for our graph generator with $n = 2^{15}$, $d = 1$, and an edge weight range of 1 to 100000000.

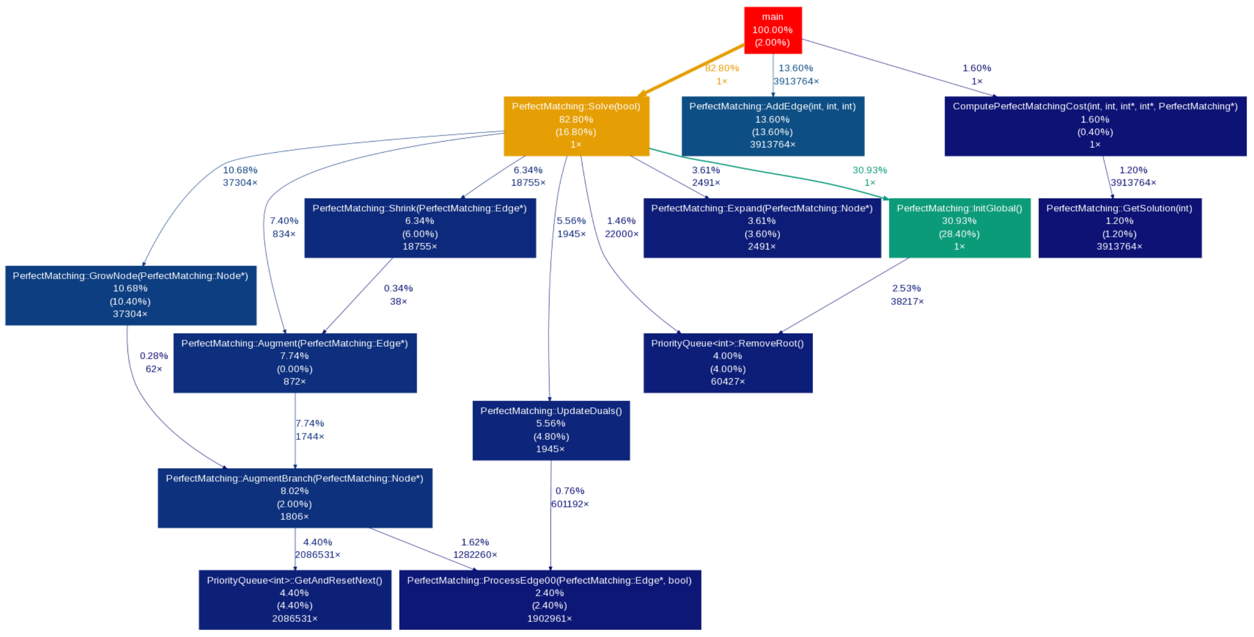

main
100.00%
(2.00%)
82.80%
1×
13.60%
3913764×
1.60%
1×
PerfectMatching::Solve(bool)
82.80%
(16.80%)
1×
PerfectMatching::AddEdge(int, int, int)
13.60%
(13.60%)
3913764×
ComputePerfectMatchingCost(int, int, int*, int*, PerfectMatching*)
1.60%
(0.40%)
1×
10.68%
37304×
6.34%
18755×
3.61%
2491×
30.93%
1×
1.20%
3913764×
7.40%
834×
PerfectMatching::Shrink(PerfectMatching::Edge*)
6.34%
(6.00%)
18755×
5.56%
1945×
1.46%
22000×
PerfectMatching::Expand(PerfectMatching::Node*)
3.61%
(3.60%)
2491×
PerfectMatching::InitGlobal()
30.93%
(28.40%)
1×
PerfectMatching::GetSolution(int)
1.20%
(1.20%)
3913764×
PerfectMatching::GrowNode(PerfectMatching::Node*)
10.68%
(10.40%)
37304×
0.34%
38×
2.53%
38217×
0.28%
62×
PerfectMatching::Augment(PerfectMatching::Edge*)
7.74%
(0.00%)
872×
PriorityQueue<int>::RemoveRoot()
4.00%
(4.00%)
60427×
PerfectMatching::UpdateDuals()
5.56%
(4.80%)
1945×
7.74%
1744×
0.76%
601192×
PerfectMatching::AugmentBranch(PerfectMatching::Node*)
8.02%
(2.00%)
1806×
4.40%
2086531×
1.62%
1282260×
PriorityQueue<int>::GetAndResetNext()
4.40%
(4.40%)
2086531×
PerfectMatching::ProcessEdge00(PerfectMatching::Edge*, bool)
2.40%
(2.40%)
1902961×

Figure 4.2: Tree diagram for the SSCA2 generator with $n = 2^{15}$.

The results in Tables 4.1, 4.4, and 4.6 all indicate that the majority of total execution time is composed of initialization, which is consistent with the results produced by gprof. Figure 4.1 shows the call hierarchy for a graph created by our graph generator. Although we were unable to produce consistent testable graphs in SSCA2, we examined a call graph for a sample that worked. Despite a larger call graph due to graph shape, initialization remains the most significant portion.

4.1 Effect of Graph Density on Initialization Time and Total Execution Time

The timing results in Table 4.1 indicate that as graph density increases, both initialization time and total execution time of a graph tend to increase, regardless of the number of nodes or weight range. For smaller graphs, the increase appears to be minimal. For example, a graph with $n = 2048$ nodes, density $d = 1$, and weight range r_i from 1 to 10^2 has an initialization time of 0.168 seconds and a total execution time of 0.204 seconds. In comparison,

Table 4.1: Initialization time and total execution time for graphs with various numbers of nodes, densities, and weight ranges.

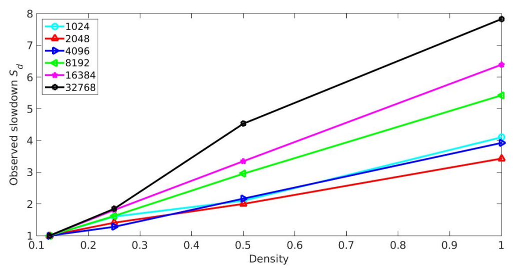
a) Initialization time in seconds

n	m	d	$1-10^2$	$1-10^4$	$1-10^6$	$1-10^8$
1024	65472	0.125	0.010	0.013	0.012	0.014
1024	130944	0.250	0.016	0.023	0.023	0.024
1024	261888	0.500	0.021	0.055	0.061	0.063
1024	523776	1.000	0.041	0.175	0.192	0.204
2048	262016	0.125	0.049	0.077	0.078	0.080
2048	524032	0.250	0.069	0.211	0.234	0.222
2048	1048064	0.500	0.098	0.630	0.655	0.659
2048	2096128	1.000	0.168	1.500	1.684	1.756
4096	1048320	0.125	0.169	0.779	0.778	0.804
4096	2096640	0.250	0.216	1.879	2.217	2.176
4096	4193280	0.500	0.366	4.001	4.891	5.099
4096	8386560	1.000	0.663	7.577	9.832	10.122
8192	4193792	0.125	0.471	6.028	6.531	6.353
8192	8387584	0.250	0.763	11.274	13.531	13.719
8192	16775168	0.500	1.390	20.977	27.102	29.909
8192	33550336	1.000	2.554	45.742	53.652	53.014
16384	16776192	0.125	1.605	30.605	38.502	40.013
16384	33552384	0.250	2.902	60.614	76.217	79.609
16384	67104768	0.500	5.368	136.237	156.232	153.970
16384	134209536	1.000	10.254	314.655	304.438	310.850
32768	67106816	0.125	6.074	181.449	219.804	226.965
32768	134213632	0.250	11.207	433.533	439.950	434.774
32768	268427264	0.500	27.505	1236.385	1077.068	905.743
32768	536854528	1.000	47.547	729.372	2649.447	2677.173

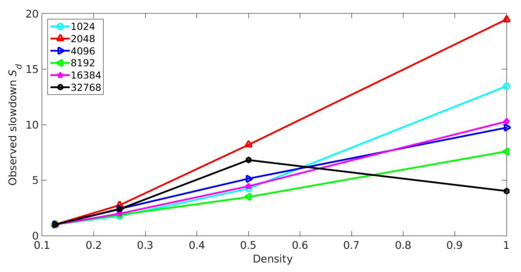
b) Total execution time in seconds

n	m	d	$1-10^2$	$1-10^4$	$1-10^6$	$1-10^8$
1024	65472	0.125	0.010	0.014	0.014	0.016
1024	130944	0.250	0.020	0.025	0.024	0.025
1024	261888	0.500	0.031	0.061	0.065	0.065
1024	523776	1.000	0.047	0.200	0.202	0.213
2048	262016	0.125	0.054	0.079	0.081	0.084
2048	524032	0.250	0.091	0.222	0.250	0.230
2048	1048064	0.500	0.148	0.660	0.673	0.704
2048	2096128	1.000	0.204	1.564	1.739	1.839
4096	1048320	0.125	0.204	0.819	0.806	0.832
4096	2096640	0.250	0.336	1.932	2.285	2.259
4096	4193280	0.500	0.476	4.210	4.965	5.244
4096	8386560	1.000	0.808	8.175	10.193	10.433
8192	4193792	0.125	0.605	6.143	6.642	6.460
8192	8387584	0.250	1.204	11.753	13.746	13.918
8192	16775168	0.500	1.571	21.819	27.428	30.382
8192	33550336	1.000	2.933	47.768	53.763	54.049
16384	16776192	0.125	2.420	31.273	38.811	40.279
16384	33552384	0.250	4.074	63.530	77.030	80.552
16384	67104768	0.500	6.399	141.237	158.295	155.207
16384	134209536	1.000	10.788	326.793	308.734	313.352
32768	67106816	0.125	8.262	185.352	220.753	227.823
32768	134213632	0.250	13.006	444.490	441.151	438.229
32768	268427264	0.500	29.046	1288.294	1090.053	918.233
32768	536854528	1.000	49.087	878.405	2670.155	2700.346

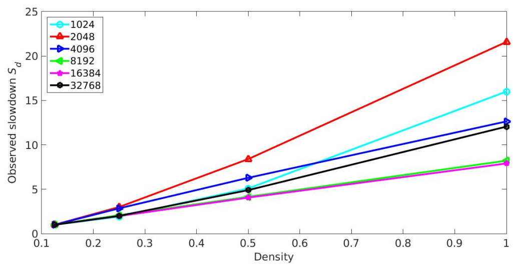
a graph with the same number of nodes and weight range, but density of $d = 0.125$, has an initialization time of 0.049 seconds and a total execution time of 0.054 seconds. Hence,



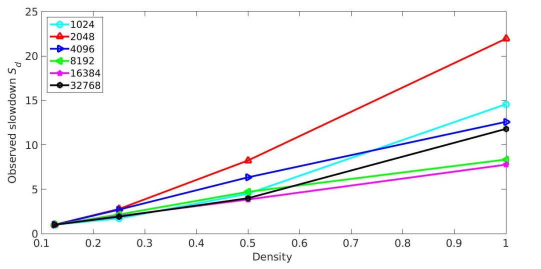
(a) Graphs with weight range $1 - 10^2$.



(b) Graphs with weight range $1 - 10^4$.



(c) Graphs with weight range $1 - 10^6$.



(d) Graphs with weight range $1 - 10^8$.

Figure 4.3: Slowdown in initialization time as graph density increases.

the difference in initialization time between the two graphs is only 0.119 seconds, and the difference in total execution time between the two graphs is only 0.150 seconds. For larger graphs, the increase appears to be more significant. For example, a graph with $n = 32768$ nodes, density $d = 1$, and weight range r_i from 1 to 10^2 has an initialization time of 47.547 seconds and a total execution time of 49.087 seconds. In comparison, a graph with the same number of nodes and weight range, but density of $d = 0.125$, has an initialization time of 6.074 seconds and a total execution time of 8.262 seconds. Hence, the difference in initialization time between the two graphs is 41.500 seconds, and the difference in total execution time between the two graphs is 40.825 seconds.

Though the increase in initialization time and total execution time appears to be more significant for larger graphs than for smaller graphs, this does not accurately indicate how much slower the times are as density increases for various sizes of graphs. For example, a graph with $n = 1024$ nodes, density $d = 1$, and weight range r_i from 1 to 10^6 has an initialization time 16.000 times slower and a total execution time 14.429 times slower than a graph with $n = 1024$ nodes, density $d = 0.125$, and weight range r_i from 1 to 10^6. In comparison, a graph with $n = 32768$ nodes, density $d = 1$, and weight range r_i from 1 to 10^6 has an initialization time 12.054 times slower and a total execution time 12.906 times slower than a graph with $n = 32768$ nodes, density $d = 0.125$, and weight range r_i from 1 to 10^6. The data can be found in Table 4.2, which contains observed slowdown values that were calculated using equations (3.2) and (3.3). The results in Table 4.2 indicate how much slower initialization time and total execution time are as density increases for each combination of n and r_i. These results are presented graphically in Figures 4.2 and 4.4, which also display the overall trend that both initialization time and total execution time tend to increase as density increases, regardless of the number of nodes and weight range.

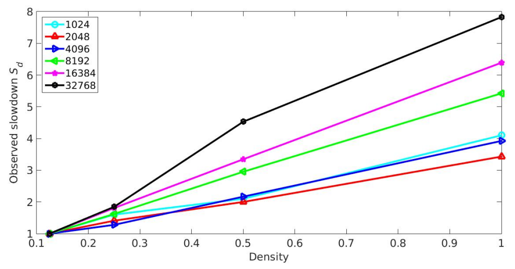
Table 4.2: Slowdown in initialization time and total execution time as graph density increases.

a) Slowdown in initialization time

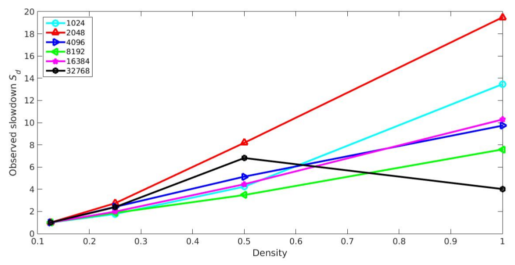
n	m	d	$1-10^2$	$1-10^4$	$1-10^6$	$1-10^8$
1024	65472	0.125	1.000	1.000	1.000	1.000
1024	130944	0.250	1.600	1.769	1.917	1.714
1024	261888	0.500	2.100	4.231	5.083	4.500
1024	523776	1.000	4.100	13.462	16.000	14.571
2048	262016	0.125	1.000	1.000	1.000	1.000
2048	524032	0.250	1.408	2.740	3.000	2.775
2048	1048064	0.500	2.000	8.182	8.397	8.238
2048	2096128	1.000	3.429	19.481	21.590	21.950
4096	1048320	0.125	1.000	1.000	1.000	1.000
4096	2096640	0.250	1.278	2.412	2.850	2.706
4096	4193280	0.500	2.166	5.136	6.287	6.342
4096	8386560	1.000	3.923	9.727	12.638	12.590
8192	4193792	0.125	1.000	1.000	1.000	1.000
8192	8387584	0.250	1.620	1.870	2.072	2.159
8192	16775168	0.500	2.951	3.480	4.150	4.708
8192	33550336	1.000	5.423	7.588	8.215	8.345
16384	16776192	0.125	1.000	1.000	1.000	1.000
16384	33552384	0.250	1.808	1.981	1.980	1.990
16384	67104768	0.500	3.345	4.451	4.058	3.848
16384	134209536	1.000	6.389	10.281	7.907	7.769
32768	67106816	0.125	1.000	1.000	1.000	1.000
32768	134213632	0.250	1.845	2.389	2.002	1.916
32768	268427264	0.500	4.528	6.814	4.900	3.991
32768	536854528	1.000	7.828	4.020	12.054	11.796

b) Slowdown in total execution time

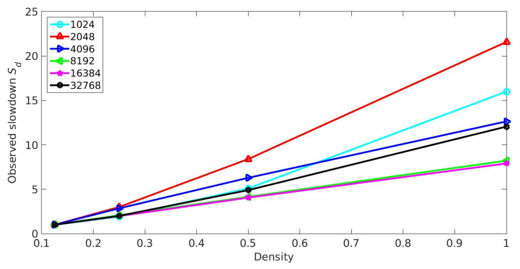
n	m	d	$1-10^2$	$1-10^4$	$1-10^6$	$1-10^8$
1024	65472	0.125	1.000	1.000	1.000	1.000
1024	130944	0.250	2.000	1.786	1.714	1.563
1024	261888	0.500	3.100	4.357	4.643	4.063
1024	523776	1.000	4.700	14.286	14.429	13.313
2048	262016	0.125	1.000	1.000	1.000	1.000
2048	524032	0.250	1.685	2.810	3.086	2.738
2048	1048064	0.500	2.741	8.354	8.309	8.381
2048	2096128	1.000	3.778	19.797	21.469	21.893
4096	1048320	0.125	1.000	1.000	1.000	1.000
4096	2096640	0.250	1.647	2.359	2.835	2.715
4096	4193280	0.500	2.333	5.140	6.160	6.303
4096	8386560	1.000	3.961	9.982	12.646	12.540
8192	4193792	0.125	1.000	1.000	1.000	1.000
8192	8387584	0.250	1.990	1.913	2.070	2.154
8192	16775168	0.500	2.597	3.552	4.129	4.703
8192	33550336	1.000	4.848	7.776	8.094	8.367
16384	16776192	0.125	1.000	1.000	1.000	1.000
16384	33552384	0.250	1.683	2.031	1.985	2.000
16384	67104768	0.500	2.644	4.516	4.079	3.853
16384	134209536	1.000	4.458	10.450	7.955	7.780
32768	67106816	0.125	1.000	1.000	1.000	1.000
32768	134213632	0.250	1.574	2.398	1.998	1.924
32768	268427264	0.500	3.516	6.951	4.938	4.030
32768	536854528	1.000	5.941	4.739	12.096	11.853



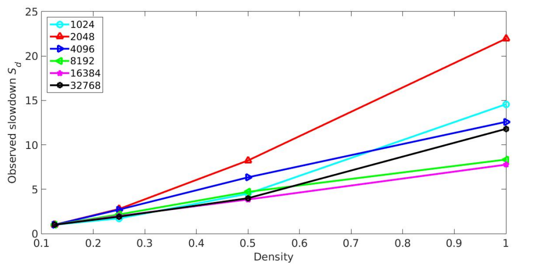
(a) Graphs with weight range $1 - 10^2$.



(b) Graphs with weight range $1 - 10^4$.



(c) Graphs with weight range $1 - 10^6$.



(d) Graphs with weight range $1 - 10^8$.

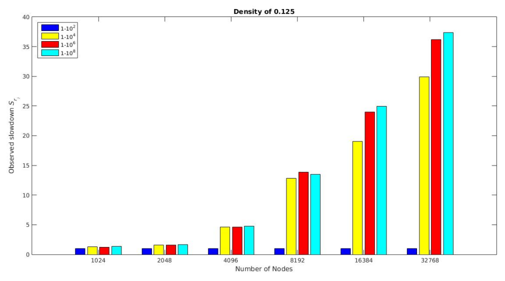
Figure 4.4: Slowdown in total execution time as graph density increases.

4.2 Effect of Weight Range on Initialization Time and Total Execution Time

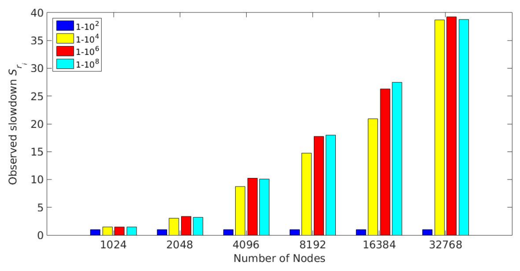
The timing results in Table 4.1 also indicate that as the weight range increases, both initialization time and total execution time tend to increase, regardless of the number of nodes or density. For smaller graphs, the increase is minimal. For example, a graph with $n = 1024$ nodes, density $d = 1$, and weight range r_i from 1 to 10^2 has an initialization time of 0.041 seconds and a total execution time of 0.047 seconds. In comparison, a graph with $n = 1024$ nodes, density $d = 1$, and weight range r_i from 1 to 10^8 has an initialization time of 0.204 seconds and a total execution time of 0.213 seconds. Hence, the difference in initialization time is only 0.163 seconds, and the difference in total execution time is only 0.166 seconds. For larger graphs, the increase is more significant. For example, a graph with $n = 32768$ nodes, density $d = 1$, and weight range r_i from 1 to 10^2 has an initialization time of 47.547 seconds and a total execution time of 49.087 seconds. In comparison, a graph with $n = 32768$ nodes, density $d = 1$, and weight range r_i from 1 - 10^8 has an initialization time of 2677.173 seconds and a total execution time of 2700.346 seconds. Hence, the difference in initialization time is 2629.626 seconds, and the difference in total execution time is 2651.259 seconds.

To more accurately measure the increases in initialization time and total execution time, we calculated slowdown values using equations (3.4) and (3.5). The results can be found in Table 4.3 and are presented graphically in Figures 4.5 and 4.6.

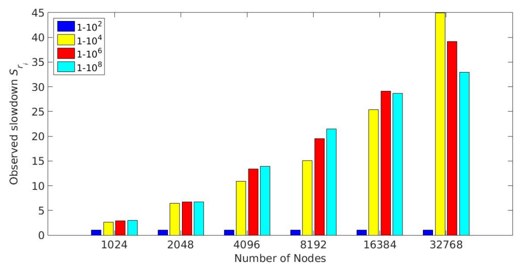
Overall, Figures 4.5 and 4.6 reinforce that as the weight range increases, both initialization time and total execution time tend to increase, regardless of the number of nodes or density. Moreover, they reinforce that as the weight range increases, slowdown in initialization time and total execution time is much smaller for small n, such as $n = 1024$ and $n = 2048$; and, slowdown is very large for large n, such as $n = 16384$ and $n = 32768$.



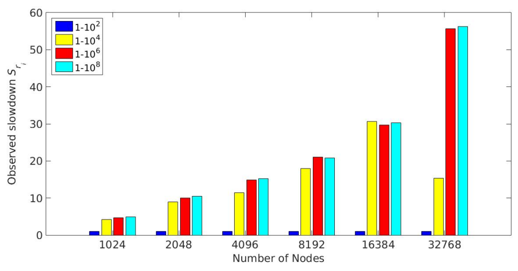
(a) Graphs with density = 0.125.



(b) Graphs with density = 0.25.



(c) Graphs with density = 0.5.



(d) Graphs with density = 1.

Figure 4.5: Slowdown in initialization time as weight range increases.

Figures 4.5 and 4.6 also reveal an interesting anomaly when the weight range increases from 1 to 10^2 to 1 to 10^4. For the majority of the graphs, there appears to be a spike in slowdown of initialization time and total execution time as the weight range is increased from 1 to 10^2 to 1 to 10^4; but, slowdown values do not change nearly as much when the weight range is increased from 1 to 10^6 and from 1 to 10^8. Further investigation is needed to determine why this happens.

4.3 Effect of Weight Scaling on Initialization Time and Total Execution Time

Given that initialization time and total execution time tend to increase as weight range increases, we hypothesized that scaling down the weights of the original graphs would lead to speedup in both times. We believed execution times would decrease because the scaled-down weight ranges consisted of real-valued numbers in the half-open interval (0,1], and thus the scaled-down weight ranges would be much smaller than any of the original weight ranges. Thus, speedup values were calculated using equations (3.6) and (3.7) and the results were recorded in Table 4.5. Speedup less than 1 indicates that a graph with a scaled-down weight range had a longer initialization time and/or total execution time than its corresponding original graph. Speedup equal to 1 indicates no change in initialization time and/or total execution time between a graph with a scaled-down weight range and its corresponding original graph. Speedup greater than 1 indicates that a graph with a scaled-down weight range had a shorter initialization time and/or total execution time than its corresponding original graph.

The results in Table 4.5 indicate that there are no discernible trends in speedup. Overall, there is a mix of speedup values less than 1, equal to 1, and greater than 1 for both

Table 4.3: Slowdown in initialization time and total execution time as weight range increases.

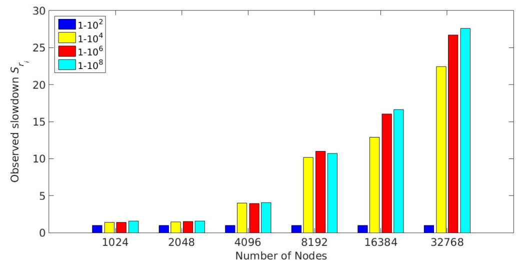
a) Slowdown in initialization time

n	m	d	$1 - 10^2$	$1 - 10^4$	$1 - 10^6$	$1 - 10^8$
1024	65472	0.125	1.000	1.300	1.200	1.400
1024	130944	0.250	1.000	1.438	1.438	1.500
1024	261888	0.500	1.000	2.619	2.905	3.000
1024	523776	1.000	1.000	4.268	4.683	4.976
2048	262016	0.125	1.000	1.571	1.592	1.633
2048	524032	0.250	1.000	3.058	3.391	3.217
2048	1048064	0.500	1.000	6.429	6.684	6.724
2048	2096128	1.000	1.000	8.929	10.024	10.452
4096	1048320	0.125	1.000	4.609	4.604	4.757
4096	2096640	0.250	1.000	8.699	10.264	10.074
4096	4193280	0.500	1.000	10.932	13.363	13.932
4096	8386560	1.000	1.000	11.428	14.830	15.267
8192	4193792	0.125	1.000	12.798	13.866	13.488
8192	8387584	0.250	1.000	14.776	17.734	17.980
8192	16775168	0.500	1.000	15.091	19.498	21.517
8192	33550336	1.000	1.000	17.910	21.007	20.757
16384	16776192	0.125	1.000	19.069	23.989	24.930
16384	33552384	0.250	1.000	20.887	26.264	27.432
16384	67104768	0.500	1.000	25.379	29.104	28.683
16384	134209536	1.000	1.000	30.686	29.690	30.315
32768	67106816	0.125	1.000	29.873	36.188	37.367
32768	134213632	0.250	1.000	38.684	39.257	38.795
32768	268427264	0.500	1.000	44.951	39.159	32.930
32768	536854528	1.000	1.000	15.340	55.723	56.306

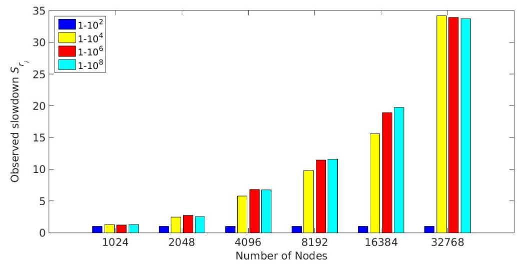
b) Slowdown in total execution time

n	m	d	$1 - 10^2$	$1 - 10^4$	$1 - 10^6$	$1 - 10^8$
1024	65472	0.125	1.000	1.400	1.400	1.600
1024	130944	0.250	1.000	1.250	1.200	1.250
1024	261888	0.500	1.000	1.968	2.097	2.097
1024	523776	1.000	1.000	4.255	4.298	4.532
2048	262016	0.125	1.000	1.463	1.500	1.556
2048	524032	0.250	1.000	2.440	2.747	2.527
2048	1048064	0.500	1.000	4.459	4.547	4.757
2048	2096128	1.000	1.000	7.667	8.525	9.015
4096	1048320	0.125	1.000	4.015	3.951	4.078
4096	2096640	0.250	1.000	5.750	6.801	6.723
4096	4193280	0.500	1.000	8.845	10.431	11.017
4096	8386560	1.000	1.000	10.118	12.615	12.912
8192	4193792	0.125	1.000	10.154	10.979	10.678
8192	8387584	0.250	1.000	9.762	11.417	11.560
8192	16775168	0.500	1.000	13.889	17.459	19.339
8192	33550336	1.000	1.000	16.286	18.330	18.428
16384	16776192	0.125	1.000	12.923	16.038	16.644
16384	33552384	0.250	1.000	15.594	18.908	19.772
16384	67104768	0.500	1.000	22.072	24.737	24.255
16384	134209536	1.000	1.000	30.292	28.618	29.046
32768	67106816	0.125	1.000	22.434	26.719	27.575
32768	134213632	0.250	1.000	34.176	33.919	33.694
32768	268427264	0.500	1.000	44.354	37.529	31.613
32768	536854528	1.000	1.000	17.895	54.396	55.011

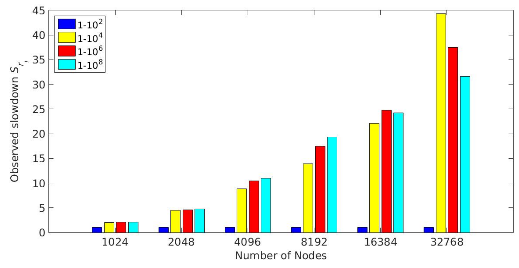
initialization time and total execution time. Of the 192 speedup values presented in Table



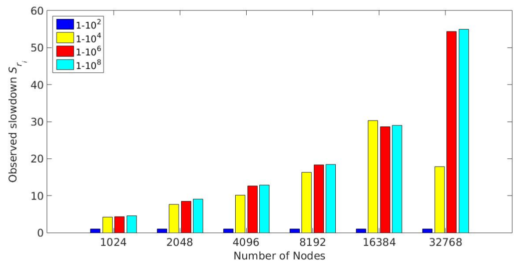
(a) Graphs with density = 0.125.



(b) Graphs with density = 0.25.



(c) Graphs with density = 0.5.



(d) Graphs with density = 1.

Figure 4.6: Slowdown in total execution time as weight range increases.

4.5, only 40 are greater than 1. This indicates that for the majority of the graphs, using the scaled-down weight ranges led to initialization times and total execution times that were either the same or slower than when the original weight ranges were used. But among the 152 speedup values less than or equal to 1, 114 were greater than or equal to 0.900 and 97 were greater than or equal to 0.950, indicating that slowdown was trivial in many cases. Additionally, the maximum speedup value in Table 4.5 was only 1.240, indicating that speedup was also trivial when it did occur for graphs with scaled-down weight ranges.

4.4 Effect of Modified Integer Weight Ranges on Initialization Time and Total Execution Time

We originally hypothesized that using modified integer weight ranges would lead to increased initialization times and total execution times. Thus, slowdown results were calculated using equations 3.8 and 3.9 and were stored in Table 4.7. "Slowdown(I)" stands for slowdown in initialization time, while "Slowdown(T)" stands for slowdown in total execution time.

No definitive conclusions can be drawn from this data. For example, slowdown values were higher for graphs with weight ranges 101 to 200, 301 to 400, 501 to 600, 701 to 800, and 901 to 1000 than graphs with weight ranges 1000001 to 2000000, 3000001 to 4000000, 5000001 to 6000000, 7000001 to 8000000, and from 9000001 to 10000000. Additionally, every slowdown value was between 0.836 and 1.266 until we tested graphs with the weight ranges from 700000001 to 800000000 and from 900000001 to 1000000000, where speedup spiked to over 14 in both cases. In order to best determine how these types of weight range modifications affect initialization time and total execution time, future studies should be more comprehensive. Graphs of various sizes should be used and more weight ranges should be tested.

Table 4.4: Initialization times and total execution times for graphs with various numbers nodes, densities, and scaled-down weight ranges.

a) Initialization time in seconds

n	m	d	$10^{-2}-1$	$10^{-4}-1$	$10^{-6}-1$	$10^{-8}-1$
1024	65472	0.125	0.012	0.013	0.012	0.014
1024	130944	0.250	0.018	0.024	0.023	0.025
1024	261888	0.500	0.022	0.057	0.059	0.065
1024	523776	1.000	0.042	0.188	0.192	0.205
2048	262016	0.125	0.057	0.080	0.078	0.081
2048	524032	0.250	0.068	0.225	0.235	0.221
2048	1048064	0.500	0.099	0.705	0.653	0.663
2048	2096128	1.000	0.167	1.655	1.677	1.755
4096	1048320	0.125	0.170	0.829	0.780	0.790
4096	2096640	0.250	0.216	2.056	2.225	2.185
4096	4193280	0.500	0.367	4.656	4.872	5.083
4096	8386560	1.000	0.676	9.711	9.934	10.140
8192	4193792	0.125	0.471	6.906	6.565	6.381
8192	8387584	0.250	0.763	14.334	13.584	13.745
8192	16775168	0.500	1.391	27.508	27.206	29.879
8192	33550336	1.000	2.599	58.726	54.024	53.059
16384	16776192	0.125	1.607	40.278	39.052	40.019
16384	33552384	0.250	2.912	79.871	77.390	79.822
16384	67104768	0.500	5.369	169.058	160.150	153.852
16384	134209536	1.000	10.251	370.340	432.790	310.871
32768	67106816	0.125	6.064	232.685	223.389	226.415
32768	134213632	0.250	11.194	493.116	458.500	434.452
32768	268427264	0.500	23.184	1313.296	936.438	1035.865
32768	536854528	1.000	60.461	849.706	2879.156	2838.738

b) Total execution time in seconds

n	m	d	$10^{-2}-1$	$10^{-4}-1$	$10^{-6}-1$	$10^{-8}-1$
1024	65472	0.125	0.013	0.014	0.014	0.014
1024	130944	0.250	0.022	0.026	0.025	0.026
1024	261888	0.500	0.025	0.064	0.063	0.066
1024	523776	1.000	0.048	0.209	0.203	0.214
2048	262016	0.125	0.059	0.082	0.080	0.084
2048	524032	0.250	0.107	0.233	0.251	0.230
2048	1048064	0.500	0.149	0.733	0.671	0.708
2048	2096128	1.000	0.204	1.707	1.730	1.837
4096	1048320	0.125	0.239	0.855	0.807	0.817
4096	2096640	0.250	0.336	2.091	2.287	2.269
4096	4193280	0.500	0.477	4.796	4.936	5.229
4096	8386560	1.000	0.794	10.053	10.292	10.447
8192	4193792	0.125	0.611	7.006	6.677	6.483
8192	8387584	0.250	1.187	14.569	13.800	13.929
8192	16775168	0.500	1.572	27.708	27.529	30.354
8192	33550336	1.000	2.873	59.723	54.143	54.097
16384	16776192	0.125	2.424	40.582	39.401	40.286
16384	33552384	0.250	4.086	80.815	78.213	80.771
16384	67104768	0.500	6.400	171.273	161.874	155.092
16384	134209536	1.000	10.757	379.232	438.403	313.418
32768	67106816	0.125	8.253	233.212	224.242	227.361
32768	134213632	0.250	12.987	498.499	459.942	437.986
32768	268427264	0.500	24.410	1341.316	944.145	1050.224
32768	536854528	1.000	61.962	960.990	2891.076	2856.454

Table 4.5: Speedup in initialization time and total execution time of graphs with scaled-down weight ranges.

a) Speedup in initialization time

n	m	d	$10^{-2}-1$	$10^{-4}-1$	$10^{-6}-1$	$10^{-8}-1$
1024	65472	0.125	0.833	1.000	1.000	1.000
1024	130944	0.250	0.889	0.958	1.000	0.960
1024	261888	0.500	0.955	0.965	1.034	0.969
1024	523776	1.000	0.976	0.931	1.000	0.995
2048	262016	0.125	0.860	0.963	1.000	0.988
2048	524032	0.250	1.015	0.938	0.996	1.005
2048	1048064	0.500	0.990	0.894	1.003	0.994
2048	2096128	1.000	1.006	0.906	1.004	1.001
4096	1048320	0.125	0.994	0.940	0.997	1.018
4096	2096640	0.250	1.000	0.914	0.996	0.996
4096	4193280	0.500	0.997	0.859	1.004	1.003
4096	8386560	1.000	0.981	0.780	0.990	0.998
8192	4193792	0.125	1.000	0.873	0.995	0.996
8192	8387584	0.250	1.000	0.787	0.996	0.998
8192	16775168	0.500	0.999	0.763	0.996	1.001
8192	33550336	1.000	0.983	0.779	0.993	0.999
16384	16776192	0.125	0.999	0.760	0.986	1.000
16384	33552384	0.250	0.997	0.759	0.985	0.997
16384	67104768	0.500	1.000	0.806	0.976	1.001
16384	134209536	1.000	1.000	0.850	0.703	1.000
32768	67106816	0.125	1.002	0.780	0.984	1.002
32768	134213632	0.250	1.001	0.879	0.960	1.001
32768	268427264	0.500	1.186	0.941	1.150	0.874
32768	536854528	1.000	0.786	0.858	0.920	0.943

b) Speedup in total execution time

n	m	d	$10^{-2}-1$	$10^{-4}-1$	$10^{-6}-1$	$10^{-8}-1$
1024	65472	0.125	0.769	1.000	1.000	1.143
1024	130944	0.250	0.909	0.962	0.960	0.962
1024	261888	0.500	1.240	0.953	1.032	0.985
1024	523776	1.000	0.979	0.957	0.995	0.995
2048	262016	0.125	0.915	0.963	1.013	1.000
2048	524032	0.250	0.850	0.953	0.996	1.000
2048	1048064	0.500	0.993	0.900	1.003	0.994
2048	2096128	1.000	1.000	0.916	1.005	1.001
4096	1048320	0.125	0.854	0.958	0.999	1.018
4096	2096640	0.250	1.000	0.924	0.999	0.996
4096	4193280	0.500	0.998	0.878	1.006	1.003
4096	8386560	1.000	1.018	0.813	0.990	0.999
8192	4193792	0.125	0.990	0.877	0.995	0.996
8192	8387584	0.250	1.014	0.807	0.996	0.999
8192	16775168	0.500	0.999	0.787	0.996	1.001
8192	33550336	1.000	1.021	0.800	0.993	0.999
16384	16776192	0.125	0.998	0.771	0.985	1.000
16384	33552384	0.250	0.997	0.786	0.985	0.997
16384	67104768	0.500	1.000	0.825	0.978	1.001
16384	134209536	1.000	1.003	0.862	0.704	1.000
32768	67106816	0.125	1.001	0.795	0.984	1.002
32768	134213632	0.250	1.001	0.892	0.959	1.001
32768	268427264	0.500	1.190	0.960	1.155	0.874
32768	536854528	1.000	0.792	0.914	0.924	0.945

Table 4.6: Initialization times and total execution times for graphs with modified integer weights.

r_i	r_m	Initialization Time (s)	Total Execution Time (s)
$1 - 10^2$	$101 - 200$	59.702	61.017
$1 - 10^2$	$301 - 400$	59.876	61.405
$1 - 10^2$	$501 - 600$	59.903	62.053
$1 - 10^2$	$701 - 800$	59.131	61.223
$1 - 10^2$	$901 - 1000$	60.188	62.008
$1 - 10^4$	$10001 - 20000$	734.177	757.254
$1 - 10^4$	$30001 - 40000$	712.700	734.272
$1 - 10^4$	$50001 - 60000$	713.916	834.620
$1 - 10^4$	$70001 - 80000$	695.741	942.709
$1 - 10^4$	$90001 - 100000$	707.156	754.112
$1 - 10^6$	$1000001 - 2000000$	2561.781	2577.299
$1 - 10^6$	$3000001 - 4000000$	2433.891	2457.337
$1 - 10^6$	$5000001 - 6000000$	2652.002	2676.802
$1 - 10^6$	$7000001 - 8000000$	2428.703	2457.649
$1 - 10^6$	$9000001 - 10000000$	2323.484	2357.401
$1 - 10^8$	$100000001 - 200000000$	2743.963	2757.626
$1 - 10^8$	$300000001 - 400000000$	2708.643	2726.601
$1 - 10^8$	$500000001 - 600000000$	2907.024	2920.235
$1 - 10^8$	$700000001 - 800000000$	38254.665	38265.777
$1 - 10^8$	$900000001 - 1000000000$	38717.599	38732.945

Table 4.7: Slowdown of initialization time and total execution time for graphs with modified integer weight ranges.

r_i	r_m	Slowdown(I)	Slowdown(T)
$1 - 10^2$	$101 - 200$	1.256	1.243
$1 - 10^2$	$301 - 400$	1.259	1.251
$1 - 10^2$	$501 - 600$	1.260	1.264
$1 - 10^2$	$701 - 800$	1.244	1.247
$1 - 10^2$	$901 - 1000$	1.266	1.263
$1 - 10^4$	$10001 - 20000$	1.007	0.862
$1 - 10^4$	$30001 - 40000$	0.977	0.836
$1 - 10^4$	$50001 - 60000$	0.979	0.950
$1 - 10^4$	$70001 - 80000$	0.954	1.073
$1 - 10^4$	$90001 - 100000$	0.970	0.859
$1 - 10^6$	$1000001 - 2000000$	0.967	0.965
$1 - 10^6$	$3000001 - 4000000$	0.919	0.920
$1 - 10^6$	$5000001 - 6000000$	1.001	1.002
$1 - 10^6$	$7000001 - 8000000$	0.917	0.920
$1 - 10^6$	$9000001 - 10000000$	0.877	0.883
$1 - 10^8$	$100000001 - 200000000$	1.025	1.021
$1 - 10^8$	$300000001 - 400000000$	1.012	1.010
$1 - 10^8$	$500000001 - 600000000$	1.086	1.081
$1 - 10^8$	$700000001 - 800000000$	14.289	14.171
$1 - 10^8$	$900000001 - 1000000000$	14.462	14.344

4.5 Effect of Graph Density on Memory Usage

Table 4.8 contains total memory usage (in gigabytes) for graphs with the initial integer weight ranges and graphs with the scaled-down real-valued weight ranges. Below the table are the associated R^2 values for the predicted values. Results are rounded to the nearest one-thousandth of a gigabyte. Ultimately, we found that varying the weight range did not change memory usage in many cases, and if it did it was by 0.001 gigabytes. Hence, the memory results in Table 4.8 are for graphs with two arbitrary weight ranges: 1 to 10^8 and 10^{-8} to 1.

Whether a graph has integer weights or real-valued weights, it is evident that memory usage is largely dependent on the number of nodes and edges in a graph. For large graphs (like the ones we generated), memory usage approximately doubles as density doubles. Additionally, the results indicated that for almost every n and d, a graph with integer weights uses less memory than a graph of the same size with real-valued weights. This is not surprising given that a real-valued weight is stored as a `double`, which requires more bytes than an `int`.

Using these results, we observed whether it is possible to estimate the amount of memory used from the number of nodes and edges. Performing a `sizeof` operation on the node and edge structs, we find that 72 bytes of memory are allocated for each node and 96 bytes of memory are allocated for each edge. Taking our example with $n = 32768$ and $m = 536854528$, 51540393984 bytes, or 51.540 GB, should be allocated. Compared to our linear-regression approximation of 57.990 GB, the calculation yields an underestimation; however, it provides a good foundation for the minimum amount of memory one should expect to use.

Table 4.8: Total memory usage (GB) for graphs with initial integer weight ranges and graphs with scaled-down real-valued weight ranges.

n	m	d	Initial Weight Ranges	Scaled-down Weight Ranges
1024	65472	0.125	0.007	0.007
1024	130944	0.250	0.014	0.015
1024	261888	0.500	0.028	0.029
1024	523776	1.000	0.057	0.059
2048	262016	0.125	0.029	0.030
2048	524032	0.250	0.057	0.059
2048	1048064	0.500	0.113	0.118
2048	2096128	1.000	0.227	0.235
4096	1048320	0.125	0.114	0.118
4096	2096640	0.250	0.227	0.235
4096	4193280	0.500	0.453	0.470
4096	8386560	1.000	0.906	0.940
8192	4193792	0.125	0.454	0.471
8192	8387584	0.250	0.907	0.940
8192	16775168	0.500	1.813	1.880
8192	33550336	1.000	3.624	3.758
16384	16776192	0.125	1.813	1.881
16384	33552384	0.250	3.625	3.760
16384	67104768	0.500	7.249	7.517
16384	134209536	1.000	14.496	15.033
32768	67106816	0.125	7.251	7.519
32768	134213632	0.250	14.498	15.035
32768	268427264	0.500	28.995	30.069
32768	536854528	1.000	57.990	60.137
			$R^2 = 0.99999999$	$R^2 = 0.99820064$

5 Conclusions

We determined that as graph density increases, initialization time and total execution time of a graph both tend to increase. Additionally, we determined that total memory usage of the algorithm is largely dependent on the number of nodes and edges in a graph. These conclusions were not surprising to us and were anticipated. Because Blossom V is a search algorithm, it seems logical that adding more edges to traverse would lead to increases in initialization time and total execution time. Also, it seems logical that adding more nodes and edges to a graph would lead to an increase in memory usage.

On the other hand, we also reached some very important conclusions that we had not previously anticipated. We determined that as the weight range of a graph increases, initialization time and total execution time tend to increase. Additionally, we determined that scaling down the initial integer weight ranges to real-valued weight ranges in the half-open interval (0,1] has a limited effect on initialization time and total execution time.

These conclusions indicate where future work should be focused when trying to improve the Blossom V implementation. Future implementations of the algorithm should address the issue of significantly increased run-time that results from having larger edge weights, especially since scaling down the weights shows no significant effect on run-time. Additionally, future implementations should aim to be more memory efficient in order to increase the range of computational problem sizes that can be run on a given hardware configuration. Finally, future investigations should consider parallelization of the algorithm with a focus on improving initialization time, which is where the majority of execution time was spent for the graphs that we generated.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (hpcreu.umbc.edu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2015. This program is funded by the National Science Foundation (NSF), the National Security Agency (NSA), and the Department of Defense (DOD), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-author Gabrielle Salib was supported, in part, through a contract from the National Security Agency (NSA), UMBC, the Meyerhoff Scholars Program. Graduate assistants Ari Rapkin Blenkhorn, Jonathan Graf, and Samuel Khuvis were supported during Summer 2015 by UMBC.

References

[1] David A. Bader and Kamesh Madduri. *GTgraph*: A synthetic graph generator suite. http://www.cse.psu.edu/~kxm85/software/GTgraph/gen.pdf. Accessed: 2015-16-07.

[2] UMBC High Performance Computing Facility. System description. http://hpcf.umbc.edu/system-description/. Accessed: 2015-14-07.

[3] José Fonseca. *gprof2dot*. https://github.com/jrfonseca/gprof2dot, 2015.

[4] H.N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. *In Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1(1):434–443, 1990.

[5] Free Software Foundation Inc. *GNU gprof*. https://sourceware.org/binutils/docs/gprof/. Accessed: 2015-30-07.

[6] Vladimir Kolmogorov. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Math. Prog. Comp.*, 1(1):43–67, 2009.

[7] Lszl Lovsz and Michael D. Plummer. *Matching Theory*. Akadmiai Kiad, 1986.

[8] Julian Seward and Nicholas Nethercote. Valgrind's tool suite. http://valgrind.org/info/tools.html. Accessed: 2015-12-07.