Evaluating the Performance of Shortest-Path Algorithms

Comparing Dijkstra's and Bellman-Ford's algorithms on runtime and memory usage

Konrad Wiley
Computer Science
University of South Carolina – Upstate
Spartanburg, South Carolina United States
rkwiley@email.uscupstate.edu

1. INTRODUCTION

Since the 1950's, computer algorithms for finding the shortest path between two nodes on a graph have been an area of significant theoretical and applied research in the field of computer science. Of the algorithms which emerged during the initial flurry of discovery, two have played a most prominent role in continued development. Dijkstra's algorithm, first proposed by the Dutch computer scientist Edsger Dijkstra in 1956, is an exceptionally efficient algorithm for solving the shortest-path problem but is only able to operate on graphs with exclusively positive edge lengths. For graphs which may contain negative edge lengths, the Bellman-Ford algorithm, published in 1958 and 1956 by Richard Bellman and Lester Ford respectively, defines the standard approach to calculating shortest-paths and detecting negative cycles in a graph. A negative cycle exists if a path may begin and terminate at a single node with the distance of all edges traveled being < 1. If such a cycle exists, any path between two nodes could be made "shorter" by repeating an additional trip around the negative cycle, indicating that no shortest path exists on the graph.

While variations of the standard Bellman-Ford algorithm have been created to improve efficiency for most graphs, no algorithm has been able to improve worst-case performance on graphs containing negative edges. Two significant approaches to improving the average-case performance of Bellman-Ford include the Shortest Path Faster Algorithm (SPFA) [1] and the hybrid Bellman-Ford-Dijkstra algorithm [2], both of which use the Bellman-Ford negative-cycle detection but attempt to improve performance by reducing the average number of relaxation operations performed.

It has been well established in computer science theory that Dijkstra's algorithm performs significantly faster on positive-edge graphs than Bellman-Ford's algorithm [3], however in this paper we will deliver real-world tests to demonstrate the performance differences between the algorithms and examine the effects of limited resources on running times. Much can be said about the loss of the art of code optimization in the present age of high-performance computers, and while many programmers may choose more general solutions to problems, sometimes a scalpel is worth more than a Swiss army knife. To this end, we aim to empirically demonstrate the efficiency of the scalpel – Dijkstra's algorithm – versus the Swiss army knife of Bellman-Ford. We will also implement and test several of the variants of Bellman-Ford to plot their data against Dijkstra's and highlight improvements where they appear. We hope this data will help inform rising computer science students of the value of selecting the most correct algorithms for the tasks they need to solve, rather than simply settling for code that works.

As we feel it is necessary to have a firm understanding of the two shortest-path algorithms to understand the intent of this paper, we will provide a brief overview of each of them in turn below.

1.1 Dijkstra's Algorithm

Dijkstra's Algorithm is commonly referred to as a "Greedy Algorithm," because it constantly pursues the lowest-cost route available to it while building its shortest-paths tree (as opposed to a depth-first or breadth-first traversal). The initialization process for Dijkstra consists of creating an empty list <dist> of distances from the source node to each node in the graph. These distances are all initialized to infinity, except for the distance to the source node which is initialized to zero (this intuitively makes sense, as we know that the shortest path to a node is itself). A queue <Q> is made which contains every node in the graph, and another list <S> is initialized as empty which will store nodes which have been visited during the traversal.

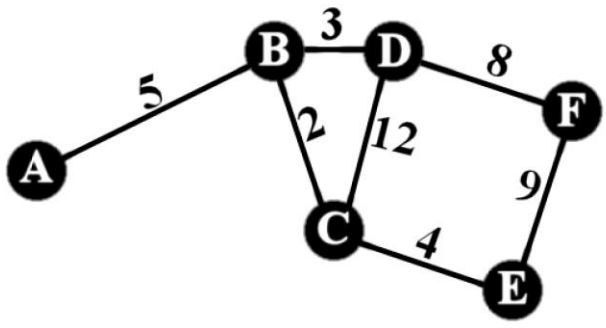

Figure 1.1: A simple undirected graph with positive edge weights.

<dist>		<Q>	<S>
A	0	A	
B	INF	B	
C	INF	C	
D	INF	D	
E	INF	E	
F	INF	F	

Figure 1.2: Dijkstra's initialization for the graph in Figure 1.1

.

The node selection process is rather straight-forward: while <*Q*> is not empty, select from <*Q*> the node <*v*> which has the smallest value in our distances list <*dist*> while also not yet being in our visted-list <*S*>. For our first iteration, the source node will be the only non-infinity value in <*dist*> while also not yet being in <*S*>, so it will be extracted.

Next, the selected node <*v*>'s neighbors are updated in the distances list <*dist*> using the following formula: if the distance of the selected node <*v*> + the edge weight of the neighbor <u> is less than the current distance value for <*u*>, then that sum of the distance of <*v*> plus the edge-weight of <*u*> replaces the current distance of <*u*>. If the sum is not less than the previous distance to <*u*>, then no update is made to the distances list <*dist*>.

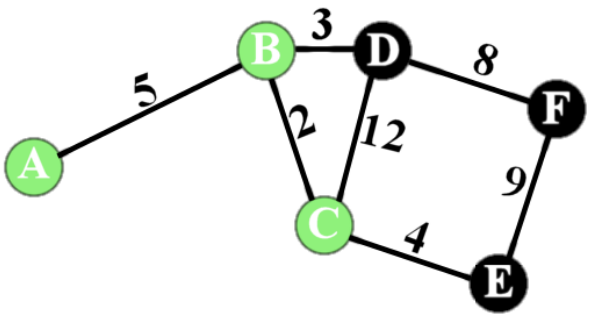

Figure 1.3: Three iterations of Dijkstra's, visited nodes highlighted

<dist>		<Q>	<S>
A	0	D	A
B	5	E	B
C	7	F	C
D	8		
E	9		
F	INF		

Figure 1.4: State of data structures at 3 iterations. Note that even though C is the most recently visited vertex, D's cost is still 8 because A->B->C was a more expensive route than A->B->D.

This process continues until all nodes have been removed from the queue <*Q*>, at which point <*dist*> will contain the shortest-paths tree. If you wish to be able to retrace the steps for the shortest path, then you simply need to add an additional level to your <*dist*> list, where each element would store both a distance and a reference to its parent. In the step where you update the distance of vertex <*v*>'s neighbors, you would replace <*u*>'s parent in <*dist*> with a reference to <*v*>. In this way, you can easily trace the graph back from your desired node and collect the shortest path.

<dist>			<Q>	<S>
node	dist	parent	D	A
A	0	NULL	E	B
B	5	A	F	C
C	7	B		
D	8	B		
E	9	C		
F	INF	NULL		

Figure 1.5: Example of data from Figure 1.4 with parent data included. Again, note that D's parent is B even though C had the most recently visited route.

1.2 Bellman-Ford Algorithm

Where Dijkstra's Algorithm is described as "Greedy," Bellman-Ford is best described as meticulous. Since it is designed to work on a broader range of graphs than Dijkstra's, it must be very careful to avoid infinite loops or mistakes in its traversal path, since negative edge weights introduce the possibility of infinite loops, and clear "shortest" priority is not necessarily apparent. At its core, Bellman-Ford's essential operation is based on the same calculation as Dijkstra's, namely the "triangle inequality."

In Bellman-Ford, this inequality is used during so-called "relaxation" procedures, where, like with Dijkstra, the distance from the source to a vertex <*v*> is updated if taking an adjacent vertex <*v*> edge to the vertex <*u*> would lead to a shorter distance than the current distance for <*v*>. Similarly to what we mentioned in the end of the section on Dijkstra's, Bellman-Ford also keeps a reference to the parent node after each of those updates.

The difference between Dijkstra's and Bellman-Ford lies in how they choose which vertices on which to perform this "relaxation" operation. While Dijkstra's selects the lowest-weight edge attached to a vertex still in its queue, Bellman-Ford instead iterates over *every edge in the graph* and relaxes the vertices attached to those edges. This process is performed |V| - 1 times, where V is the number of vertices in the graph. In a simple graph such as the one shown in Figure 1, Bellman-Ford would correctly update all the distances in the first iteration, but it would continue iterating over the graph four more times, performing the relaxation operation on each edge, despite there being no change to the shortest-path tree after the first iteration. This behavior is necessary in cases where edges may have negative values.

If, for instance, our graph from Figure 1.1 instead had a -8 edge connecting vertex D and F (see Figure 1.6), our first iteration of Bellman-Ford would not in fact result in the correct values for all distances, but would instead resemble Figure 1.7 (below).

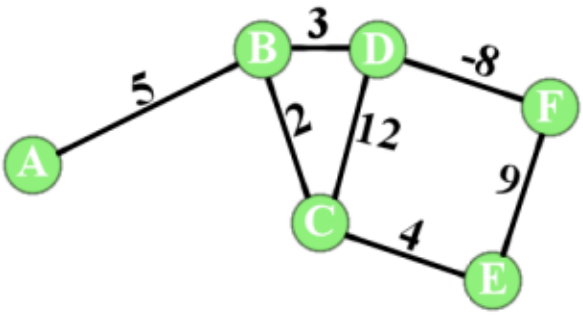

Figure 1.6: An undirected graph with a negative edge. All edges have been visited once by Bellman-Ford.

<dist>		
node	dist	parent
A	0	NULL
B	-5	D
C	4	D
D	-8	F
E	8	C
F	0	D

Figure 1.7: After one iteration of Bellman-Ford, the distances are far from correct, due to a negative edge cycle (undirected graph).

The remaining four iterations of Bellman-Ford would not result in a valid shortest-paths tree but would instead rely on another crucial feature of this algorithm – negative cycle detection. Because we have used an undirected graph, the path between D and F may be taken any number of times to simply reduce the cost of any other path around the graph. To detect issues such as this, Bellman-Ford performs one final iteration around the entire tree, checking if any edge meets the relaxation criteria. If any edges can still be relaxed after |V| - 1 relaxations, then there exists a negative cycle in the graph, and a shortest-paths tree cannot be built. As Bellman-Ford would reach the negative cycle detection loop, the last trip around would result in a "invalid" triangle inequality between D and F, alerting the algorithm to the presence of a negative edge cycle.

2. LITERATURE REVIEW

To assist us in the design and analysis of our experiment, we have selected five relevant papers, each focusing on at least one of the algorithms we will be studying. While all five papers are worth taking the time to read, we will provide a brief summary of their relevance to our research project below.

2.1 Hybrid Bellman–Ford–Dijkstra algorithm

This paper, authored by Yefim Dinitz and Rotem Itzhak [2], centers on the proposition of a new "hybrid" algorithm which allows for a much lower average-case runtime than Bellman-Ford through clever application of Dijkstra's algorithm, while still providing negative-cycle detection. Much attention is given to theoretical performance and mathematical proofs in the paper, as the authors took a deep-dive into the fundamentals of graph theory to defend the expected performance of their proposed "Bellman-Ford-Dijkstra" [BFD] hybrid algorithm.

We found the theoretical evaluation metrics to be well explained and well argued, but as the aim of our research project is to collect experimental data rather than providing theoretical proofs, the general applicability of this paper to our own research was the enrichment of our understanding of the underlying mechanics of graphing algorithms. No experimental data was provided in the paper, and no experiments were proposed. Despite this, there may be experimental value in an implementation of the BFD algorithm proposed, which would offer practical support for the authors' theoretical proofs.

2.2 Expected Performance of Dijkstra's Shortest Path Algorithm

This very short paper, authored by Andrew V. Goldberg and Robert E. Tarjan [4], offers a proof for the number of decrease-key operations performed by Dijkstra's algorithm. The value of this research is the insight it offers into the performance of Dijkstra implementations based on various data structures. Significant attention was drawn to the various operations which make up Dijkstra's algorithm, including which operations are performed in with what frequency. It is this analysis which provides the insight into the performance of decrease-key operations on a Fibonacci heap, as well as demonstrating why the reduced cost of such operations in a Fibonacci heap does not generally lead to performance increases in the overall algorithm, because comparatively few decrease-key operations are performed.

The core of the paper is a probabilistic analysis of the fundamental operations of Dijkstra's algorithm, yielding a theoretical time-complexity metric for each operation as a subset of Dijkstra's time-complexity for a given graph. The strongly theoretical nature of this paper helped us better understand the operations of Dijkstra's algorithm, and provided us with a clear choice in data structures for our experimental implementation of Dijkstra's, namely binary heaps. As our research objective is to provide experimental data confirming the theoretical performance of Dijkstra and Bellman-Ford, this paper aids us in understanding the theoretical performance of Dijkstra's.

2.3 Improvement And Experimental Evaluation Bellman-Ford Algorithm

This paper, authored by Wei Zhang et al. [5], also proposes a new algorithm based on the Shortest Path Faster Algorithm (SPFA) first published in 1994 by Fanding Duan [1]. Unfortunately, the proposed algorithm is not clearly explained in the course of the paper, however experimental results are provided comparing the proposed algorithm to SPFA and Dijkstra's algorithm on two different types of graphs. Of interest to the authors is the proposed algorithm's performance on grid-type maps, where their algorithm

demonstrates a significant improvement over SPFA. SPFA maintains an advantage on random maps, while the proposed algorithm's iterative approach is too costly in very large random maps.

We found the experimental methodologies helpful in the design and implementation of our own algorithm, as well as being benefitted by the explanation provided in this paper on the mechanics of SPFA, as it is an algorithm based on the Bellman-Ford Algorithm we examine in this paper. While we were not able to use the proposed algorithm due to its lack of clear explanation in this paper, the discussion of other algorithms and testing methods have been of great assistance in our work.

2.4 An improved Bellman-Ford algorithm based on SPFA

This paper, by Hao Chen et al. [6], builds on the proposed algorithm from 2.3 and provides a much clearer explanation of the algorithm being proposed. They propose an algorithm which updates the shortest-path tree for all nodes on a graph in an iterative manner, rather than calculating each shortest-path individually. In this way, a significant performance gain can be made over Bellman-Ford or even SPFA in highly interconnected graphs. They once again perform similar tests to those done in 2.3, however this time more results are gathered to build confidence in the final analysis. As before, the proposed algorithm outperforms even SPFA on grid-type graphs but is substantially slower on large random graphs of large depth values, due to the iterative nature of the algorithm. We find this paper useful in its explanation of the algorithm proposed in 2.3 and have benefited from its discussion of graphing algorithm metrics and the specifics of implementing and testing various shortest-path algorithms.

2.5 Two-Levels-Greedy: a generalization of Dijkstra's shortest path algorithm

This paper, authored by Domenico Cantone and Simone Faro [3], presents another novel implementation of Dijkstra's algorithm which allows input graphs with negative edges, but not negative cycles. The authors discuss at some length the fundamentals of graphing algorithms and standard approaches to shortest-path solutions. While Dijkstra's algorithm is typically referred to as a "greedy" algorithm, due to how it chooses which edges to explore, the proposed algorithm is described as "two-levels-greedy" because it selects the smallest strongly connected component in a topological ordering, and then selects the node with the minimal potential to begin scanning. This algorithm retains the asymptotic time complexity of Dijkstra's algorithm, while also providing linear complexity in the case of acyclic graphs.

The authors present experimental data comparing their TLG algorithm with several noteworthy graphing algorithms, including Bellman-Ford-Moore (BFM) and another variant of Dijkstra's algorithm as proposed by Dial. To perform the experimental tests, they utilized two classes of graphs: Rand-Len and Acyc-P2n. The former consists of strongly connected and dense graphs, and the latter are acyclic random graphs with variable fractions of negative edges. The two classes of graphs allow the performance of TLG to be measured across dissimilar graphs for which other algorithms in their experiment were known to have strong or weak performances. Their results demonstrate very strong performance from TLG compared to the other algorithms in their experiment and performance which was insensitive to the class of graph being scanned. TLG also had the lowest number of scan operations out of the tested algorithms.

The discussion of graph classes is highly relevant to our own experiment and aids us in the design of the graphs we use. Much of the formulation of the experiment provides a useful framework for building and implementing our experiment. However, this paper does not focus on a standard implementation of Dijkstra's and is instead concerned with graphing algorithms which could operate on graphs containing negative edges.

3. METHODOLOGY

As our goal was to gain experimental results comparing various shortest-paths algorithms to each other, we first needed to select which algorithms we wished to study, create adequate implementations of them, and then design a testing system whereby we could collect performance metrics on each algorithm in our experiment.

3.1 Shortest-Paths Algorithms

To this end, we selected both Dijkstra's Algorithm and Bellman-Ford's Algorithm. Dijkstra's was selected since it is the gold-standard for finding shortest-paths in graphs containing non-negative edges. Its usage in OSPF is fundamental to the performance of that protocol, as it provides what is effectively the lower-bound of shortest-path tree creation. Bellman-Ford is the worst-case standard for solving shortest-paths in graphs which may contain negative edges, as it provides negative cycle detection to avoid inaccurate results in cases where an infinitely decreasing loop exists within the graph. We elected to include this algorithm because it is known to be inefficient compared to Dijkstra's over graphs with only positive edge values, and we were curious as to the real-world performance of the algorithms compared to each other.

3.2 Evaluation Metrics

For our metrics, we had an obvious choice in measuring time-to-complete between the algorithms, but also were interested in any significant differences in memory usage. It should be noted that while Dijkstra's can be used discretely to find the shortest path between two nodes without building a full shortest-paths tree, we felt that for the purposes of a direct comparison, we should implement Dijkstra's to scan all nodes and build a shortest-paths tree for each node on the graph. In practice, ability to terminate upon reaching the destination is another real-world strength of Dijkstra's but not one we will evaluate in this experiment.

3.3 Coding Decisions

We chose to use Python 3.72 for our coding language, as we were comfortable with Python3 and felt confident we could implement the algorithms and testing systems. One benefit of Python is its fine control over memory resources being utilized by its scripts, and we were able to find a powerful library which let us extract the memory usage of individual method calls within our program. Within our testing system, we generate graphs of distinct vertex and edge counts, with the edge weights randomly distributed between 0 and 500 (inclusive) and with each edge's source and destination vertices randomly distributed throughout the graph. This provides us with a Rand-Len class graph as discussed in 2.5. The values for the vertex and edge counts are entered in two separate arrays, and each pair (0/0, 1/1, 2/2, etc., …) are used to generate a total of 50 random graphs, with all shortest-paths algorithms being evaluated over each graph.

All tests were run on a PC with Windows 10 version 1909, a quad-core i5-7600K @ 3.8GHz processor, and 16GBs of RAM.

3.4 Data Collection

To collect our data, each novel execution of our testing system generated a unique CSV file into which we inserted our experimental results. Each line in the CSV consists of the name of the algorithm to which it corresponds, the vertices and edge counts for the result, the total runtime of the algorithm, and the maximum memory usage of the algorithm. This data was then collated with data from other results of our testing system and used to produce tables and graphs to highlight trends in algorithm performance.

3.5 Graph Density

Since both Dijkstra's and Bellman-Ford's algorithms have runtimes strongly associated with the number of edges and vertices in the graphs on which they are being run, it was necessary to choose appropriate values for our randomly generated graphs. We ran several tests and quickly noticed that performance was significantly impacted by the ratio of edges to vertices, which is not unexpected when the algorithms' functionality is understood. We elected to use graphs of (100, 1000, and 10000) vertices and ran tests with two separate sets of edge values, one with edges equal to the number of vertices (100, 1000, 10000) and one with edges equal to five times the number of vertices (500, 5000, 50000). In an early and naïve attempt at our experiment, we initiated a test with 100,000 vertices and 500,000 edges, but we were unable to get Bellman-Ford to generate a result within an acceptable timeframe. We will discuss this further in the results analysis.

3.6 Algorithm

Below is the pseudo-code for our testing system

```
procedure Shortest-Paths-Tester(vertices_array, density)
1  csv = new csv file
2  csv.writeHeaders('Algorithm', 'Vertices',
3                   'Edges', 'Time (ms)',
4                   'Memory (MB)')
5  for vertices in vertices_array:
6      edges = vertices * density
7      graph = new Graph(vertices)
8      graph.createEdges(edges)
9      for index in range(50):
10         source = random(0, size – 1)
11         dest = random(0, size – 1)
12
13         dijkstraTime = graph.timerDijkstra(source, dest)
14         dijkstraMem = graph.memDijkstra(source, dest)
15         csv.writeRow('Dijkstra', vertices, edges,
                        dijkstraTime, dijkstraMem)
16
17         bellmanTime = graph.timerBellman(source, dest)
18         bellmanMem = graph.memBellman(source, dest)
19         csv.writeRow('Bellman', vertices, edges,
                        bellmanTime, bellmanMem)
```

4. RESULTS ANALYSIS

As discussed in section 3.5, we performed tests over two sets of graph densities – one with E = |V| and another with E = 5 * |V| where E is the number of edges in the graph and V is the number of vertices in the graph. For each density, we performed 50 tests with each algorithm over increasing values for V (100, 1000, 10000). A separate procedure was created to collect the memory usage by each algorithm, since we found the time results to be heavily affected by the memory profiler.

Because the results from our two sets of graph densities are difficult to integrate visually, we have generated separate graphs highlighting the differences between our two shortest-paths algorithms over the graph densities.

4.1 Hypothesis

Our initial hypothesis was that Bellman-Ford would take exponential time relative to the input graph size, while Dijkstra's would take logarithmic time relative to the input graph size. These estimates were based on the theoretical O(|V|*|E|) performance of Bellman-Ford's algorithm compared with the theoretical O(E + |V|log|V|) performance of Dijkstra's algorithm. We did not predict any significant difference in memory usage between the algorithms, since Bellman-Ford and Dijkstra both perform essentially similar functions in memory, without creating significantly impactful data structures. While Bellman-Ford would be expected to require its memory resources for longer, due to its expected runtimes, the maximum and average memory usage was not expected to be noticeably different.

4.2 Experimental Results

As mentioned previously, we have separated the data for the E = |V| and E = 5*|V| graphs to allow for cleaner data presentation. Figure 4.1 shows the runtimes on a logarithmic graph, since Bellman-Ford's performance was indeed exponential, and a standard graph resulted in the growth rate of Dijkstra's being buried by the results of Bellman-Ford. By utilizing a logarithmic graph, we were able to clearly present both sets of data without introducing unnecessary visual clutter or losing critical data.

Our results (see Figure 4.1) for graphs containing equivalent edges and vertices demonstrate a significant difference in runtimes

between Bellman-Ford and Dijkstra which grows proportional to the number of edges and vertices in the graphs.

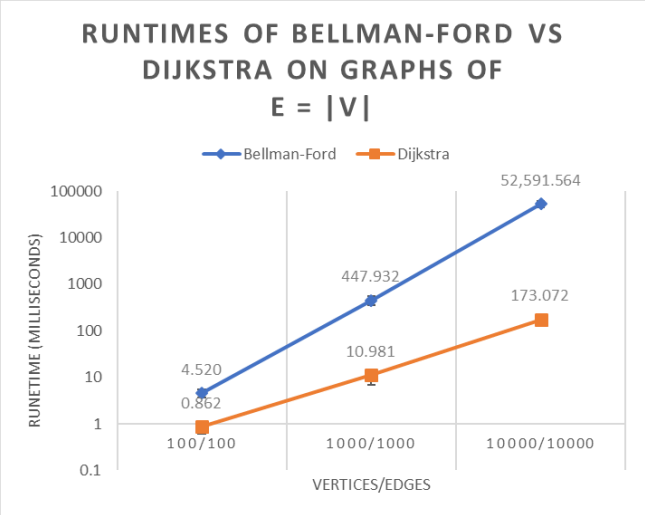

Figure 4.1: Results from graphs with equal edges and vertices. (n=50)

This growth is significant, as we got a P-value of far less than 0.05 for each vertex count (see Figure 4.4). At no point did our mean plus standard deviation approach each other for our entire runtime analysis. The results in table form are shown below as Figure 4.2 and Figure 4.3.

Vertices/Edges	Dijkstra	
	Time (ms)	Memory (MB)
100/100	0.86 ± 0.24	29.43 ± 0.03
1000/1000	10.98 ± 4.01	30.34 ± 0.02
10000/10000	173.07 ± 48.04	37.55 ± 1.16

Figure 4.2: Dijkstra results table for Figure 4.1 with standard deviation. n = 50

Vertices/Edges	Bellman-Ford	
	Time (ms)	Memory (MB)
100/100	4.52 ± 0.89	29.36 ± 0.03
1000/1000	447.93 ± 102.35	29.83 ± 0.00
10000/10000	52,591.56 ± 8328.12	33.15 ± 0.10

Figure 4.3: Bellman-Ford results table for Figure 4.1 with standard deviation. n = 50

Vertices/Edges	Confidence Value (P)	
	Time	Memory
100/100	4.09E-39	0.3846
1000/1000	4.02E-34	0.1824
10000/10000	2.10E-41	1.0000

Figure 4.4: Confidence values for Figure 4.2 and 4.3.

We also analyzed the memory usage of both algorithms during our testing. Our results for the graphs with equal edges and vertices did not indicate any significant difference between the algorithms in terms of total memory usage (Figure 4.2 – 4.4). There was an anomaly in Figure 4.4 where the 10,000 x 10,000 graph such a low significance in the variance between the two data sets as to result in a P-value of 1. This can be observed in the data presented in Figure 4.2 and Figure 4.3 where the values for the 4th line overlap for both sets of data.

The data sets with E = 5 * |V| provided even stronger evidence of the performance differences inherent in the two algorithms. At this density, the "iterate over every edge in E a total of V-1 times" process for edge relaxation within Bellman-Ford caused runtimes to become overwhelming. In an early attempt at building data sets, we tried to gather performance data on a 100,000 x 500,000 graph but were unable to complete a single test of Bellman-Ford over the course of no less than 12 hours. Another attempt was made for an additional 8 hours but also did not yield any results. After completing shorter tests, a brief analysis of the trend lines led us to understand how time consuming that graph would be, and since we needed to gather data from multiple test runs and also run the tests multiple times with the memory profiler attached (which added significant time to every run), we decided this was not a test which would provide benefits that would outweigh the costs.

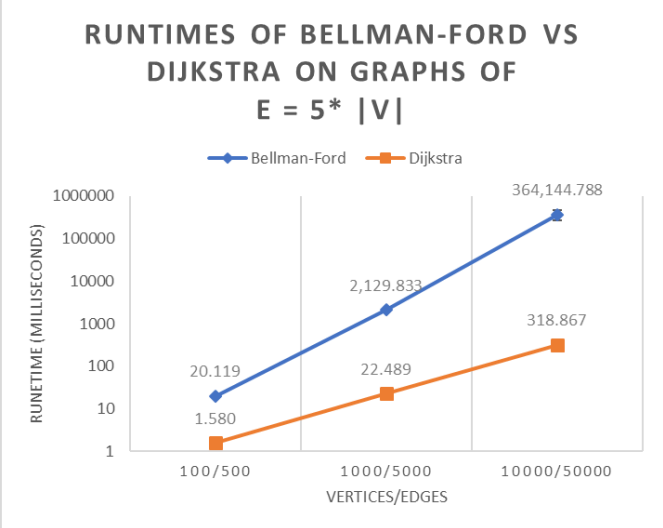

Figure 4.5: Results from graphs with 5 times more edges than vertices. (n=50)

As you can see in Figure 4.5, Bellman-Ford's runtime was increasing at an alarming rate, especially since our memory-profiler added significant overhead to each overall test. Even at a very generalized expectation of the performance at 100k x 500k, our approximate estimate of minimum runtime for Bellman-Ford was 5.5 hours for a single test (out of 50 for our data set) – before the memory profiler would run. Our real-world results indicated that there were additional bottlenecks causing performance to stall, since in both our tests at that size we waited longer than 5.5 hours without getting a result.

The results for this data set also indicated significant differences in the performances of the two algorithms, with Dijkstra showing very small increases in total runtime compared to the extreme runtimes produced by Bellman-Ford at the larger graph sizes.

Vertices/Edges	Dijkstra	
	Time (ms)	Memory (MB)
100/500	1.58 ± 0.03	29.43 ± 0.03
1000/5000	22.49 ± 0.85	30.34 ± 0.02
10000/50000	318.87 ± 49.02	37.55 ± 1.16

Figure 4.6: Dijkstra results table for Figure 4.5 with standard deviation. n = 50

Vertices/Edges	Bellman-Ford	
	Time (ms)	Memory (MB)
100/500	20.12 ± 0.25	29.44 ± 0.03
1000/5000	2,129.83 ± 48.56	30.32 ± 0.02
10000/50000	364,144.79 ± 96,564.37	37.17 ± 2.46

Figure 4.7: Bellman-Ford results table for Figure 4.5 with standard deviation. n = 50

Memory usage between the two algorithms was again negligibly different, with no statistically significant difference present across the test (see Figure 4.8). As with the 10,000 x 10,000 graph mentioned previously, we again achieved a P-value of 1 for one of our data sets, indicating no statistical difference between the two sets of data -- the memory usage for Bellman-Ford on a 1000 x 5000 graph was not distinguishable from the memory usage of Dijkstra on the same graph, and this held for 50 different graphs of that size.

Confidence values for runtimes were also statistically significant, with P-values approximating 0 in all cases (Figure 4.8). This indicates that there is effectively 100% confidence that the values for each algorithm's results are distinct.

Vertices/Edges	Confidence Value	
	Time	Memory
100/500	3.97E-93	0.073185
1000/5000	3.25E-82	1.000000
10000/50000	8.06E-31	0.143803

Figure 4.8: Confidence values for Figure 4.6 and 4.7

5. CONCLUSION

Our results indicate that our experimental hypothesis stating that we expected to find a significant difference in runtime between Bellman-Ford and Dijkstra's algorithms proportional to the size of the input graphs was valid. We reject the null hypothesis that there is no significant difference between the runtimes of the two algorithms over the graphs tested in our experiment. We also accept the null hypothesis stating that no significant difference exists between the memory usage of the two algorithms over our experimental graphs.

In the simplest sense these results were hardly surprising, since Bellman-Ford is widely known to be less efficient than Dijkstra's algorithm on graphs containing only positive edges. However, the magnitude of the difference in our results was astonishing to us. In practice, Bellman-Ford would deliver cripplingly slow performance over large and densely connected graphs, with days needed for even powerful computers to calculate the shortest-paths tree. It is clear why so much effort has been invested in the intervening years since this algorithm's invention trying to improve its average case and reduce the number of worst cases. If OSPF had been implemented with Bellman-Ford instead of Dijkstra, router calculations would be incredibly slow to update as network density increased. A 100,000 node network with 500,000 connections is not an unreasonable concept for a core router to need to service, but as our experiment here has shown, calculating shortest-paths as nodes and connections are added or removed would require tremendous computing power if OSPF had to take into consideration some form of "negative" connection type.

Fortunately, there are many situations where "negative edge weights" are not of any significant concern, and Dijkstra's algorithm is sufficient for these purposes. Hopefully more efficient improvements to Bellman-Ford such as SPFA and TLG are able to help bridge the gap in performance in the years to come.

REFERENCES

[1] Fanding Duan. 1994. A Faster Algorithm For Shortest Path-SPFA. *Journal of Southwest Jiaotong University*, 29, 6 (1994), 207-212.

[2] Yefim Dinitz, Rotem Itzhak. 2017. Hybrid Bellman–Ford–Dijkstra algorithm. *Journal of Discrete Algorithms*, 42 (2017), 35-44. DOI: https://doi.org/10.1016/j.jda.2017.01.001.

[3] Domenico Cantone and Simone Faro. 2004. Two-Levels-Greedy: a generalization of Dijkstra's shortest path algorithm. *Electronic Notes in Discrete Mathematics*, 17 (2004), 81-86, DOI: https://doi.org/10.1016/j.endm.2004.03.019.

[4] Andrew Goldberg and Robert Tarjan. June 1996. *Expected Performance of Dijkstra's Shortest Path Algorithm*. NEC Technical Report TR-96-062. NEC Research Institute, Princeton, NJ.

[5] Wei Zhang, Hao Chen, Chong Jiang, and Lin Zhu. 2013. *Improvement And Experimental Evaluation Bellman-Ford Algorithm.* DOI: https://doi.org/10.2991/icaicte.2013.29.

[6] Hao Chen, Yanji Liu, Xi Li and Heejong Suh. 2012. An Improved Bellman-Ford Algorithm Based on SPFA. *The Journal of the Korea institute of electronic communication sciences*, 7 (2012), 524-525.
DOI: https://doi.org/10.13067/JKIECS.2012.7.4.721