An Analysis of Shortest Path Algorithms

Brendan Gibson, Connie Bernard, Tyler Trogden, Wes Robbins

Due: December 10, 2021

1 Introduction

The single-source shortest path is a rather famous computer science problem, with numerous real world applications and countless proposed algorithms. Given a weighted graph, or a set of nodes and weighted edges, these algorithms must determine the shortest path from a source node to all other nodes.

One real world application of this problem is route finding. Google Maps, Waze, and Apple Maps are all different applications that allow users to find the shortest path from their current location to a destination of their choosing. The roads represent edges, their weight being the distance from one node to the next, and the nodes are the intersections of roads.

Perhaps two of the most well known shortest path algorithms are Dijkstra's algorithm and the Bellman-Ford algorithm. Each has their own strengths and weaknesses, and both have inspired several variations that improve upon the time and space complexities of the originals. This paper will examine the two original algorithms, an improvement for each, implement them in Python, analyze them mathematically, and then use the code to run timed experiments. Doing so will reveal the pros and cons of each algorithm, and the specific applications each might be best suited for.

2 Algorithms

In this section, we overview three essential single-source shortest path algorithms. Additionally, we provide an in-depth discussion of an algorithm that uses a few alternate heaps to accelerate Dijkstra's algorithm.

2.1 Bellman-Ford Original

The Bellman-Ford algorithm is a fundamental algorithm to the single-source shortest path (SSSP) problem. The algorithm solves SSSP for all graphs—weighted or unweighted, directed or undirected, and negative or non-negative. The algorithm iterates through each vertex in a graph and performs a nested loop through each edge leaving each vertex, relaxing edges throughout the iteration (discussed further in Section 3.1). Intuitively, we can view the algorithm as spreading out from the source node in a breadth-first manner. In *Algorithms*, Jeff Erickson (2019) describes this as a wave front emanating from the source node. In practice, Bellman-Ford is slow—$\Theta(VE)$—and rarely used. However, the improvements are primarily obtained from improving node-visiting order, improving the underlying data structure, or using heuristics—*not* changing the underlying principle. I.e., edges must be relaxed to solve SSSP. In this sense, all other shortest path algorithms are Bellman-Ford-like.

The algorithm was first proposed in 1955 by Alfonso Shimbel (Shimbel, 1955). The algorithm was proposed separately twice more—by Lester Ford in 1956 (Ford, 1956) and Richard Bellman in 1958 (Bellman, 1958). It is not definitively known why Shimbel is not in the algorithms name.

2.2 Moore-Bellman-Ford Improvement

This Bellman-Ford variation was first proposed in 1959 by Edward Moore in Proceedings of the International Symposium on the Theory of Switching (Moore, 1959). The algorithm operates very similarly to the original Bellman-Ford. There are a collection of 'tense edges', edges that have a potential shorter route from node A to node B, and the algorithm works to turn these tense edges into relaxed edges, the shortest route from A to B. Also known as the Shortest Path Faster Algorithm (SPFA) (Yen, 1970), this update differs from the original in one key aspect, that is, "instead of trying all vertices blindly, SPFA maintains a queue of candidate vertices and adds a vertex to the queue only if that vertex is relaxed" (Shortest path faster algorithm, n.d.). Like the original Bellman-Ford algorithm, its input is a directed weighted graph and the weights may be negative (Erickson, 2019).

2.3 Dijkstra's Original

Dijkstra's algorithm is an algorithm used to find the single-source shortest path of a weighted graph $G = (V, E, w)$ given a source vertex $s \in V$. The algorithm must be implemented on a directed graph. However, if a given graph is not directed, then we can simply replace every undirected edge with two directed edges of the same weight. Further, Dijkstra's algorithm is a special case of the Bellman-Ford algorithm where the weights on each edge are arbitrary. The difference in Dijkstra's algorithm is that the edge weights must be non-negative, i.e., for every $(u, v) \in E$, $w(u, v) \geq 0$. The output for Dijkstra's is a shortest-path tree which gives all connected paths from s to every other $v \in V$ with its shortest distance.

Like most (if not all) SSSP algorithms, Dijkstra's relies on the idea of tense edges, that is, for some given vertices $u, v \in V$ with approximate minimum distance $dist[v]$ and minimum distance $dist[u]$, if $dist[v] > dist[u]+w(u, v)$, then we update $dist[v]$ to $dist[v] = dist[u]+w(u, v)$. Dijkstra's works by searching the vertices not yet part of the shortest-path tree to find the one which has the current smallest distance from s, this vertex is then added to a priority queue where we check its neighbors, and so on. For this reason, Dijkstra's is considered a greedy algorithm. It chooses the vertex with the smallest distance from s and then recurses.

2.4 Improvement On Dijkstra's Algorithm

Like Bellman-Ford, computer theorists have sought to improve Dijkstra's algorithm. How can Dijkstra be improved though? Well, Ahuja, Mehlhorn, Orlin, and Tarjan (1990), through some input constraints and data-structure augmentation, managed to successfully quicken Dijkstra's runtime.

The first change from the standard Dijkstra's algorithm to the improvement is the use of a Radix heap (Ahuja et al., 1990, pp. 215-217). Dijkstra's algorithm usually runs by utilizing some form of a priority queue, including heaps, like a binary heap (Sedgewick & Wayne, 2011). Hence, to quicken Dijkstra's algorithm, using a different data structure is a good first step. There are some key differences between a Radix heap and the binary heap normally used in Dijkstra's. To start, a Radix heap is a monotone priority queue, which means any new key inserted into the heap must be greater than the last key extracted from the heap (Akiba, 2015). A Radix heap also consists of buckets with specific ranges and sizes dependent on the index of the queue (Ahuja et al., 1990, p. 215-216). Vertices inserted or decreased will be moved to specific buckets that match the range of the key (p. 216)[1]. To delete the minimum vertex, if not already in the smallest bucket, a Radix heap will search in the next smallest buckets and find the minimum there (p. 216). If this bucket has more than one vertex, the vertices are distributed throughout the heap through decreased calls (p. 216). With these buckets, a Radix

[1]All page 216 references are from the same reference Ahuja et al., 1990. The citations were condensed for sentence clarity.

heap is considered to be one-leveled. A two-level Radix heap includes everything prior, but also has equally spaced (and functionally sized) segments within the buckets to decrease the amount of reinserts needed when moving vertices (Ahuja et al., 1990, p. 217). Understanding this alternative data structure is key to understanding the accelerated form of Dijkstra.

Another change made to Dijkstra's algorithm was adding a Fibonacci heap along with the Radix heap. Fibonacci heaps are an unorthodox heap structure. They have four pointers at each node: One for the parent, one for the root, and one each for the neighboring children (Moore, n.d.)[2]. Fibonacci heaps are also composed of multiple heap based trees. The four pointers mentioned earlier maintain their connections between these trees. When the minimum node is deleted, all trees of equal rank are merged into the fewest trees of unequal rank. Each parent node in a Fibonacci heap can have up to $\log(n)$ number of children. The second child, and any child onward, removed from any parent becomes its own separate tree too. This is managed by marking parent nodes and checking their marked status. Lastly, like the binary heap, Fibonacci heaps are a doubly-linked list, which means it can insert and delete in constant time. Instead of using a Fibonacci heap alone though, Ahuja et al. (1990) decided to combine the Radix heap and an extended Fibonacci heap together to locate non-empty segments better than either could do on their own, further improving runtime (pp. 218-219).

The algorithm locates and manages these vertices by extending the Fibonacci heap's functions. Each vertex in a segment of a bucket would be put into a set $S(i)$ (i being an index based on the segment and bucket in question) and the Fibonacci heap (Ahuja et al., 1990 p. 218, 220). This helps differentiate which nodes are representatives of the set, which trees to *activate* or turn *passive*, and which nodes to delete or decrease (Ahuja et al., 1990, p. 220). Each function of the Fibonacci heap manipulates the heap trees and sets to differentiate what vertices are essential to check and scan. Each play a part and interact in the final algorithm.

The algorithm works as follows[3]:

1. Create a combined Radix heap and a Fibonacci heap.
2. Insert the source vertex s into the heap.
   (a) To insert a vertex, make it a root of a one-node tree.
   (b) If the set $S(i)$ is empty, make the vertex a representative and activate its tree.
   (c) If the set $S(i)$ is not empty, insert the vertex but keep its tree passive.
3. Delete the minimum vertex. This vertex, v, is now considered fully scanned.
   (a) When deleting the minimum, proceed like a regular Fibonacci heap, but only scan the roots of the active trees.
   (b) Find the minimum of the active trees. Delete this vertex from the set containing it.
   (c) Should the set be non-empty, choose another vertex in the set to be the new representative of the set.
   (d) Activate the tree with the root that is this new representative vertex.
   (e) Merge all active trees of equal rank.
4. Inspect all neighbor vertices $n \in N$ connected to this new scanned vertex v.

[2]See previous footnote. This is with respect to Moore (n.d.) though. Subsequent sentences of the paragraph is referenceable from Moore (n.d.) unless otherwise cited.

[3]Pseudocode was not provided. Instead, a list of instructions are provided. Author Brendan Gibson felt it was the next best method to relay the algorithmic steps. The steps are based on the original Ahuja et al. (1990) paper with supplementary help from Moore (n.d) for clarity on Fibonacci heaps.

5. If a neighboring vertex distance, $d(n)$, is infinite, let $d(n) = d(v) + edgeCost(v, w)$ and insert this vertex into the heap. If $d(n)$ is less than infinity and a new cost $d(v) + edgeCost(v, w) < d(n)$, then decrease this vertex in the heap with the new distance.
   (a) To insert a vertex, refer to step 2.
   (b) To decrease a vertex, update the key of the vertex to the new tentative weight. Simply decrease the key if heap structure is not violated.
   (c) If the heap structure is compromised, do the following:
      i. Cut link from v and its parent node.
      ii. If the parent node is not a root and not marked, mark it.
      iii. If the parent is marked, remove the node from its parent and mark it if unmarked. If marked, repeat this process of removing marked parents until an unmarked parent is reached or the root of the tree is reach. If the unmarked node reached is not a root, mark it.
   (d) After a decrease, do the following:
      i. Move v from $S(i)$ to some appropriate set $S(j)$.
      ii. If $v \in S(j)$ is the only vertex in the set, activate its respective tree.
      iii. If the set $S(i)$ is not empty after the transfer, select a new representative, vertex y, and activate its respective tree.
      iv. For any tree made by the series of cuts required for the decrease, activate them.
6. Repeat steps 2 through 5 until all vertices in the graph are scanned.

The most important caveat and substantial change from the standard Dijkstra' algorithm though was only considering a specific problem set. Dijkstra's algorithm works with any edge weight that is in the set of positive real numbers, but Ahuja et al. (1990) decided to constrain the inputs by only allowing positive, moderately sized integers bounded by a constant (pp. 214-215). Most likely, this is due to the quantity and size of buckets being dependent on a function with the bounded integer constraint as part of its input. The number of buckets, B, for a Radix heap is "$\lceil \log_2(C + 1) \rceil + 2$", and the size of the last bucket created (indexed bucket B) is "$nC + 1$" where n is the number of vertices in the graph. (Ahuja et al., 1990, pp. 215-216). At a basic level, this is reasonable, because data structures are discrete structures. After all, there is no such thing as half an index. A two-level Radix heap is similar but with the number of buckets being determined by $\log_K$, K being any integer parameter and used later for segment sizes (Ahuja et al., 1990, p. 217). Perhaps real numbers may not be able to create enough sufficient buckets for a seamless run of Dijkstra, but this is personal conjecture and not corroborated by the authors. Still, integer constraints are a factor to consider for applicably.

Now, when Dijkstra's algorithm under these bounded conditions uses a one-level Radix heaps, it runs at an asymptotic speed of $O(E + V \log C)$ (Ahuja et al, 1990, pg. 216). Should integer $C < V$, then it is faster than a standard binary heap. A binary heap has a runtime of $O(E + V \log V)$ (Ahuja et al., 1990, p. 214). When Dijkstra's algorithm uses a mix of a two-level Radix heap and a Fibonacci heap, it runs at an asymptotic speed of $O(E + V\sqrt{\log C})$ (Ahuja et al., pg. 219). This augmented version of Dijkstra's algorithm is clearly faster than the one referenced in Erickson's Algorithms (2019, pg. 288)[4] and runs faster than both prior heaps used. Should a programmer find themselves in a situation where they are given a shortest path problem with integer weighted digraphs and a known max weight, this algorithm would be a prime choice.

[4]Erickson does comment about priority queues, heaps, and the improved running time $O(E + V \log V)$ in his footnote on the same page.

However, the input parameters are a drawback. According to Ahuja et al. (1990), the runtime is achievable give the condition that the edges are moderately sized positive integers bounded by a constant (p. 215). Hence, the algorithm is inflexible. Real, non-negative numbers are not considered nor used. Therefore, the speed of this augmented algorithm is only useful when the input parameters fit the constraints.

The input constraint is not the only drawback though. Programming overhead and runtime incongruity may also be an issue. Fibonacci heaps have a reputation for being challenging to implement, and their methods in particular sometimes do not match their theoretical runtime (Fredman, Sedgewick, Sleator, & Tarjan, 1986, p. 112). The reason Fibonacci heaps have inconsistent runtimes is because they use four pointers for each node, as opposed to two or three for other heaps (Fredman et al., 1986). Heap distortion is also at play. Fibonacci heaps are a collection of heap structured trees, so the time efficiency starts to decay the more distorted the heap structure becomes (Moore, n.d.). Even an alternative heap, like a paring heap, is practically faster (Moore, n.d.). Hence, there are no guarantees that this algorithm would be realistically faster. Should the Fibonacci heap match its theoretical runtime though, then it is the faster algorithm. However, this disconnect between theory and practicality harms the potential application of this algorithm.

This faster version of Dijkstra was not included into our statistical analysis because of the aforementioned overhead and a lack of pseduocode from the original source. To program this algorithm without pseduocode as a reference is not impossible, but it would have taken significantly more time. Meanwhile, the other three algorithms are already well-developed, referenceable, and reputable. Thus, we can only compare by theoretical runtimes, data-structures, and potential applicability. This algorithm was still worth mentioning though, as a means of showcasing how standard, well-known algorithms can be augmented and manipulated to decrease their runtime. Research is perpetual and ever expanding, even with algorithms that are thought to be fully optimized.

To summarize, this faster version of Dijkstra's algorithm seamlessly takes advantage of extended, augmented heap methods and integer constraints, but at the cost of input flexibility, runtime certainty, and manageable overhead. Whether these sacrifices are worth the decrease in runtime is up to the programmers discretion. A standard Dijkstra's algorithm is sufficient to find a single-source shortest path solution, but if the problem sets of graphs are more specific (positive integers only), then this algorithm is definitely worth considering.

3 Implementations and Comparison

In this section, we discuss our implementation of SSSP algorithms. Additionally, we compare the algorithms asymptotically and based on observed experimental results.

3.1 Bellman-Ford Original

This implementation served as the basis for both the Moore-Bellman-Ford algorithm and one possible implementation of Dijkstra's algorithm (Figure 3). Shown in Figure 1, the algorithm took an adjacency list input which represented the graph and returned an array containing the shortest distance from the source node to all other nodes. This implementation utilizes a queue to keep track of vertices in the graph. As long as the queue contains a vertex u, whose distance to the source node needs to be evaluated, it will continue running. On line 11, a for-loop ensures that each node connected to node u is evaluated for a shorter distance. On line 12, if the answer already contains a distance from the source node to the given node, check to see if the new distance using the node most recently added to the queue would give a shorter overall distance. If it would update the answer, add the edge to the queue. On line 16 if the answer does not contain any distance from the source to a given node B, then it is set based on the

current node B and edge u. The vertex is then added to the queue.

```
7   def ford(self, graph, answer):
8       queue = [0]
9       while len(queue) != 0:
10          u = queue.pop()
11          for v,weight in graph[u]:
12              if answer[v] != None:
13                  if answer[v] > answer[u]+weight:
14                      answer[v] = answer[u]+weight
15                      queue.insert(0,v)
16              else:
17                  answer[v] = answer[u]+weight
18                  queue.insert(0,v)
```

Figure 1: Example implementation of the Bellman-Ford Algorithm.

3.2 Moore-Bellman-Ford Improvement

```
10  def mooreFord(self, graph, answer):
11      queue = [0]
12      while len(queue) != 0:
13          u = queue.pop()
14          for v,weight in graph[u]:
15              if answer[v] != None:
16                  if answer[v] > answer[u]+weight:
17                      answer[v] = answer[u]+weight
18                      if v not in queue:
19                          queue.insert(0,v)
20              else:
21                  answer[v] = answer[u]+weight
22                  if v not in queue:
23                      queue.insert(0,v)
```

Figure 2: Example implementation of the Shortest Path Faster Algorithm.

A few differences between the implementation of Bellman-Ford and Moore-Bellman-Ford include two additional if statements, one on line 18 and one on line 22. These if statements ensure that a vertex is added to the queue only if it is the most optimal distance, cutting down the runtime.

3.3 Dijkstra

We had two implementations of Dijkstra's algorithm. One that closely followed the implementation of the Bellman-Ford algorithm (Figure 3) and one that differed (Figure 4).

The implementation modeled after the Bellman-Ford implementation was used for our experimental testing. The main difference between the Ford implementation and this version of Dijkstra occurs in line 11. Rather than simply using the next element of the queue, like in Bellman-Ford, Dijkstra's finds the minimum of the queue. This cuts down on time because the minimum of the queue is more likely to yield the shortest path than whatever is on the top of the queue. Intuitively it makes sense to use a path that is known to be shorter to calculate additional shortest paths rather than using a longer path.

The code found in Figure 4 is a direct implementation of Dijkstra's algorithm as found in *Introduction to Algorithms* by Cormen, Leiseron, Rivest, and Stein (2009, p. 658), where we keep track of our shortest-path tree using lists, namely S and Q. Here, S is the current shortest-path tree and Q is the queue of vertices not yet traversed. Similar to what was mentioned above about Dijkstra's algorithm, this specific implementation will recurse until every reachable vertex from the source $s \in V$ is in S. To traverse our queue, we choose the vertex $u \in V$ with the current shortest distance from s using the subroutine `getMin` on the difference $Q - S$. We add u to our list of vertices in the shortest-path tree S and remove it from Q. We then relax every tense edge from u to all of its neighboring vertices using the subroutine `relax`.

```
8   def dijkstra(self, graph, answer):
9       queue = [0]
10      while len(queue) != 0:
11          u = min(queue,key=lambda x:answer[x])
12          for v,weight in graph[u]:
13              if answer[v] != None:
14                  if answer[v] > answer[u]+weight:
15                      answer[v] = answer[u]+weight
16                      if v not in queue:
17                          queue.insert(0,v)
18              else:
19                  answer[v] = answer[u]+weight
20                  if v not in queue:
21                      queue.insert(0,v)
```

Figure 3: Dijkstra's Implementation using Bellman-Ford Base.

```
67  def dijkstra(self):
68      S= []
69      Q= self.G[:]
70
71      if Q== []* self.n:
72          while len(Q)!= 0:
73              D= [item for item in Q if item not in S]
74              u= self.getMin(D)
75              S.append(u)
76              Q.remove(u)
77
78              for v in u:
79                  self.relax(u, v)
80      else:
81          return
```

Figure 4: Alternate Dijkstra's Implementation.

3.4 Asymptotic Runtimes

Algorithm	Excepted Runtime
Dijkstra	$O(E \log(V))$
Faster Dijkstra	$O(E + V\sqrt{\log C})$
Bellman-Ford	$O(VE)$
Moore-Bellman-Ford	$O(V + E)$

The above table presents the asymptotic runtime for each algorithm. Here, we give a brief description of the asymptotic runtime of each algorithm in addition to the previous discussions.

Bellman-Ford runs in $\Theta(VE)$ because the algorithm iterates through the edges and then performs a nested loop through the edges. Moore-Bellman-Ford runs in $O(V + E)$ as it only loops through the edges leaving each vertex, not all the edges. In the worst case graph, Moore-Bellman-Ford runs with the same time complexity as Bellman-Ford. Dijkstra's only needs to iterate $O(E)$ times because Dijkstra's is immediately able to decide if an edge belongs in the shortest path. However, the runtime of Dijkstra's is also affected by the insertion into the priority queue, which is a $\log V$ operation. Faster Dijkstra's uses a Radix heap to improve the time needed to maintain the priority queue. See Section 2.4 for a complete discussion on this improvement.

3.5 Experimental

In order to test each implementation from above, we need to first generate numerous input graphs to run on. Our graphs were generated randomly and composed of different categories, namely, directed or undirected, weighted or unweighted, with the density of edges varying for each. In order to create graphs with different edge densities we used the following procedure:

$$E = \begin{cases} w & r \leq p \\ 0 & else \end{cases} \mid \forall\, uv \in G$$

where E is a data structure of edges, w is a a random positive edge weight (1 if unweighted), r is a sample from a binomial probability distribution defined by probability p, and uv is a pair of vertices in graph G. This can be adjusted to build directed graphs as follows:

$$E = \begin{cases} w & r \leq p \\ 0 & else \end{cases} \mid \forall\, u \to v \in G$$

where $u \to v$ is a directed edge. For our experiments we use all edge probabilities $p \in P$ for the set $P = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

Then, each of every possible type of graph was generated and used in our analysis. We recorded the execution times of the three algorithms—Bellman-Ford, Moore-Ford-Bellman, and Dijkstra's. We ran each algorithm on 10 similar graphs, and then averaged those times for a more accurate estimate. This was done several times in order to generate a data-set on which we could run inferential statistics, namely, an analysis of variance test (ANOVA test).

Experimental Results

With our results, we were able to compare execution times between the three algorithms mentioned above. Additionally, we did more fine-grained comparisons between algorithms based on specific graph types. For all presented comparisons, we used graphs with 1,000 nodes. For all graph types, we created 10 separate graphs. Each algorithm ran all 10 graphs and then calculated the average execution time. Below, we present the most relevant findings from these trials.

First, we look at performance on graphs that are weighted and undirected. These results can be seen in Figure 5. It can be seen that the other two algorithms outperform Ford, which is expected behaviour based on the theoretical runtimes. It can also be seen that this differences widens as the graphs gets more dense (the x-axis in the plot). When comparing between Moore-Bellman-Ford and Dijkstra's, we see that Dijkstra's out performs Moore-Bellman-Ford in sparse graphs. However, at around $p = 0.3$, Moore-Bellman-Ford becomes the better performing algorithm. Recall the asymptotic run times from Table 1—Dijkstra's is linear with respect to E ($O(E \log V)$) and Moore-Bellman-Ford (in the average case) is not ($O(E + V)$). Thus, it

is expected that Moore-Bellman-Ford improves relative to Dijkstra's as edge density increases. This exact behaviour was also observed for directed graphs (with cycles).

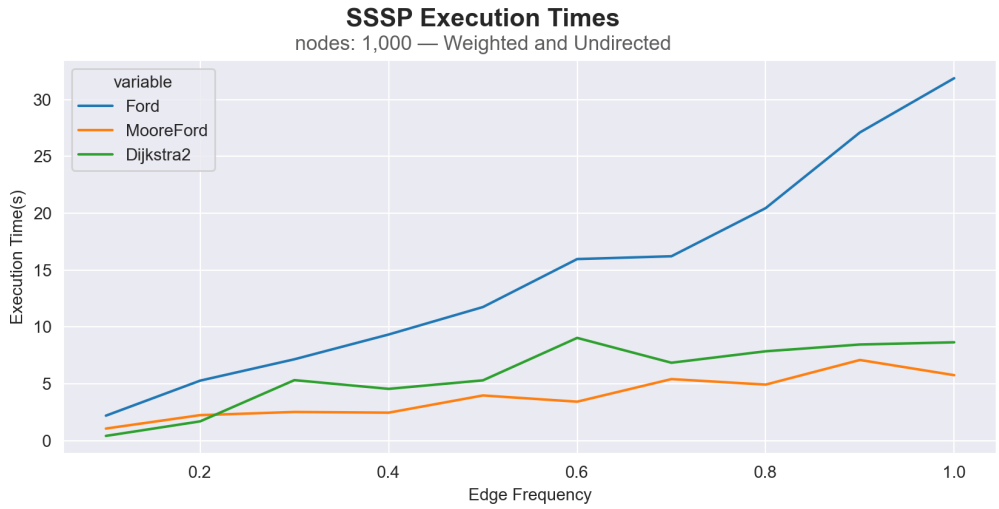

Figure 5: Execution times on weighted and undirected graphs

Next, we compare the performance of unweighted and undirected graphs. These results are shown in Figure 6. Interestingly, the execution times are similar across algorithms. This can not be explained from asymptotic analysis. We conjecture that this difference between unweighted and weighted graphs is because of the following observation: A relaxed edge in a weighted graph is more likely to be re-relaxed later in execution because a relatively short distance (sum of weights) may occur from a relatively long path (number of edges in the path). Dijkstra's can, on the other hand, in an unweighted graph. If an edge connects two vertices, that edge *is* the shortest path between those two vertices. That edge will never be relaxed again for a path. Our implementation of Bellman-Ford (which uses a queue), is able to take advantage of this, along with other algorithms.

Figure 6: Execution times on unweighted and undirected graphs

Lastly, we compare the performance of weighted, directed acyclic graphs. These results are shown in Figure 7. In this graph, we observe the same primary behaviour found in the weighted and undirected (potted in Figure 5)—Bellman-Ford is the worst performing algorithm, Dijkstra's is the best performing on sparse graphs, and Moore-Bellman-Ford is the best performing on dense graphs. However, by comparing Figure 5 and Figure 7, we can see that Bellman-Ford

performance is better on directed acyclic graphs than on cyclic graphs.

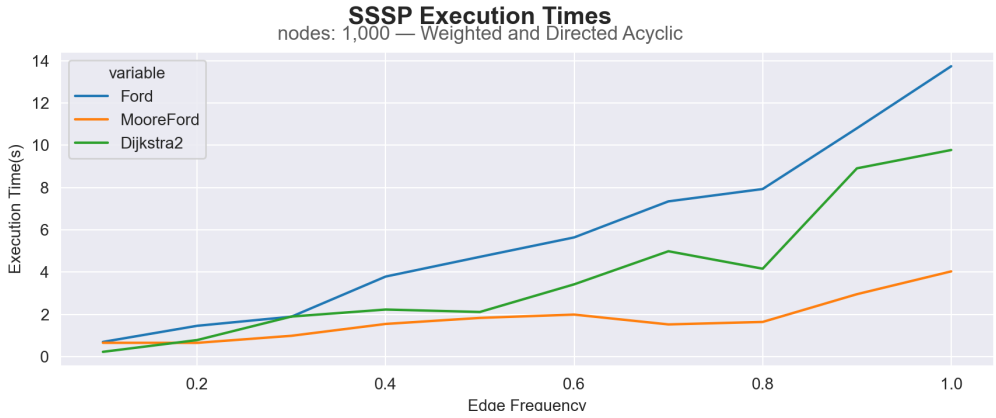

Figure 7: Execution times on weighted and directed acyclic graphs

Statistical Conclusions

An ANOVA test is an inferential tool used to test whether or not observed data, from several groups, comes from the same population or not. For example, taking the times generated from Bellman-Ford and Dijkstra's algorithms, we can compare the means of the two samples and see whether or not they are the same. If $\mu_1, \mu_2 \in \mathbb{R}$ are the mean runtimes of Bellman-Ford and Dijkstra's, respectively, then our hypotheses are as follows:

$$H_0 : \mu_1 = \mu_2$$
$$H_1 : \mu_1 \neq \mu_2.$$

That is, we assume there is no difference in the means of the runtimes initially and then we update our belief to H_1 if there is enough evidence to do so. The strength of our evidence is determined by a specific statistic which is the ratio of the variances of the data, i.e., the F-statistic.

There exist three assumptions, however, that must be met in order for an ANOVA test to be considered reliable. Namely, the data is assumed to be approximately normal, the samples are independent of one another, and the variance between sample groups is approximately equal. Using `R` software, a three way ANOVA test, for example, is implemented using a model of the form

$$Y_{ij} = \mu_i + \epsilon_{ij},$$

where μ_i is the mean of each regressor, and ϵ_{ij} is the j-th residual of the i-th regressor. The first and last assumptions from above can now be interpreted using the residuals of the ANOVA model. In terms of the residuals, our first assumption is satisfied if the residuals are normally distributed, and our third assumption is satisfied if the residuals have a linear variance.

Using our sample data we completed an ANOVA analysis in order to verify our assumptions of the runtimes of each algorithm. According to the theoretical asymptotic runtimes of each algorithm, we should expect our analysis to conclude that the means of each algorithm are not the same. The results of our analysis can be seen in Figure 8.

Term	DoF	F-value	p-value
alg	2	209.8708	0
n	1	2977.3489	0
m	1	1500.0251	0
alg:n	2	495.3933	0
alg:m	2	381.5341	0
Residuals	141	NA	NA

Figure 8: Results of the factorial ANOVA analysis comparing means of Bellman-Ford, Moore-Bellman-Ford, and Dijkstra's algorithms.

From Figure 8 we see the results of our analysis in the final column. The smaller the p-value the more significant the test statistics; therefore, we reject the null hypothesis H_0 in favor of its alternative H_1 at a certain level of significance $\alpha = .05$. We conclude that depending on which algorithm is used, the average runtime is indeed different, as we can see from Figures 5, 6, and 7 above.

As was mentioned previously, in order for an ANOVA analysis to be considered reliable the data must satisfy the above criteria. Unfortunately, in our analysis, at least one of the assumptions was not satisfied, namely, the residuals are not normally distributed, shown in Figure 9.

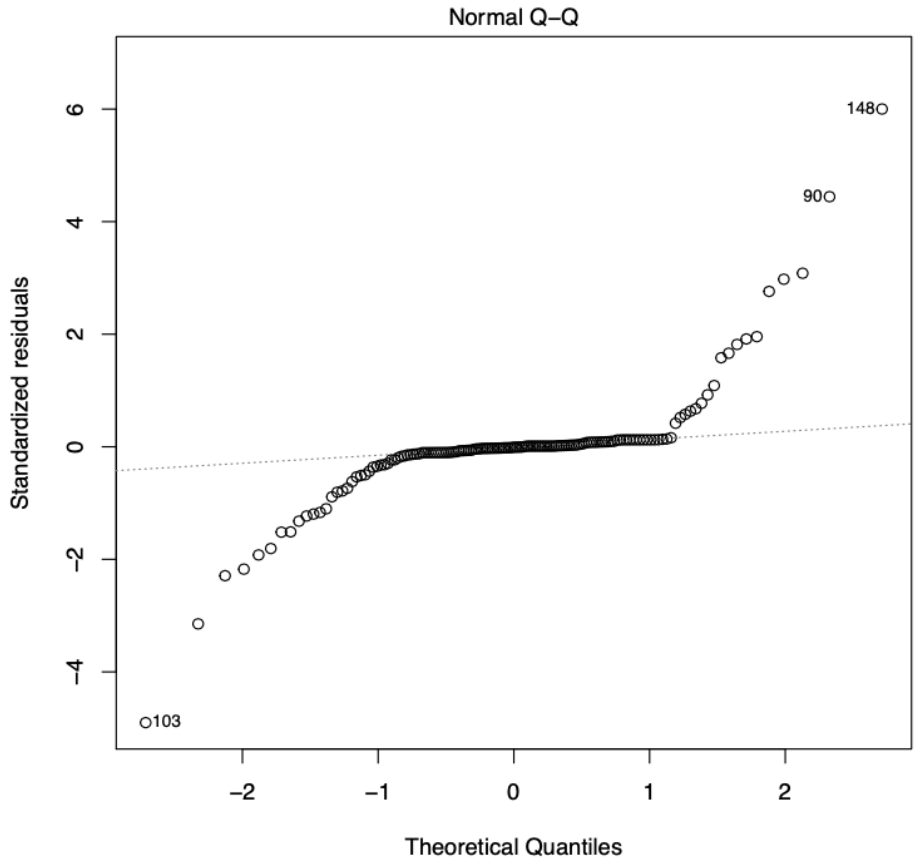

Figure 9: Standardized residuals plotted against what would be their corresponding quantiles of a normal distribution.

Given that our test may currently be considered unreliable, we believe that this is an artifact of how the data was generated and is not indicative of a false assumption, namely, that each algorithm produces statistically distinguishable runtimes.

4 Conclusion

The shortest path problem has a variety of algorithms, each which find suitable solutions via different means. While the original Bellman-Ford algorithm is robust and relatively easy to implement, its improvements are able to solve the problem much faster. These improvements however, are harder to implement, and some can only be applied to specific problem sets. However should an improved algorithm be able to solve a given set, it is our belief that it is worth the extra time and effort to implement, given the huge improvements on time complexity. A future topic of research or discussion would be to implement the Faster Dijkstra algorithm and run this in parallel with the other algorithms discussed. Running these algorithms on graphs that would provide sufficient data for the ANOVA statistical analysis would allow us to see how they function in real time. Additionally implementing the Faster Dijkstra's algorithm would allow us to see how the Fibonacci heaps preform in real time, since their expected runtime and actual runtime can differ significantly. Until then, there is still much flexibility in choice of algorithm to solve the single-source shortest path.