Single-Source Shortest Path Algorithms: Survey, Implementation, and Analysis

Jonny Pei jonnypei@
Vivek Verma vivekverma@
Young-Jin Park yjp20@

Abstract

The single-source shortest paths (SSSP) problem is one of the most fundamental and long-standing graph problems in history, with deep rooted applications in navigation, networking and robotics. Despite classic algorithms such as Bellman-Ford and Dijkstra's Algorithm being around for decades, new research has improved the bounds to $O(m \log^8(n) \log W)$ [2], and very recently $O(m \log^2(n) \log(nW) \log \log n)$ [3].

We survey existing SSSP algorithms, and analyze their effectiveness in real-life and randomized scenarios. We provide implementations for Bernstein-Nanongkai-Wulff-Nilsen (BNW) Algorithm, and compare it's constant factors to traditional and recent algorithms.

Contents

1 Introduction

In this paper, we survey the single-source shortest paths (SSSP) problem with arbitrary edge weights. Given a weighted graph $G = (V, E)$, weight function $w(e)$ and a source vertex s, the SSSP problem aims to compute the shortest distance from s to all vertices $v \in V$. We then contribute an implementation of the BNW Algorithm, which we test and compare to traditional shortest paths algorithms. We note that the general shortest paths problem can be broken down into three cases:

1. Single-source shortest paths, which aims to find the shortest path from a vertex $s \in V$ to all other vertices.
2. Single-destination shortest paths, which aims to find the shortest path from a vertex $s \in V$ to a vertex $t \in V$.
3. All-pairs shortest paths, which aims to find the shortest path between every pair of vertices $(s, t) \in V \times V$.

Each problem can be further broken down into special cases - positive edge weights, directed acyclic graphs (DAGs), negative cycles, etc. While we only focus on the SSSP variant, we observe that many of the algorithms discussed can be generalized to other variants of the problem, and vice versa. As such, we hope that a discussion of SSSP algorithms can shed light onto more efficient solutions to other shortest paths variants. Designing efficient shortest paths algorithms is one of the most fundamental problems in algorithmic research, but state-of-the-art aims for two primary goals: deterministic, near-linear runtime for any graph and simplistic design.

We first attempt to survey a subset of the field of shortest paths algorithms, providing brief explanations of the techniques used for each one. Then, we evaluate effectiveness of modern algorithm design, and compare it to traditional algorithms. Finally, we suggest directions for further shortest paths research, and discuss implementation strategies for sophisticated algorithms, and choices we made when implementing the BNW algorithm.

Algorithm Name	Negative Weights	Time Complexity
Dijikstra	No	$O(m + n \log n)$
Dial	No	$O(m + nW)$
Orlin et al.	No	$O(\min\{m + nK, m \log \frac{nK}{m}\})$
Bellman-Ford	Yes	$O(nm)$
Goldberg	Yes	$O(m\sqrt{n} \log W)$
Bernstein-Nanongkai-Wulff-Nilsen	Yes	$O(m \log^8(n) \log W)$
Bringmann-Cassis-Fischer	Yes	$O(m \log^2(n) \log(nW) \log \log n)$

2 SSSP Algorithms for non-negative Weights

In this section, we survey several algorithms that solve SSSP problem on graphs with strictly non-negative edge weights.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm [6] is a well-known and widely used algorithm for solving the single-source shortest paths problem in a weighted, directed graph. It was developed by Dutch computer scientist Edsger W. Dijkstra in 1956 and has since become a fundamental algorithm in the field of graph theory and network analysis. Dijkstra's algorithm is particularly efficient when applied to graphs with non-negative edge weights.

2.1.1 Algorithm Description

Dijkstra's algorithm [6] operates by iteratively expanding the frontier of the search from the source vertex, gradually finding the shortest paths to all other vertices. The algorithm maintains a priority queue (often implemented using a min-heap) that stores vertices along with their

tentative distances from the source. Initially, the distance to the source vertex is set to zero, and the distances to all other vertices are set to infinity.

In each iteration, the algorithm selects the vertex with the smallest tentative distance from the priority queue and examines all its outgoing edges. It relaxes the edges by updating the distance values if a shorter path is found. This process continues until all vertices have been processed or until the priority queue becomes empty. We provide the pseudocode for the algorithm below:

Algorithm 1 Dijkstra's Algorithm for Shortest Paths

```
dist(v) = ∞ ∀v ∈ V
q is a priority queue of vertices with priority dist(v)
Add s onto q
dist(s) = 0
while q is not empty do
  Pop u from q
  for all (u, v) ∈ E do
    if dist(u) + w(u, v) < dist(v) then
      dist(v) = dist(u) + w(u, v)
      Queue v onto q, or update it's priority
    end if
  end for
end while
```

2.1.2 Time Complexity

The time complexity of Dijkstra's algorithm depends on the implementation of the priority queue. When implemented with a binary min-heap, the time complexity is $O((n + m) \log n)$, where n represents the number of vertices and m represents the number of edges in the graph. The algorithm requires n iterations, and in each iteration, it performs a logarithmic time operation to extract the vertex with the minimum distance from the priority queue and updates the distances of its adjacent vertices.

If a Fibonacci heap is used as the priority queue, the time complexity can be improved to $O(n \log n + m)$. However, Fibonacci heaps have a more complex implementation and may have higher constant factors in practice, making the binary min-heap implementation more commonly used.

2.1.3 Advantages and Limitations

Dijkstra's algorithm has several advantages that contribute to its popularity and wide usage. It guarantees the optimality of the shortest paths it computes, providing the shortest path tree from the source vertex to all other vertices. Additionally, it can handle graphs with non-negative edge weights, which makes it suitable for many real-world applications.

One limitation of Dijkstra's algorithm is its inability to handle negative-weight edges. If a graph contains negative-weight edges, the algorithm may produce incorrect results or get stuck in an infinite loop due to its greedy nature. In such cases, alternative algorithms like the Bellman-Ford (3) Algorithm or the A* algorithm may be more appropriate.

2.2 Dial's Algorithm

Dial's [5] algorithm is a variation of Dijkstra's algorithm that is specifically designed to handle graphs with non-negative integer edge weights. It was proposed by Yefim Dinitz and Shimon Even in 1970 as an improvement over Dijkstra's algorithm for certain types of graphs. Dial's algorithm is particularly efficient when applied to graphs with a limited range of edge weights.

2.2.1 Algorithm Description

Dial's algorithm operates by maintaining a set of buckets, each representing a range of distances from the source vertex. The buckets are organized in such a way that each bucket contains vertices whose distances fall within a particular range. Initially, all buckets are empty except for the bucket representing a distance of zero, which contains the source vertex.

In each iteration, the algorithm selects the non-empty bucket with the smallest index and examines its vertices. It then relaxes the outgoing edges of those vertices and updates the distances accordingly. If the distance of a relaxed vertex falls into a new range, it is moved to the corresponding bucket. This process continues until all buckets are empty.

To ensure efficient retrieval of the non-empty buckets, Dial's algorithm uses an array of pointers that point to the first non-empty bucket. These pointers are dynamically updated during the execution of the algorithm, allowing for efficient bucket selection. We provide the pseudocode for the algorithm below:

Algorithm 2 Dial's Algorithm

```
dist(v) = ∞ ∀v ∈ V
dist(s) = 0
B is a list of buckets
Add s to bucket 0
while B contains a non-empty bucket do
  Find the first non-empty bucket b, starting at bucket 0
  u = Top vertex from b
  for all (u, v) ∈ E do
    if dist(u) + w(u, v) < dist(v) then
      if dist(v) ≠ ∞ then
        Remove v from bucket dist(v)
      end if
      dist(v) = dist(u) + w(u, v)
      Add v to bucket dist(v)
    end if
  end for
end while
```

2.2.2 Time Complexity

The time complexity of Dial's algorithm depends on the number of buckets required to cover the range of distances in the graph. Let the maximum weight of any edge in the graph be W. Then, the maximum distance between any two nodes in the graph is $(n-1) \cdot W$. So, the number of buckets required is $O(n \cdot W)$, and each vertex is examined and relaxed once for each of its outgoing edges. Hence, the overall time complexity of Dial's algorithm is $O(m + n \cdot W)$.

2.2.3 Advantages and Limitations

Dial's algorithm offers several advantages over Dijkstra's algorithm in certain scenarios. It is particularly efficient when applied to graphs with a limited range of non-negative integer edge weights. By organizing vertices into buckets based on their distances, the algorithm avoids unnecessary operations on vertices that are unlikely to produce shorter paths.

Additionally, Dial's algorithm can be more memory-efficient than Dijkstra's algorithm. Instead of storing and updating the distances for all vertices, it only requires storing the vertices in the buckets. This can be advantageous when dealing with large graphs or limited memory resources.

However, Dial's algorithm has limitations as well. It can only handle non-negative integer edge weights, which restricts its applicability to graphs with other types of weights. If a graph

contains non-integer or negative weights, alternative algorithms such as Dijkstra's algorithm or the Bellman-Ford algorithm should be considered.

Furthermore, the efficiency of Dial's algorithm heavily depends on the range of distances in the graph. If the range is significantly larger than the number of vertices, the algorithm may not provide significant improvements over Dijkstra's algorithm. Careful analysis of the graph characteristics and edge weight distributions is essential to determine whether Dial's algorithm is suitable for a specific problem instance.

2.3 Orlin et al.

In 2009, Orlin et al. [9], proposed a solution to SSSP that shines when used on graphs with a small number of unique edge weights. It was motivated by the "gossip" problem in social networks - consider a collection of graphs with edge weight 1, connected by edges with weight l. Here, each graph represents a tightly-knit social cluster, and the higher weight edges between them represent the difficulty of information to travel between clusters. The goal is to then determine the minimum distance a "gossip" emerged at a particular source vertex has to travel to reach every other vertex, or in other words, perform SSSP on this graph.

2.3.1 Algorithm Description

The algorithm works as a modification to Dijkstra's Algorithm. Let the number of unique edge weights be K. Under the assumption that K is relatively small, we maintain a set of data structures in order to efficiently find the vertex with the minimum distance at each step. First, we initialize S to be the set of containing s, the source vertex, and $T = V \setminus S$ to be a set containing all other vertices. We use S to store all the vertices with finalized distances, and T to store all other distances. In addition, we maintain $E_t(S) = \{(u, v) \in E \mid u \in S, w(u, v) = t\}$, which stores all the edges of a given weight t in S. Each $(u, v) \in E_t(S)$ is sorted by when u was added into S. Then, we maintain a map CurrentEdge(t), where t is the weight of an edge. CurrentEdge(t) stores the first edge (u, v) added into $E_t(S)$ such that $v \in T$. Finally, if CurrentEdge$(t) = (u, v)$ we maintain $f(t) = d(u) + w(u, v)$. Thus, in order to determine the vertex with the minimum distance, it now suffices to just perform $\text{argmin}_e\{f(t) \mid t \in [0, K]\}$.

If the value of K is larger, we first examine the value $q = \frac{nK}{m}$. Then, we construct $\frac{K}{q}$ binary heaps $H_1, ..., H_{K/q}$, where the i^{th} binary heap stores vertices with distances from $(i-1) \cdot q + 1$ to $i \cdot q$. In order to find the vertex with the minimum distance, we first linearly scan the heaps and pop as necessary.

2.3.2 Time Complexity

First, we analyze the algorithm when K is small. In order to find the vertex with minimum weight, we linearly scan f, which has a runtime of $O(K)$. Since this is executed $O(n)$ times, the total runtime of finding the minimum vertex is $O(nK)$. To calculate the running time of the updates, notice that on an iteration where CurrentEdge(t) does not change, it suffices to update $E_t(S)$, so the total running time for these iterations is $O(nK)$. For the cases when CurrentEdge(t) does change, notice that when an edge is changed in CurrentEdge(t), it is never revisited in the linear scan. Thus, the runtime for these updates is $O(m + nK)$. So, the total runtime of the algorithm is $O(m + nK)$ and if $nK \leq m$, then the runtime is $O(m)$. For the algorithm when $nK > m$, notice that each binary heap update is $O(\log q)$, so the total runtime of the algorithm is $O(m \log \frac{nK}{m})$.

2.3.3 Advantages and Limitations

This algorithm provides a significant advantage when the graph is sparse. A sparse graph implies a small number of unique edge weights, which gives a lower runtime. As per the figures [9] in the original paper, the algorithm gives a significant advantage over BFS/Dijkstra's on long meshes, square meshes and sparse random graphs.

However, if the number of distinct edge weights grows, the runtime bound becomes quite slow. In particular, if every edge weight is unique, the runtime grows to $O(m \log n)$, which is equivalent to Dijkstra's, yet has the overhead of all the updates.

3 SSSP Algorithms for arbitrary weights

In this section, we survey several algorithms that solve SSSP problem on graphs with arbitrary edge weights.

3.1 Bellman-Ford Algorithm

The Bellman-Ford algorithm [1, 7] is a classical algorithm used to solve the single-source shortest paths problem in a weighted, directed graph. It was proposed by Richard Bellman and Lester Ford Jr. in 1958 and has since become an essential algorithm in the field of graph theory and network analysis. The Bellman-Ford algorithm is particularly useful when the graph contains negative-weight edges, as it can handle such scenarios unlike some other popular algorithms like Dijkstra's algorithm.

3.1.1 Algorithm Description

The Bellman-Ford algorithm operates by iteratively relaxing the edges of the graph, gradually improving the estimates of the shortest paths from the source vertex to all other vertices. The algorithm maintains an array of distances, initially set to infinity for all vertices except the source, which is set to zero. In each iteration, it examines all edges in the graph and relaxes them by updating the distance values if a shorter path is found.

The relaxation process continues for $n - 1$ iterations, where n is the number of vertices in the graph. This ensures that the algorithm has enough iterations to propagate the shortest path information to all vertices. If after $n - 1$ iterations, there is still a relaxation that results in a shorter path, then the graph contains a negative-weight cycle, and the algorithm detects this cycle in the n^{th} iteration. In such cases, the algorithm cannot produce a valid shortest path tree. We provide the pseudocode for the algorithm below:

Algorithm 3 The Bellman-Ford Algorithm

```
dist(v) = ∞ ∀v ∈ V
dist(s) = 0
for n − 1 iterations do
  for (u, v) ∈ E do
    if dist(u) + w(u, v) < dist(v) then
      dist(v) = dist(u) + w(u, v)
    end if
  end for
end for
```

3.1.2 Time Complexity

The time complexity of the Bellman-Ford algorithm is $O(n \cdot m)$, where n represents the number of vertices and m represents the number of edges in the graph. This complexity arises from the need to relax all edges $n - 1$ times. In the worst case, the algorithm examines every edge in each iteration, resulting in a total of $n - 1$ iterations. However, if the graph contains a negative-weight cycle, the algorithm can terminate early during the n^{th} iteration when the distances are updated, indicating the presence of a negative cycle.

3.1.3 Advantages and Limitations

One significant advantage of the Bellman-Ford algorithm is its ability to handle graphs with negative-weight edges. This property makes it suitable for scenarios where negative weights

are involved, such as modeling certain types of transportation networks or financial systems. Additionally, the algorithm can detect negative-weight cycles, which is valuable in identifying inconsistencies or potential problems within a graph.

However, the Bellman-Ford algorithm has some limitations. One major drawback is its relatively high time complexity compared to other algorithms like Dijkstra's algorithm. The need to relax all edges $n-1$ times can result in slower performance, especially for large graphs. Additionally, the algorithm's time complexity makes it less suitable for dense graphs with many edges, where more efficient algorithms, such as the Floyd-Warshall algorithm, may be preferred. Despite its limitations, the Bellman-Ford algorithm remains an essential algorithm in the field of graph theory due to its ability to handle negative-weight edges and detect negative-weight cycles. Its simplicity and versatility make it a valuable tool for various applications, and it has paved the way for the development of more advanced algorithms and techniques in the realm of single-source shortest paths.

3.2 LP formulation

Linear Programming (LP) is a mathematical optimization technique whose goal is to maximize a linear objective function with linear constraints. We attempt to formulate the single-source shortest paths problem as an LP in order to employ traditional LP solvers.

3.2.1 Algorithm Description

For an arbitrary graph, consider the following LP formulation:

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in V} d_v \\ \text{such that} \quad & d_v \leq d_u + w(u,v), \forall e = (u,v) \in E \\ & d_s = 0 \end{aligned}$$

To see that this works, notice that for a given vertex, the optimal path must emerge from one of the edges directed towards it. By induction, if we assume that the distances to the other vertices are accurate, the shortest path outputted by the LP is optimal.

3.2.2 Time Complexity

The time complexity of solving an LP is $O^*((n^\omega + n^{2.5-\alpha/2} + n^{2+1/6}) \log(n/\delta))$ [4] where where ω is the exponent of matrix multiplication, α is the dual exponent of matrix multiplication, and δ is the relative accuracy.

3.2.3 Advantages and Limitations

The most significant advantage of the LP formulation is its abstractness - if a new LP algorithm drastically improves the bound for solving an LP, the bound for solving SSSP will go down as well. However, with current LP solutions, the overhead and complexity of LP solvers does not outweigh more traditional algorithms. We note that this is largely due to the solver being so generalized, whereas other algorithms described were developed primarily for SSSP.

3.3 Goldberg's Scaling Algorithm [8]

In 1995, Andrew V. Goldberg proposed solving SSSP with arbitrary edge weights by using a "price function", which attempts to find a suitable transformation of the graph that preserves the shortest paths such that all edge weights are positive. After this transformation, it suffices to just run Dijsktra's to output the shortest path.

We define a price function as a map $\Phi : V \to \mathbb{R}$ that scales the weight of each each edge $(u,v) \in E$ via $w_\phi(u,v) = w(u,v) + \phi(u) - \phi(v)$. Through this transformation, we note that shortest paths are preserved, as for an arbitrary path P, the weight $w(P)$ is only scaled by the price of the beginning and the end. As such, the shortest path between two vertices $u, v \in V$ in G is the same as G_ϕ, the transformed graph.

3.3.1 Algorithm Description

In order to come up with a suitable price function, in which the transformed graph has non-negative edge weights, we introduce three subroutines: Decycle, Eliminate-Chain, Cut-Relabel.

First, we duplicate the graph with zeroed-out edge weights. Then, for each bit in W, the maximum edge weight, we multiply all the edge weights and prices by two, the subtract one where necessary based on the weight of the edge. As such, we've reduced finding a price function to resolving the -1 edge weights at each step.

The first step of each scaling step is Decycle, which "contracts" all the vertices by a 0 weight edge into one. During this process, it can also tell when a negative-weight cycle occurs. Then, we define Cut-Relabel, which reduces the price functions of all nodes in the closed set S by one. A key intuition is that if the set S consists of all reachable nodes in G_p for some v, then the edges going out from v is no longer negative. In addition, we define Eliminate-Chain, which enhances Cut-Relabel by noticing that for some chain of admissible edges creating a path that visits nodes $w_1, w_2, w_3, \cdots, w_i$, the set S_i of all reachable nodes in G_p for w_i are subsets of each other.

For each iteration of the scaling algorithm, Goldberg calls Decycle, then iteratively calls Cut-Relabel or Eliminate-Chain until no negative-weight edges remain. Each iteration reduces the number of *improvable* nodes k (nodes with negative weight edges coming out of them), by $\sqrt{k}$.

3.3.2 Time Complexity

There are a total of $\log W$ iterations of scaling, which require $O(\sqrt{n})$ iterations of calling either Cut-Relabel or Eliminate-Chain, which each take $O(m)$ time. This results in a time complexity of $O(m\sqrt{n}\log W)$. An alternative, but similar approach can yield $O(n\sqrt{m}\log W)$.

3.3.3 Advantages and Limitations

Goldberg's scaling algorithm is a combinatorial algorithm that is relatively simple to implement that can handle negative weight edges. Better algorithms exist, but given the large constant factors that affect other combinatorial algorithms, Goldberg's scaling algorithm may be useful alternative to both BNW and Bellman-Ford in the right circumstances.

3.4 Bernstein-Nanongkai-Wulff-Nilsen Algorithm

In October of 2022, Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen published a paper describing an algorithm (BNW) [2] that solves the general SSSP problem in near-linear time. Their main technical contribution is showing how to use low-diameter decompositions of graphs to develop an efficient recursive scaling algorithm that has time complexity $O(m\log^8(n)\log W)$.

3.4.1 Algorithm Description

BNW borrows Goldberg's scaling framework in some ways, where we look at the varying levels of precision as we progress through the algorithm iteratively. The scaling algorithm itself differs in the sense that we do not "resolve" edges with weights of -1 at that level of precision, but instead roughly halve the magnitude of the negative weight edges on each iteration.

Using this method of scaling, we cannot get negative weights to zero, but we can get them *almost* to zero. This is accomplished by multiplying all the weights by a large factor, so that a negative weight of -1 is relatively insignificant. In particular, if we multiply all weights by $2n$, we can show that the shortest path remains the same regardless of whether or not the negative weight edges are -1 or 0.

Each iteration of the scaling algorithm is mainly composed of a call to ScaleDown, which makes all weights in the graph $w_i \geq -D$ to $w_i \geq -D/2$.

At a high level, each ScaleDown call consists of separating the graph into SCCs with a weak-diameter of D using LowDiameterDecomposition, then calling ScaleDown recursively on

each smaller SCC. The edges between the SCCs are then taken care of with FixDagEdges, while the edges that were removed by LowDiameterDecomposition are fixed by ElimNeg.

3.4.2 Time Complexity

For the sake of simplicity, we first focus on the case that the input graph does not contain any negative cycles. The expected runtime of SPmain is dominated by $\log(B) = O(\log(n))$ calls to ScaleDown, each with expected runtime $O(m \log^3(n) \log(\Delta)) = O(m \log^4(n))$. Thus, SPmain has expected runtime $O(m \log^5(n) \log(W))$ after accounting for the edge weight scaling (Goldberg's Theorem [8]).

Now, in the case that there are negative cycles, as described in 3.4.1, SPmain may never terminate. The SPLasVegas wrapper routine resolves this issue at the cost of an extra $O(\log^3(n))$ runtime factor. This yields the overall expected runtime of $O(m \log^8(n) \log W)$.

3.4.3 Advantages and Limitations

BNW is the first combinatorical algorithm to solve the general negative-weight SSSP problem with in near-linear expected time, with very low variance in time complexity. It is also the first algorithm that takes advantage of low-diameter decomposition in a classical static setting, and proves its effectiveness in the paper.

Multiple components of BNW are not fully optimized (in particular, negative-weight cycle detection subroutine is excessive), which lead to several extra log factors in its runtime. The algorithm described in Section 3.5 proposes an extension of BNW that optimizes for these log factors. Furthermore, the algorithm is difficult to implement, as the main method relies on several subroutines that each have highly specific inputs and probabilistic output guarantees. As a result, BNW has large constant factors, which may diminish its performance in practical settings; we obtain empirical results in Section 5 that support this implication.

3.5 Bringmann-Cassis-Fischer Algorithm [3]

Just 6 months after its release, the BNW algorithm is revisited by Karl Bringmann, Alejandro Cassis, and Nick Fischer in BCF, an incremental algorithm that improves BNW to shave off nearly six runtime log-factors. To do this, they propose a new construction for computing low-diameter decompositions and design a much better algorithm for detecting negative-cycles. Along with a few other smaller optimizations, these contributions enable them to achieve an improved runtime of $O(m \log^2(n) \log(nW) \log \log n)$.

4 BNW Implementation

In our implementation of BNW, we only consider the case where there does not exist a negative cycle in the input graph. Hence, we only implement SPmain (and the routines it relies upon) and not SPLasVegas. We explain our implementation details in Section 4.1 and choice of constants in Section 4.2

We provide our code in the following github repository: `https://github.com/vivek3141/bnw-implementation`. Please feel free to follow the instructions in the `README.md` file to test out our implementation.

4.1 Implementation Details and Design Choices

We implemented BNW using python as it was the most accessible programming language for our team. However, we acknowledge that it is not the most effective language for optimal performance and benchmarking.

For our graph data structure, we use a standard adjacency list representation, and define a "subgraph" subclass (with a vertex subset argument) for storing e.g. SCC DAG decompositions and balls of a graph. We update the edge weight functions of graphs (e.g. performing $w \mapsto w^B$ or $w \mapsto w_\phi$) by constructing a new copy of the graph with the updated edge weights.

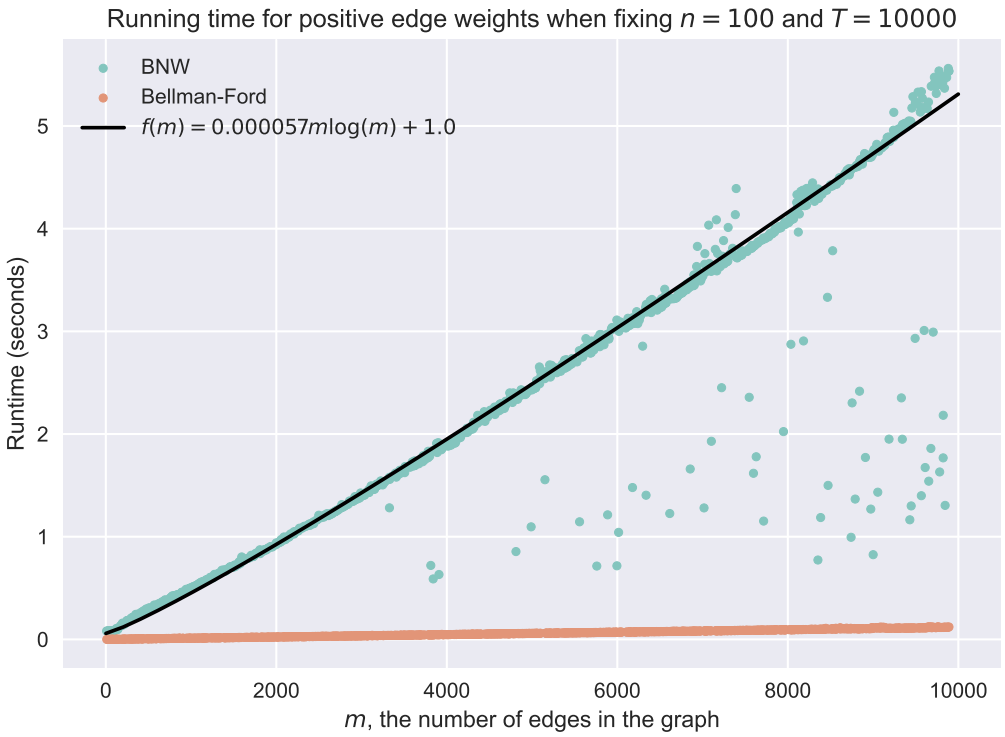

Figure 1: Runtime of BNW and Bellman-Ford for positive edge weight graphs when fixing $n = 100$ and $T = 10000$, comparing m, the number of edges to runtime. The curve of best fit is $f(m) = 0.000057m \log m + 1$ after removing outliers.

Our implementations of each subroutine closely follow the pseudocode provided in the appendices of [2]. We note that we use the set data structure religiously throughout our code to make merging or removing graph components efficient.

4.2 Constants

For our implementation of Low Diameter Decomposition, when setting k we choose to set $c = 1$ for simplicity.

5 Experiments

5.1 Data and Methods

In order to test the runtime of the BNW implementation, we employ a subroutine GENERATE_RANDOM_GRAPH, which takes in two parameters: n, the number of vertices, and p, a probability threshold. The subroutine generates a random graph with positive edge weights by accepting each edge in the graph with probability p.

In random graphs with negative weights, we first create a DAG, which allows us to find the distance to each node via topological sort and dynamic programming. We add back edges by (and therefore cycles) without generating negative cycles by limiting the negative weight to be the difference in the two distances.

5.2 Empirical Results

We first attempt to approximate the constant factor of the big-O bound provided in the BNW paper [2]. We begin by fixing $n = 1e2$ and letting the max number of iterations to run be $I = 1e5$. For each iteration $i \in [1, I]$, we generate a random graph with n vertices, $p = \frac{i-1}{L-i}$ and positive

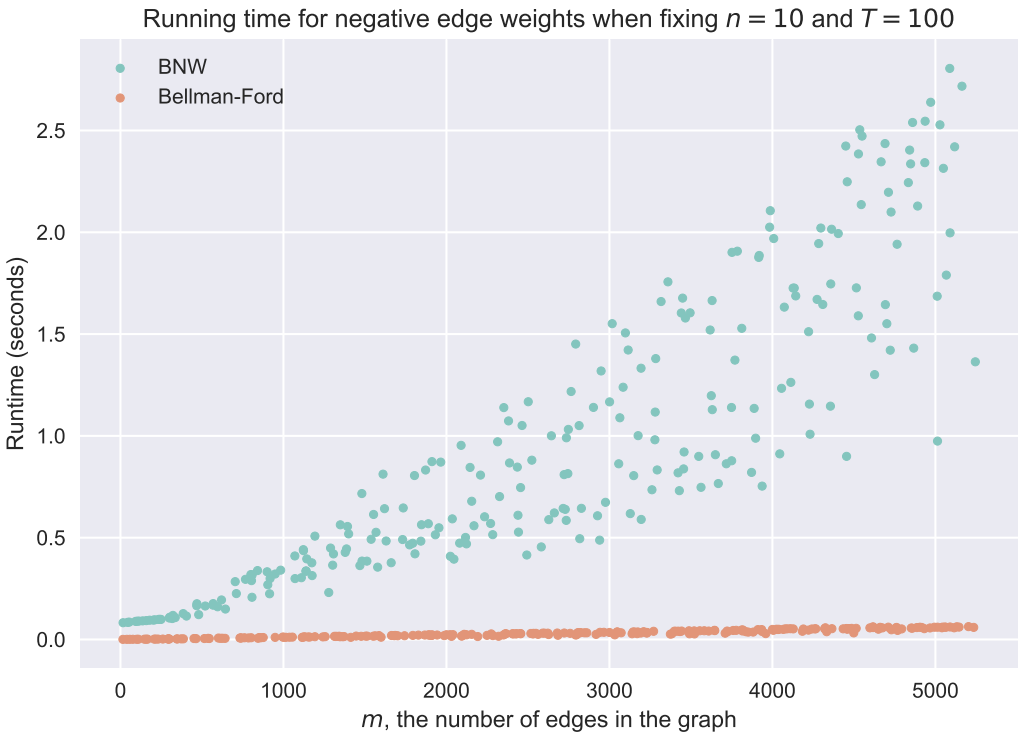

Figure 2: Runtime of BNW and Bellman-Ford for positive edge weight graphs when fixing $n = 10$ and $T = 1000$, comparing m, the number of edges to runtime. We note that this data is noisy due to the time required to run BNW on larger graphs.

edge weights. We then plot m, the number of edges in the graph with respect to the time BNW took to run and compare it with Bellman-Ford in Figure 1. Then, we re-create the same plot but with negative edge weights, avoiding negative cycles using the method described in Section 5.1.

6 Discussion

As mentioned in Section 3.4.1, we expect the constant factor to cause increased overhead when running BNW. As per Figure 1, this overhead is quite significant. We also note the results in Figure 2, which demonstrate that BNW is significantly slower when considering negative weights as well.

7 Conclusion and Future Directions

Going forward, we believe that an important line of work is reducing memory usage for practical implementation. We noticed this to be particularly difficult to deal with when implementing BNW. In particular, the repeated copying of graphs caused a significant overhead shown in Figures 1 and 2. However, it is evident that the field of algorithms has significantly grown by the number of solutions and drastic improvements to the bounds of SSSP. We hope that such bounds continue to improve in the future, and these results lead to impactful applications.

References

[1] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[2] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–611, 2022.

[3] Karl Bringmann, Alejandro Cassis, and Nick Fischer. Negative-weight single-source shortest paths in near-linear time: Now faster! *ArXiv*, abs/2304.05279, 2023.

[4] Michael B. Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time, 2020.

[5] Robert B. Dial. Algorithm 360: Shortest-path forest with topological ordering [h]. *Commun. ACM*, 12(11):632–633, nov 1969.

[6] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[7] Lester Randolph Ford. Network flow theory. 1956.

[8] Andrew V Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[9] James B. Orlin, Kamesh Madduri, K. Subramani, and M. Williamson. A faster algorithm for the single source shortest path problem with few distinct positive lengths. *Journal of Discrete Algorithms*, 8(2):189–198, 2010. Selected papers from the 3rd Algorithms and Complexity in Durham Workshop ACiD 2007.