COSC-520: Breaking the Sorting Barrier for Directed Single-Source Shortest Path

GHAITH CHRIT*, BAKDAUREN NARBAYEV*, and HAMZA MUHAMMAD ANWAR*, The University of British Columbia, Canada

The Single-Source Shortest Path (SSSP) problem is a fundamental problem in graph theory, with classical solutions such as Dijkstra's algorithm relying on priority queues and sorting of vertices. While efficient in practice for sparse graphs with non-negative weights, such methods face scalability limitations in large graphs. Bellman–Ford provides a more general approach, capable of handling negative edge weights, but its higher computational cost makes it less practical for large instances. This report provides an exposition of the Bounded Multi-Source Shortest Path (BMSSP) algorithm introduced by Duan et al. [4]. The BMSSP framework bypasses the need for sorting by employing a recursive divide-and-conquer strategy and specialized data structures to maintain distance estimates. We present the key assumptions, data structures, and algorithmic procedures, offer intuition behind its asymptotic complexity of $O(m \log^{2/3} n)$, and outline planned implementation and benchmarking against Dijkstra and, time permitting, Bellman–Ford algorithms. Our code is available at https://github.com/BakdaurenNarbayev/BMSSP. A demo of the GUI can be found here.

1 Introduction and Motivation

The Single Source Shortest Path (SSSP) problem is a central task in graph algorithms, with widespread applications in routing, network analysis, and optimization. Classical approaches, most notably Dijkstra's algorithm [3], determine shortest paths by repeatedly selecting the vertex with the smallest tentative distance. Although effective, this selection step relies on efficient priority management, creating a sorting-related bottleneck that limits performance on large graphs. Bellman–Ford [2] removes this dependence by relaxing all edges in each iteration, enabling it to handle negative weights, but its higher time complexity makes it less suitable for large-scale use.

A recent development by Duan et al. [4] introduces the Bounded Multi-Source Shortest Path (BMSSP) algorithm, which breaks the traditional sorting barrier while achieving a running time of $O(m \log^{2/3} n)$. This improves the theoretical guarantee over Dijkstra's algorithm and earned the work the Best Paper Award at STOC '25. This theoretical advance motivated our empirical question: Does the asymptotic improvement translate into observable speedups in practice?

To examine this question rigorously, we implement BMSSP directly from its formal description, as many open-source implementations diverge from the data structures prescribed in the original work and can therefore obscure the algorithm's intended performance characteristics. A statement outlining the contributions of each group member is provided in Appendix A.

The report is structured as follows:

(1) A background on the Bellman-Ford and Dijkstra's algorithms
(2) An introduction to the BMSSP algorithm.
(3) Intuition behind the theoretical analysis of the runtime complexity of the BMSSP algorithm.
(4) An empirical comparison between the Bellman-Ford, Dijkstra's, and BMSSP algorithms.
(5) A description of the implemented graphical user interface (GUI)
(6) A conclusion summarizing the findings of the report.

*All authors contributed equally to this report.

Authors' Contact Information: Ghaith Chrit, gchrit@student.ubc.ca; Bakdauren Narbayev, bakdauren.narbayev@ubc.ca; Hamza Muhammad Anwar, hanwar02@student.ubc.ca, The University of British Columbia, Okanagan, British Columbia, Canada.

2 Background

In this section, the algorithms used in the BMSSP (and whose records it beat) will be introduced.

2.1 Notation

Throughout this report, the following notation is used for clarity:

- $G = (V, E)$: the graph under consideration.
- V: the set of vertices (nodes).
- E: the set of edges.
- $n = |V|$: the number of vertices in the graph.
- $m = |E|$: the number of edges in the graph.
- $S \subseteq V$: the set of source vertices.
- B: an upper bound on the distance values considered during computation.
- $\mathcal{Q}$: a min-heap priority queue for managing distance estimates in Dijkstra's algorithm.
- $\mathcal{D}$: the custom block-based linked-list data structure used for managing distance estimates.
- $O(\cdot)$: standard asymptotic notation for upper bounds on time or space complexity.

2.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm was independently published in 1956 and 1958 by mathematicians Lester Ford Jr. and Richard Bellman [2, 6]. It finds the shortest path from a source node to all other nodes in a weighted digraph. It is capable of handling edges with negative weights and even detecting and reporting negative-weight cycles, which are cycles where the sum of the edge weights is less than zero. Suppose such a cycle is present in the graph. In that case, the shortest path becomes undefined, as you can continuously loop through the negative cycle to decrease the path's total weight, resulting in a cost of negative infinity [1, 8].

2.2.1 Time and Space Complexity. The time complexity of the Bellman-Ford algorithm is $O(nm)$. It is proportional to the number of edges, as the algorithm only traverses through each edge once, iterating through all the vertices [9]. The distance initialization of each node other than the source node to infinity takes $O(n)$ time, and iterating through all edges and performing relaxation takes $O(m)$ time in each iteration, making the total time complexity $O(nm)$.

The space complexity of the Bellman-Ford algorithm is $O(n)$, because the distances from the source vertex to all other ones need to be stored [9].

As shown in Algorithm 1, the Bellman-Ford algorithm works by updating the distances along every edge from the source node each iteration. Each edge is relaxed, meaning it is examined to see if it lessens the current shortest path distance. If it does, then that distance is stored as the new current shortest path distance. In the last, n^{th} iteration, the algorithm goes through all the edges one more time to check for any negative cycle. If the distance to any vertex can still be reduced during this final check, it means there is a negative-weight cycle reachable from the source. If not, then no negative cycles were found [2, 8].

2.3 Dijkstra's Algorithm

Dijkstra's algorithm was first published in 1959 by Edsger W. Dijkstra [3]. Like the Bellman-Ford algorithm, it also finds the shortest paths between nodes in a weighted graph. But unlike Bellman-Ford, Dijkstra's algorithm is greedy. Due to this, it cannot handle negatively weighted edges. It assumes that once a node's shortest path is found, it's final; however, a negative-weight edge discovered later could create an even shorter path to that node, violating this core assumption. This makes the algorithm's "final" shortest path incorrect and can lead to it never terminating if there

Algorithm 1 Bellman-Ford Algorithm

```
1:  procedure BellmanFord(G, s)
2:      V ← vertices of G
3:      E ← edges of G
4:      for each vertex v ∈ V do                          ▷ Initialize distances
5:          d[v] ← ∞
6:      end for
7:      d[s] ← 0
8:      for i = 1 to |V| − 1 do                           ▷ Relax edges |V| − 1 times
9:          for each edge (u, v) ∈ E with weight w(u, v) do
10:             if d[u] + w(u, v) < d[v] then
11:                 d[v] ← d[u] + w(u, v)
12:             end if
13:         end for
14:     end for
15:     for each edge (u, v) ∈ E with weight w(u, v) do   ▷ Check for negative cycles
16:         if d[u] + w(u, v) < d[v] then
17:             return Negative cycle detected
18:         end if
19:     end for
20:     return d
21: end procedure
```

are negative cycles, as the path weight can decrease indefinitely by repeatedly traversing the cycle [11].

2.3.1 Time and Space Complexity. The time complexity of Dijkstra's algorithm is $O((n + m) \log n)$ when utilizing a priority queue. This data structure is optimized for this task, as extracting the minimum-distance vertex V times is more efficient, taking $O(n \log n)$ time [10]. Also, decreasing the key for E edges contributes $O(m \log n)$ time, making the total complexity $O((n + m) \log n)$.

Similar to the Bellman-Ford algorithm, the space complexity of Dijkstra's algorithm is $O(V)$, because the distances from the source vertex to all other ones need to be stored, along with additional space for data structures like priority queues [10].

As shown in Algorithm 2, Dijkstra's algorithm uses a priority queue, such as a min heap, to store the nodes and their distances. It first selects the unvisited node with the currently smallest known distance from the source and marks it as visited. This becomes the new current node. Then, it calculates the total distance from the start node through the current node to the neighbour. If the new path is shorter than the neighbour's current recorded distance, update the neighbour's distance and record that the path comes from the current node. The algorithm terminates when all nodes are visited [8].

3 BMSSP Algorithm

3.1 Assumptions

The analysis and description of the BMSSP–based algorithm rely on a few simplifying assumptions that are standard in the theoretical formulation presented in [4].

Algorithm 2 Dijkstra's Algorithm

```
1:  procedure DIJKSTRA(G, s)
2:      V ← vertices of G
3:      E ← edges of G
4:      for each vertex v ∈ V do                ▷ Initialize distances and priority queue
5:          d[v] ← ∞
6:      end for
7:      d[s] ← 0
8:      Q ← priority queue containing all vertices in V
9:      while Q is not empty do                 ▷ Process vertices in order of distance
10:         u ← vertex in Q with minimum d[u]
11:         Remove u from Q
12:         for each neighbor v of u do
13:             alt ← d[u] + w(u, v)
14:             if alt < d[v] then
15:                 d[v] ← alt
16:                 Update priority of v in Q
17:             end if
18:         end for
19:     end while
20:     return d
21: end procedure
```

Constant-Degree Graph. We assume that the input graph has constant in-degree and out-degree. This assumption is important because it guarantees that each relaxation step affects only a constant number of edges, simplifying both the analysis and the implementation. However, this assumption does not restrict generality. Any arbitrary directed weighted graph $G = (V, E)$ can be transformed into an equivalent constant-degree graph $G' = (V', E')$ while preserving the shortest paths. The transformation proceeds as follows [7]:

(1) Substitute each vertex $v \in V$ with a cycle of vertices that are strongly connected using zero-weight edges.
(2) For every incoming or outgoing neighbor w of v, create a vertex x_{vw} on this cycle.
(3) For every edge $(u, v) \in E$ with weight w_{uv}, add a directed edge from vertex x_{uv} to x_{vu} with the same weight w_{uv}.

This transformation preserves all shortest paths in the original graph G, since all zero-weight edges are internal to the cycles and do not alter path costs. Furthermore, every vertex in the transformed graph G' has both in-degree and out-degree at most 2. Because each edge in G introduces a constant number of new vertices and edges, the transformed graph has

$$|V'| = O(m) \quad \text{and} \quad |E'| = O(m).$$

Total Order of Paths. Another assumption made for simplicity is that all paths have a total order by their lengths. In other words, no two distinct paths share the exact same total weight. This guarantees that comparisons between tentative distances are unambiguous and that the algorithm's selection steps are well-defined. In practice, this assumption is non-restrictive: any deterministic tie-breaking rule (such as ordering by vertex ID or edge index) can be used to handle equal-weight paths without affecting correctness or asymptotic performance.

3.2 General Algorithm

While the algorithm ultimately solves the *single-source shortest path* (SSSP) problem, it is more intuitively explained in terms of the *bounded multi-source shortest path* (BMSSP) formulation described in [4]. The BMSSP problem can be stated as follows: given a set of sources S with their initial distances and a distance bound B, the goal is to report, for every vertex $x \in V$, the distance $d[x]$ from any source if $d[x] < B$.

The algorithm approaches this problem using a divide-and-conquer strategy by recursively partitioning the computation into t subproblems.[1] The first subproblem considers the subset $\{x \in S \mid d[x] < B_1\}$, while each subsequent subproblem includes $\{x \in S \mid B_{i-1} \leq d[x] < B_i\}$ along with the out–neighbors of the vertices processed in the previous subproblem. To improve the running time, the size of S in each subproblem is reduced by a factor of k. As shown in [4], the optimal value is $k = \log^{1/3}(n)$.

The pruning step follows the procedure outlined in [4, Lemma 3.2]. The aim is to identify a set of *pivots*, $P \subseteq S$, of size at most n/k. This is achieved by performing k iterations of the Bellman–Ford algorithm from each source. The roots of the resulting forest of shortest paths containing at least k nodes serve as the pivots which will then be split into further subproblems.

In the theoretical formulation, it is assumed that the distance bounds can be divided into equal intervals. In practice, however, many distances are initially unknown, and the split points B_i are estimated dynamically using known distance values. A custom data structure is maintained to store and update these estimates. Whenever a subproblem becomes too large[2], the computation halts and returns control to its parent, which subdivides it further into smaller subproblems based on the available distance approximations. At the base level, the problem size includes a single source node, where a *mini-Dijkstra*, a Dijkstra algorithm limited to at most k vertices, is executed to complete the local computation.

Although the method is introduced as a multi-source algorithm, the SSSP version can be viewed as a special case with $|S| = 1$. In this setting, the recursion begins with a single source, progressively subdividing the problem until reaching the base case. After executing a mini-Dijkstra, several vertices are processed and become active sources for the parent level. By maintaining multiple sources across recursion levels and applying pruning at each step, the algorithm efficiently balances the number of source vertices, leading to improved control over both computation and memory usage.

3.3 Data Structure

As described in Section 3.2, a specialized data structure $\mathcal{D}$ is employed to maintain and efficiently update the estimated distance bounds during recursion.

The structure consists of two sequences of blocks, $\mathcal{D}_0$ and $\mathcal{D}_1$, as shown in Figure 1. Each block is implemented as a linked list of key–value pairs, where the key corresponds to a vertex and the value represents its current estimated distance. While the elements within each block are unsorted, the blocks themselves are ordered by their upper bounds using a balanced binary search tree (BST), enabling logarithmic-time access to the correct block.

Each block in $\mathcal{D}_1$ contains at most M key–value pairs, where M depends on the recursion depth l and is typically defined as $M = 2^{(l-1)t}$. Thus, higher-level subproblems (with larger l) manage larger blocks, while deeper subproblems handle smaller ones. Each subproblem maintains an independent instance of the data structure, ensuring localized updates without interference between recursion levels.

[1] Following [4], $t = \log^{2/3}(n)$.

[2] In practice, the threshold is set to $k \cdot 2^{lt}$, where l denotes the current recursion depth.

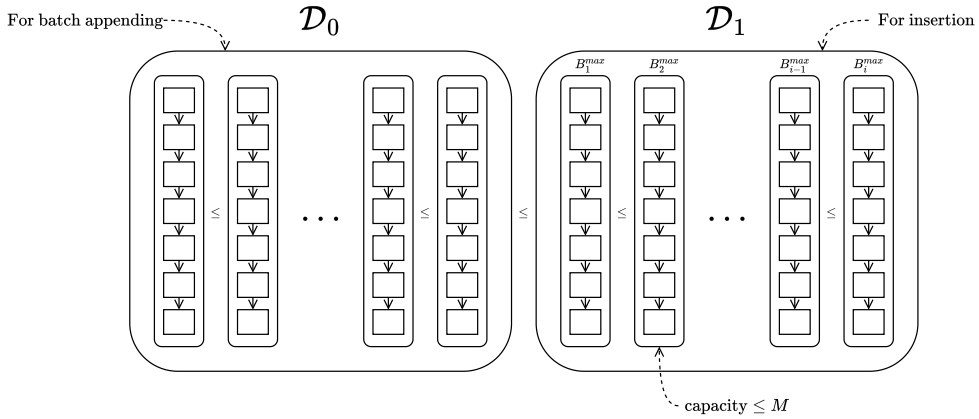

Fig. 1. Illustration of the data structure used in the BMSSP-based explanation of the algorithm.

Inserting a new key–value pair involves identifying the appropriate block in $\mathcal{D}_1$, which can be done in $O(\log(N/M))$ time, where N is the maximum number of elements to be stored. If a block exceeds its capacity, M, a split operation is triggered, followed by rebalancing of the BST.

The data structure supports efficient *pull* and *batch-append* operations. A pull operation retrieves up to M elements, starting from $\mathcal{D}_0$ and then accessing $\mathcal{D}_1$ if necessary. The $\mathcal{D}_0$ sequence allows the appending of a batch of L key–value pairs whose values are smaller than any currently stored element, allowing quick propagation of improved distance estimates from deeper recursion levels.

Overall, $\mathcal{D}$ offers a balance between flexibility (through linked lists) and ordered access (via the BST), enabling the BMSSP–based approach to efficiently maintain and update distance estimates while controlling insertion and lookup costs.

4 Complexity analysis

4.1 Time Complexity

As mentioned earlier, this section provides an intuitive overview of the time complexity of the BMSSP algorithm. For a more rigorous and detailed analysis, we refer the reader to the original paper [4]. For this section, we assume that all the nodes are sources and all boundaries are available to simplify the analysis.

Assume that we are currently within a subproblem containing n' nodes. The pruning step, following the assumptions in Section 3.1 (constant–degree graph), requires $O(k^2)$ time. After pruning, for each pivot we must determine to which of the t subproblems it belongs. Using binary search, this step takes $O(\log t)$ time per pivot. Since there are at most n'/k pivots, the total cost per subproblem is:

$$O\left(\frac{n'}{k}\log t + k^2\right).$$

At the current recursion level, there can be up to n/n' such subproblems. Therefore, the total cost per level is:

$$O\left(\frac{n}{k}\log t + \frac{n}{n'}k^2\right).$$

For simplicity, we use the height of the *unpruned* recursion tree as an upper bound on the number of levels in the pruned case. The height is approximately:

$$\frac{\log n}{\log t}.$$

At each level, the work required to assign sources to the next level is $O\left(\frac{n}{k}\log t\right)$. Over all levels, this accumulates to:

$$O\left(\frac{n}{k}\frac{\log n}{\log t}\log t\right) = O\left(\frac{n}{k}\log n\right).$$

For the pruning phase, the total work over all levels is:

$$O\left(k^2(1 + t + t^2 + \cdots + t^\alpha)\right) = O(nk^2),$$

since the geometric series sums to $O(t^\alpha)$, which equals to $O(n)$.

Balancing the two dominant terms, the work of splitting subproblems and pruning, leads to:

$$n\frac{\log n}{k} = nk^2,$$

which implies:

$$k = \log^{1/3}(n).$$

Substituting this optimal value of k into the expressions above yields the overall complexity:

$$O\big(n\log^{2/3}(n)\big) = O\big(m\log^{2/3}(n)\big),$$

where m denotes the number of edges in the graph.

Thus, under the stated assumptions, the BMSSP–based approach achieves a sublinear logarithmic factor improvement over the standard Bellman–Ford or Dijkstra formulations when analyzed in this bounded multi-source framework.

4.2 Space Complexity

In the original paper [4], the space complexity has not been evaluated. Here we provide our rough intuition on what is lower bound on space complexity.

In Sections 3.2 and 3.3, we referred to a specialized data structure $\mathcal{D}$ that is used to update the estimated distance bounds during recursion. However, this data structure is not shared among sub-problem calls (refer to [4, Algorithm 3]). Each call to BMSSP solver maintains its own $\mathcal{D}$, which we believe is a main contributor for auxiliary space usage (ignoring data structures to maintain graph, distance estimates, and node predecessors). For a particular level l and its sub-problem of size n', BMSSP solver for this sub-problem will maintain a data structure $\mathcal{D}$, which maintains distance estimates for this sub-problem. Hence, its size will be $\Omega(n')$ when all distances are estimated for this sub-problem. For the level l, we would have n/n' such sub-problems, which results in total auxiliary space of $\Omega(n)$. Since the number of levels is $\log^{1/3}(n)$ [4], the total auxiliary space complexity is $\Omega(n\log^{1/3}(n))$.

Due to approaches we took in implementing $\mathcal{D}$, the expected space complexity is much larger since each $\mathcal{D}$ maintains a list of references to all nodes. This is done to allow for expected time complexities mentioned in the paper [4], while allowing for expected functionality of the data structure. Since implementation details are not mentioned in the paper, we took liberty in certain decisions, which resulted in $O(n)$ auxiliary space for each sub-problem. Assuming that each node will end up being a singleton or base sub-problem, the total number of sub-problems on the lowest level would be n. Since sub-problems form a tree (with at least two branches on non-leaf sub-problems), we can safely assume that the total number of sub-problems is $O(n)$. Hence, total auxiliary space complexity is $O(n^2)$ in our implementation.

5 Empirical Comparison

5.1 Experimental Setup

All experiments were performed on a machine with the hardware and software configuration summarized in Table 1.

Component	Specification
CPU	Intel(R) Core(TM) i5-12600K, 64-bit, max 3.7 GHz
Memory	16 GB
Operating System	Windows 11
Python Version	3.13
Random Seed	42

Table 1. Experimental system configuration.

While exact numerical results may vary due to inherent system performance fluctuations, the observed trends and relative patterns are expected to be consistent under a stable system load. In some rare cases, the BMSSP will not visit some terminal nodes. In that case, we simply run the base case of the algorithm, which is a mini-version of Dijkstra as explained in Section 3.2.

5.2 Dataset

For the purposes of the benchmark, graph datasets were generated synthetically to allow precise control over their size and density. The number of nodes was varied logarithmically from 10 to 10^7, and for each node size, the number of edges was determined according to an edge ratio selected from $\{1.2, 1.5, 1.8\}$. Larger graphs (e.g., with 10^9 nodes) were not considered due to computational constraints, as the performance patterns observed at lower scales were already representative. This methodology enables the systematic evaluation of algorithmic performance across a wide range of graph sizes and densities.

5.3 Empirical Results

Dijkstra's empirical running times, reported in Table 2, closely track the expected $O(n \log n)$ behaviour: when plotted on the same axes the measured timing curve for Dijkstra exhibits the shallower growth and slope consistent with the $O(n \log n)$ model, and its absolute wall-clock costs remain substantially lower than the other methods across all tested node sizes. Bellman–Ford, by contrast, displays a markedly steeper timing curve that agrees with the theoretical $O(n\,m)$ bound. Given that in our experiments, the edge counts was proportional to node counts (constant edge–ratio), $O(n\,m)$ in practice behaves like $O(n^2)$, and the slope obtained from the empirical data aligns closely with the quadratic trend visible in Figure 2. Comparing Dijkstra and Bellman–Ford directly therefore highlights the expected divergence in scalability: Dijkstra scales much more favourably as n grows, whereas Bellman–Ford's run times escalate rapidly under our edge model and approach the quadratic regime predicted by theory. On the other hand, BMSSP consistently requires more time than Bellman–Ford for the sizes we measured, as shown in Table 2, and extrapolating its measured pattern suggests growth that is substantially faster than a fixed polynomial yet slower than purely exponential – i.e., quasi-polynomial – which is compatible with its theoretical characterization of $O(m \log^{2/3}(n))$. However, the practical penalty for BMSSP is amplified by implementation and bookkeeping costs. For example, following the reference implementation, it performs many recursive calls and allocates new data structures per call (rather than reusing containers in an

Algorithm	Edge Ratio	Node Sizes						
		10	100	1,000	10,000	100,000	1,000,000	10,000,000
Dijkstra	1.2	4.66e-05	2.426e-04	3.4853e-03	3.9945e-02	5.246449e-01	7.459169	96.2596233
	1.5	4.96e-05	2.616e-04	4.1235e-03	4.55632e-02	5.784414e-01	7.9979997	107.040923
	1.8	5.20e-05	3.119e-04	4.3736e-03	4.71263e-02	6.327364e-01	8.6951988	113.7238705
Bellman–Ford	1.2	9.71e-05	9.0322e-03	8.833487e-01	86.7029292	–	–	–
	1.5	1.03e-04	1.06599e-02	1.0937101	108.2363484	–	–	–
	1.8	1.116e-04	1.33582e-02	1.3143024	130.4091597	–	–	–
BMSSP	1.2	7.2089e-03	1.8137201	117.2841189	–	–	–	–
	1.5	7.6483e-03	1.4638407	120.2204348	–	–	–	–
	1.8	6.981e-03	2.1279052	170.8601043	–	–	–	–

Table 2. Median run times (seconds) of different shortest-path algorithms for varying node sizes and edge ratios across 3 different seeds. Entries marked '–' indicate unavailable data due to computational constraints.

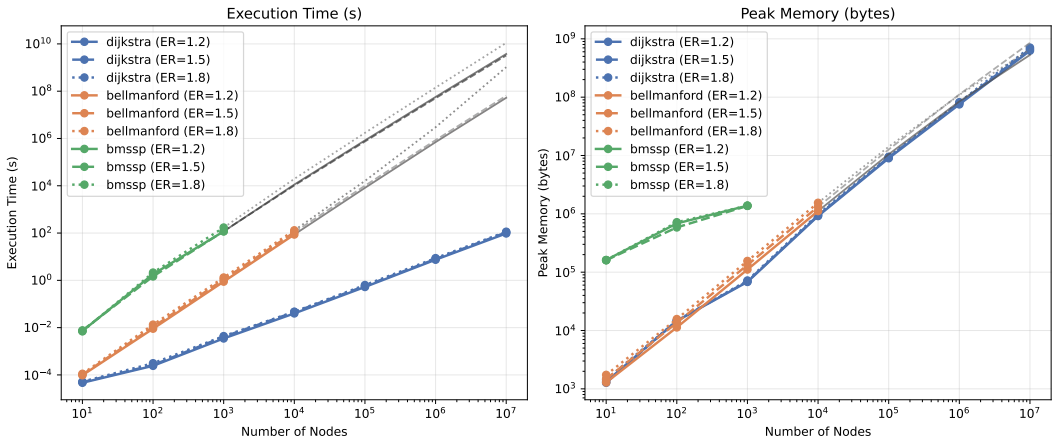

Fig. 2. Left: Runtime of the algorithm. Right: Peak memory usage of the models. Gray regions indicate extrapolated values obtained by fitting a best polynomial or quasi-polynomial, depending on the algorithm, to illustrate expected future trends.

iterative formulation), so per-call allocation, copying, and recursion overhead inflate the observed running time relative to what an idealized asymptotic bound would suggest.

Regarding memory, the observed ordering of peak usage, reported in Table 3, shows Dijkstra and Bellman–Ford as moderate and BMSSP as much larger, which matches expectations from the algorithms' working storage requirements ($O(n + m)$ for the former two and higher nominal transient usage for BMSSP because of recursive frames and temporary structures). However, peak memory measurements should be treated as an implementation-level indicator rather than a strict measurement of algorithmic space complexity, since allocator behaviour, fragmentation, transient buffers, and OS/runtime overheads all affect the recorded peaks. Overall, the empirical results illustrated in Tables 2 and 3 and in Figure 2 confirm the theoretical trends while highlighting practical overheads.

6 Graphical User Interface

To visualize and compare the results of the Bellman-Ford, Dijkstra's, and BMSSP algorithms, we built a graphical user interface (GUI). Figure 3 shows the main menu of the GUI. To launch it, simply run `python gui/run.py` in the terminal.

Algorithm	Edge Ratio	Node Sizes						
		10	100	1,000	10,000	100,000	1,000,000	10,000,000
Dijkstra	1.2	1,328	14,592	68,096	917,128	8,974,320	75,154,768	625,431,032
	1.5	1,272	14,792	69,392	940,784	9,304,680	78,870,232	662,889,856
	1.8	1,440	14,672	71,984	985,096	9,503,368	81,912,320	707,038,352
Bellman–Ford	1.2	1,288	11,304	111,048	1,116,136	–	–	–
	1.5	1,480	13,576	132,968	1,321,672	–	–	–
	1.8	1,736	15,720	155,624	1,545,992	–	–	–
BMSSP	1.2	161,520	684,968	1,380,232	–	–	–	–
	1.5	160,262	584,744	1,367,808	–	–	–	–
	1.8	160,904	718,112	1,381,448	–	–	–	–

Table 3. Median peak memory usage (bytes) of different shortest-path algorithms for varying node sizes and edge ratios across 3 different seeds. Entries marked '–' indicate unavailable data due to computational constraints.

On the top, you have the options to choose a random graph to generate, as shown in Figure 4, as well as which algorithm you want to run on that graph (Figure 5). You can choose any graph and press the "Open Image" button to view it. "Generate New Graph" will generate a new random graph of the chosen type. Then you can choose the algorithm you want to run on that graph and press the blue "Run Algorithm" button. The keyboard shortcuts shown in the lower left can also be used instead of pressing the buttons with the mouse.

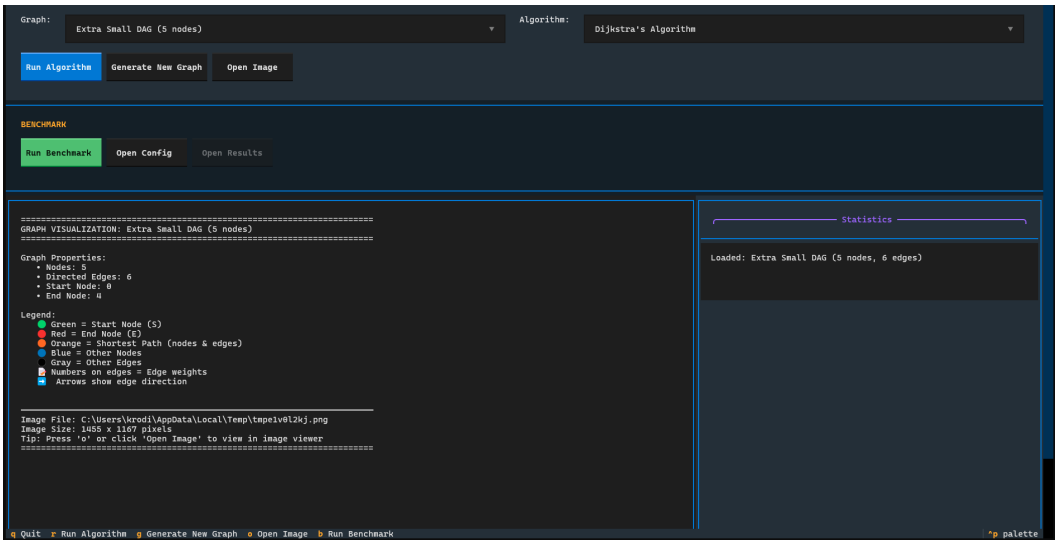

Fig. 3. The main menu of the GUI showing all the options.

Below the top header is the Benchmark header, which contains the buttons for benchmarking the algorithms with each other. The "Open Config" button opens the file where the configuration options can be set. The README of our GitHub contains more details about these options.

The lower left box contains information about the Graph properties and the legend related to the colour of the nodes. The box on the right contains statistics of the algorithm run, including its execution time and number of operations. At the bottom of it, the history of the commands run can be seen.

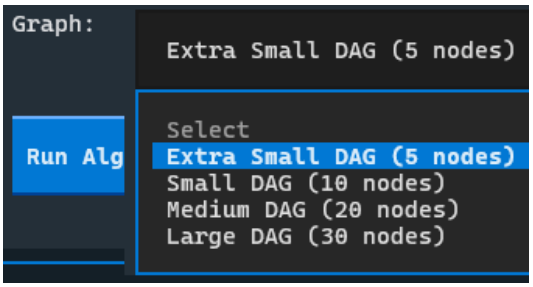

Fig. 4. The four types of random graphs that the GUI can generate

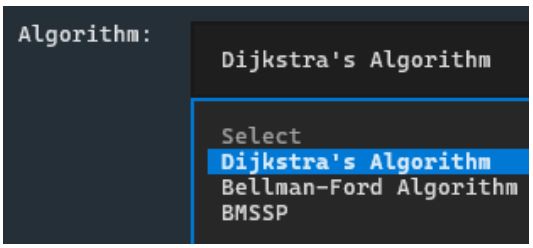

Fig. 5. The Algorithms' drop-down menu showing the three shortest-path algorithms.

7 Conclusion

This report provided an overview and detailed exposition of the BMSSP algorithm for solving the Single-Source Shortest Path problem, along with a background of the Bellman-Ford and Dijkstra's algorithms. By framing the problem in the bounded multi-source setting, we highlighted how the algorithm eliminates the traditional dependence on complete sorting, thereby achieving improved asymptotic performance of $O(m \log^{2/3} n)$ under the stated assumptions. We implemented the BMSSP, and due to the expensive bookkeeping and recursive calls on relatively small graphs, it will be worse than Bellman-Ford. Given its quasi-polynomial pattern, we expect it to beat Dijkstra's algorithm on extremely sparse graphs. We also described the graphical user interface developed to run and compare the three algorithms on a number of randomly generated graphs.

Acknowledgments

The authors acknowledge that large language models (LLMs) were used to improve the clarity and flow of the text in this report and in the complementary code. All mathematical results, analyses, and conclusions presented were independently verified by the authors. The authors also wish to recognize that their understanding of the BMSSP problem was heavily influenced by the talk of one of the co-authors Xiao Mao [5].

References

[1] Jørgen Bang-Jensen and Gregory Z. Gutin. 2009. Distances. In *Digraphs: Theory, Algorithms and Applications*, Jørgen Bang-Jensen and Gregory Z. Gutin (Eds.). Springer, London, 87–126. doi:10.1007/978-1-84800-998-1_3

[2] Richard Bellman. 1958. On a routing problem. *Quart. Appl. Math.* 16, 1 (Apr 1958), 87–90. doi:10.1090/qam/102435

[3] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (01 Dec 1959), 269–271. doi:10.1007/BF01386390

[4] Ran Duan, Jiayi Mao, Xiao Mao, Xinkai Shu, and Longhui Yin. 2025. Breaking the Sorting Barrier for Directed Single-Source Shortest Paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing* (Prague, Czechia) *(STOC '25)*. Association for Computing Machinery, New York, NY, USA, 36–44. doi:10.1145/3717823.3718179

[5] EnCORE. 2025. *Breaking the Sorting Barrier for Directed Single Source Shortest Paths.* YouTube. https://www.youtube.com/watch?v=LzvvcadKbd0

[6] L. R. Ford. 1956. *Network Flow Theory.* Technical Report.

[7] Greg N. Frederickson. 1983. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. Association for Computing Machinery, New York, NY, USA, 252–257. doi:10.1145/800061.808754

[8] Jon Kleinberg and Eva Tardos. 2005. *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., USA.

[9] tarunsarawgi_gfg. 16:06:25+00:00. Time and Space Complexity of Bellman–Ford Algorithm. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-bellman-ford-algorithm/.

[10] tarunsarawgi_gfg. 17:46:06+00:00. Time and Space Complexity of Dijkstra's Algorithm. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-dijkstras-algorithm/.

[11] templatetypedef. 2011. Answer to "Negative Weights Using Dijkstra's Algorithm".

A Roles

Author	Contributions
Ghaith Chrit	Benchmark logic implementation
	GUI logic implementation
	Empirical Comparison between the different methods
Bakdauren Narbayev	BMSSP implementation
	Unit tests for BMSSP
Hamza Muhammad Anwar	Background information
	GUI logic implementation
	Unit tests for Bellman-Ford and Dijkstra.

Table 4. Roles and contributions of each author.

Tab. 4 summarizes the contributions of each author. Each author wrote their specific sections in the report and presentation, while other sections, such as the introduction, were a shared effort.