Implementation and Brief Experimental Analysis of the Duan et al. (2025) Algorithm for Single-Source Shortest Paths

Lucas Castro
Instituto de Computação - UFAM, Brazil

Thailsson Clementino
Instituto de Computação - UFAM, Brazil

Rosiane de Freitas
Instituto de Computação - UFAM, Brazil

Abstract

We present an implementation and a brief experimental analysis of the deterministic algorithm proposed by Duan et al. (2025) for the Single-Source Shortest Path (SSSP) problem, which achieves the best known asymptotic upper bound in the comparison-addition model, with running time $O(m \log^{2/3} n)$. We provide a faithful C++ implementation of this algorithm, following all structural details described in the original paper, and compare its empirical performance with the classical Dijkstra's algorithm using binary heaps. The experiments were conducted on both synthetic sparse random graphs and real-world road network instances from the DIMACS benchmark. Our implementation achieves the proposed running time in the worst case. However, our results show that, despite the new algorithm's superior asymptotic complexity, its large constant factors significantly affect its practical performance, making Dijkstra's algorithm faster for all tested sparse graph sizes, including instances with tens of millions of vertices. (This is a work in progress.)

2012 ACM Subject Classification Theory of computation → Shortest paths

Keywords and phrases Algorithms, Dijkstra's Algorithm, Experimental Analysis, Heap and Hash Data Structures, Single Source Shortest Path, Time Complexity.

Acknowledgements This research was partially supported by the Coordination for the Improvement of Higher Education Personnel - Brazil (CAPES-PROEX) - Funding Code 001, the National Council for Scientific and Technological Development (CNPq), and the Amazonas State Research Support Foundation - FAPEAM - through the POSGRAD 2024-2025 project.

1 Introduction

Let $G = (V, E)$, $|V| = n, |E| = m$, be a connected graph with a weight function $w : E \to \mathbb{R}_{\geq 0}$ and a source vertex $s \in V$. The **Single Source Shortest Path Problem** (SSSP) consists of determining, for each vertex $v \in V$, the minimum distance $d(v) = d(s, v)$ of a path in G that starts at s and ends at v.

As one of the most classical algorithmic problems in graph theory, the SSSP is widely studied in the literature. Until the end of the last century, the algorithm proposed by Dijkstra (1959), was still considered the state of the art for solving the SSSP. When combined with an efficient priority queue, like binary heap (Williams, 1964), Dijkstra's algorithm solves the problem with a time complexity of $O((n + m) \log n)$. In 2024, Haeupler et al. (2024) showed that Dijkstra's algorithm is universally optimal for the natural problem of ordering the vertices according to their distances $d(v)$ from the source s. In sparse graphs, where $m = O(n)$, this complexity matches the sorting barrier in the comparison-addition computation model.

To overcome the bottleneck associated with vertex ordering, some works have proposed algorithms with lower time complexity by avoiding sorting the vertices. In the RAM computation model, Thorup (1999) presented an algorithm with time complexity $O(m)$ for undirected graphs whose weights fit into a machine word, and later (Thorup, 2004) extended the approach to directed graphs, achieving $O(m + n \log \log n)$. In the comparison-addition model, Duan et al. (2023) proposed a randomized algorithm for undirected graphs with time complexity $O(m\sqrt{\log n \log \log n})$.

Recently, Duan et al. (2025) presented the first deterministic algorithm with $o(n \log n)$ complexity for sparse graphs, achieving an upper bound of $O(m \log^{2/3} n)$.

This work aims to empirically evaluate the deterministic algorithm proposed by Duan et al. (2025) and compare its practical performance with Dijkstra's classical algorithm. We implement both algorithms in `C++` (Dijkstra's algorithm using a binary heap, and Duan et al. (2025) algorithm faithfully following the implementation details described in their paper).

We acknowledge the recent Rust implementation by Valko et al. (2025) and their evaluation on Lightning Network topologies. Their work provides a practical approach, using sorting as their selection algorithm. This differs from the theoretical proposal in Duan et al. (2025), which assumes a worst-case linear-time selection algorithm. As a result of this implementation choice, the algorithm's complexity aligns with that of Dijkstra, though with higher constant factors. This implementation was evaluated on graphs of up to approximately 15k nodes, offering useful insights for networks of that particular size and topology.

During the development of this work, we also became aware that Makowski et al. (2025) are developing a Python package that aims to faithfully implement the algorithm proposed by Duan et al. (2025). They did experiments on grid and geospatial networks to compare their implementations with other Python packages which implement Dijkstra's algorithm.

To the best of our knowledge, this is the first work to present a faithful implementation of the Duan et al. (2025) algorithm and to provide a practical performance analysis on random and road networks up to approximately 10 million vertices.

The remainder of this paper is organized as follows: Section 2 outlines the main ideas of the new algorithm; Section 3 analyzes the expected behavior of the comparison; Section 4 presents key implementation details; Section 5 describes the experimental setup; Section 6 discusses the obtained results; and Section 7 concludes the paper.

2 Duan et al. (2025)'s Algorithm

The algorithm proposed by Duan et al. (2025) addresses an extension of the SSSP called *Bounded Multi-source Shortest Path* (BMSSP). In BMSSP, the goal is to compute, for each vertex $v \in V$, the minimum distance $d(v)$ from some vertex $u \in S \subseteq V$, subject to the constraint $d(v) < B$. When $S = \{s\}$ and $B = \infty$, BMSSP reduces to the standard SSSP.

The algorithm uses two parameters, k and t, and is based on a divide-and-conquer approach over the vertex set. The vertex set is recursively divided into 2^t roughly equal parts across $O((\log n)/t)$ recursion levels, until reaching the base case where a subproblem contains a single vertex x. In the base case, Dijkstra's algorithm is executed to compute the shortest distances from x to its k nearest vertices.

The algorithm faces two main bottlenecks:

1. selecting the new set S to connect sequential subproblems in a partition, and
2. controlling the size of S to prevent degeneration of the overall complexity.

To address the first bottleneck, Duan et al. (2025) use a specialized data structure to select the $2^{(l-1)t}$ vertices with the smallest distance in each partition at recursion level l. This data structure has been fully implemented in this work. To address the second bottleneck, the set S is reduced to a smaller set of *pivots* $P \subseteq S$ that suffice as sources. This reduction is performed through k iterations of a Bellman-Ford-like algorithm (Bellman, 1958).

The parameters k and t are chosen to balance the cost of the Bellman-Ford iterations with the recursion depth, resulting in a final time complexity of $O(m \log^{2/3} n)$.

3 Asymptotic versus Empirical Analysis under Big Constants

Asymptotic complexity characterizes an algorithm's growth rate by focusing on the dominant term of its running time and ignoring constants and lower-order terms. However, in practice, these constants can significantly impact performance on real computers. In this section, we present an analysis of the constants in the BMSSP algorithm to allow a more precise comparison with Dijkstra's algorithm. (For simplicity, we focus only on the dominant term and neglect the other ones.)

For sparse graphs, Dijkstra's algorithm has time complexity $O(n \log n)$, which can be approximated as $c_1 \cdot n \log n$, where c_1 is a constant reflecting the implementation overhead.

Similarly, the BMSSP algorithm has complexity $O(n \log^{2/3} n)$, which we approximate as $c_2 \cdot n \log^{2/3} n$, where c_2 is a constant reflecting the implementation overhead.

Since $O(n \log n)$ grows faster than $O(n \log^{2/3} n)$, there exists a threshold n_0 beyond which BMSSP will outperform Dijkstra. Formally, we define

$$n_0 = \min \left\{ n \mid c_2\, n\, \log^{2/3} n < c_1\, n\, \log n \right\},$$

which simplifies to

$$n_0 = \min \left\{ n \mid \log^{1/3} n > c_2/c_1 \right\}.$$

Assuming different ratios c_2/c_1 (that is, assuming different rates at which the BMSSP constant is greater than Dijkstra's) the threshold values n_0 are as follows:

- $c_2/c_1 = 2$: BMSSP is faster for $n > 64$
- $c_2/c_1 = 3$: BMSSP is faster for $n \gtrsim 10^8$
- $c_2/c_1 = 4$: BMSSP is faster for $n \gtrsim 10^{19}$
- $c_2/c_1 = 5$: BMSSP is faster for $n \gtrsim 10^{38}$
- $c_2/c_1 = 6$: BMSSP is faster for $n \gtrsim 10^{65}$
- $c_2/c_1 = 7$: BMSSP is faster for $n \gtrsim 10^{103}$

This analysis predicts that even if the constant in a BMSSP implementation is only five times larger than that of Dijkstra's algorithm, Dijkstra will remain faster for almost all practical graph sizes.

4 Implementation Details

We provide implementations for both algorithms discussed in this work. All implementations are written in `C++20`, compiled with `g++` using the `-O3` optimization flag. The code is publicly available at `https://github.com/lcs147/bmssp`.

The algorithm proposed by Duan et al. (2025) requires that all simple paths have distinct lengths. To satisfy this requirement in a general setting, we must break ties consistently when paths have equal length. As described in the original paper, we represent the distance to a vertex v as the tuple

$$\hat{d}[v] = \big(\text{length}[v],\ \text{hops}[v],\ v,\ \text{pred}[v]\big).$$

Here, length$[v]$ is the total weight of the current shortest path to v, hops$[v]$ is the number of edges in this path, and pred$[v]$ is the predecessor of v in this path. Paths are compared using the lexicographic order on these tuples, which guarantees a unique ordering even if their length values are identical.

Dijkstra's algorithm was implemented using the binary heap provided by the `C++` standard library (`std::priority_queue`). We selected this data structure because binary heaps are often more efficient in practice than Fibonacci heaps (Cherkassky et al., 1996).

Our implementation closely follows the algorithm proposed by Duan et al. (2025). We implemented all three subroutines and the core data structure as specified, ensuring that the theoretical worst-case time complexity of $O(m \log^{2/3} n)$ is preserved. Below, we list some implementation details:

Additional memory usage

The block-based data structure in the original paper stores at most N key-value pairs. It must support finding any key and its associated data in $O(1)$ time.

Instead of using a hash table to find the keys (which provides expected $O(1)$ operations), we employ global direct-address tables (DATs) of size $O(n)$, where n is the number of vertices. Naively, each recursive call of BMSSP (Algorithm 3) would require its own DAT, leading to an infeasible number of tables. However, due to the sequential nature of the divide-and-conquer recursion, we can reuse DATs across calls. Specifically, we maintain only one DAT *per recursion level*, as recursive calls at the same level are processed sequentially and can share the same memory. With $O(\log^{1/3} n)$ recursion levels and a DAT of size $\Theta(n)$ per level, the total space complexity becomes $\Theta(n \log^{1/3} n)$.

Set operations

To guarantee $O(1)$ worst-case operations for set operations (such as union, membership tests, and duplicate removal), we consistently use global direct-address tables (DATs) throughout our implementation. However, unlike in the bucket data structure, the other DATs do not increase the asymptotic space complexity.

FindPivots forest building.

In the FindPivots subroutine, we do not explicitly build the directed forest F. Since we only need to identify roots whose trees contain at least k vertices, we maintain an array root$[v]$ that stores the root of the tree to which vertex v belongs. Whenever an edge (u, v) is relaxed, we update root$[v] :=$ root$[u]$. This approach allows us to efficiently determine the final set of pivots without the overhead of building the full forest and traversing any vertex.

Disjointness of U_i sets

Experimentally, we encountered a subtle discrepancy with Remark 3.8 in Duan et al. (2025), which asserts that the sets U_i in a recursive call of BMSSP (Algorithm 3) are disjoint. This property may fail. Consider a vertex x that enters the data structure D with an initial distance d_{old}. If x is later marked as complete by a recursive call with a shorter distance d_{new} (so that $x \in U_i$) but is never extracted from D, it remains in D with the outdated key d_{old}. When x is eventually extracted (still with key d_{old}), it becomes a frontier vertex, causing its shortest-path subtree to be rediscovered with the correct distance d_{new}. Consequently, x and its subtree may be assigned to another set U_j, violating disjointness.

We restore the disjointness property by removing a vertex from D as soon as it is marked complete. Specifically, before line 15 of subroutine BMSSP (Algorithm 3), we perform D.`erase`(u). Assuming `erase` has complexity $O(\max\{1, \log(N/M)\})$ when the element is present and $O(1)$ otherwise, the asymptotic complexity of the algorithm is preserved. With this modification, the implementation satisfies the disjointness claim of Remark 3.8.

5 Experiments

All experiments were conducted on a computer with `32 GB` of memory and an `Intel Core i5-10400F` processor running at `2.90 GHz`, under *Linux Mint 21.3 Cinnamon*.

We tested two types of instances:

- **Sparse Random Graphs:** randomly generated graphs with sizes $2^7, 2^8, \ldots, 2^{25}$. These graphs have a mean out-degree of 3 and a maximum out-degree of 4. The vertices were numbered from 1 to n, and it is guaranteed that vertex 1 can reach all other vertices. The code used to generate these instances is available in the repository.
- **Road Graphs:** 12 graphs representing road networks in different regions of the United States, obtained from the 9th DIMACS Implementation Challenge (Demetrescu et al., 2006). Edge weights correspond to the average time required to travel along the respective street. Since these are road networks, all graphs are sparse. The number of vertices and edges for each instance is reported in Table 2.

For all instances, shortest paths were computed using vertex 1 as the source. Each algorithm was executed five times independently on each instance to reduce the influence of external factors such as operating system variations or timing fluctuations. The average execution time, measured in milliseconds, was used as the main performance metric. The results are discussed in Section 6.

6 Results

6.1 Sparse Random Graphs

Table 1 reports execution times for sparse random graphs with mean out-degree of 3. Experiments were conducted on graphs with up to 2^{25} vertices, the maximum size that could be accommodated within the available memory. Attempts to process a graph with 2^{26} vertices resulted in memory exhaustion. The scaling of average execution time (on a logarithmic scale) across graph sizes is illustrated in Figure 1.

Figure 2 illustrates the ratio of execution times between the BMSSP and Dijkstra algorithms on random graphs. For the smallest graph size (128 vertices), BMSSP is substantially slower, measuring approximately 24 times slower than Dijkstra. However, as the graph size increases, this performance disparity significantly narrows, with the ratio dropping to approximately 3 for some graphs. This decreasing ratio indicates that BMSSP becomes relatively more efficient as the input size grows. Overall, across all random graph instances, BMSSP maintains an average execution time approximately four times higher than Dijkstra.

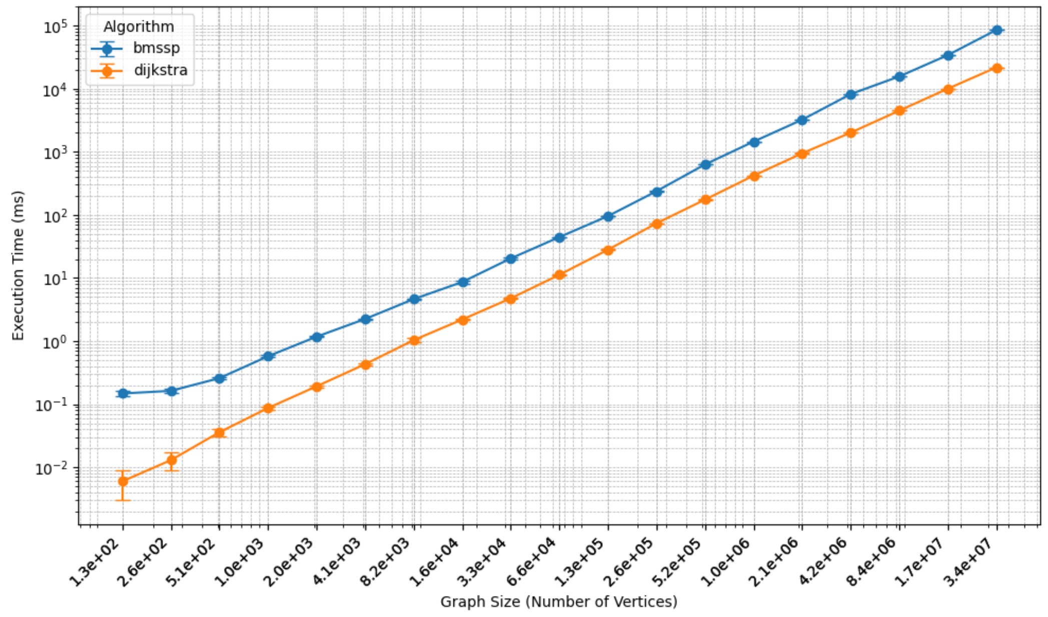

Figure 1 Execution time for random sparse graphs of increasing size, on a log-log scale.

Table 1 Execution times (in milliseconds) of our implementations of Dijkstra and Duan et al. algorithms on random sparse graphs.

Instance	n	m	Time (ms)		Ratio
			Dijkstra (1959)	Duan et al. (2025)	
random128D3	2^7	3×2^7	0.006 ± 0.003	0.148 ± 0.012	24.667
random256D3	2^8	3×2^8	0.013 ± 0.004	0.163 ± 0.010	12.538
random512D3	2^9	3×2^9	0.036 ± 0.005	0.087 ± 0.011	7.194
random1024D3	2^{10}	3×2^{10}	0.087 ± 0.006	0.575 ± 0.021	6.609
random2048D3	2^{11}	3×2^{11}	0.192 ± 0.006	1.175 ± 0.028	6.120
random4096D3	2^{12}	3×2^{12}	0.431 ± 0.017	2.244 ± 0.038	5.206
random8192D3	2^{13}	3×2^{13}	1.036 ± 0.081	4.634 ± 0.058	4.473
random16384D3	2^{14}	3×2^{14}	2.199 ± 0.040	8.639 ± 0.410	3.929
random32768D3	2^{15}	3×2^{15}	4.769 ± 0.068	20.455 ± 0.539	4.289
random65536D3	2^{16}	3×2^{16}	11.237 ± 0.143	44.527 ± 0.413	3.963
random131072D3	2^{17}	3×2^{17}	28.035 ± 0.654	95.711 ± 1.126	3.414
random262144D3	2^{18}	3×2^{18}	73.609 ± 0.903	237.379 ± 2.058	3.225
random524288D3	2^{19}	3×2^{19}	174.674 ± 2.412	631.082 ± 1.475	3.613
random1048576D3	2^{20}	3×2^{20}	418.830 ± 2.058	1450.863 ± 0.565	3.464
random2097152D3	2^{21}	3×2^{21}	946.968 ± 7.659	3226.209 ± 8.977	3.407
random4194304D3	2^{22}	3×2^{22}	2012.560 ± 3.322	8218.205 ± 13.655	4.083
random8388608D3	2^{23}	3×2^{23}	4494.410 ± 5.503	15646.055 ± 11.004	3.481
random16777216D3	2^{24}	3×2^{24}	9998.167 ± 6.375	34104.239 ± 188.505	3.411
random33554432D3	2^{25}	3×2^{25}	21765.062 ± 14.522	85382.069 ± 49.101	3.923

To check if the implementation matches the expected behavior of the algorithm, we can use the asymptotic complexities of the algorithm to simulate the same ratio as the graph size increases:

$$f(n) = \frac{\text{Time(BMSSP)}}{\text{Time(Dijkstra)}} = \frac{n \log^{2/3} n}{n \log n} = \log^{-1/3} n.$$

Figure 3 illustrates the expected ratio based on asymptotic analysis, and it is consistent with the ratio obtained experimentally.

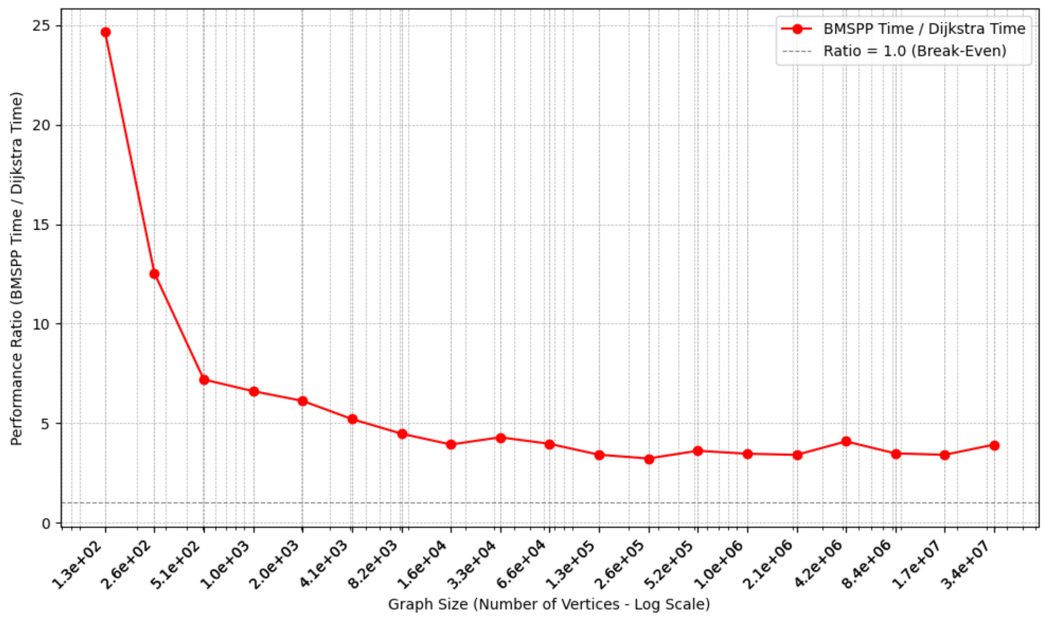

Figure 2 Experimental ratio of BMSSP to Dijkstra execution times on random sparse graphs.

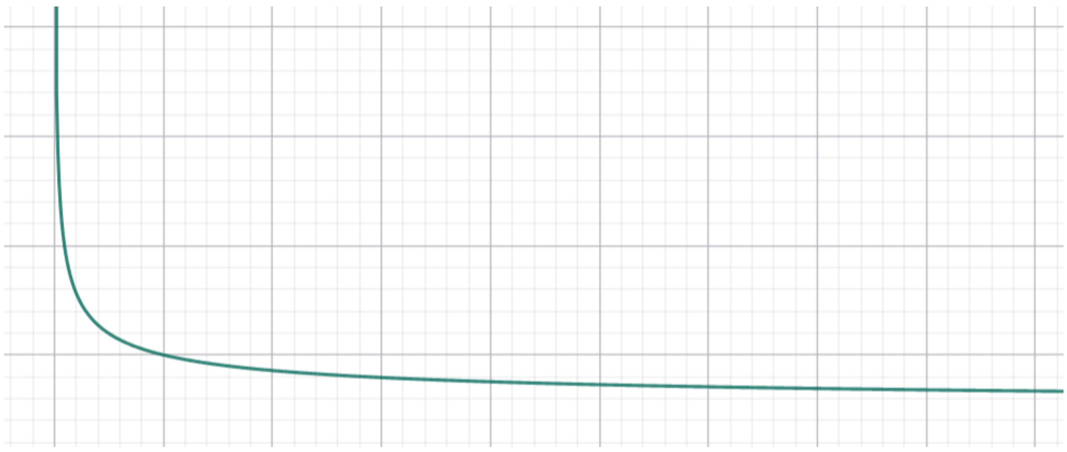

Figure 3 Theoretical ratio of BMSSP to Dijkstra execution times based on asymptotic analysis.

6.2 Road Graphs

Table 2 reports execution times for the 12 road network instances, and Figure 4 illustrates the growth of execution time with increasing graph size. The results show that Dijkstra's algorithm consistently outperforms the BMSSP algorithm by an average factor of four across all twelve road network instances. The performance ratio remains remarkably stable, ranging only from 3.67 to 4.89, despite graph sizes varying over two orders of magnitude.

The observed four-fold performance ratio is similar to the ratio previously measured on random graphs. Dijkstra maintains a clear advantage; the performance gap seems stable as the graph size increases.

Table 2 Execution times (in milliseconds) of our implementations of Dijkstra and Duan et al. algorithms on 12 USA road network instances.

Instance	n **(approx.)**	m **(approx.)**	**Time (ms)**		**Ratio**
			Dijkstra (1959)	Duan et al. (2025)	
New York City	2.6×10^5	7.3×10^5	33.694 ± 0.266	138.444 ± 0.731	4.109
San Francisco Bay Area	3.2×10^5	8.0×10^5	39.672 ± 0.535	163.748 ± 1.228	4.128
Colorado	4.3×10^5	1.0×10^6	51.731 ± 0.356	253.033 ± 1.556	4.891
Florida	1.0×10^6	2.7×10^6	126.690 ± 1.733	546.35 ± 5.189	4.313
Northwest USA	1.2×10^6	2.8×10^6	153.513 ± 1.658	621.822 ± 2.873	4.051
Northeast USA	1.5×10^6	3.8×10^6	228.126 ± 1.654	836.506 ± 3.755	3.667
California and Nevada	1.8×10^6	4.6×10^6	258.069 ± 2.587	1006.176 ± 2.887	3.899
Great Lakes	2.7×10^6	6.8×10^6	378.239 ± 3.004	1626.529 ± 3.147	4.300
Eastern USA	3.5×10^6	8.7×10^6	541.000 ± 3.923	2413.533 ± 4.204	4.461
Western USA	6.2×10^6	1.5×10^7	1012.382 ± 2.023	4549.724 ± 9.840	4.494
Central USA	1.4×10^7	3.4×10^7	3355.727 ± 12.259	14104.518 ± 21.586	4.203
Full USA	2.3×10^7	5.8×10^7	4566.797 ± 4.672	21261.355 ± 10.304	4.656

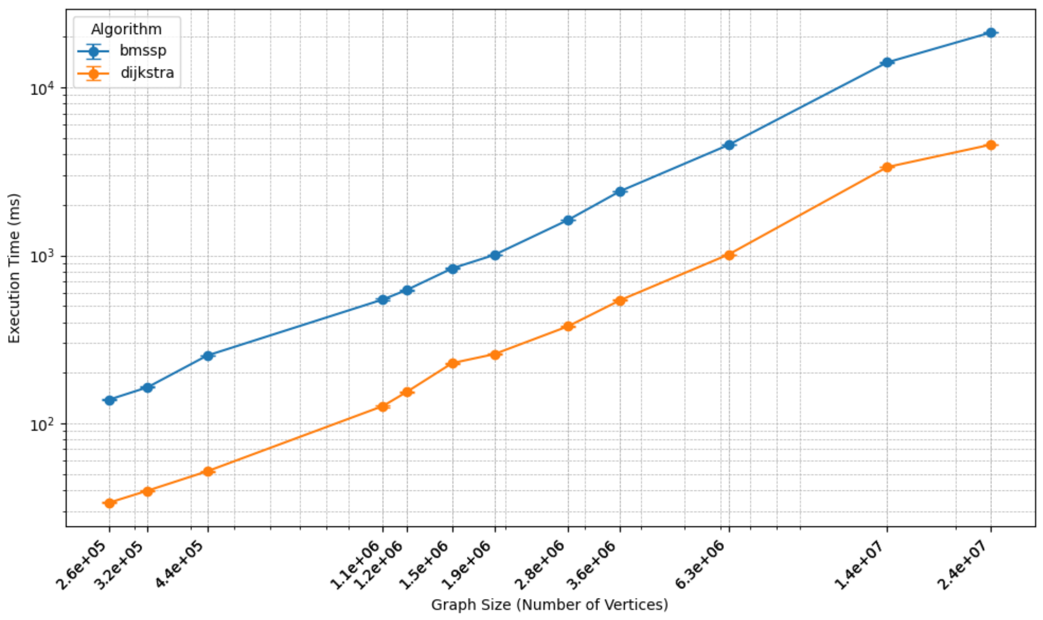

Figure 4 Execution time versus graph size for the USA road network instances, plotted on log-log scale.

6.3 General analysis

After running the tests, it was observed that the classical algorithm of Dijkstra (1959) achieved superior performance in all evaluated instances. Although, in theory, the algorithm proposed by Duan et al. (2025) has a better asymptotic upper bound, its implementation is considerably more complex, involving multiple recursive calls and nuances that introduce a large constant factor in the running time. In practice, this overhead causes the algorithm to perform worse than Dijkstra's algorithm, which is simple, direct, and highly efficient—especially when using, for example, a *binary heap* structure implemented directly in an array.

If we assume that the constant of our BMSSP implementation is four times higher than Dijkstra's ($c_2/c_1 = 4$), a ratio derived from the road network experiment, then the same analysis used in Section 3 predicts that our implementation will only be faster when the graph size surpasses 10^{19}.

These results indicate that the theoretical improvement obtained through asymptotic analysis does not necessarily translate into practical gains, due to the large hidden constants and structural complexity of the new algorithm. In most real-world environments, using Dijkstra's algorithm is therefore substantially more efficient.

7 Concluding Remarks

We presented an experimental analysis comparing Dijkstra's classical algorithm (Dijkstra, 1959) with the recent algorithm by Duan et al. (2025), which establishes the best known deterministic asymptotic SSSP upper bound in the comparison-addition model.

Our experiments show that Dijkstra's algorithm remains the preferred choice for graphs with up to ten million vertices, covering most practical applications. Furthermore, we predict that Dijkstra's algorithm should be the preferred choice for graphs significantly larger than that.

Dijkstra's algorithm remains the standard for real-world graph problems. However, the work of Duan et al. (2025) illustrates that improvements may be achievable, inspiring the exploration of algorithms that combine strong asymptotic performance with low practical overhead.

Several avenues for further investigation remain open. First, we plan to refine the BMSSP implementation by replacing its current linear-time selection routine (Blum et al., 1973) with a more efficient deterministic selection algorithm that offers better constant factors (Alexandrescu, 2017). This modification may reduce the algorithm's overhead and narrow the performance gap observed in our experiments.

Additionally, we will expand our experimental analysis in several directions. We intend to evaluate both algorithms on structured grid graphs, on which the Bellman-Ford algorithm has previously been shown to outperform Dijkstra (Cherkassky et al., 1996). We also plan to conduct systematic comparisons between expected-case and worst-case performance, implement a Fibonacci heap version of Dijkstra's algorithm as an additional baseline, and test the impact of the degree-normalization preprocessing step required by the Duan et al. (2025) algorithm to ensure constant out-degree.

Note

This is a work in progress.

References

Alexandrescu, A. (2017). Fast deterministic selection. In *16th International Symposium on Experimental Algorithms (SEA 2017)*, pages 24–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Bellman, R. (1958). On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461.

Cherkassky, B. V., Goldberg, A. V., and Radzik, T. (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174.

Demetrescu, C., Goldberg, A., and Johnson, D. (2006). 9th dimacs implementation challenge–shortest paths. *American Mathematical Society.*

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 50:269–271.

Duan, R., Mao, J., Mao, X., Shu, X., and Yin, L. (2025). Breaking the sorting barrier for directed single-source shortest paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pages 36–44.

Duan, R., Mao, J., Shu, X., and Yin, L. (2023). A randomized algorithm for single-source shortest path on undirected real-weighted graphs. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 484–492. IEEE.

Haeupler, B., Hladík, R., Rozhoň, V., Tarjan, R. E., and Tetěk, J. (2024). Universal optimality of dijkstra via beyond-worst-case heaps. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2099–2130. IEEE.

Makowski, C., Guter, W., Russell, T., and Saragih, A. (2025). Bmsspy: A python package and empirical comparison of bounded multi-source shortest path algorithm. *MIT Center for Transportation & Logistics Research Paper*, (2025/034).

Thorup, M. (1999). Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394.

Thorup, M. (2004). Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69:330–353.

Valko, D., Paranjpe, R., and Gómez, J. M. (2025). Outperforming dijkstra on sparse graphs: The lightning network use case. *arXiv preprint arXiv:2509.13448.*

Williams, J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348.