Breaking the Sorting Barrier for Directed Single-Source Shortest Paths - An Experimental Evaluation

Seminar zu Algorithmen

Omar Chatila

February 1, 2026

1 Introduction

In the realm of single-source shortest path (SSSP) algorithms, Dijkstra's algorithm has been the dominant approach since 1959 for computing the shortest paths from a single source in an n-vertex m-edge graph with non-negative edge weights [Dij59]. Recently, Haeupler et al. proved that a Dijkstra implementation using a Fibonacci Heap [FT87], resulting in a runtime of $O(m + n \log n)$, is optimal in the comparison-addition model, if the output distances are required to be reported in sorted order [Hae+24]. This lower bound is commonly referred to as the sorting barrier.
In their 2025 paper "Breaking the Sorting Barrier for Directed Single-Source Shortest Paths", Duan et al. demonstrate that this barrier can be overcome by relaxing the requirement that the distances be output in sorted order [Dua+25]. They present a deterministic SSSP algorithm, *Bounded Multi-Source Shortest Path* (BMSSP), with a running time of $O(m \log^{2/3}(n))$, which is the first algorithm to asymptotically improve upon Dijkstra's algorithm for directed sparse graphs.

The goal of this work is to present the BMSSP algorithm from an implementation-oriented perspective, describing its core ideas and data structures, and evaluate its practical performance through empirical benchmarks.

2 Preliminaries

A directed graph $G = (V, E)$ is defined as having $n = |V|$ vertices and $m = |E|$ edges, with a non-negative weight function $w : E \to \mathbb{R}_{\geq 0}$, where w_{uv} denotes the weight of the edge $(u, v) \in E$. Let $s \in V$ be a source vertex; the goal of SSSP algorithms is to compute the shortest-path distances from s to each vertex $v \in V$.

2.1 Assumptions

For their algorithm BMSSP, Duan et al. require the graph G to have the following properties:

1. Every vertex in G is reachable from the source s. This can be assumed without loss of generality, as unreachable vertices have infinite distances to s and are therefore irrelevant for the SSSP. Such vertices can be identified and removed in $\mathcal{O}(n+m)$ time using a simple depth-first search procedure, which does not affect the asymptotic running time of the algorithm.

2. Constant-Degree Graph: BMSSP requires the vertices in G to have constant in-degrees and out-degrees. This constraint is also w.l.o.g., as any graph G can be transformed to a graph G', such that distances are preserved and the number of vertices and edges in G' only grows linearly [Fre83].

3. Comparison-Addition Model: The BMSSP algorithm operates under the comparison-addition model, as does Dijkstra's. This means that only comparisons and additions between edge weights are allowed. In this model, both operations are assumed to take unit time and other operations are not permitted.

4. Total order of path lengths: For the sake of simplicity, all paths considered in the algorithm are assumed to have different lengths. This restriction also comes without loss of generality, as paths can be extended by a lexicographic order, retaining the constant time complexity of comparisons between vertices with the same tentative distance.

2.2 Classical Approaches

In essence, two commonly used algorithms are employed to solve the single-source shortest path problem. Modified versions of these algorithms are used in the BMSSP algorithm:

Dijkstra's algorithm uses a priority queue to extract the vertex u with the minimal distance to the source s to relax the outgoing edges from u for each of the n vertices. The outcome of this sorting process is that the running time is at least $\Theta(n \log n)$ [Dij59].

The Bellman-Ford algorithm relaxes all m edges for repeatedly for $n-1$ iterations. It therefore has a time complexity of $\mathcal{O}(nm)$ [Bel58]. Limiting it for paths of length at most k, this results in a running time of $\mathcal{O}(km)$ [Dua+25].

2.3 Labels and Concepts

Labels As with Dijkstra's algorithm, the BMSSP algorithm maintains a global tentative distance value $\hat{d}[v]$ for each $v \in V$ as an upper bound for the actual distance from s to v, denoted as $d(v)$. Throughout the execution of BMSSP $\hat{d}[v]$ is updated through the relaxation of edges in a monotone non-increasing manner until $\hat{d}[v] = d(v)$. A vertex is called *complete*, if its tentative distance matches its actual distance, otherwise it's called *incomplete*.

Frontier and Sorting Barrier During the execution of Dijkstra's algorithm, a dynamic set of vertices, the so called *frontier* S, is processed through the priority queue. It consists of all discovered, yet not necessarily complete vertices.

For each incomplete vertex u, the path from s to u, has to at least contain one complete vertex in S. The algorithm picks the vertex in S, that is closest to the source and relaxes its outgoing edges.
The central running time bottleneck results from the fact that, in the worst case, the frontier contains $\Theta(n)$ vertices and that in each step, a total ordering of these vertices is required. This implicit sorting requirement, the so called *sorting barrier*, leads to a lower bound for the time complexity of $\Omega(n \log n)$.

The core idea of the BMSSP algorithm is to circumvent this barrier by relinquishing this dynamic intractable frontier. Instead, they use a divide and conquer approach, where the vertex set V is subdivided into disjoint sets, re-expressing the frontier as a set of source vertices S on these subsets. The frontier becomes static and admits a pivot-pruning subroutine, required to improve on the running time of $O(n \log n)$. The details of this approach are described in the following section.

3 Bounded Multi-Source Shortest Path

3.1 Overview and Recursive Structure

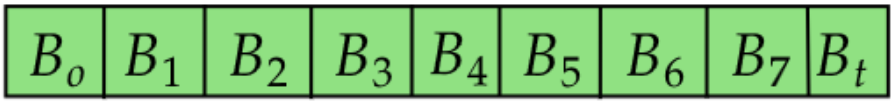

Figure 1: Abstract view of a graph in one dimension as a partition of its vertices in bounds of their distances to the source s

We first consider the SSSP problem in abstract terms as a one-dimensional ordering problem based on the actual distances of the nodes from the start node. Let $t \coloneqq \lfloor \log^{2/3} n \rfloor$ be a parameter of BMSSP. Suppose, we can find suitable bounds $B_0 < B_1 < \ldots < B_t$, such that the vertices in U are evenly distributed to these intervals as displayed in Figure 1. To be precise, a vertex set U is divided into 2^t vertex sets $U = U_1 \cup U_2 \cup \ldots \cup U_{2^t}$, such that $U_i \coloneqq \{v \in V | B_{i-1} \leq d(v) < B_i\}$. Each subset is recursively partitioned until it shrinks to a single element after around $l \coloneqq (\log n)/t$ recursion levels. This establishes a partial ordering of vertices into distance classes as opposed to Dijkstra's global ordering.

To calculate the actual distances for vertices v in U_i with distances $d(v) \in [B_{i-1}, B_i)$, it is necessary to determine the set of vertices that shortest paths to vertices in U_i pass through. Assume that all vertices with distance less than B_{i-1} are already complete and that their outgoing edges have been relaxed.

Consider now the set of vertices whose current tentative distances satisfy $B_{i-1} \leq \hat{d}[v] < B_i$. This set forms a local *frontier* S. For any incomplete vertex v' with $d(v') \in [B_{i-1}, B_i)$, the shortest path from the source to v' must pass through some complete vertex in S.

Therefore, in order to compute the true distances of all vertices with $d(v) \in [B_{i-1}, B_i)$, it suffices to compute shortest paths starting from the vertices in S, considering only paths of total length less than B_i. This subproblem is called

the *bounded multi-source shortest path* problem (BMSSP) and is summarized by the following lemma.

Lemma 1 (Bounded Multi-Source Shortest Path) *Given a directed graph, a source vertex s, a set of vertices S, and an upper bound B. Suppose that all vertices u with $d(u) < b$ are complete, and that for every incomplete vertex v with $b \leq d(v) < B$, there exists a shortest path from s to v that passes through at least one vertex in S.*

Then there exists an algorithm that computes a bound $B' \leq B$ and a set of vertices U such that: U contains exactly all vertices v with $d(v) < B'$ whose shortest paths pass through S, and all vertices in U are completely settled after execution.

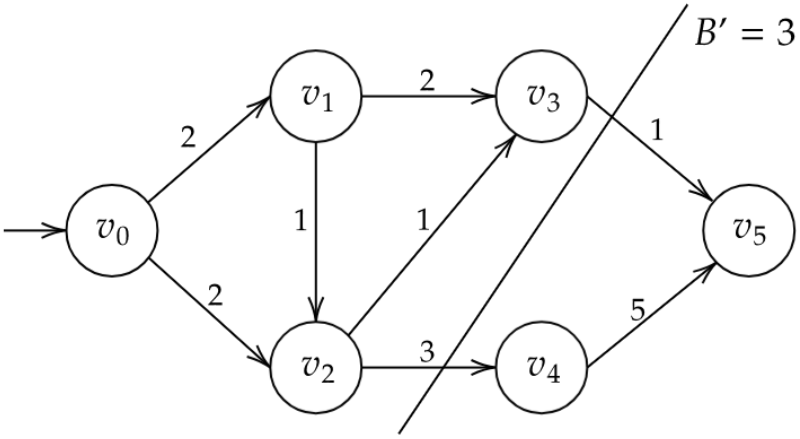

Figure 2: Graph divided into two parts depending on the distances $d(v)$ with regard to a bound B'

The example in Figure 2 illustrates this divide-and-conquer approach. We divide the subgraph into two subproblems: the first contains vertices with $\hat{d}[x] < B'$, and the second contains vertices with $B' \leq \hat{d}[x] < B$. The first subproblem is equivalent to the BMSSP problem with source set $\{x \in S \mid \hat{d}[x] < B'\}$. The source set for the second subproblem is the union of the set of vertices with $B' \leq \hat{d}[x] < B$ and the set of out-neighbors of the vertices computed in the first subproblem.

However, so far we have not improved the running time barrier set by Dijkstra: In the worst case, the source set S contains all the vertices. In a t-way divide-and-conquer approach, we have to decide for each subproblem, for each vertex in its source set, which sub-problem it falls into. Even with binary search this takes $\log(t)$ time. Assuming that the time complexity of each vertex is $\log(t)$ per level, and that a t-way divide-and-conquer algorithm has $\frac{\log(n)}{\log(t)}$ levels, it follows that the total time spent by the algorithm on each vertex is still $\log(n)$, resulting in a lower bound for the overall time complexity of $\Omega(n \log n)$. Therefore, it is necessary to prune the frontier S. This topic will be addressed in the following section.

3.2 Finding Pivots

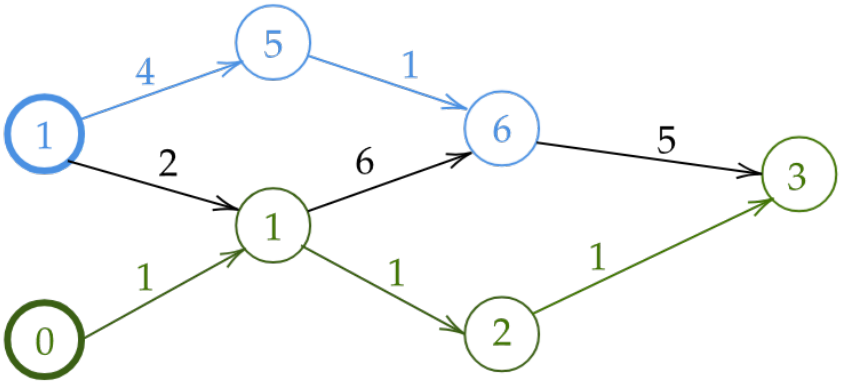

Figure 3: Frontier vertices have thick markers. Displayed are the trees extending from frontier vertices for $k = 3$ and $B' = 6$

The goal of the pivot selection step is to limit the amount of recursive work performed by the BMSSP subroutine. At this stage of the algorithm, vertices whose real distances are smaller than B_{i-1}, have already been finalized and their outgoing edges relaxed. Let S denote the current frontier, consisting of vertices x with $B_{i-1} \leq \hat{d}[x] < B_i$. Essentially, we want to identify a small set of vertices $P \subset S$, called *pivots*, such that every remaining vertex whose distance has not yet been finalized can be reached via at least one of these pivots.
Figure 3 illustrates the effect of the pivot selection: shallow shortest-path trees (blue) are fully resolved locally within k iterations, while only the roots of large trees are retained for further recursive processing.

Lemma 2 (Find Pivots) *For any k, there exists an algorithm that operates in $O(\min\{k^2|S|, k|U_i|\})$ time and identifies a set of pivots $P \subset S$ of size at most $|S|/k$. Furthermore, for any vertex x with $d(x) < B$, if the shortest path to x originates from a non-pivot vertex $s \in S \setminus P$, then the algorithm also computes $d(x)$. Otherwise, the shortest path to x passes through at least one vertex $y \in P$.*

The algorithm (Algorithm 1) simultaneously relaxes vertices in S for at most k steps (lines 7–13). This is done via a truncated Bellman-Ford procedure that terminates when paths exceed the bound B_i (line 18). After k relaxation steps, every vertex u with at most k intermediate vertices and $d(u) \in [B_{i-1}, B_i)$, is guaranteed to be complete. These vertices are collected into the set W (line 17). If this is the case for all frontier vertices, the algorithm terminates and returns S as the pivot set and W as a set of complete vertices reachable from S (lines 22–23).
The remaining incomplete vertices must be located in shortest-path trees that originate at vertices in S and contain at least k other vertices (lines 27–30). These trees are vertex-disjoint, and they only cover vertices with distances less than B_i. Therefore, there can be at most $|U_i|/k$ large trees. As a consequence, only the roots of these trees need to be considered in recursive calls. The set of roots forms the pivot set, P.

Algorithm 1 FindPivots(B, S)

Require: Directed graph $G = (V, E)$, parameter $k \geq 1$, distance bound B
Require: Frontier set $S = \{(v, \hat{d}[v])\}$ with $\hat{d}[v] < B$
Ensure: Sets (P, W) such that every vertex x with $d(x) < B$ is either complete in W or depends on some pivot in P

$W \leftarrow S, \quad W_{\text{prev}} \leftarrow S$
initialize arrays root$[v] \leftarrow \perp$, treeSz$[v] \leftarrow 0$
initialize array visited$[v] \leftarrow$ false
for all $(u, \hat{d}[u]) \in S$ **do**
    root$[u] \leftarrow u$
    visited$[u] \leftarrow$ true
for $i \leftarrow 1$ to k **do**
    $W_i \leftarrow \emptyset$
    for all $(u, \hat{d}[u]) \in W_{\text{prev}}$ **do**
        for all $(u, v, w_{uv}) \in E$ **do**
            cand $\leftarrow \hat{d}[u] + w_{uv}$
            if cand $< B$ **and** cand $\leq \hat{d}[v]$ **then**
                $\hat{d}[v] \leftarrow$ cand
                root$[v] \leftarrow$ root$[u]$
                if not visited$[v]$ **then**
                    visited$[v] \leftarrow$ true
                    $W_i \leftarrow W_i \cup \{(v, \hat{d}[v])\}$
    if $W_i = \emptyset$ **then**
        break
    $W \leftarrow W \cup W_i$
    $W_{\text{prev}} \leftarrow W_i$
    if $|W| > k \cdot |S|$ **then**
        return (S, W) ▷ All frontier vertices become pivots
for all $(v, \_) \in W$ **do**
    treeSz[root$[v]$] $\leftarrow$ treeSz[root$[v]$] $+ 1$
$P \leftarrow \emptyset$
for all $(u, \hat{d}[u]) \in S$ **do**
    if treeSz$[u] \geq k$ **then**
        $P \leftarrow P \cup \{(u, \hat{d}[u])\}$
return (P, W)

In contrast to the original description in [Dua+25], the directed forest induced by the Bellman-Ford iterations in lines 10–14 does not need to be explicitly constructed. It suffices to maintain an array of root nodes and their corresponding tree sizes, such that whenever an edge (u, v) is relaxed, the root of v is set to the root of u (line 14). This shortcut avoids the overhead of traversing the forest, while maintaining the correctness of the pivot selection, as only the roots of trees larger than k are of interest [CCF25].

Analysis During the Bellman-Ford procedure, either all vertices in U are discovered or only those with a hop distance of at most k from each vertex in S. Therefore the size of W is bounded by $O(\min\{k|S|, |U|\})$. The for-loop in line 7 runs k times. Given that each iteration processes at most the vertices in W, each iteration takes $O(|W|) = O(\min\{k|S|, |U|\})$ time. Consequently, the overall time complexity for Algorithm 1 is $O(\min\{k^2|S|, k|U|\})$. The selection of the parameter k results in two opposing effects. A reduced value k decreases the workload during the pivot finding procedure. In contrast, an increase in the parameter k results in a more aggressive reduction of the frontier S, our main bottleneck, by a factor of at least $1/k$. This, in turn, reduces the number of recursive subproblems generated per recursion level. Given that BMSSP works on constant degree graphs with $O(m) = O(n)$, the workload per recursion level is bounded by $O(km)$, dominated by the Algorithm 1 algorithm. Recall from Section 3 our parameters $t = \lfloor \log^{2/3} n \rfloor$ and $l = (\log n)/t$. If we set $k \coloneqq \lfloor \log^{1/3} n \rfloor$, the total running time of BMSSP is bounded by

$$O(l \cdot km) = O\left(m \frac{\log n}{\log^{2/3} n} \log^{1/3} n \right) = O(m \log^{2/3} n)$$

.

To meet the stated running time bounds, a special data structure is required to support the efficient partitioning and extraction of blocks of vertices separated by their respective distance bounds.

3.3 Block-Based Linked List

The data structure $\mathcal{D}$ supports a small set of primitive operations on key/value pairs $\langle a, b \rangle$, each tailored to the access patterns of the algorithm. Overall N key value pairs are contained by $\mathcal{D}$ in the form of blocks, which are linked lists with upper bounds B and size of at most M. A key/value pair $\langle a, b \rangle$ is stored in the block whose upper bound is the smallest value B satisfying $b \leq B$. The blocks are stored in containers $\mathcal{D}_0$ and $\mathcal{D}_1$. The number of blocks in $\mathcal{D}_1$ is bounded by $O(\max\{1, N/M\})$. While the blocks are ordered according to their upper bounds within their containers, there is no such requirement for the elements within the blocks. Blocks in $\mathcal{D}_1$ are referenced through a balanced binary search tree (a Red-black tree [GS78] for example) by their upper bounds. The nodes of the binary search tree store pointers to the blocks corresponding to their respective bounds. Additionally, $\mathcal{D}$ stores a global upper bound for all key/value pairs. For the purposes of the BMSSP algorithm, we define the key/value pairs as vertex/tentative-distance pairs denoted as $\langle v, \hat{d}[v] \rangle$. Our implementation employs additional auxiliary data-structures to facilitate amortized constant

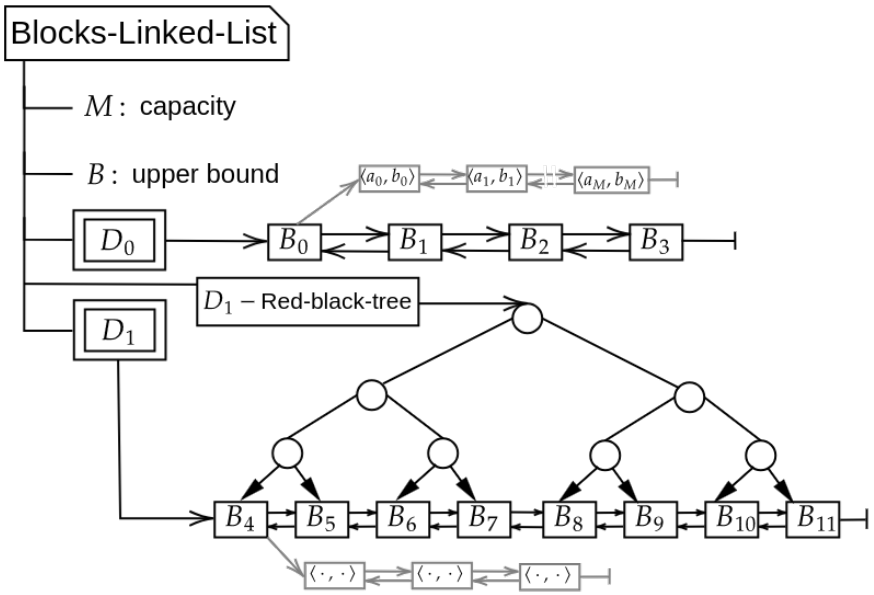

Figure 4: Overview of the members of the Blocks-Linked-List

block-membership tests and access for pairs. These implementation-specific details do not affect the asymptotic guarantees of the data structure and are therefore deferred to Section 4.1.

Initialize(M, B) $O(1)$**:** Sets the global upper bound of $\mathcal{D}$ to B, initializes $\mathcal{D}_\infty$ with an empty block with upper bound B and adds its pointer to the D_1-tree.

Delete(a, b) $O(\max\{1, \log(N/M)\})$ ***amortized*:** The original paper states a constant time for deletion, implying that for each pair a direct pointer is stored to its corresponding node within its linked-list. If a block becomes empty after deletion, its reference must be removed from the $\mathcal{D}_1$-tree which can be done in $O(\max\{1, \log N/M\})$ time.

Insert(a, b) $O(\max\{1, \log(N/M)\})$ ***amortized*:** Elements from insert operations are exclusively stored in $\mathcal{D}_1$. Firstly, we check if a pair with key a already exists in $\mathcal{D}_1$ with our `present`-array in unit time. If it already exists, we delete the old pair $\langle a, b' \rangle$ only if $b < b'$. If it does not exist, we locate the block with the lowest bound $B > b$. If such a block exists the pair is inserted to the end of its list in $O(1)$. If necessary, the upper bound of that block is updated to b. If the block does not exist, a new block with upper bound b is appended to $\mathcal{D}_1$ and its reference is stored in the $\mathcal{D}_1$-tree. The time complexity of an insertion is bounded by the find and add operations on the $\mathcal{D}_1$ and therefore takes $O(\max\{1, \log N/M\})$ time. If the size of a block exceeds M after an insertion, a split operation must be performed.

Split() $O(M)$**:** When the size of a block in $\mathcal{D}_1$ exceeds M elements, it is split into two new blocks. To this end, the median value B_{mid} of its elements is determined in $O(M)$ time [Blu+73]. Elements with values values $b \leq B_{mid}$ are saved in the lower block and pairs with $b > B_{mid}$ are saved in the upper blocks. Both blocks are added to $\mathcal{D}_1$ and the $\mathcal{D}_1$-tree is updated accordingly.

BatchPrepend($\mathcal{L}$) $O(|\mathcal{L}| \log(|\mathcal{L}|/M))$**:** If $|\mathcal{L}| \leq M$, a block is created for its elements and added to the beginning of $\mathcal{D}_0$. Otherwise, L/M new blocks are inserted at the start of $\mathcal{D}_0$ by recursively partitioning $\mathcal{L}$ at its median value such that each block contains at most $\lceil M/2 \rceil$ elements. This takes $O(L \log(L/M))$ time.

Pull() $O(M)$ ***amortized*****:** The pull operation extracts the M smallest elements from $\mathcal{D}_0 \cup \mathcal{D}_1$ starting with elements from $\mathcal{D}_0$ and additionally returns an upper bound x of these elements. If M is greater than the number of elements in $\mathcal{D}$, x is set to the global bound B. Otherwise x is set to the lowest bound of the remaining blocks in $\mathcal{D}$, which takes $O(M)$ time. The deletion of M elements also takes amortized $O(M)$ time.

3.4 Base Case of BMSSP

As outlined in Section 3.1, BMSSP follows a recursive divide-and-conquer strategy that progressively reduces the size of the active frontier. Once the recursion depth reaches $l = 0$, the frontier S consists of a single complete vertex x. In this situation, the bounded multi-source shortest path problem degenerates to a bounded single-source shortest path problem. The goal of the base case is therefore to explore the local neighborhood of x up to the current distance bound B and to identify at most k vertices with smallest distances.

This is achieved using a truncated variant of Dijkstra's algorithm, which terminates either when the priority queue is exhausted or after $k + 1$ vertices have been settled.

Algorithm 2 BaseCase-BMSSP(B, S)

Require: Directed graph $G = (V, E)$, parameter $k \geq 1$, distance bound B
Require: Frontier $S = \{(x, d(x)\}$ with and $d(x) < B$ and x is complete
Ensure: Updated bound B' and set U of at most k vertices with $d(v) < B'$

1: Initialize priority queue $\mathcal{H}$
2: Insert $(x, d(x))$ into $\mathcal{H}$
3: $U \leftarrow \emptyset$
4: **while** $H \neq \emptyset$ **and** $|U| < k + 1$ **do**
5: Extract $(u, \hat{d}[u])$ with minimum key from $\mathcal{H}$
6: **if** $\hat{d}[u] > \text{dist}[u]$ **then**
7: **continue**
8: Add $(u, \hat{d}[u])$ to U
9: **for all** $(u, v, w_{uv}) \in E$ **do**
10: cand $\leftarrow \hat{d}[u] + w_{uv}$
11: **if** cand $< B$ **and** cand $\leq \text{dist}[v]$ **then**
12: dist$[v] \leftarrow$ cand
13: Insert (v, cand) into $\mathcal{H}$
14: **if** $|U| \leq k$ **then**
15: **return** (B, U)
16: $B' \leftarrow \max_{v \in U} \hat{d}[v]$
17: Remove vertex attaining B' from U
18: **return** (B', U)

Algorithm 2 settles the distances of at most $k+1$ vertices in the neighborhood of its source, whose actual distances are smaller than B, and collects them in U (lines 4–13). If at most k vertices are collected, U is returned with the unchanged bound B. Otherwise, the distance $d(v)$ of the $(k+1)$-st vertex v defines a new bound B', which is returned along with $U \setminus \{v\}$ (lines 16–18). By construction, all vertices in U are complete. In higher recursion levels, these sets U_i serve as the units by which completeness is propagated.

3.5 General Case of BMSSP

Algorithm Algorithm 3 describes the recursive BMSSP procedure. Given a frontier set S and a distance bound B, it computes a set of newly complete vertices with distances below a refined bound, while maintaining the invariants required for deeper recursion levels. To match the requirements of SSSP algorithms, BMSSP is initially invoked with parameters $l = \lceil (\log n)/t \rceil$, $S = \{s\}$, and $B = \infty$.

Algorithm 3 BMSSP(l, B, S)

Require: Directed graph $G = (V, E)$, recursion level l, distance bound B
Require: Frontier set $S = \{(v, \hat{d}[v])\}$ with $\hat{d}[v] < B$
Ensure: Updated bound B' and a set U of complete vertices with $d(v) < B'$

if $l = 0$ **then**
  return BaseCase$(S[0], B)$
$(P, W) \leftarrow$ FindPivots(B, S)
$M \leftarrow 2^{(l-1)t}$
Initialize data structure $\mathcal{D}$ with block size M and bound B
$B' \leftarrow B$
for all $(x, \hat{d}[x]) \in P$ **do**
  Insert $(x, \hat{d}[x])$ into $\mathcal{D}$
  $B' \leftarrow \min(B', \hat{d}[x])$
$U \leftarrow \emptyset, \quad \text{cap} \leftarrow 2^{lt}$
while $|U| < \text{cap}$ **and** $\mathcal{D}$ not empty **do**
  $(S_i, B_i) \leftarrow$ Pull$(\mathcal{D})$
  if $S_i = \emptyset$ **then**
    break
  $(B_i', U_i) \leftarrow \text{BMSSP}(l-1, B_i, S_i)$
  $U \leftarrow U \cup U_i$
  $K \leftarrow \emptyset$
  for all $(u, d[u]) \in U_i$ **do**
    Mark u as complete
    Erase$(\mathcal{D}, u)$
    for all $(u, v, w_{uv}) \in E$ **do**
      $\text{cand} \leftarrow d[u] + w_{uv}$
      if $\text{cand} \le \hat{d}[v]$ **then**
        $\hat{d}[v] \leftarrow \text{cand}$
        if $\text{cand} \in [B_i, B)$ **then**
          Insert (v, cand) into $\mathcal{D}$
        else if $\text{cand} \in [B_i', B_i)$ **then**
          Add (v, cand) to K
  for all $(x, \hat{d}[x]) \in S_i$ **do**
    if $\hat{d}[x] \in [B_i', B_i)$ **then**
      Add $(x, \hat{d}[x])$ to K
  BatchPrepend$(\mathcal{D}, K, B_i')$
  $B' \leftarrow B_i'$
if $\mathcal{D}$ is empty **then**
  $B_{\text{res}} \leftarrow B$
else
  $B_{\text{res}} \leftarrow B'$
for all $x \in W$ **do**
  if x is not complete **and** $\hat{d}[x] < B_{\text{res}}$ **then**
    Mark x as complete
    Add $(x, \hat{d}[x])$ to U
return (B_{res}, U)

For $l > 0$ the current frontier S is reduced to a pivot-set $P \subset S$ and a set W of vertices reached within at most k relaxation steps from S, whose tentative distances have been refined but are not necessarily final. The povots serve as an entry point for recursive calls and guarantee that every incomplete vertex v with $d(v) < B$ can be reached via at least one of the pivots. Subsequently, the block-linked-list $\mathcal{D}$ is initialized with $M := 2^{(l-1)t}$ and the pivot vertices $x \in P$ are inserted as $\langle x, \hat{d}[x] \rangle$ pairs. $\mathcal{D}$ organizes vertices in an ascending order of their tentative distances and enables us to iteratively extract subsets S_i with similar distance bounds. In every iteration i of the while-loop, BMSSP is called recursively on the subset S_i. The return value consists of a refined distance bound B'_i, as well as a set U_i of newly settled vertices. For each newly settled vertex $u \in U_i$, all outgoing edges (u, v) are relaxed. If a relaxation is successful, i.e.,

$$\hat{d}[v] \leftarrow \min\{\hat{d}[v], \hat{d}[u] + w_{uv}\},$$

the vertex v is handled as follows:

- If $\hat{d}[v] \in [B'_i, B_i)$, the pair $\langle v, \hat{d}[v] \rangle$ is stored in a temporary set K.
- If $\hat{d}[v] \in [B_i, B)$, the pair $\langle v, \hat{d}[v] \rangle$ is inserted directly into $\mathcal{D}$.

Thereafter, vertices in $x \in S_i$ satisfying $\hat{d}[x] \in [B'_i, B_i)$ are batch-prepended along with vertices in K. The while-loop terminates under one of the following conditions:

1. $\mathcal{D}$ becomes empty, indicating a successful execution. In this case, $B' \leftarrow B$.
2. $|U| > k \cdot 2^{lt}$, indicating excessive workload. The execution is stopped prematurely and $B' \leftarrow B'_i$.

Finally, all vertices $x \in W$ with $\hat{d}[x] < B'$, that have not yet been marked complete at level l, are added to U and returned with B'. In the event of a successful execution, these vertices coincide with the entire set W.

4 Implementation and Benchmarks

We have created a faithful implementation of the BMSSP algorithm along with the block-based linked list $\mathcal{D}$ in C++ for the purpose of benchmarking [Cha25]. The following section provides an overview of the aspects where our implementation deviates from, or makes concrete choices beyond, the original paper.

4.1 Implementation Details

Assumptions The fourth assumption from Section 2.1 states that all paths inside the graph have different lengths. To satisfy this assumption in practice, we add a small random value in the range of $[10^{-8}, 10^{-4}]$ to each edge weight in the graph's initialization stage. This ensures, that no computational overhead is created to settle tie-breaks between distances, while also maintaining fairness when comparing the running times with Dijkstra as both algorithms operate on the same graph. While this measure suffices in practice, we additionally use the vertices' unique IDs as a deterministic tie-breaker if required.

Find Pivots In the original paper, a forest of shortest path trees rooted in vertices from S is explicitly constructed during the relaxation process. Instead, we use two local arrays `pivot_roots` and `pivot_tree_sizes`, which are updated as described in Algorithm 1 [CCF25].

Block-Based Linked-List In our implementation, vertices are assigned a fixed lexicographic order $(v_0, \ldots, v_n)$ during graph construction and their presence in $\mathcal{D}$ can be tracked via a boolean array `present[`v`]` in unit time. Furthermore, we use dynamic arrays instead of linked-lists for the blocks due to their better cache-locality. Deletions can still be managed in constant time by swapping with the last element and deleting the last element [cpp25] and insertions take amortized $O(1)$ time. To locate a key's block and its position inside its block in amortized $O(1)$ time, we use a Hash-map of `(block_id, element_id)` pairs for each container $\mathcal{D}_0$ and $\mathcal{D}_1$.

BMSSP Overall, our implementation follows Algorithm 3 closely. However, instead of sets like Hash-sets we take advantage of the lexicographic order we created during the graph's creation and use array-backed structures, guaranteeing access in $O(1)$ time instead of merely amortized constant time. Additionally, settled vertices are immediately erased from $\mathcal{D}$ (line 20), an operation that is not explicitly mentioned in the original paper's description.

4.2 Benchmarks

All benchmarks in this work were run on a 64-bit EndeavourOS Linux operating system (version 2024.06.25) with an AMD Ryzen 5 5500 processor and 31.1 GiB RAM. `C++` version 20 with the compiler `(GCC) 15.2.1 20260103` was used to implement the algorithms and Google Benchmarks was used for time measurement.

4.2.1 Setup

Reference For comparison, two versions of Dijkstra's algorithm were implemented. The first one uses Fibonacci heaps. It is well known that Fibonacci heaps, while optimal in theory, often perform poorly in practice due to large constant factors [CGR96]. To account for this, we additionally evaluate a variant using a binary heap based on `std::priority_queue`.

Dataset Synthetic randomized graphs are created via a python script. Graphs consists of n vertices in the range of $[2^7, \ldots, 2^{23}]$. They are constant-degree directed graphs with out-degrees ranging from $d \in [2^1, \ldots, 2^7]$. For each graph, 10 source vertices are randomly selected with a fixed seed. Each benchmark configuration is executed 10 times, meaning the same graph instance is executed 100 times for each algorithm. The measured time in microseconds comprises the construction of the algorithm instance and the complete execution of the SSSP algorithm returning the distances from s to each vertex. The program is compiled using aggressive optimization via the flag
`-O3`, however results are shielded from optimization using Google benchmark's commands `benchmark::DoNotOptimize` and `benchmark::ClobberMemory`.

4.2.2 Results

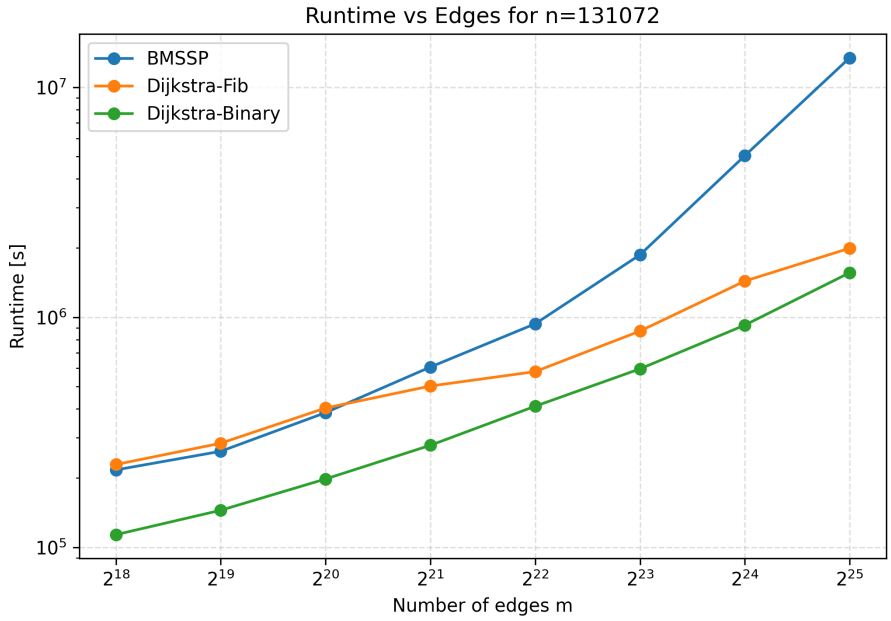

Figure 5: Running time comparison of BMSSP, Dijkstra with Fibonacci Heaps and Dijkstra with a binary heaps for $n = 131072$ vertices and out-degrees d ranging from 2^1 to 2^7

Fixed vertices n, variable out-degrees d As demonstrated in Figure 1, the BMSSP's execution time displays a substantial increment in proportion to the increase in edge counts. However, it is noteworthy that for m in the range 2^{18} to 2^{20}, i.e., out-degrees between 2 and 8, BMSSP managed to outperform Dijkstra's implementation with a Fibonacci heap. While the running times of both Dijkstra variants increase relatively moderately, BMSSP's running time rises very strongly after 2^6 edges and dominates here. This is expected as Dijkstra with its time complexity of $O(m+n \log n)$ only has an additive dependence on the number of edges while BMSSP with its time complexity of $O(m \log^{2/3} n)$ is expected to increase linearly with increasing m.

Variable vertices n, fixed out-degrees d As demonstrated in Figure 6 and Figure 7, the substantial constant factors of BMSSP dominate in the lower vertex count range, specifically between 2^7 and 2^{10} vertices. However, as n increases, the difference between the BMSSP and Dijkstra algorithms decreases, an effect that can be expected due to the BMSSP's reduced time complexity. For increasing n, the constant factors become less significant. Dijkstra's algorithm, implemented with binary heaps, has been demonstrated to be the most efficient of the three algorithms. This efficiency can be attributed to the use of a flat array as the underlying structure, which minimizes overhead. The algorithm's ability to operate in-place is also a contributing factor to its effectiveness.

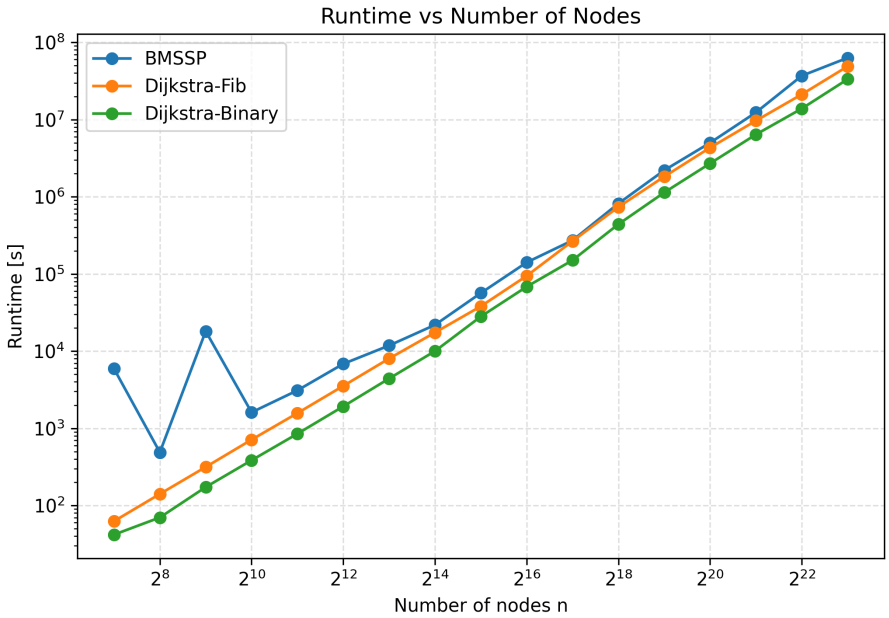

Figure 6: Running time comparison of BMSSP, Dijkstra with Fibonacci Heaps and Dijkstra with a binary heaps for a variable number of vertices n and fixed $d = 4$ out-degrees

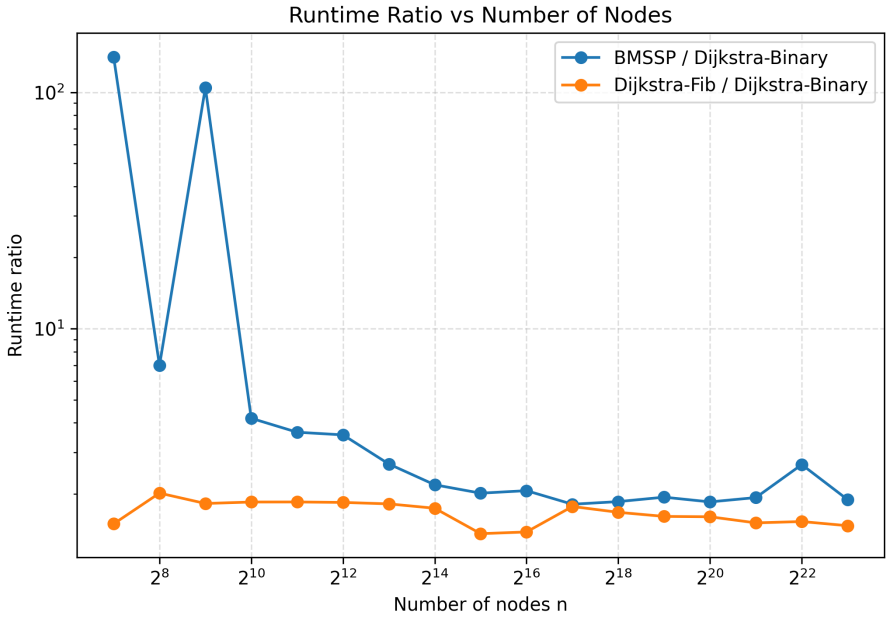

Figure 7: Ratio of running times of Dijkstra with a Fibonacci Heap and BMSSP to the running times of Binary Heap-Dijkstra for $d = 4$ out-degrees and a variable number of vertices n.

5 Conclusion and Future Work

The results in Section 4.2 clearly indicate that, in practice, a faithful implementation of BMSSP has too much overhead to outperform Dijkstra's algorithm, even for very sparse graphs. This discrepancy is especially evident in the context of dense graphs. Nonetheless, certain instances of low out-degrees have show superior performance in comparison to the Dijkstra algorithm. Additionally, as the number of vertices increases, the difference in run-time becomes progressively smaller. Future work may examine the refinement of the underlying block-based data structure D, or the parallelization of the processing of vertex subsets, U_i. Given that the theoretically best implementation of Dijkstra, namely the one using Fibonacci heaps, is consistently slower than one with (STL) binary heaps, a simplified or refined data-structure for BMSSP probably has the potential to significantly improve its practical viability.

References

[Bel58] Richard Bellman. "On a routing problem". In: *Quarterly of applied mathematics* 16.1 (1958), pp. 87–90.

[Blu+73] Manuel Blum et al. "Time bounds for selection". In: *J. Comput. Syst. Sci.* 7.4 (1973), pp. 448–461.

[CCF25] Lucas Castro, Thailsson Clementino, and Rosiane de Freitas. *Implementation and Brief Experimental Analysis of the Duan et al. (2025) Algorithm for Single-Source Shortest Paths*. 2025. arXiv: `2511.03007 [cs.DS]`. URL: `https://arxiv.org/abs/2511.03007`.

[CGR96] Boris V Cherkassky, Andrew V Goldberg, and Tomasz Radzik. "Shortest paths algorithms: Theory and experimental evaluation". In: *Mathematical programming* 73.2 (1996), pp. 129–174.

[Cha25] Omar Chatila. *Experimental Implementation and Evaluation of BMSSP*. 2025. URL: `https://github.com/Omar-Chatila/BMSSP-cpp` (visited on 01/31/2026).

[cpp25] cppreference.com. *std::vector::pop_back*. Accessed via cppreference. 2025. URL: `https://en.cppreference.com/w/cpp/container/vector/pop_back` (visited on 01/31/2026).

[Dij59] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[Dua+25] Ran Duan et al. *Breaking the Sorting Barrier for Directed Single-Source Shortest Paths*. 2025. arXiv: `2504.17033 [cs.DS]`. URL: `https://arxiv.org/abs/2504.17033`.

[Fre83] Greg N Frederickson. "Data structures for on-line updating of minimum spanning trees". In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 1983, pp. 252–257.

[FT87] Michael L Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[GS78] Leo J Guibas and Robert Sedgewick. “A dichromatic framework for balanced trees”. In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE. 1978, pp. 8–21.

[Hae+24] Bernhard Haeupler et al. “Universal optimality of dijkstra via beyond-worst-case heaps”. In: *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2024, pp. 2099–2130.

A Affidavit

I hereby declare that I have completed this work independently and without any unauthorized assistance from third parties, that I have not used any aids other than those mentioned and explicitly permitted, and that I have generally conducted myself in a manner consistent with the examination requirements. I also declare that when using IT/AI-supported writing tools, I have listed these tools in full in the overview of aids used, including their product name, my source of reference, and ... and/or have marked the relevant passages in the paper as having been written with AI-generated support. I am aware that deception or attempts at deception will be punished in accordance with the examination regulations applicable to me.