Breaking the Sorting Barrier for Directed Single-Source Shortest Paths

Ran Duan
Tsinghua University
Beijing, China
duanran@mail.tsinghua.edu.cn

Jiayi Mao
Tsinghua University
Beijing, China
mjy22@mails.tsinghua.edu.cn

Xiao Mao
Stanford University
Stanford, USA
matthew99a@gmail.com

Xinkai Shu
Max Planck Institute for Informatics
Saarbrücken, Germany
xshu@mpi-inf.mpg.de

Longhui Yin
Tsinghua University
Beijing, China
ylh21@mails.tsinghua.edu.cn

Abstract

We give a deterministic $O(m \log^{2/3} n)$-time algorithm for single-source shortest paths (SSSP) on directed graphs with real non-negative edge weights in the comparison-addition model. This is the first result to break the $O(m + n \log n)$ time bound of Dijkstra's algorithm on sparse graphs, showing that Dijkstra's algorithm is not optimal for SSSP.

CCS Concepts

• **Theory of computation** → **Shortest paths**; *Data structures design and analysis.*

Keywords

Shortest paths, graph algorithms

ACM Reference Format:
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 (STOC '25), June 23–27, 2025, Prague, Czechia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3717823.3718179

1 Introduction

In an m-edge n-vertex directed graph $G = (V, E)$ with a non-negative weight function $w : E \to \mathbb{R}_{\geq 0}$, single-source shortest path (SSSP) considers the lengths of the shortest paths from a source vertex s to all $v \in V$. Designing faster algorithms for SSSP is one of the most fundamental problems in graph theory, with exciting improvements since the 50s.

The textbook Dijkstra's algorithm [7], combined with advanced data structures such as the Fibonacci heap [13] or the relaxed heap [8], solves SSSP in $O(m + n \log n)$ time. It works in the *comparison-addition* model, natural for real-number inputs, where only comparison and addition operations on edge weights are allowed, and each operation consumes unit time. For undirected graphs, Pettie and Ramachandran [22] proposed a hierarchy-based algorithm which runs in $O(m\alpha(m, n) + \min\{n \log n, n \log\log r\})$ time in the comparison-addition model, where α is the ~~inverse-Ackermann~~ function and r bounds the ratio between any two edge weights.

Dijkstra's algorithm also produces an ordering of vertices by distances from the source as a byproduct. A recent contribution by Haeupler, Hladík, Rozhoň, Tarjan and Tětek [18] showed that Dijkstra's algorithm is optimal if we require the algorithm to output the order of vertices by distances. If only the distances and not the ordering are required, a recent result by Duan, Mao, Shu and Yin [10] provided an $O(m\sqrt{\log n \log\log n})$-time randomized SSSP algorithm for undirected graphs, better than $O(n \log n)$ in sparse graphs. However it remains to break such a sorting barrier in directed graphs.

1.1 Our Results

In this paper, we present the first SSSP algorithm for directed real-weighted graphs that breaks the sorting bottleneck on sparse graphs.

Theorem 1.1. *There exists a deterministic algorithm that takes $O(m \log^{2/3}(n))$ time to solve the single-source shortest path problem on directed graphs with real non-negative edge weights.*

Note that the algorithm in [10] is randomized, thus our result is also the first deterministic algorithm to break such $O(m + n \log n)$ time bound even in undirected graphs.

Technical Overview. Broadly speaking, there are two traditional algorithms for solving the single-source shortest path problem:

- Dijkstra's algorithm [7]: via a priority queue, it each time extracts a vertex u with the minimum distance from the source, and from u relaxes its outgoing edges. It typically sorts vertices by their distances from the source, resulting in a time complexity of at least $\Theta(n \log n)$.
- Bellman-Ford algorithm [1]: based on dynamic programming, it ~~relaxes all edges~~ for several steps. For finding shortest paths with at most k edges, the Bellman-Ford algorithm can achieve this in $O(mk)$ time without requiring sorting.

Our approach merges these two methods through a recursive partitioning technique, similar to those used in bottleneck path algorithms as described in [5, 9, 16].

At any point during the execution of the Dijkstra's algorithm, the priority queue (heap) maintains a "frontier" S of vertices such that if a vertex u is "incomplete" — if the current distance estimate

This work is licensed under a Creative Commons Attribution 4.0 International License.
STOC '25, Prague, Czechia
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1510-5/25/06
https://doi.org/10.1145/3717823.3718179

$\hat{d}[u]$ is still greater than the true distance $d(u)$ — the shortest s-u path must visit some complete vertex $v \in S$. In this case, we say u is "dependent" on $v \in S$. (However, vertices in S are not guaranteed to be all complete.) The Dijkstra's algorithm simply picks the vertex in S closest to source, which must be complete, and then relaxes all edges from that vertex.

The running time bottleneck comes from the fact that sometimes the frontier may contain $\Theta(n)$ vertices. Since we constantly need to pick the vertex closest to source, we essentially need to maintain a total order between a large number of vertices, and are thus unable to break the $\Omega(n \log n)$ sorting barrier. Our most essential idea is a way to reduce the size of the frontier. Suppose we want to compute all distances that are less than some upper bound B. Let $\tilde{U}$ be the set of vertices u with $d(u) < B$ and the shortest s-u path visits a vertex in S. It is possible to limit the size of our frontier $|S|$ to $|\tilde{U}|/\log^{\Omega(1)}(n)$, or $1/\log^{\Omega(1)}(n)$ of the vertices of interest. Given a parameter $k = \log^{\Omega(1)}(n)$, there are two possible cases:

- If $|\tilde{U}| > k|S|$, then our frontier already has size $|\tilde{U}|/k$;
- Otherwise, suppose $|\tilde{U}| \le k|S|$. By running Bellman-Ford step k times from vertices in S, every vertex $u \in \tilde{U}$ whose shortest s-u path containing $< k$ vertices in $\tilde{U}$ is complete. Otherwise the vertex $v \in S$ which u is dependent on must have a shortest path tree rooted at it with $\ge k$ vertices in $\tilde{U}$, so we can shrink the frontier S to the set of "pivots", each of which has a shortest path tree of size $\ge k$, and the number of such pivots is bounded by $|\tilde{U}|/k$.

Our algorithm is based on the idea above, but instead of using a Dijkstra-like method where the frontier is dynamic and thus intractable, we use a divide-and-conquer procedure that consists of $\log n/t$ levels, each containing a set of frontier vertices and an upper bound B, such that a naive implementation would still spend $\Theta(t)$ time per frontier vertex and the running time would remain $\Theta(\log n)$ per vertex. We can however apply the aforementioned frontier reduction on each level so that the $\Theta(t)$ work only applies to the pivots, about $1/\log^{\Omega(1)}(n)$ of the frontier vertices. Thus the running time per vertex gets reduced to $\log n/\log^{\Omega(1)}(n)$, which is a significant speedup.

1.2 Related Works

For the SSSP problem, if we allow the algorithm to run in the word RAM model with integer edge weights, a sequence of improvements beyond $O(n \log n)$ [14, 15, 19, 23, 24, 26, 27] culminated in Thorup's linear-time algorithm for undirected graphs [25] and $O(m+n \log\log \min\{n, C\})$ time for directed graphs [28], where C is the maximum edge weight. For graphs with negative weights, recent breakthrough results include near-linear time $O(m^{1+o(1)} \log C)$ algorithms for SSSP with negative integer weights [2, 4, 6], and strongly subcubic time algorithm for negative real weights [11, 20].

Based on the lower bound of $\Omega(n \log n)$ for comparison-based sorting algorithms, it was generally believed that such *sorting barrier* exists for SSSP and many similar problems. Researchers have broken such a sorting barrier for many graph problems. For example, Yao [30] gave a minimum spanning tree (MST) algorithm with running time $O(m \log\log n)$, which is further improved to randomized linear time [21]. Gabow and Tarjan [16] solves s-t bottleneck path problem in $O(m \log^* n)$-time, later improved to randomized $O(m\beta(m, n))$ time [5], where $\beta(m, n) = \min\{k \ge 1 : \log^{(k)} n \le \frac{m}{n}\}$. For the single-source all-destination bottleneck path problem, there is a randomized $O(m\sqrt{\log n})$-time algorithm by Duan, Lyu, Wu and Xie [9]. For the single-source nondecreasing path problem, Virginia V.Williams [29] proposed an algorithm with a time bound of $O(m \log\log n)$.

2 Preliminaries

We consider a directed graph $G = (V, E)$ with a non-negative weight function $w : E \to \mathbb{R}_{\ge 0}$, also denoted by w_{uv} for each edge $(u, v) \in E$. Let $n = |V|$, $m = |E|$ be the number of vertices and edges in the graph. In the single-source shortest path problem, we assume there is a source vertex in the graph denoted by s. The goal of our algorithm is to find the length of shortest path from s to every vertex $v \in V$, denoted by $d(v)$. Without loss of generality, we assume that every vertex in G is reachable from s, so $m \ge n - 1$.

Constant-Degree Graph. In this paper we assume that the algorithm works on a graph with constant in-degrees and out-degrees. For any given graph G we may construct G' by a classical transformation (similar to [12]) to accomplish this:

- Substitute each vertex v with a cycle of vertices strongly connected with zero-weight edges. For every incoming or outgoing neighbor w of v, there is a vertex x_{vw} on this cycle.
- For every edge (u, v) in G, add a directed edge from vertex x_{uv} to x_{vu} with weight w_{uv}.

It's clear that the shortest path is preserved by the transformation. Each vertex in G' has in-degree and out-degree at most 2, while G' being a graph with $O(m)$ vertices and $O(m)$ edges.

Comparison-Addition Model. Our algorithm works under the *comparison-addition* model, where all edge weights are subject to only comparison and addition operations. In this model, each addition and comparison takes unit time, and no other computations on edge weights are allowed.

Labels Used in the Algorithm. For a vertex $v \in V$, denote $d(u)$ as the length of shortest path from s to u in the graph. Similar to the Dijkstra's algorithm, our algorithm maintains a global variable $\hat{d}[u]$ as a sound estimation of $d(u)$ (that is, $\hat{d}[u] \ge d(u)$). Initially $\hat{d}[s] = 0$ and $\hat{d}[v] = \infty$ for any other $v \in V$. Throughout the algorithm we only update $\hat{d}[v]$ in a non-increasing manner by relaxing an edge $(u, v) \in E$, that is, assign $\hat{d}[v] \leftarrow \hat{d}[u] + w_{uv}$ when $\hat{d}[u] + w_{uv}$ is no greater than the old value of $\hat{d}[v]$. Therefore each possible value of $\hat{d}[v]$ corresponds to a path from s to v. If $\hat{d}[x] = d(x)$, we say x is *complete*; otherwise, we say x is *incomplete*. If all vertices in a set S are complete, we say S is complete. Note that completeness is sensitive to the progress of the algorithm. The algorithm also maintains a shortest path tree according to current $\hat{d}[\cdot]$ by recording the predecessor $\textsc{Pred}[v]$ of each vertex v. When relaxing an edge (u, v), we set $\textsc{Pred}[v] \leftarrow u$.

Total order of Paths. Like in many papers on shortest path algorithms, we make the following assumption for an easier presentation of our algorithm:

Assumption 2.1. *All paths started from s have different lengths.*

This assumption is required for two purposes:

(1) To ensure that $\textsc{Pred}[v]$ for all vertices $v \in V$ keep a tree structure throughout the algorithm;
(2) To provide a relative order of vertices with the same value of $\hat{d}[]$.

Next, we show that this assumption does not lose generality since we can provide a total order for all paths. We treat each path of length l that traverses α vertices $v_1 = s, v_2, ..., v_\alpha$ as a tuple $\langle l, \alpha, v_\alpha, v_{\alpha-1}, ..., v_1\rangle$ (note that the sequence of vertices is reversed in the tuple). We sort all paths in the lexicographical order of the tuple in principle. That is, first compare the length l. When we have a tie, compare α. If there is still a tie, compare the sequence of vertices from v_α to v_1. Comparison between the tuples can be done in only $O(1)$ time with extra information of $\textsc{Pred}[]$ and α stored for each $\hat{d}[v]$:

Relaxing an edge (u, v): If $u \neq \textsc{Pred}[v]$, even if there is a tie in l and α, it suffices to compare between u and $\textsc{Pred}[v]$, and if $u = \textsc{Pred}[v]$, then $\hat{d}[u]$ is updated previously and $\hat{d}[v]$ needs to get updated;

Comparing two different $\hat{d}[u]$ and $\hat{d}[v]$ for $u \neq v$: Even if there is a tie in l and α, it suffices to compare the endpoints u and v only.

Therefore, in the following sections, we may assume that each path started from s has a unique length.

3 The Main Result

3.1 The Algorithm

Recall that we work on SSSP from source s in constant-degree graphs with $m = O(n)$. Let $k := \lfloor \log^{1/3}(n) \rfloor$ and $t := \lfloor \log^{2/3}(n) \rfloor$ be two parameters in our algorithm. Our main idea is based on divide-and-conquer on vertex sets. We hope to partition a vertex set U into 2^t pieces $U = U_1 \cup U_2 \cdots \cup U_{2^t}$ of similar sizes, where vertices in earlier pieces have smaller distances, and then recursively partition each U_i. In this way, the size of the sub-problem shrinks to a single vertex after roughly $(\log n)/t$ recursion levels. To construct our structure dynamically, each time we would try to compute the distances to a set of closest vertices (without necessarily recovering a complete ordering between their true distances), and report a boundary indicating how much progress we actually make.

Suppose at some stage of the algorithm, for every u with $d(u) < b$, u is complete and we have relaxed all the edges from u. We want to find the true distances to vertices v with $d(v) \geq b$. To avoid the $\Theta(\log n)$ time per vertex in a priority queue, consider the "frontier" S containing all current vertices v with $b \leq \hat{d}[v] < B$ for some bound B (without sorting them). We can see that the shortest path to every incomplete vertex v' with $b \leq d(v') < B$ must visit some complete vertex in S. Thus to compute the true distance to every v' with $b \leq d(v') < B$, it suffices to find the shortest paths from vertices in S and bounded by B. We call this subproblem *bounded multi-source shortest path* (BMSSP) and design an efficient algorithm to solve it. The following lemma summarizes what our BMSSP algorithm achieves.

Lemma 3.1 (Bounded Multi-Source Shortest Path). *We are given an integer level $l \in [0, \lceil \log(n)/t \rceil]$, a set of vertices S of size $\leq 2^{lt}$, and an upper bound $B > \max_{x \in S} \hat{d}[x]$. Suppose that for every incomplete vertex v with $d(v) < B$, the shortest path to v visits some complete vertex $u \in S$.*

Then we have a sub-routine BMSSP(l, B, S) (Algorithm 3) in $O((kl + tl/k + t)|U|)$ time that outputs a new boundary $B' \leq B$ and a vertex set U that contains every vertex v with $d(v) < B'$ and the shortest path to v visits some vertex of S. At the end of the sub-routine, U is complete. Moreover, one of the following is true:

Successful execution $B' = B$.

Partial execution due to large workload $B' < B$, *and* $|U| = \Theta\left(k2^{lt}\right)$.

On the top level of divide and conquer, the main algorithm calls BMSSP with parameters $l = \lceil (\log n)/t \rceil$, $S = \{s\}$, $B = \infty$. Because $|U| \leq |V| = o(kn)$, it must be a successful execution, and the shortest paths to all vertices are found. With chosen k and t, the total running time is $O(m \log^{2/3} n)$.

BMSSP procedure on level l works by recursion, it first tries to "shrink" S to size $|U|/k$ by a simple Bellman-Ford-like method (Lemma 3.2), then it makes several recursive calls on level $(l-1)$ until the bound reaches B or the size of U reaches $\Theta\left(k2^{lt}\right)$. We always make sure that a recursive call solves a problem that is smaller by a factor of roughly $1/2^t$, so the number of levels of the recursion is bounded by $O((\log n)/t)$. The main ideas are:

- Every time we only select about $2^{(l-1)t}$ vertices for our next recursive call, so we use a partial sorting heap-like data structure as described in Lemma 3.3 to improve the running time.
- If we used all of S in the recursive calls, then after all remaining levels S could be fully sorted and nothing is improved. Thus FindPivots (Algorithm 1) procedure is crucial here, as it shows that only at most $|U|/k$ vertices of S are useful in the recursive calls.
- Partial executions may be more complicated to analyze, so we introduce some techniques like BatchPrepend operation in Lemma 3.3.

Finding Pivots. Recall that in the current stage, every u such that $d(u) < b$ is complete, and we have relaxed all the edges from such u's, and the set S includes every vertex v with current $b \leq \hat{d}[v] < B$. Thus, the shortest path of every incomplete vertex v such that $d(v) < B$ visits some complete vertex in S. (This is because the first vertex w in the shortest path to v with $d(w) \geq b$ is complete and included in S.)

The idea of finding pivots is as follows: we perform relaxation for k steps (with a bound B). After this, if the shortest s-v path with $b \leq d(v) < B$ goes through at most k vertices w with $b \leq d(w) < B$, then v is already complete. Observe that the number of large shortest path trees from S, consisting of at least k vertices and rooted at vertices in S, is at most $|\tilde{U}|/k$ here, where $\tilde{U}$ is the set of all vertices v such that $d(v) < B$ and the shortest path of v visits some complete vertex in S. So only the roots of such shortest-path trees are needed to be considered in the recursive calls, and they are called "pivots".

Lemma 3.2 (Finding Pivots). *Suppose we are given a bound B and a set of vertices S. Suppose that for every incomplete vertex v*

such that $d(v) < B$, the shortest path to v visits some complete vertex $u \in S$.

Denote by $\tilde{U}$ the set that contains every vertex v such that $d(v) < B$ and the shortest path to v passes through some vertex in S. The sub-routine FindPivots(B, S) *(Algorithm 1) finds a set $W \subseteq \tilde{U}$ of size $O(k|S|)$ and a set of pivots $P \subseteq S$ of size at most $|W|/k$ such that for every vertex $x \in \tilde{U}$, at least one of the following two conditions holds:*

- *At the end of the sub-routine, $x \in W$ and x is complete;*
- *The shortest path to x visits some complete vertex $y \in P$.*

Moreover, the sub-routine runs in $O(k|W|) = O(\min\{k^2|S|, k|\tilde{U}|\})$ time.

Algorithm 1 Finding Pivots

function FindPivots(B, S)
• **requirement:** for every incomplete vertex v with $d(v) < B$, the shortest path to v visits some complete vertex in S
• **returns:** sets P, W satisfying the conditions in Lemma 3.2
  $W \leftarrow S$
  $W_0 \leftarrow S$
  for $i \leftarrow 1$ to k **do** ▷ Relax for k steps
    $W_i \leftarrow \emptyset$
    for all edges (u, v) with $u \in W_{i-1}$ **do**
      if $\hat{d}[u] + w_{uv} \le \hat{d}[v]$ **then**
        $\hat{d}[v] \leftarrow \hat{d}[u] + w_{uv}$
        if $\hat{d}[u] + w_{uv} < B$ **then**
          $W_i \leftarrow W_i \cup \{v\}$
    $W \leftarrow W \cup W_i$
    if $|W| > k|S|$ **then**
      $P \leftarrow S$
      return P, W
  $F \leftarrow \{(u, v) \in E : u, v \in W, \hat{d}[v] = \hat{d}[u] + w_{uv}\}$ ▷ F is a directed forest under Assumption 2.1
  $P \leftarrow \{u \in S : u$ is a root of a tree with $\ge k$ vertices in $F\}$
  return P, W

Proof. Note that all vertices visited by Algorithm 1 are in $\tilde{U}$, and those in W are not guaranteed to be complete after the procedure. Also note that every vertex in $\tilde{U}$ must visit some vertex in S which was complete before Algorithm 1, since any incomplete vertex v at that time with $d(v) < B$ must visit some complete vertex in S.

If the algorithm returns due to $|W| > k|S|$, we still have $|W| = O(k|S|)$ since the out-degrees of vertices are constant. For every incomplete vertex v such that $d(v) < B$, we know that the shortest path to v visits some complete vertex $u \in S$. Since $P = S$, we must also have $u \in P$ and conditions in the lemma are thus satisfied.

If $|W| \le k|S|$, P is derived from F. For any vertex $x \in \tilde{U}$, consider the first vertex $y \in S$ on the shortest path to x which was complete before Algorithm 1, then y must be a root of a tree in F. If there are no more than $k - 1$ edges from y to x on the path, x is complete and added to W after k relaxations. Otherwise, the tree rooted at y contains at least k vertices, therefore y is added to P. Additionally, $|P| \le |W|/k \le |\tilde{U}|/k$ since each vertex in P covers a unique subtree of size at least k in F.

To evaluate the time complexity, note that the size of W is $O(\min\{k|S|, |\tilde{U}|\})$, so each iteration takes $O(\min\{k|S|, |\tilde{U}|\})$ time. Computing P after the for-loop runs in $O(|W|)$ time for final W. Therefore, FindPivots finishes in $O(\min\{k^2|S|, k|\tilde{U}|\})$ time. □

We also use the following data structure to help adaptively partition a problem into sub-problems, specified in Lemma 3.3:

Lemma 3.3. *Given at most N key/value pairs to be inserted, an integer parameter M, and an upper bound B on all the values involved, there exists a data structure that supports the following operations:*

Insert *Insert a key/value pair in $O(\max\{1, \log(N/M)\})$ amortized time. If the key already exists, update its value.*

Batch Prepend *Insert L key/value pairs such that each value in L is smaller than any value currently in the data structure, in amortized $O(L \cdot \max\{1, \log(L/M)\})$ time. If there are multiple pairs with the same key, keep the one with the smallest value.*

Pull *Return a subset S' of keys where $|S'| \le M$ associated with the smallest $|S'|$ values and an upper bound x that separates S' from the remaining values in the data structure, in amortized $O(|S'|)$ time. Specifically, if there are no remaining values, x should be B. Otherwise, x should satisfy $\max(S') < x \le \min(D)$ where D is the set of elements in the data structure after the pull operation.*

Proof. We introduce a block-based linked list data structure to support the required operations efficiently.

Specifically, the data is organized into two sequences of blocks, $\mathcal{D}_0$ and $\mathcal{D}_1$. $\mathcal{D}_0$ only maintains elements from batch prepends while $\mathcal{D}_1$ maintains elements from insert operations, so every single inserted element is always inserted to $\mathcal{D}_1$. Each block is a linked list containing at most M key/value pairs. The number of blocks in $\mathcal{D}_1$ is bounded by $O(\max\{1, N/M\})$, while $\mathcal{D}_0$ does not have such a requirement. Blocks are maintained in the sorted order according to their values, that is, for any two key/value pairs $\langle a_1, b_1\rangle \in B_i$ and $\langle a_2, b_2\rangle \in B_j$, where B_i and B_j are the i-th and j-th blocks from a block sequence, respectively, and $i < j$, we have $b_1 \le b_2$. For each block in $\mathcal{D}_1$, we maintain an upper bound for its elements, so that the upper bound for a block is no more than any value in the next block. We adopt a self-balancing binary search tree (e.g. Red-Black Tree [17]) to dynamically maintain these upper bounds, with $O(\max\{1, \log(N/M)\})$ search/update time.

For cases where multiple pairs with the same key are added, we record the status of each key and the associated value. If the new pair has a smaller value, we first delete the old pair then insert the new one, ensuring that only the most advantageous pair is retained for each key. For the operations required in Lemma 3.3:

Initialize(M, B) Initialize $\mathcal{D}_0$ with an empty sequence and $\mathcal{D}_1$ with a single empty block with upper bound B. Set the parameter M.

Delete(a, b) To delete the key/value pair $\langle a, b\rangle$, we remove it directly from the linked list, which can be done in $O(1)$ time. Note that it's unnecessary to update the upper bounds of blocks after a deletion. However, if a block in $\mathcal{D}_1$ becomes empty after deletion, we need to remove its upper bound in the binary search tree in $O(\max\{1, \log(N/M)\})$ time. Since Insert takes $O(\max\{1, \log(N/M)\})$ time for $\mathcal{D}_1$, deletion time will be amortized to insertion time with no extra cost.

Insert(a, b) To insert a key/value pair $\langle a, b\rangle$, we first check the existence of its key a. If a already exists, we delete original pair $\langle a, b'\rangle$ and insert new pair $\langle a, b\rangle$ only when $b < b'$.
Then we insert $\langle a, b\rangle$ to $\mathcal{D}_1$. We first locate the appropriate block for it, which is the block with the smallest upper bound greater than or equal to b, using binary search (via the binary search tree) on the block sequence. $\langle a, b\rangle$ is then added to the corresponding linked list in $O(1)$ time. Given that the number of blocks in $\mathcal{D}_1$ is $O(\max\{1, N/M\})$, as we will establish later, the total time complexity for a single insertion is $O(\max\{1, \log(N/M)\})$.
After an insertion, the size of the block may increase, and if it exceeds the size limit M, a split operation will be triggered.

Split When a block in $\mathcal{D}_1$ exceeds M elements, we perform a split. First, we identify the median element within the block in $O(M)$ time [3], partitioning the elements into two new blocks each with at most $\lceil M/2\rceil$ elements — elements smaller than the median are placed in the first block, while the rest are placed in the second. This split ensures that each new block retains about $\lceil M/2\rceil$ elements while preserving inter-block ordering, so the number of blocks in $\mathcal{D}_1$ is bounded by $O(N/M)$. (Every block in $\mathcal{D}_1$ contains $\Theta(M)$ elements, including the elements already deleted.) After the split, we make the appropriate changes in the binary search tree of upper bounds in $O(\max\{1, \log(N/M)\})$ time.

BatchPrepend$(\mathcal{L})$ Let L denote the size of $\mathcal{L}$. When $L \le M$, we simply create a new block for $\mathcal{L}$ and add it to the beginning of $\mathcal{D}_0$. Otherwise, we create $O(L/M)$ new blocks in the beginning of $\mathcal{D}_0$, each containing at most $\lceil M/2\rceil$ elements. We can achieve this by repeatedly taking medians which completes in $O(L \log(L/M))$ time.

Pull() To retrieve the smallest M values from $\mathcal{D}_0 \cup \mathcal{D}_1$, we collect a sufficient prefix of blocks from $\mathcal{D}_0$ and $\mathcal{D}_1$ separately, denoted as S'_0 and S'_1, respectively. That is, in $\mathcal{D}_0$ ($\mathcal{D}_1$) we start from the first block and stop collecting as long as we have collected all the remaining elements or the number of collected elements in S'_0 (S'_1) has reached M. If $S'_0 \cup S'_1$ contains no more than M elements, it must contain all blocks in $\mathcal{D}_0 \cup \mathcal{D}_1$, so we return all elements in $S'_0 \cup S'_1$ as S' and set x to the upper bound B, and the time needed is $O(|S'|)$. Otherwise, we want to make $|S'| = M$, and because the block sizes are kept at most M, the collecting process takes $O(M)$ time.
Now we know the smallest M elements must be contained in $S'_0 \cup S'_1$ and can be identified from $S'_0 \cup S'_1$ as S' in $O(M)$ time. Then we delete elements in S' from $\mathcal{D}_0$ and $\mathcal{D}_1$, whose running time is amortized to insertion time. Also set returned value x to the smallest remaining value in $\mathcal{D}_0 \cup \mathcal{D}_1$, which can also be found in $O(M)$ time.

□

We now describe our BMSSP algorithm (Algorithm 3) in detail. Recall that the main algorithm calls BMSSP with parameters $l = \lceil (\log n)/t\rceil, S = \{s\}, B = \infty$ on the top level.

In the base case of $l = 0$, S is a singleton $\{x\}$ and x is complete. We run a mini Dijkstra's algorithm (Algorithm 2) starting from x to find the closest vertices v from x such that $d(v) < B$ and the shortest path to v visits x, until we find $k + 1$ such vertices or no more vertices can be found. Let U_0 be the set of them. If we do not find $k + 1$ vertices, return $B' \leftarrow B, U \leftarrow U_0$. Otherwise, return $B' \leftarrow \max_{v\in U_0} d(v), U \leftarrow \{v \in U_0 : d(v) < B'\}$.

Algorithm 2 Base Case of BMSSP

```
function BaseCase(B, S)
  • requirement 1: S = {x} is a singleton, and x is complete
  • requirement 2: for every incomplete vertex v with d(v) < B, the shortest path to v visits x
  • returns 1: a boundary B' ≤ B
  • returns 2: a set U
    U_0 ← S
    initialize a binary heap H with a single element ⟨x, d̂[x]⟩
    while H is non-empty and |U_0| < k + 1 do
        ⟨u, d̂[u]⟩ ← H.ExtractMin()
        U_0 ← U_0 ∪ {u}
        for edge e = (u, v) do
            if d̂[u] + w_uv ≤ d̂[v] and d̂[u] + w_uv < B then
                d̂[v] ← d̂[u] + w_uv
                if v is not in H then
                    H.Insert(⟨v, d̂[v]⟩)
                else
                    H.DecreaseKey(⟨v, d̂[v]⟩)
    if |U_0| ≤ k then
        return B' ← B, U ← U_0
    else
        return B' ← max_{v∈U_0} d̂[v], U ← {v ∈ U_0 : d̂[v] < B'}
```

If $l > 0$, first we use FindPivots from Lemma 3.2 to obtain a pivot set $P \subseteq S$ and a set W. We initialize the data structure $\mathcal{D}$ from Lemma 3.3 with $M := 2^{(l-1)t}$ and add each $x \in P$ as a key associated with value $\hat{d}[x]$ to $\mathcal{D}$. For simplicity, we write $\mathcal{D}$ as a set to refer to all the keys (vertices) in it.

Set $B'_0 \leftarrow \min_{x\in P} \hat{d}[x]; U \leftarrow \emptyset$. The rest of the algorithm repeats the following iteration of many phases, during the i-th iteration, we:

(1) Pull from $\mathcal{D}$ a subset S_i of keys associated with the smallest values and B_i indicating a lower bound of remaining value in $\mathcal{D}$;
(2) Recursively call BMSSP$(l-1, B_i, S_i)$, obtain its return, B'_i and U_i, and add vertices in U_i into U;
(3) Relax every edge (u, v) for $u \in U_i$ ($\hat{d}[v] \leftarrow \min\{\hat{d}[v], \hat{d}[u] + w_{uv}\}$). If the relax update is valid ($\hat{d}[u] + w_{uv} \le \hat{d}[v]$), do the following (even when $\hat{d}[v]$ already equals $\hat{d}[u] + w_{uv}$):
   (a) if $\hat{d}[u] + w_{uv} \in [B_i, B)$, then simply insert $\langle v, \hat{d}[u] + w_{uv}\rangle$ into $\mathcal{D}$;
   (b) if $\hat{d}[u] + w_{uv} \in [B'_i, B_i)$, then record $\langle v, \hat{d}[u] + w_{uv}\rangle$ in a set K;
(4) Batch prepend all records in K and $\langle x, \hat{d}[x]\rangle$ for $x \in S_i$ with $\hat{d}[x] \in [B'_i, B_i)$ into $\mathcal{D}$;
(5) If $\mathcal{D}$ is empty, then it is a successful execution and we return;
(6) If $|U| > k2^{lt}$, set $B' \leftarrow B'_i$. We expect a large workload and we prematurely end the execution.

Finally, before we end the algorithm, we update U to include every vertex x in the set W returned by FindPivots with $\hat{d}[x] < B'$.

Algorithm 3 Bounded Multi-Source Shortest Path

1: **function** BMSSP(l, B, S)
• **requirement 1:** $|S| \le 2^{lt}$
• **requirement 2:** for every incomplete vertex x with $d(x) < B$, the shortest path to x visits some complete vertex $y \in S$
• **returns 1:** a boundary $B' \le B$
• **returns 2:** a set U
2: **if** $l = 0$ **then**
3: **return** $B', U \leftarrow$ BaseCase(B, S)
4: $P, W \leftarrow$ FindPivots(B, S)
5: $\mathcal{D}$.Initialize(M, B) with $M = 2^{(l-1)t}$ ▷ $\mathcal{D}$ is an instance of Lemma 3.3
6: $\mathcal{D}$.Insert($\langle x, \hat{d}[x]\rangle$) **for** $x \in P$
7: $i \leftarrow 0$; $B'_0 \leftarrow \min_{x \in P} \hat{d}[x]$; $U \leftarrow \emptyset$ ▷ If $P = \emptyset$ set $B'_0 \leftarrow B$
8: **while** $|U| < k2^{lt}$ and $\mathcal{D}$ is non-empty **do**
9: $i \leftarrow i + 1$
10: $B_i, S_i \leftarrow \mathcal{D}$.Pull()
11: $B'_i, U_i \leftarrow$ BMSSP($l - 1, B_i, S_i$)
12: $U \leftarrow U \cup U_i$
13: $K \leftarrow \emptyset$
14: **for** edge $e = (u, v)$ where $u \in U_i$ **do**
15: **if** $\hat{d}[u] + w_{uv} \le \hat{d}[v]$ **then**
16: $\hat{d}[v] \leftarrow \hat{d}[u] + w_{uv}$
17: **if** $\hat{d}[u] + w_{uv} \in [B_i, B)$ **then**
18: $\mathcal{D}$.Insert($\langle v, \hat{d}[u] + w_{uv}\rangle$)
19: **else if** $\hat{d}[u] + w_{uv} \in [B'_i, B_i)$ **then**
20: $K \leftarrow K \cup \{\langle v, \hat{d}[u] + w_{uv}\rangle\}$
21: $\mathcal{D}$.BatchPrepend($K \cup \{\langle x, \hat{d}[x]\rangle : x \in S_i$ and $\hat{d}[x] \in [B'_i, B_i)\}$)
22: **return** $B' \leftarrow \min\{B'_i, B\}$; $U \leftarrow U \cup \{x \in W : \hat{d}[x] < B'\}$

Remark 3.4. *Note that on line 7 of Algorithm 1, line 8 of Algorithm 2 and line 15 of Algorithm 3, the conditions are "$\hat{d}[u] + w_{uv} \le \hat{d}[v]$". The equality is required so that an edge relaxed on a lower level can be re-used on upper levels.*

Remark 3.5. *Using methods in Lemma 3.3 to implement data structure $\mathcal{D}$ in Algorithm 3 at level l, where $M = 2^{(l-1)t}$, and $|S| \le 2^{lt}$, the total number of insertions N is $O(k2^{lt})$, because of the fact that $|U| = O(k2^{lt})$, the constant-degree property, and the disjointness of U_i's (as established later in Remark 3.8). Also, size of K each time is bounded by $O(|U_i|) = O(k2^{(l-1)t})$. Thus, insertion to $\mathcal{D}$ takes $O(\log k + t) = O(t)$ time, and Batch Prepend takes $O(\log k) = O(\log\log n)$ time per vertex in K.*

3.2 Observations and Discussions on the Algorithm

We first give some informal explanations on the algorithm, and then in the next subsections, we will formally prove the correctness and running time of the algorithm.

What can we get from the recursion? Let's look at the recursion tree $\mathcal{T}$ of Algorithm 3 where each node x denotes a call of the BMSSP procedure. Let l_x , B_x, and S_x denote the parameters l, B, and S at node x, B'_x and U_x denote the returned values B' and U at node x, P_x and W_x denote P and W (on line 4) returned by the FindPivots procedure at node x, respectively, and W'_x denote the set of vertices $v \in W_x$ satisfying $d(v) < B'_x$. Then we expect:

(1) Since we assume that all vertices are reachable from s, in the root r of $\mathcal{T}$, we have $U_r = V$;
(2) $|S_x| \le 2^{l_x t}$; therefore the depth of the $\mathcal{T}$ is at most $(\log n)/t = O(\log^{1/3} n)$;
(3) We break when $|U_x| \ge k2^{l_x t} > |S_x|$. Intuitively, the size of U_x increases slowly enough so we should still have $|U_x| = O(k2^{l_x t})$ (formally shown in Lemma 3.9);
(4) If node x is a successful execution, in the execution of FindPivots, $\tilde{U}$ in Lemma 3.2 is equal to U_x, so $|P_x| \le |U_x|/k$ by Lemma 3.2;
(5) If node x is a partial execution, $|U_x| \ge k2^{l_x t} \ge k|S_x|$, so $|P_x| \le |S_x| \le |U_x|/k$;
(6) For each node x of $\mathcal{T}$ and its children $y_1, \cdots, y_q$ in the order of the calls, we have:
- $U_{y_1}, \cdots, U_{y_q}$ are disjoint. For $i < j$, distances to vertices in U_{y_i} are smaller than distances to vertices in U_{y_j};
- $U_x = W'_x \cup U_{y_1} \cup \cdots \cup U_{y_q}$;
- Consequently, the U_x's for all nodes x at the same depth in $\mathcal{T}$ are disjoint, and total sum of $|U_x|$ for all nodes x in $\mathcal{T}$ is $O(n(\log n)/t)$.

Correctness. In a call of BMSSP, let $\tilde{U}$ denote the set that contains all vertices v with $d(v) < B$ and the shortest path to v visits some vertex in S. Then BMSSP should return $U = \tilde{U}$ in a successful execution, or $U = \{u \in \tilde{U} : d(u) < B'\}$ in a partial execution, with all vertices in U complete.

In the base case where $l = 0$, S contains only one vertex x, a Dijkstra-like algorithm starting from x completes the task.

Otherwise we first shrink S to a smaller set of pivots P to insert into $\mathcal{D}$, with some vertices complete and added into W. Lemma 3.2 ensures that the shortest path of any remaining incomplete vertex v visits some complete vertex in $\mathcal{D}$. For any bound $B_i \le B$, if $d(v) < B_i$, the shortest path to v must also visit some complete vertex $u \in \mathcal{D}$ with $d(u) < B_i$. Therefore we can call subprocedure BMSSP on B_i and S_i.

By inductive hypothesis, each time a recursive call on line 11 of Algorithm 3 returns, vertices in U_i are complete. After the algorithm relaxes edges from $u \in U_i$ and inserts all the updated out-neighbors x with $\hat{d}[x] \in [B'_i, B)$ into $\mathcal{D}$, once again any remaining incomplete vertex v now visits some complete vertex in $\mathcal{D}$.

Finally, with the complete vertices in W added, U contains all vertices required and all these vertices are complete.

Running time. The running time is dominated by calls of FindPivots and the overheads inside the data structures $\mathcal{D}$. By (4) and (5), the running time of FindPivots procedure at node x is bounded by $O(|U_x|k)$, so the total running time over all nodes on one depth of $\mathcal{T}$ is $O(nk)$. Summing over all depths we get $O(nk \cdot (\log n)/t) = O(n\log^{2/3} n)$.

For the data structure $\mathcal{D}$ in a call of BMSSP, the total running time is dominated by INSERT and BATCH PREPEND operations. We analyze the running time as follows.

- Vertices in P can be added to $\mathcal{D}$ on line 6. Since $|P_x| = O(|U_x|/k)$, the total time for nodes of one depth of $\mathcal{T}$ is $O(n/k \cdot (t + \log k)) = O(nt/k)$. Summing over all depths we get $O((n \log n)/k) = O(n \log^{2/3} n)$.
- Some vertices already pulled to S_i can be added back to $\mathcal{D}$ through BATCHPREPEND on line 21. Here we know in every iteration i, $|S_i| \le |U_i|$, so the total time for adding back is bounded by $\sum_{x \in \mathcal{T}} |U_x| \cdot \log k = O(n \log k \cdot (\log n)/t) = O(n \cdot \log^{1/3} n \cdot \log \log n)$.
- A vertex v can be inserted to $\mathcal{D}$ through edge (u, v) directly on line 18 if $B_i \le \hat{d}(u) + w_{uv} < B$, or through K as on line 20 if $B'_i \le \hat{d}(u) + w_{uv} < B_i$. (Note that $u \in U_i$ is already complete.) Every edge (u, v) can only be relaxed once in each level by (6), but it can be relaxed in multiple levels in an ancestor-descendant path of the recursion tree $\mathcal{T}$. (Note that the condition on line 15 is $\hat{d}[u] + w_{uv} \le \hat{d}[v]$.) However, we can show every edge (u, v) can only lead v to be directly inserted to $\mathcal{D}$ by the INSERT operation on one level.
  - It can be easily seen that if y is a descendant of x in the recursion tree $\mathcal{T}$, $B_y \le B_x$.
  - If (u, v) leads v to be directly inserted to $\mathcal{D}$ on line 18, then $\hat{d}(u) + w_{uv} \ge B_i$. Since $u \in U_i$, for every descendant y of the current call in the recursion tree $\mathcal{T}$ satisfying $u \in U_y$, we have $B_y \le B_i \le \hat{d}(u) + w_{uv}$, so v will not be added to $\mathcal{D}$ through (u, v) in any way in lower levels.
- Since every edge (u, v) can only lead v to be directly inserted to $\mathcal{D}$ by the INSERT operation once in the algorithm, the total time is $O(m(\log k + t)) = O(m \log^{2/3} n)$. The time for all edges (u, v) leading v to be inserted to $\mathcal{D}$ through K in one level is $O(m \log k)$, and in total is $O(m \log k \cdot (\log n)/t) = O(m \cdot \log^{1/3} n \cdot \log \log n)$.

3.3 Correctness Analysis

To present the correctness more accurately and compactly, we introduce several notations on the "shortest path tree". Due to the uniqueness of shortest paths, the shortest path tree T rooted at the source s can be constructed unambiguously. Define $T(u)$ as the subtree rooted at u according to $\hat{d}(\cdot)$. An immediate observation is that *the shortest path to v passes through u if and only if v is in the subtree rooted at u.*

For a vertex set S, define $T(S) = \bigcup_{v \in S} T(v)$, namely the union of the subtrees of $T(s)$ rooted at vertices in S, or equivalently, the set of vertices whose shortest paths pass through some vertex in S. For a set of vertices $S \subseteq V$, denote $S^* = \{v \in S : v \text{ is complete}\}$. Then $T(S^*) = \bigcup_{v \in S^*} T(v)$ is the union of subtrees of $T(s)$ rooted at S^*, or the set of vertices whose shortest paths pass through some complete vertex in S. Obviously, $T(S^*) \subseteq T(S)$. Note that $T(S^*)$ is sensitive to the progress of the algorithm while $T(S)$ is a fixed set. Another observation is that, a complete vertex would remain complete, so S^* and $T(S^*)$ never lose a member.

For a bound B, denote $T_{<B}(S) = \{v \in T(S) : \hat{d}(v) < B\}$. Also note that $T_{<B}(S)$ coincides with $\tilde{U}$ mentioned above. For an interval $[b, B)$, denote $T_{[b,B)}(S) = \{v \in T(S) : \hat{d}(v) \in [b, B)\}$.

LEMMA 3.6 (PULL MINIMUM). *Suppose every incomplete vertex v with $\hat{d}(v) < B$ is in $T(S^*)$. Suppose we split S into $X = \{x \in S : \hat{d}[x] < \mathcal{B}\}$ and $Y = \{x \in S : \hat{d}[x] \ge \mathcal{B}\}$ for some $\mathcal{B} < B$. Then every incomplete vertex v with $\hat{d}(v) < \mathcal{B}$ is in $T(X^*)$. Moreover, for any $\mathcal{B}' < \mathcal{B}$, $T_{<\mathcal{B}'}(S) = T_{<\mathcal{B}'}(X)$.*

PROOF. For any incomplete vertex v with $\hat{d}(v) < \mathcal{B}$, as $\mathcal{B} < B$, by definition, $v \in T(u)$ for some complete vertex $u \in S$. Therefore $\hat{d}[u] = \hat{d}(u) \le \hat{d}(v) < \mathcal{B}$, so $u \in X$ and thus $v \in T(X^*)$.

For the second statement, it is clear that $T_{<\mathcal{B}'}(X) \subseteq T_{<\mathcal{B}'}(S)$. For any $v \in T_{<\mathcal{B}'}(S)$, since $\hat{d}(v) < \mathcal{B}' < \mathcal{B} < B$, the shortest path to v passes through some vertex $x \in S^*$. Now that $\hat{d}[x] = \hat{d}(x) \le \hat{d}(v) < \mathcal{B}'$, we have $x \in X$ and $v \in T_{<\mathcal{B}'}(X)$. □

Now we are ready to prove the correctness part of Lemma 3.1.

LEMMA 3.7. *We prove the correctness of Algorithm 3 by proving the following statement (the size of U is dealt with in Lemma 3.9), restated from Lemma 3.1: Given a level $l \in [0, \lceil (\log n)/t \rceil]$, a bound B and a set of vertices S of size $\le 2^{lt}$, suppose that every incomplete vertex v with $\hat{d}(v) < B$ is in $T(S^*)$.*

Then, after running Algorithm 3, we have: $U = T_{<B'}(S)$, and U is complete.

PROOF. We prove by induction on l. When $l = 0$, since $S = \{x\}$ is a singleton, x must be complete. Then clearly the classical Dijkstra's algorithm (Algorithm 2) finds the desired $U = T_{<B'}(S)$ as required. Suppose correctness holds for $l - 1$, we prove that it holds for l. Denote $\mathcal{D}_i$ as the set of vertices (keys) in $\mathcal{D}$ just before the i-th iteration (at the end of the $(i-1)$-th iteration), and $\mathcal{D}_i^*$ as the set of complete vertices in $\mathcal{D}_i$ at that time. Next, we prove the following two propositions by induction on increasing order of i:

Immediately before the i-th iteration,

(a) Every incomplete vertex v with $\hat{d}(v) < B$ is in $T_{[B'_{i-1}, B)}(P)$.
(b) $T_{[B'_{i-1}, B)}(P) = T_{<B}(\mathcal{D}_i) = T_{<B}(\mathcal{D}_i^*)$;

In the base case $i = 1$, by Lemma 3.2, for every incomplete vertex v with $\hat{d}(v) < B$ (including $v \in P$), $v \in T_{<B}(P^*)$. Then $\hat{d}(v) \ge \min_{x \in P^*} \hat{d}(x) = \min_{x \in P^*} \hat{d}[x] \ge B'_0$. Thus every such v is in $T_{[B'_0, B)}(P)$ (actually $T_{<B'_0}(P)$ is an empty set) and $T_{[B'_0, B)}(P) = T_{<B}(P) = T_{<B}(P^*)$. Because $\mathcal{D}_1 = P$, the base case is proved.

Suppose both propositions hold for i. Then each incomplete vertex v with $\hat{d}(v) < B$ is in $T_{[B'_{i-1}, B)}(P) \subseteq T(\mathcal{D}_i^*)$. By Lemma 3.3, the suppositions of Lemma 3.6 are met for $X := S_i$, $Y := \mathcal{D}_i \setminus S_i$, and $\mathcal{B} := B_i$, so every incomplete vertex v with $\hat{d}(v) < B_i$ is in $T({S_i}^*)$; also note $|S_i| \le 2^{(l-1)t}$. By induction hypothesis on level $l - 1$, the i-th recursive call is correct and we have $U_i = T_{<B'_i}(S_i)$ and is complete. Note that S_i includes all the vertices v in $\mathcal{D}_i$ such that $\hat{d}[v] < B'_i$ and that $B'_i \le B$. Thus by Lemma 3.6 and proposition (b) on case i, $U_i = T_{<B'_i}(S_i) = T_{<B'_i}(\mathcal{D}_i) = T_{[B'_{i-1}, B'_i)}(P)$ and is complete (thus proving Remark 3.8). Then, every incomplete vertex v with $\hat{d}(v) < B$ is in $T_{[B'_{i-1}, B)}(P) \setminus T_{[B'_{i-1}, B'_i)}(P) = T_{[B'_i, B)}(P)$. Thus, we proved proposition (a) for the $(i+1)$-th iteration.

Now we prove proposition (b) for the $(i+1)$-th iteration. Since $B'_i \ge B'_{i-1}$, from proposition (b) of the i-th iteration, we have $T_{[B'_i, B)}(P) = T_{[B'_i, B)}(\mathcal{D}_i) = T_{[B'_i, B)}(\mathcal{D}_i^*)$. Suppose we can show that $T_{[B'_i, B)}(\mathcal{D}_i^*) \subseteq T_{<B}(\mathcal{D}_{i+1}^*)$ and $T_{<B}(\mathcal{D}_{i+1}) \subseteq T_{[B'_i, B)}(\mathcal{D}_i)$. By

definition $T_{<B}(\mathcal{D}^*_{i+1}) \subseteq T_{<B}(\mathcal{D}_{i+1})$, then:

$$T_{[B'_i,B)}(P) = T_{[B'_i,B)}(\mathcal{D}^*_i) \subseteq T_{<B}(\mathcal{D}^*_{i+1}) \subseteq T_{<B}(\mathcal{D}_{i+1}) \subseteq T_{[B'_i,B)}(\mathcal{D}_i) = T_{[B'_i,B)}(\mathcal{D}^*_i),$$

and by sandwiching, proposition (b) holds for $(i+1)$. Thus it suffices to prove $T_{[B'_i,B)}(\mathcal{D}^*_i) \subseteq T_{<B}(\mathcal{D}^*_{i+1})$ and $T_{<B}(\mathcal{D}_{i+1}) \subseteq T_{[B'_i,B)}(\mathcal{D}_i)$.

For any vertex $y \in \mathcal{D}^*_i \setminus \mathcal{D}^*_{i+1}$, we have $y \in S_i$. Note that vertices v of S_i with $\hat{d}[v] \geq B'_i$ are batch-prepended back to $\mathcal{D}_{i+1}$ on line 21, so we have $\hat{d}[y] < B'_i$. Thus $y \in U_i$ and is complete. For any vertex $x \in T_{[B'_i,B)}(y)$, as $x \notin U_i$, along the shortest path from y to x, there is an edge (u,v) with $u \in U_i$ and $v \notin U_i$. During relaxation, v is then complete and is added to $\mathcal{D}_{i+1}$, so $x \in T_{[B'_i,B)}(v) \subseteq T_{<B}(\mathcal{D}^*_{i+1})$.

For any vertex $v \in \mathcal{D}_{i+1} \setminus \mathcal{D}_i$, since v is added to $\mathcal{D}_{i+1}$, there is an edge (u,v) with $u \in U_i = T_{<B'_i}(\mathcal{D}_i)$ and the relaxation of (u,v) is valid ($\hat{d}[u] + w_{uv} \leq \hat{d}[v]$). Pick the last valid relaxation edge (u,v) for v. If $v \in T(u)$, then v is complete, $d(v) = \hat{d}[v] \geq B'_i$, and $v \in T_{[B'_i,B)}(\mathcal{D}_i)$; if v is incomplete, then by proposition (a), $v \in T_{[B'_i,B)}(P) = T_{[B'_i,B)}(\mathcal{D}_i)$; or if v is complete but $v \notin T(u)$, we have a contradiction because it implies that relaxation of (u,v) is invalid.

Now that we have proven the propositions, we proceed to prove that the returned $U = T_{<B'}(S)$ and that U is complete. Suppose there are q iterations. We have shown that every $U_i = T_{[B'_{i-1},B'_i)}(P)$ and is complete, so U_i's are also disjoint. By Lemma 3.2, W contains every vertex in $T_{<B}(S \setminus P)$ (as they are not in $T(P)$) and they are complete; besides, all vertices v in W but not in $T_{<B}(S \setminus P)$ with $d(v) < B'_q$ are complete (because they are in $T_{<B'_q}(P)$). Therefore, $\{x \in W : \hat{d}[x] < B'_q\}$ contains every vertex in $T_{<B'_q}(S \setminus P)$ and is complete. Thus finally $U := (\bigcup_{i=1}^{q} U_i) \cup \{x \in W : \hat{d}[x] < B'_q\}$ equals $T_{<B'_q}(S)$ and is complete. □

Remark 3.8. *From the proof of Lemma 3.7, $U_i = T_{[B'_{i-1},B'_i)}(P)$ and they are disjoint and complete. As a reminder, Lemma 3.6 and proposition (b) of Lemma 3.7 prove this.*

3.4 Time Complexity Analysis

Lemma 3.9. *Under the same conditions as in Lemma 3.7, where every incomplete vertex v with $d(v) < B$ is in $T_{<B}(S^*)$. After running Algorithm 3, we have $|U| \leq 4k2^{lt}$. If $B' < B$, then $|U| \geq k2^{lt}$.*

Proof. We can prove this by induction on the number of levels. The base case $l = 0$ clearly holds.

Suppose there are q iterations. Before the q-th iteration, $|U| < k2^{lt}$. After the q-th iteration, the number of newly added vertices $|U_q| \leq 4k2^{(l-1)t}$ (by Lemma 3.7, the assumptions for recursive calls are satisfied), and because $|W| \leq k|S| \leq k2^{lt}$, we have $|U| \leq 4k2^{lt}$. If $\mathcal{D}$ is empty, the algorithm succeeds and quits with $B' = B$. Otherwise, the algorithm ends partially because $|U| \geq k2^{lt}$. □

Lemma 3.10. *Immediately before the i-th iteration of Algorithm 3, $\min_{x \in \mathcal{D}} d(x) \geq B'_{i-1}$.*

Proof. From the construction of $\mathcal{D}$, immediately before the i-th iteration of Algorithm 3, $\min_{v \in \mathcal{D}} \hat{d}[v] \geq B'_{i-1}$. For $v \in \mathcal{D}$, if v is complete, then $d(v) = \hat{d}[v] \geq B'_{i-1}$. If v is incomplete, by proposition (a) of Lemma 3.7, because $v \in T_{[B'_{i-1},B)}(P)$, $d(v) \geq B'_{i-1}$. □

Lemma 3.11. *Denote by $\tilde{U} := T_{<B}(S)$ the set that contains every vertex v with $d(v) < B$ and the shortest path to v visits some vertex of S. After running Algorithm 3, $\min\{|\tilde{U}|, k|S|\} \leq |U|$ and $|S| \leq |U|$.*

Proof. If it is a successful execution, then $S \subseteq \tilde{U} = U$. If it is partial, then $k|S| \leq k2^{lt} \leq |U|$. □

For a vertex set U and two bounds $c < d$, denote $N^+(U) = \{(u,v) : u \in U\}$ as the set of outgoing edges from U, and similarly $N^+_{[c,d)}(U) = \{(u,v) : u \in U \text{ and } d(u) + w_{uv} \in [c,d)\}$.

Lemma 3.12 (Time Complexity). *Suppose a large constant C upper-bounds the sum of the constants hidden in the big-O notations in all previously mentioned running times: in find-pivots, in the data structure, in relaxation, and all other operations. Algorithm 3 solves the problem in Lemma 3.1 in time*

$$C(k + 2t/k)(l+1)|U| + C(t + l\log k)|N^+_{[\min_{x\in S} d(x),B)}(U)|.$$

With $k = \lfloor \log^{1/3}(n) \rfloor$ and $t = \lfloor \log^{2/3}(n) \rfloor$, calling the algorithm with $l = \lceil (\log n)/t \rceil$, $S = \{s\}$, $B = \infty$ takes $O(m \log^{2/3} n)$ time.

Proof. We prove by induction on the level l. When $l = 0$, the algorithm degenerates to the classical Dijkstra's algorithm (Algorithm 2) and takes time $C|U| \log k$, so the base case is proved. Suppose the time complexity is correct for level $l-1$. Now we analyze the time complexity of level l.

By Remark 3.5, each insertion takes amortized time Ct (note that $\log k < t$); each batch prepended element takes amortized time $C \log k$; each pulled term takes amortized constant time. But we may ignore Pull's running time because each pulled term must have been inserted/batch prepended, and the constant of Pull can be covered by C in those two operations.

By Remark 3.8, U_i are disjoint and $\sum_{i \geq 1} |U_i| \leq |U|$.

Now we are ready to calculate total running time of Algorithm 3 on level l by listing all the steps.

By Lemma 3.2, the FindPivots step takes time $C \cdot \min\{|\tilde{U}|, k|S|\}k$ and $|P| \leq \min\{|\tilde{U}|/k, |S|\}$. Inserting P into $\mathcal{D}$ takes time $C|P|t$. And by Lemma 3.11, their time is bounded by $C(k + t/k)|U|$.

In the i-th iteration, the sub-routine takes $C(k + 2t/k)l|U_i| + C(t + (l-1)\log k)|N^+_{[\min_{x\in S_i} d(x),B_i)}(U_i)|$ time by the induction hypothesis. Taking the sum, by Lemma 3.10, $N^+_{[\min_{x\in S_i} d(x),B_i)}(U_i) \subseteq N^+_{[B'_{i-1},B_i)}(U_i)$. Then the sum of time spent by all the sub-routines is bounded by

$$C(k + 2t/k)l|U| + \overbrace{C(t + (l-1)\log k) \sum_{i \geq 1} |N^+_{[B'_{i-1},B_i)}(U_i)|}^{\mathcal{N}_1}.$$

In the following relaxation step, for each edge (u,v) originated from U_i, if $\hat{d}[u] + w_{uv} \in [B_i, B)$, they are directly inserted; if $\hat{d}[u] + w_{uv} \in [B'_i, B_i)$, they are batch prepended. Again by Remark 3.8, all U_i's are complete, so for the previous statements we can replace

$\hat{d}[\cdot]$ with $d(\cdot)$. Thus in total, it takes time

$$\overbrace{Ct\sum_{i\geq 1}|N^+_{[B_i,B)}(U_i)|}^{\mathcal{N}_2}+\overbrace{C(\log k)\sum_{i\geq 1}|N^+_{[B'_i,B_i)}(U_i)|}^{\mathcal{N}_3}.$$

Vertices in S_i were also batch prepended, and this step takes time $O(|\{x\in S_i : \hat{d}[x]\in[B'_i,B_i)\}|\log k)$. This operation takes non-zero time only if $B'_i < B_i$, i.e., the i-th sub-routine is a partial execution. Thus in total this step takes time $C\sum_{\text{partial } i}|S_i|\log k \leq (C|U|\log k)/k \leq C|U|t/k$.

Therefore, the total running time on the level l is

$$C(k+2t/k)(l+1)|U|+\mathcal{N},$$

where $\mathcal{N}$ is the sum

$$\mathcal{N}=\overbrace{C(t+(l-1)\log k)\sum_{i\geq 1}|N^+_{[B'_{i-1},B_i)}(U_i)|+Ct\sum_{i\geq 1}|N^+_{[B_i,B)}(U_i)|}^{\mathcal{N}_1+\mathcal{N}_2}+\overbrace{C(\log k)\sum_{i\geq 1}|N^+_{[B'_i,B_i)}(U_i)|}^{\mathcal{N}_3}.$$

For $\mathcal{N}_1+\mathcal{N}_2$, because

$$\sum_{i\geq 1}|N^+_{[B'_{i-1},B_i)}(U_i)|+\sum_{i\geq 1}|N^+_{[B_i,B)}(U_i)| = \sum_{i\geq 1}|N^+_{[B'_{i-1},B)}(U_i)| \leq |N^+_{[B'_0,B)}(U)|,$$

and $B'_0 \geq \min_{x\in S} d(x)$, it is bounded by

$$C(t+(l-1)\log k)|N^+_{[\min_{x\in S} d(x),B)}(U).$$

$\mathcal{N}_3$ is bounded by

$$C(\log k)|N^+_{[B'_0,B)}(U)| \leq C(\log k)|N^+_{[\min_{x\in S} d(x),B)}(U)|.$$

Therefore $\mathcal{N} \leq C(t+l\log k)|N^+_{[\min_{x\in S} d(x),B)}(U)|$. □

References

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

[2] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. 2022. Negative-Weight Single-Source Shortest Paths in Near-linear Time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 600–611. doi:10.1109/FOCS54457.2022.00063

[3] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. 1973. Time bounds for selection. *J. Comput. System Sci.* 7, 4 (1973), 448–461. doi:10.1016/S0022-0000(73)80033-9

[4] Karl Bringmann, Alejandro Cassis, and Nick Fischer. 2023. Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster! . In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 515–538. doi:10.1109/FOCS57990.2023.00038

[5] Shiri Chechik, Haim Kaplan, Mikkel Thorup, Or Zamir, and Uri Zwick. 2016. Bottleneck Paths and Trees and Deterministic Graphical Games.. In *STACS (LIPIcs, Vol. 47)*, Nicolas Ollinger and Heribert Vollmer (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Wadern, Germany, 27:1–27:13. doi:10.4230/LIPIcs.STACS.2016.27

[6] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 612–623. doi:10.1109/FOCS54457.2022.00064

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

[8] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. 1988. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation. *Commun. ACM* 31, 11 (Nov 1988), 1343–1354. doi:10.1145/50087.50096

[9] Ran Duan, Kaifeng Lyu, Hongxun Wu, and Yuanhang Xie. 2018. Single-Source Bottleneck Path Algorithm Faster than Sorting for Sparse Graphs. *CoRR* abs/1808.10658 (2018), 15 pages. doi:10.48550/arXiv.1808.10658

[10] R. Duan, J. Mao, X. Shu, and L. Yin. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 484–492. doi:10.1109/FOCS57990.2023.00035

[11] Jeremy T. Fineman. 2024. Single-Source Shortest Paths with Negative Real Weights in $\tilde{O}(mn^{8/9})$ Time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing* (Vancouver, BC, Canada) *(STOC 2024)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/3618260.3649614

[12] 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

[13] M. L. Fredman and R. E. Tarjan. 1987. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *JACM* 34, 3 (1987), 596–615. doi:10.1145/28869.28874

[14] Michael L. Fredman and Dan E. Willard. 1993. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.* 47, 3 (1993), 424–436. doi:10.1016/0022-0000(93)90040-4

[15] Michael L. Fredman and Dan E. Willard. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.* 48, 3 (1994), 533–551. doi:10.1016/S0022-0000(05)80064-9

[16] Harold N Gabow and Robert E Tarjan. 1988. Algorithms for two bottleneck optimization problems. *Journal of Algorithms* 9, 3 (1988), 411–417. doi:10.1016/0196-6774(88)90031-4

[17] Leo J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE Computer Society, Los Alamitos, CA, USA, 8–21. doi:10.1109/SFCS.1978.3

[18] Bernhard Haeupler, Richard Hladik, Vaclav Rozhon, Robert E. Tarjan, and Jakub Tetek. 2024. Universal Optimality of Dijkstra Via Beyond-Worst-Case Heaps . In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 2099–2130. doi:10.1109/FOCS61266.2024.00125

[19] Torben Hagerup. 2000. Improved Shortest Paths on the Word RAM. In *Automata, Languages and Programming*, Ugo Montanari, José D. P. Rolim, and Emo Welzl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–72. doi:10.1007/3-540-45022-X_7

[20] Yufan Huang, Peter Jin, and Kent Quanrud. 2025. Faster single-source shortest paths with negative real weights via proper hop distance. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, USA, 5239–5244. doi:10.1137/1.9781611978322.178

[21] David R. Karger, Philip N. Klein, and Robert E. Tarjan. 1995. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *J. ACM* 42, 2 (Mar 1995), 321–328. doi:10.1145/201019.201022

[22] Seth Pettie and Vijaya Ramachandran. 2005. A Shortest Path Algorithm for Real-Weighted Undirected Graphs. *SIAM J. Comput.* 34, 6 (2005), 1398–1431. doi:10.1137/S0097539702419650

[23] Rajeev Raman. 1996. Priority Queues: Small, Monotone and Trans-Dichotomous. In *Proceedings of the Fourth Annual European Symposium on Algorithms (ESA '96)*. Springer-Verlag, Berlin, Heidelberg, 121–137. doi:10.1007/3-540-61680-2_51

[24] Rajeev Raman. 1997. Recent Results on the Single-Source Shortest Paths Problem. *SIGACT News* 28, 2 (Jun 1997), 81–87. doi:10.1145/261342.261352

[25] Mikkel Thorup. 1999. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. ACM* 46, 3 (May 1999), 362–394. doi:10.1145/316542.316548

[26] Mikkel Thorup. 2000. Floats, Integers, and Single Source Shortest Paths. *J. Algorithms* 35, 2 (May 2000), 189–201. doi:10.1006/jagm.2000.1080

[27] Mikkel Thorup. 2000. On RAM Priority Queues. *SIAM J. Comput.* 30, 1 (2000), 86–109. doi:10.1137/S0097539795288246

[28] Mikkel Thorup. 2004. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. System Sci.* 69, 3 (2004), 330–353. doi:10.1016/j.jcss.2004.04.003 Special Issue on STOC 2003.

[29] Virginia Vassilevska Williams. 2010. Nondecreasing Paths in a Weighted Graph or: How to Optimally Read a Train Schedule. *ACM Trans. Algorithms* 6, 4, Article 70 (Sep 2010), 24 pages. doi:10.1145/1824777.1824790

[30] Andrew Chi-Chih Yao. 1975. An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees. *Inform. Process. Lett.* 4, 1 (1975), 21–23. doi:10.1016/0020-0190(75)90056-3

Received 2024-11-04; accepted 2025-02-01