Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time

Mikkel Thorup
AT&T Labs—Research

The single source shortest paths problem (SSSP) is one of the classic problems in algorithmic graph theory: given a positively weighted graph G with a source vertex s, find the shortest path from s to all other vertices in the graph.

Since 1959 all theoretical developments in SSSP for general directed and undirected graphs have been based on Dijkstra's algorithm, visiting the vertices in order of increasing distance from s. Thus, any implementation of Dijkstra's algorithm sorts the vertices according to their distances from s. However, we do not know how to sort in linear time.

Here, a deterministic linear time and linear space algorithm is presented for the undirected single source shortest paths problem with positive integer weights. The algorithm avoids the sorting bottle-neck by building a hierarchical bucketing structure, identifying vertex pairs that may be visited in any order.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Bounded-action devices (random access machines)*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures, Sorting and searching*; G.2.2 [**Discrete Mathematics**]: Graph algorithms—*Computations on discrete structures, Sorting and searching*

General Terms: Algorithms

Additional Key Words and Phrases: shortest paths, RAM algorithms

---

1. INTRODUCTION

Let $G = (V, E)$, $|V| = n$, $|E| = m$, be an undirected connected graph with a positive integer edge weight function $\ell : E \to \mathbb{N}$ and a distinguished source vertex $s \in V$. If $(v, w) \notin E$, define $\ell(v, w) = \infty$. *The single source shortest path problem (SSSP)* is for every vertex v to find the distance $d(v) = dist(s, v)$ from s to v. This is one of the classic problems in algorithmic graph theory. In this paper, we present a deterministic linear time and linear space algorithm for undirected SSSP with

A preliminary short version of this paper appeared in *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science (FOCS'97)*, 1997, 12–21.
Most of this work was done while the author was at the University of Copenhagen.
Address: 180 Park Avenue, Florham Park, NJ 07932. E-mail: `mthorup@research.att.com`

positive integer weights. So far a linear time SSSP algorithm has only been known for planar graphs [Henzinger et al. 1997].

1.1 Model

Our algorithm runs on a RAM, which models what we program in imperative programming languages such as C. The memory is divided into addressable words of length ω. Addresses are themselves contained in words, so $\omega \geq \log n$. Moreover, we have a constant number of registers, each with capacity for one word. The basic assembler instructions are: conditional jumps, direct and indirect addressing for loading and storing words in registers, and some computational instructions, such as comparisons, addition, and multiplication, for manipulating words in registers. The space complexity is the maximal memory address used, and the time complexity is the number of instructions performed. All weights and distances are assumed to be integers represented as binary strings. For simplicity, we assume they all weights and distances each fit in one word so that the input and output size is $O(m)$; otherwise the output size may be asymptotically larger than the input size, say, if we start with a huge weight when leaving the source. Our algorithm is easily modified to run in time and space linear in the output size for arbitrarily large integer weights.

Within the RAM model, one may prefer to use only the AC^0 operations among the computational instructions. A computational instruction is an AC^0 operation if it is computable by an $\omega^{O(1)}$-sized constant depth circuit with $O(\omega)$ input and output bits. In the circuit we may have negation and and-gates and or-gates with unbounded fan-in. Addition, shift, and bit-wise boolean operations are all AC^0 operations. On the other hand, multiplication is not. Our linear time algorithm does use multiplication, but if we restrict ourselves to AC^0 operations, it can be implemented in $O(\alpha(m,n)m)$ time.

In contrast to the RAM, we have the pointer machine model, disallowing address arithmetic, and hence bucketing which is essential to our algorithm. Also, we have the comparison based model where weights may only be compared. Of all the algorithms mentioned below, it is only those from [Dijkstra 1959; Williams 1964; Fredman and Tarjan 1987] that work in any of these two restricted models. All the other algorithms assume a RAM model with integer weights, like ours.

1.2 History

Since 1959, all theoretical developments in SSSP for general directed or undirected graphs have been based on Dijkstra's algorithm [Dijkstra 1959]. For each vertex we have a super distance $D(v) \geq d(v)$. Moreover, we have a set $S \subseteq V$ such that $\forall v \in S : D(v) = d(v)$ and $\forall v \notin S : D(v) = \min_{u \in S}\{d(u) + \ell(u,v)\}$. Initially, $S = \{s\}$, $D(s) = d(s) = 0$ and $\forall v \neq s : D(v) = \ell(s,v)$. In each round of the algorithm, we *visit* a vertex $v \notin S$ minimizing $D(v)$. Then, as proved by Dijkstra, $D(v) = d(v)$, so we can move v to S. Consequently, for all $(v,w) \in E$, if $D(v) + \ell(v,w) < D(w)$, we have to decrease $D(w)$ to $D(v) + \ell(v,w)$. Dijkstra's algorithm finishes when $S = V$, returning $D(\cdot) = d(\cdot)$.

The complexity of Dijkstra's algorithm is determined by the $n-1$ times that we find a vertex $v \in V \setminus S$ minimizing $D(v)$ and the at most m times we decrement some $D(w)$. All subsequent theoretical developments in SSSP for general graphs have been based on various speed-ups and trade-offs in priority queues/heaps supporting these two operations. If we just find the minimum by searching all vertices, we solve SSSP in $O(n^2+m)$ time. Applying Williams' heap [Williams 1964], we get $O(m \log n)$ time. Fredman and Tarjan's Fibonacci heaps [Fredman and Tarjan 1987] had SSSP as a prime application, and reduced the running time to $O(m+n\log n)$. They noted that this was an optimal implementation of Dijkstra's algorithm in a comparison model since Dijkstra's algorithm visits the vertices in sorted order. Using Fredman and Willard's fusion trees, we get an $O(m\sqrt{\log n})$ randomized bound [Fredman and Willard 1993]. Their later atomic heaps give an $O(m + n \log n/ \log\log n)$ bound [Fredman and Willard 1994]. More recently, Thorup's priority queues gave an $O(m \log\log n)$ bound and an $O(m + n\sqrt{\log n}^{1+\varepsilon})$ bound [Thorup 1996]. These bounds are randomized assuming that we want linear space. Finally, Raman has obtained an $O(m + n\sqrt{\log n \log\log n})$ bound in deterministic linear space [Raman 1996] and an $O(m + n\sqrt[3]{\log n}^{1+\varepsilon})$ randomized bound [Raman 1997].

There has also been a substantial development based on the maximal edge weight C, again assuming integer edge weights, each fitting in one word. First note that using van Emde Boas's general search structure [van Emde Boas 1977; van Emde Boas et al. 1977; Melhorn and Nähler 1990], and bucketing according to $\lfloor D(v)/n \rfloor$, we get an $O(m \log\log C)$ algorithm for SSSP. Ahuja, Melhorn, Orlin, and Tarjan have found a priority queue for SSSP giving a running time of $O(m + n\sqrt{\log C})$ [Ahuja et al. 1990]. Recently, this has been improved by Cherkassky, Goldberg, and Silverstein to $O(m + n\sqrt[3]{\log C \log\log C})$ expected time [Cherkassky et al. 1997], and a further improvement to $O(m + n(\log C)^{1/4+\varepsilon})$ has been presented by Raman [Raman 1997].

For the case of undirected graphs, we end the above quest by presenting an $O(m)$ algorithm.

1.3 Techniques

As observed in [Fredman and Tarjan 1987], implementing Dijkstra's algorithm in linear time would require sorting in linear time. In fact, the converse also holds, in that Thorup has shown that linear time sorting implies that Dijkstra's algorithm can be implemented in linear time [Thorup 1996]. In this paper, we solve the undirected version of SSSP deterministically in $O(m)$ time and space. Since we do not know how to sort in linear time, this implies that we are deviating from Dijkstra's algorithm in that we do not visit the vertices in order of increasing distance from s. Our algorithm is based on a hierarchical bucketing structure, where the bucketing helps identifying vertex pairs that can be visited in any order. It should be mentioned that using bucketing is not in itself new in connection with SSSP. In 1978 Dinitz [Dinic 1978] argued that if δ is the minimum edge weight, then in Dijkstra's algorithm, we can visit any vertex v minimizing $\lfloor D(v)/\delta \rfloor$. Thus bucketing according to $\lfloor D(v)/\delta \rfloor$, we can visit the vertices in the minimal bucket in any order. In this paper, we are in some sense applying Dinitz's idea recursively, identifying cuts where the minimum weight δ of the crossing edges is large.

1.4 Contents

The paper is divided as follows. After the preliminaries in Section 2, in Section 3, we present the general idea of using bucketing to avoid the sorting bottle-neck. This idea is then implemented recursively over Sections 4-8, allowing us to conclude in Section 9 with a linear time algorithm for the undirected SSSP problem with positive integer weights. Finally, in Appendix A, we discuss how to get a linear time algorithm if the weights are not integers but floating point numbers.

2. PRELIMINARIES

Throughout the paper, we will assume that $G, V, E, \ell, s, D, d, S$ are as defined in the introduction in the description of Dijkstra's algorithm. In particular, concerning S and D, $\forall v \in S : D(v) = d(v)$ and $\forall v \in V \setminus S : D(v) = \min_{u \in S}\{d(u) + \ell(u,v)\}$. As in Dijkstra's algorithm, initially, $S = \{s\}$, $D(s) = d(s) = 0$, and $\forall v \neq s : D(v) = \ell(s,v)$. We also inherit that we can *visit* a vertex $v \notin S$ only if $D(v) = d(v)$. Visiting v implies that v is moved to S and that for all $(v,w) \in E$, $w \notin S$, we set $D(w) = \min\{D(w), D(v) + \ell(v,w)\}$. As for Dijkstra's algorithm, we have:

LEMMA 1. *If $v \in V \setminus S$ minimizes $D(v)$, $D(v) = d(v)$.*

PROOF. Let u be the first vertex outside S on a shortest path from s to v. Then $D(u) = d(u)$ by definition of S. Hence we have $D(v) \geq d(v) \geq d(u) = D(u) \geq D(v)$, implying $D(v) = d(v)$. □

However, in contrast to Dijkstra's algorithm, we may visit a vertex $v \notin S$ that does not minimize $D(v)$. Nevertheless, we inherit the following additional result from Dijkstra's algorithm:

LEMMA 2. $\min D(V \setminus S) = \min d(V \setminus S)$ *is non-decreasing.*

PROOF. If $S = V$, $\min D(V \setminus S) = \min d(V \setminus S) = \min \emptyset = \infty$. Otherwise, there is a $v \in V \setminus S$ minimizing $d(v)$. Let u be the first vertex outside S on a shortest path from s to v. Then $D(u) = d(u)$ and $d(u) \leq d(v)$. However, v minimized $d(v)$, so $d(u) = d(v)$. Hence, we have $\min D(V \setminus S) \leq D(u) = d(u) = \min d(V \setminus S)$. On the other hand, $D(w) \geq d(w)$ for all $w \in V$, so $\min D(V \setminus S) \geq \min d(V \setminus S)$. Hence, we conclude that $\min D(V \setminus S) = \min d(V \setminus S)$. Now, $\min d(V \setminus S)$ is non-decreasing because S is only increased and $d(w)$ does not change for any $w \in V$. Thus we are minimizing over a smaller and smaller set of constant d-values, implying that $\min d(V \setminus S)$ can only increase. □

We will let ω denote the word length. We will write $\lfloor x/2^i \rfloor$ as $x \gg i$ to emphasize that it may be calculated simply by shifting the i least significant bits out to the right. Note that $x \leq y \Rightarrow x \gg i \leq y \gg i$ while $x < y \Leftarrow x \gg i < y \gg i$. If f is a function on the elements from a set X, we let $f(X)$ denote $\{f(x) | x \in X\}$. We also adopt the standard that $\min \emptyset = \infty$. We define '$\gg$' to have lower precedence than 'min', '+', and '−'. For example, if $W \subseteq V$, $\min D(W) \gg i - 1 = (\min\{D(w) | w \in W\}) \gg (i-1)$.

By a *bucket*, we refer to a dynamic set B into which elements can be inserted and deleted, and from which we can pick out an unspecified element. Each operation should be supported in constant time. A bucket could, for example, be implemented as a doubly-linked list where one can just insert and pick elements from the head.

Using the indirect addressing of the RAM, we typically create an array $B(1...l)$ of buckets. Then we can insert and pick elements from an arbitrary bucket $B(i)$ in constant time. Also, in constant time, we can delete an element from whatever bucket it is in.

3. AVOIDING THE SORTING BOTTLENECK

We will now briefly indicate how the sorting bottleneck can be avoided. That is, we will discuss some simple conditions for $D(v) = d(v)$ where possibly $D(v) > \min D(V \setminus S)$. The resulting algorithm is far from efficient but over the subsequent sections, we will apply the ideas recursively, so as to achieve a linear time solution to the SSSP problem.

Lemma 3. *Suppose the vertex set V divides into disjoint subsets $V_1, ..., V_k$ and that all edges between the subsets have length at least δ. Further suppose for some $i, v \in V_i \setminus S$ that $D(v) = \min D(V_i \setminus S) \leq \min D(V \setminus S) + \delta$. Then $d(v) = D(v)$.*

Proof. To see that $d(v) = D(v)$, let u be the first vertex outside S on a shortest path from s to v. Then $d(u) = D(u)$ follows by definition of S. If $u \in V_i$, as in the proof of Lemma 1, we have $D(v) \geq d(v) \geq d(u) = D(u) \geq D(v)$, implying $d(u) = D(u)$, as desired. Now, suppose $u \notin V_i$. Since all edges from $V \setminus V_i$ to V_i are of length δ, we have $D(v) \geq d(v) \geq d(u) + \delta = D(u) + \delta$. On the other hand $D(u) + \delta \geq \min D(V \setminus S) + \delta \geq \min D(V_i \setminus S) = D(v)$, so we conclude that we have equality everywhere. In particular, this allows us to conclude that $d(v) = D(v)$. □

Approaching a first simple SSSP bucketing algorithm, suppose $\delta = 2^\alpha$. Then $\min D(V_i \setminus S) \leq \min D(V \setminus S) + \delta$ is implied by $\min D(V_i \setminus S) \gg \alpha \leq \min D(V \setminus S) \gg \alpha$. Algorithmically, we will now bucket each $i \in \{1, ..., k\}$ according to $\min D(V_i \setminus S) \gg \alpha$. That is, we have an array B of buckets where i belongs in bucket $B(\min D(V_i \setminus S) \gg \alpha)$. Note that $\min D(V \setminus S) \gg \alpha = \min_i(\min D(V_i \setminus S) \gg \alpha)$. Hence, if i is in the smallest indexed non-empty bucket, $\min D(V_i \setminus S) \gg \alpha = \min D(V \setminus S) \gg \alpha$. In more algorithmic terms, suppose ix is maintained as $\leq$ the smallest index of a non-empty bucket. If $i \in B(ix)$ and $v \in V_i \setminus S$ minimizes $D(v)$, then $D(v) = \min D(V_i \setminus S) \leq \min D(V \setminus S) + \delta$, so $D(v) = d(v)$ by Lemma 3, and hence v can be visited.

For the maintenance of ix recall that $\min D(V \setminus S)$ is non-decreasing by Lemma 2. Hence $\min D(V \setminus S) \gg \alpha$ is non-decreasing, so ix will never need to be decreased. Also, note that $D(v) < \infty$ implies that there is a path in G from s to v of length $D(v)$, and hence $D(v) \leq \sum_{e \in E} \ell(e)$. Consequently, the maximum index $< \infty$ of any non-empty bucket is bounded by $\Delta = \sum_{e \in E} \ell(e) \gg \alpha$. That is, the only bucket indices used are $0, ..., \Delta, \infty$. Viewing ∞ as the successor of Δ, we only need an array of $\Delta + 2$ buckets.

Based on the above discussion, we get the following SSSP algorithm.

Algorithm A. Solves the SSSP problem where V is partitioned into subsets $V_1, ..., V_k$ where all edges between the subsets have length at least 2^α.

A.1. $S \leftarrow \{s\}$; $D(s) \leftarrow 0$; for all $v \neq s : D(v) \leftarrow \ell(s, v)$

A.2. for $ix \leftarrow 0, 1, ..., \Delta, \infty$, $B(ix) \leftarrow \emptyset$

A.3. for $i \leftarrow 1, ..., k$, add i to $B(\min D(V_i \setminus S) \gg \alpha)$

A.4. for $ix \leftarrow 0$ to Δ,
A.4.1. while $B(ix) \neq \emptyset$,
A.4.1.1. pick $i \in B(ix)$
A.4.1.2. pick $v \in V_i \setminus S$ minimizing $D(v)$
A.4.1.3. for all $(v, w) \in E, w \notin S$,
A.4.1.3.1. let j be such that $w \in V_j$
A.4.1.3.2. $D(w) = \min\{D(w), D(v) + \ell(v, w)\}$: if $\min D(V_j \setminus S) \gg \alpha$ is thereby decreased, move j to $B(\min D(V_j \setminus S) \gg \alpha)$.
A.4.1.4. $S \leftarrow S \cup \{v\}$; if $\min D(V_i \setminus S) \gg \alpha$ is thereby increased, move i to $B(\min D(V_i \setminus S) \gg \alpha)$.

The complexity of the above algorithm is $O(m + \Delta)$ plus the cost of maintaining $\min D(V_i \setminus S)$ for each i. The latter will essentially be done recursively, and $\Delta = \sum_{e \in E} \ell(e) \gg \alpha$ will be kept small by choosing α large.

4. THE COMPONENT HIERARCHY

We are now going to present a recursive condition for concluding $D(v) = d(v)$ which will later be used in a linear time SSSP algorithm. It is based on a *component hierarchy* defined as follows. By G_i we denote the subgraph of G whose edge set is the edges e from G with $\ell(e) < 2^i$. Then G_0 consists of singleton vertices. Recall that ω denotes the word length. Hence all edge lengths and distances are $< 2^\omega$, so $G_\omega = G$. On level i in the component hierarchy, we have the components (maximal connected subgraphs) of G_i. The component on level i containing v is denoted $[v]_i$. The children of $[v]_i$ are the components $[w]_{i-1}$ with $[w]_i = [v]_i$, i.e. with $w \in [v]_i$.

Lemma 4. *If $[v]_i \neq [w]_i$, $dist(v, w) \geq 2^i$.*

Proof. Since $[v]_i \neq [w]_i$, any path from v to w contains an edge of length $\geq 2^i$. □

By $[v]_i^-$ we will denote $[v]_i \setminus S$, noting that $[v]_i^-$ may not be connected. We say that $[v]_i$ is a *min-child* of $[v]_{i+1}$ if $\min(D([v]_i^-) \gg i = \min(D([v]_{i+1}^-) \gg i$. We say that $[v]_i$ is *minimal* if $[v]_i^- \neq \emptyset$ and for $j = i, \ldots, b-1$, $[v]_j$ is a min-child of $[v]_{j+1}$. The requirement $[v]_i^- \neq \emptyset$ is only significant if $V = S$. If $V = S$, $\min D([v]_i^-) = \infty$ for all $[v]_i$, and hence all $[v]_i$ would be minimal without the requirement $[v]_i^- \neq \emptyset$; now, no $[v]_i$ is minimal if $V = S$.

Below, in Lemma 8, we will show that $D(v) = d(v)$ if $[v]_0$ is minimal. If $v \in V \setminus S$ minimizes $D(v)$, as in Dijkstra's algorithm, $\forall i : \min D([v]_i^-) \gg i = D([v]_{i+1}^-) \gg i = D(v) \gg i$, so $[v]_0$ is minimal. The point is that $[v]_0$ may be minimal even if $D(v)$ is not minimized, thus providing us with a more general condition for $D(v) = d(v)$ than the one used in Dijkstra's algorithm.

The condition for $D(v) = d(v)$ that $[v]_0$ is minimal also holds for directed graphs. Our efficient use of the condition hinges, however, on the property of undirected graphs that $\forall u, w \in [v]_i : dist(u, v) \leq \sum_{e \in [v]_i} \ell(e)$. We shall return to the latter property in Section 6, where it will be used to limit the size of an underlying bucketing structure.

We will now prove several properties of the component hierarchy, one of which is that $D(v) = d(v)$ if $[v]_0$ is minimal. All these properties will prove relevant in our later algorithmic developments.

LEMMA 5. *If $v \notin S$, $[v]_i$ is minimal, and $i \leq j \leq \omega$,* $\min D([v]_i^-) \gg j-1 = \min D([v]_j^-) \gg j-1$.

PROOF. The proof is by induction on j. If $j = i$, the statement is vacuously true. If $j > i$, inductively, $\min D([v]_i^-) \gg j-2 = \min D([v]_{j-1}^-) \gg j-2$, implying $\min D([v]_i^-) \gg j-1 = \min D([v]_{j-1}^-) \gg j-1$. Moreover, the minimality of $[v]_i$ implies that $[v]_{j-1}$ is a min-child of $[v]_j$, hence that $\min D([v]_{j-1}^-) \gg j-1 = \min D([v]_j^-) \gg j-1$. □

LEMMA 6. *Suppose $v \notin S$ and there is a a shortest path to v where the first vertex u outside S is in $[v]_i$. Then $d(v) \geq \min D([v]_i^-)$.*

PROOF. Since u is the first vertex outside S on our path, $D(u)$ is a lower bound on its length, so $D(u) \leq d(v)$. Moreover, since $u \in [v]^-$, $D(u) \geq \min D([v]_i^-)$. □

LEMMA 7. *Suppose $v \notin S$ and $[v]_{i+1}$ is minimal. If there is no shortest path to v where the first vertex outside S is in $[v]_i$, $d(v) \gg i > \min D([v]_{i+1}^-) \gg i$.*

PROOF. Among all shortest paths to v, pick one P so that the first vertex u outside S is in $[v]_k$ with k minimized. Then $k > i$ and $d(v) = \ell(P) = D(u) + dist(u,v)$.

We prove the statement of the lemma by induction on $\omega - i$. If $u \notin [v]_{i+1}$, we have $i+1 < \omega$ and the minimality of $[v]_{i+1}$ implies minimality of $[v]_{i+2}$. Hence, by induction, $d(v) \gg i+1 > \min D([v]_{i+2}^-) \gg i+1$. By minimality of $[v]_{i+1}$, $\min D([v]_{i+2}^-) \gg i+1 = \min D([v]_{i+1}^-) \gg i+1$. Thus, $d(v) \gg i+1 > \min D([v]_{i+1}^-) \gg i+1$, implying $d(v) \gg i > \min D([v]_{i+1}^-) \gg i$.

If $u \in [v]_{i+1}^-$, $D(u) \gg i \geq \min D([v]_{i+1}^-) \gg i$. Moreover, since $u \notin [v]_i$, by Lemma 4, $dist(u,v) \geq 2^i$. Hence, $d(v) \gg i = (D(u) + dist(u,v)) \gg i \geq (\min D([v]_{i+1}^-) \gg i) + 1$. □

We are now in the position to prove that the minimality of $[v]_0$ implies $D(v) = d(v)$:

LEMMA 8. *If $v \notin S$ and $[v]_i$ is minimal, $\min D([v]_i^-) = \min d([v]_i^-)$. In particular, $D(v) = d(v)$ if $[v]_0 = \{v\}$ is minimal.*

PROOF. Since $D(w) \geq d(w)$ for all w, $\min D([v]_i^-) \geq \min d([v]_i^-)$. Viewing v as an arbitrary vertex in $[v]_i$, it remains to show that $d(v) \geq \min D([v]_i^-)$.

Among all shortest paths to v, pick one P so that the first vertex u outside S is in $[v]_i$ if possible. If $u \in [v]_i$, Lemma 6 gives the result directly. If $u \notin [v]_i$, by Lemma 7, $d(v) \gg i > \min D([v]_{i+1}^-) \gg i$. However, $[v]_i$ is min-child of $[v]_{i+1}$, so $\min D([v]_{i+1}^-) \gg i = \min D([v]_i^-) \gg i$. Thus $d(v) \gg i > \min D([v]_i^-) \gg i$, implying $d(v) > \min D([v]_i^-)$. □

The above lemma gives us our basic condition for visiting vertices, moving them to S. Our algorithms will need one more consequence of Lemma 6 and 7.

LEMMA 9. *If $v \notin S$ and $[v]_i$ is not minimal but $[v]_{i+1}$ is minimal, then* $\min d([v]_i^-) \gg i > \min D([v]_{i+1}^-) \gg i$.

PROOF. Consider any $w \in [v]_i^-$. If there is no shortest path to w where the first vertex outside S is in $[v]_i$, $d(w) \gg i > \min D([v]_{i+1}^-) \gg i$ follows directly from Lemma 7. Otherwise, by Lemma 6, $d(w) \geq \min D([v]_i^-)$. Moreover, the non-minimality of $[v]_i$ implies $\min D([v]_i^-) \gg i > \min D([v]_{i+1}^-) \gg i$. Hence, we conclude that $d(w) \gg i > \min D([v]_{i+1}^-) \gg i$ for all $w \in [v]_i^-$, as desired. □

5. VISITING MINIMAL VERTICES

In this section, we will discuss the basic dynamics of visiting vertices v with $[v]_0$ minimal. First we show a series of lemmas culminating in Lemma 13, stating that if $[v]_i$ has once been minimal, $\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i$ in all future. Based on this, we will present an abstract SSSP algorithm displaying the basic order in which we want to visit the vertices of G in our later linear time algorithm.

Definition 1. In the rest of this paper, *visiting a vertex v requires that $[v]_0 = \{v\}$ is minimal.* When v is visited, it is moved to S, setting $D(w)$ to $\min\{D(w), D(v) + \ell(v,w)\}$ for all $(v,w) \in E$.

Note by Lemma 8 that $D(v) = d(v)$ whenever we visit a vertex v.

Lemma 10. *For all $[v]_i$, $\max d([v]_i \setminus [v]_i^-) \gg i - 1 \leq \min d([v]_i^-) \gg i - 1$.*

Proof. Since d is a constant function, it suffices to show that just before $w \in [v]_i^-$ is visited, $d(w) \gg i-1 = \min d([v]_i^-) \gg i-1$. By definition, $[w]_0$ is minimal just before the visit, so by Lemma 5, $D(w) \gg i-1 = \min D([w]_0^-) \gg i-1 = \min D([w]_i^-) \gg i-1$. On the other hand, by Lemma 8, $D(w) = d(w)$ and $\min D([w]_i^-) = \min d([w]_i^-)$, so we conclude that $d(w) \gg i-1 = \min d([v]_i^-) \gg i-1$, as desired. □

In the following, we will frequently study the situation before and after the event of visiting some vertex. We will then use the notation $\langle e\rangle^b$ and $\langle e\rangle^a$ to denote that the expression e should be evaluated before respectively after the event. By Lemma 10, if $j \geq i-1$, $\langle \min d([v]_i^-) \gg j\rangle^a \geq \langle \min d([v]_i^-) \gg j\rangle^b$. Hence, since $\forall w : D(w) \geq d(w)$,

$$\begin{aligned}\langle \min D([v]_i^-) \gg j\rangle^b = \langle \min d([v]_i^-) \gg j\rangle^b \;\Rightarrow\\ \langle \min D([v]_i^-) \gg j\rangle^a \geq \langle \min D([v]_i^-) \gg j\rangle^b\end{aligned} \tag{1}$$

Lemma 11. *Suppose $\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i$ and that visiting a vertex $w \in V \setminus S$ changes $\min D([v]_i^-) \gg i$. Then $w \in [v]_i$ and if $[v]_i^-$ is not emptied, the change in $\min D([v]_i^-) \gg i$ is an increase by one.*

Proof. We are studying the event of visiting the vertex w. By assumption, $\langle \min D([v]_i^-) \gg i\rangle^b = \langle \min d([v]_i^-) \gg i\rangle^b$. Hence, by (1), $\langle \min D([v]_i^-) \gg i\rangle^a \geq \langle \min D([v]_i^-) \gg i\rangle^b$. By assumption, $\langle \min D([v]_i^-) \gg i\rangle^a \neq \langle \min D([v]_i^-) \gg i\rangle^b$, so $\langle \min D([v]_i^-) \gg i\rangle^a > \langle \min D([v]_i^-) \gg i\rangle^b$. Since D-values never increase, we conclude $\langle [v]_i^-\rangle^a \subset \langle [v]_i^-\rangle^b$, hence that $w \in \langle [v]_i^-\rangle^b$ and $\langle [v]_i^-\rangle^a = \langle [v]_i^-\rangle^b \setminus \{w\}$.

Suppose $\langle [v]_i^-\rangle^a$ is non-empty. Since $[v]_i$ is connected, there must be an edge (u,x) in $[v]_i$ with $u \notin \langle [v]_i^-\rangle^a$ and $x \in \langle [v]_i^-\rangle^a$. We will now argue that

$$d(u) \gg i \leq \min \langle D([v]_i^-) \gg i\rangle^b. \tag{2}$$

If $u \notin \langle [v]_i^-\rangle^b$, (2) follows from Lemma 10. Otherwise $u = w$. By Lemma 8 and Lemma 5, the minimality of $[u]_0 = [w]_0$ implies $d(u) \gg i = D(u) \gg i = \langle \min D([v]_i^-) \gg i\rangle^b$. Thus (2) follows.

Based on (2), since $\ell(u,x) < 2^i$, we conclude

$$\langle \min D([v]_i^-) \gg i\rangle^a \le \langle D(x) \gg i\rangle^a \le (d(u)+\ell(u,x)) \gg i \le \langle \min D([v]_i^-) \gg i\rangle^b + 1.$$

□

In connection with Lemma 11, it should be noted that with directed graphs, the increase could be by more than one. This is the first time in this paper, that we use the undirectedness.

LEMMA 12. *If $[v]_i$ is minimal, it remains minimal until $\min D([v]_i^-) \gg i$ is increased, in which case $\min d([v]_i^-) \gg i$ is also increased.*

PROOF. Suppose $[v]_i$ is minimal, but visiting some vertex w stops $[v]_i$ from being minimal. If w was the last vertex not in S, the visit increases both $\min D([v]_i^-)$ and $\min d([v]_i^-)$ to ∞. Otherwise, some ancestor of $[v]_i$ is minimal, and we pick the smallest j such that $[v]_{j+1}$ is minimal. Moreover, we pick $u \in [v]_{j+1}^-$ such that $[u]_j$ is a min-child of $[v]_{j+1}$. Hence $[u]_j$ is minimal while $[v]_j$ is not minimal.

Before the visit to w, $[v]_i$ was minimal, so $\langle \min D([v]_i^-) \gg j\rangle^b = \langle \min D([v]_{j+1}^-) \gg j\rangle^b$ by Lemma 5. Also, $[v]_{j+1}$ was minimal, so by Lemma 8 and (1), $\langle \min D([v]_{j+1}^-) \gg j\rangle^a \ge \langle \min D([v]_{j+1}^-) \gg j\rangle^b$.

After the visit, since $[v]_{j+1}$ is minimal and $[v]_j$ is not a min-child of $[v]_{j+1}$, by Lemma 9, $\langle \min d([v]_j^-) \gg j\rangle^a > \langle \min D([v]_{j+1}^-) \gg j\rangle^a$. Thus

$$\begin{aligned}
\langle \min D([v]_i^-) \gg j\rangle^a &\ge \langle \min d([v]_i^-) \gg j\rangle^a \\
&\ge \langle \min d([v]_j^-) \gg j\rangle^a \\
&> \langle \min D([v]_{j+1}^-) \gg j\rangle^a \\
&\ge \langle \min D([v]_{j+1}^-) \gg j\rangle^b \\
&= \langle \min D([v]_i^-) \gg j\rangle^b \\
&= \langle \min d([v]_i^-) \gg j\rangle^b.
\end{aligned}$$

□

LEMMA 13. *If $[v]_i$ has once been minimal, in all future,*

$$\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i. \tag{3}$$

PROOF. First time $[v]_i$ turns minimal, (3) gets satisfied by Lemma 8. Now, suppose (3) is satisfied before visiting some vertex w. Since $\forall u : D(u) \ge d(u)$, (3) can only be violated by an increase in $\min D([v]_i^-)$. If $\min D([v]_i^-) \gg i$ is increased, by Lemma 11, $w \in [v]_i$ and the increase is by one. Visiting w requires that $[w]_0$ is minimal, hence that $[w]_i = [v]_i$ is minimal. If $[v]_i$ is minimal after the visit, (3) follows from Lemma 8. Also, if $[v]_i^-$ is emptied, (3) follows with $\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i = \infty$. If $[v]_i$ becomes non-minimal and $[v]_i^-$ is not emptied, by Lemma 12, $\min d([v]_i^-) \gg i$ is also increased. Since $\min d([v]_i^-) \gg i \le \min D([v]_i^-) \gg i$ and $\min D([v]_i^-) \gg i$ was increased by one, we conclude that (3) is restored. □

We are now ready to derive an algorithm for the undirected SSSP problem based on the component hierarchy. The algorithm is so far inefficient, but it shows the ordering in which we intend to visit the vertices in a later linear time algorithm. As our main routine, we have:

Algorithm B. *SSSP* is given an input graph $G = (V, E)$ with weight function ℓ and distinguished vertex s. It outputs D with $D(v) = d(v) = dist(s, v)$ for all $v \in V$.

B.1. $S \leftarrow \{s\}$

B.2. $D(s) \leftarrow 0$, for all $v \neq s : D(v) \leftarrow \ell(s, v)$

B.3. Visit($[s]_\omega$) (Algorithm C below and later Algorithm F)

B.4. return D

A recursive procedure is now presented for visiting a minimal component $[v]_i$. The goal is to visit all $w \in [v]_i^-$ with $d(w) \gg i$ equal to the call time value of $\min D([v]_i^-) \gg i$. By Lemma 13, the call time minimality of $[v]_i$ implies that we preserve $\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i$ throughout the call. Thus, $\min D([v]_i^-) \gg i$ will not increase until we have visited the last vertex w with $d(w) \gg i$ equal to the call time value of $\min D([v]_i^-) \gg i$. By Lemma 12 this in turns implies that $[v]_i$ will remain minimal until we have visited the last vertex we want to visit. We will maintain an index $ix([v]_i)$ which essentially equals $\min D([v]_i^-) \gg i - 1$. Then a child $[w]_{i-1}$ of $[v]_i$ is minimal if $\min D([w]_{i-1}^-) \gg i - 1 = ix([v]_i)$. Hence, recursively, we can visit all vertices $z \in [w]_{i-1}^-$ with $d(z) \gg i - 1 = \min D([w]_{i-1}^-) \gg i - 1$. Since $\min D([w]_{i-1}^-) \gg i - 1 = ix([v]_i) = \min D([v]_i^-) \gg i - 1$, $d(z) \gg i = \min D([v]_i^-) \gg i$, as desired. Finally, the visiting of $[v]_i$ is completed when $ix([v]_i) \gg 1 = D([v]_i^-) \gg i$ is increased. Formalizing in pseudo-code, we get

Algorithm C. *Visit*($[v]_i$) presumes that $[v]_i$ is minimal. It visits all $w \in [v]_i^-$ with $d(w) \gg i$ equal to the value of $\min D([v]_i^-) \gg i$ when the call is made.

C.1. if $i = 0$, visit v and return

C.2. if $[v]_i$ has not been visited previously, $ix([v]_i) \leftarrow \min D([v]_i^-) \gg i - 1$.

C.3. repeat until $[v]_i^- = \emptyset$ or $ix([v]_i) \gg 1$ is increased:

C.3.1. while $\exists$ child $[w]_{i-1}$ of $[v]_i$ such that $\min D([w]_{i-1}^-) \gg i - 1 = ix([v]_i)$,

C.3.1.1. let $[w]_{i-1}$ be a child of $[v]_i$ with $\min D([w]_{i-1}^-) \gg i - 1 = ix([v]_i)$

C.3.1.2. Visit($[w]_{i-1}$)

C.3.2. increment $ix([v]_i)$ by one

Correctness. We now prove that Algorithm C is correct, that is, if $[v]_i$ is minimal, Visit($[v]_i$) visits exactly the vertices $w \in [v]_i^-$ with $d(w) \gg i$ equal to the value of $\min D([v]_i^-) \gg i$ when the call is made. The proof is by induction on i.

If $i = 0$, we just visit v in Step C.1. By Lemma 8, $D(v) = d(v)$. Hence $d(v) \gg i$ equals the call time value of $\min D([v]_i^-) \gg i = D(v) \gg i$, as desired. After the visit to v, $[v]_i^- = \emptyset$, and we are done.

Now, assume $i > 0$. Inductively, if a subcall Visit($[w]_{i-1}$) (step C.3.1.2) is made with $[w]_{i-1}$ minimal, we may assume that it correctly visits all $u \in [w]_{i-1}^-$ with $d(u) \gg i - 1$ equal to the value of $\min D([w]_{i-1}^-) \gg i - 1$ when the subcall is made.

We will prove the following invariants for when $[v]_i^- \neq \emptyset$:

$$ix([v]_i) \gg 1 = \min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i \tag{4}$$

$$ix([v]_i) \leq \min d([v]_i^-) \gg i-1 \tag{5}$$

When $ix([v]_i)$ is first assigned in step C.2, it is assigned $\min D([v]_i^-) \gg i-1$. Also, at that time, $[v]_i$ is minimal, so $\min D([v]_i^-) = \min d([v]_i^-)$ by Lemma 8. Thus $ix([v]_i) = \min D([v]_i^-) \gg i-1 = \min d([v]_i^-) \gg i-1$, implying both (4) and (5). Now, assume (4) and (5) both hold at the beginning of an iteration of the repeat-loop C.3.

LEMMA 14. *If $\min D([v]_i^-) \gg i$ has not increased, $[v]_i$ remains minimal and (4) and (5) remain true.*

PROOF. By Lemma 12 and Lemma 8, $[v]_i$ remains minimal with $\min D([v]_i^-) = \min d([v]_i^-)$. Then, by (1), $\min D([v]_i^-) \gg i-1$ is non-decreasing, so a violation of (5) should be due to an increase in $ix([v]_i)$. However, $ix([v]_i)$ is only increased in step C.3.2, which is only entered if $\forall [w]_{i-1} \subseteq [v]_i : \min D([w]_{i-1}^-) \gg i-1 \neq ix([v]_i)$. In particular, before the increase, $ix([v]_i) \neq \min D([v]_i^-) \gg i-1 = \min_{[w]_{i-1} \subseteq [v]_i} (\min D([w]_{i-1}^-) \gg i-1)$. Moreover, by (5), $ix([v]_i) \leq \min D([v]_i^-) \gg i-1$. Hence, $ix([v]_i) < \min D([v]_i^-) \gg i-1 = \min d([v]_i^-) \gg i-1$, so the increase in $ix([v]_i)$ by one cannot violate (5). Moreover, since $\min D([v]_i^-) \gg i$ is not increased and $\min D([v]_i^-) = \min d([v]_i^-)$, (5) implies that (4) is preserved. □

LEMMA 15. *If a subcall Visit($[w]_{i-1}$) (step C.3.1.2) is made before $\min D([v]_i^-) \gg i$ is increased, all vertices u visited have $d(u) \gg i$ equal to the original value of $\min D([v]_i^-) \gg i$ (as required for visits within Visit($[v]_i$)).*

PROOF. By assumption, Lemma 14 applies when the subcall Visit($[w]_{i-1}$) is made, so (4) and (5) hold true. The assignment C.3.1.1 implies that $ix([v]_i) = \min D([w]_{i-1}^-) \gg i-1$, and clearly, $\min D([w]_{i-1}^-) \gg i-1 \geq \min D([v]_i^-) \gg i-1 \geq \min d([v]_i^-) \gg i-1$. Then (5) implies equality everywhere, so $\min D([w]_{i-1}^-) \gg i-1 = \min D([v]_i^-) \gg i-1$, and hence $[w]_{i-1}$ inherits the minimality of $[v]_i$. Thus by induction, Visit($[w]_{i-1}$) correctly visits the vertices $u \in [w]_i^-$ with $d(u) \gg i-1$ equal to the value of $\min D([w]_{i-1}^-) \gg i-1$ at the time of the subcall. However, at the time of the subcall, $\min D([w]_{i-1}^-) \gg i-1 = ix([v]_i)$ and by (4), $ix([v]_i) \gg 1 = \min D([v]_i^-) \gg i$, so $d(u) \gg i = \min D([v]_i^-) \gg i$. □

LEMMA 16. *$\min D([v]_i^-) \gg i$ has increased when the repeat-loop C.3 terminates.*

PROOF. If $\min D([v]_i^-) \gg i$ did not increase, (5) holds by Lemma 14, and (5) implies $ix([v]_i) \gg 1 \leq \min D([v]_i^-) \gg i$. Initially we have equality by (4). However, the repeat-loop can only terminate if $ix([v]_i) \gg 1$ increases or $[v]_i^-$ becomes empty, setting $\min D([v]_i^-) \gg i = \infty$. □

So far we will just assume termination deferring the proof of termination to the proof of efficiency in the next section. Thus, by Lemma 16, $\min D([v]_i^-) \gg i$ increases eventually. Let Visit($[w]_{i-1}$) be the subcall during which the increase happen.

By Lemma 13, $\min d([v]_i^-) \gg i$ increases with $\min D([v]_i^-) \gg i$. Hence by Lemma 15, Visit($[w]_i$) will visit no more vertices. Moreover, it implies that we have visited all vertices $u \in [v]_i$ with $d(u) \gg i$ equal to the original value of $\min D([v]_i^-)$, so we have now visited exactly the required vertices.

Since $\min d([v]_i^-) \gg i$ is increased and $\forall [w]_{i-1} \subseteq [v]_i : \min D([w]_{i-1}^-) \gg i-1 \geq \min D([v]_i^-) \gg i-1 \geq \min d([v]_i^-) \gg i-1$, $ix([v]_i)$ will now just be incremented without recursive subcalls Visit($[w]_{i-1}$) until either $[v]_i^-$ is emptied, or $ix([v]_i) \gg 1$ is increased by one.

Since no more vertices are visited after the increase of $\min D([v]_i^-) \gg i$, by Lemma 11, the increase is by one. Thus, we conclude that all of $ix([v]_i)$, $D([v]_i^-) \gg i$, and $\min d([v]_i^-) \gg i$ are increased by one, restoring the equalities of (4). Since, $ix([v]_i)$ now has the smallest value such that $ix([v]_i) \gg 1 = \min d([v]_i^-) \gg i$, we conclude that (5) is also satisfied.

By Lemma 13, in all future $\min d([v]_i^-) \gg i = \min D([v]_i^-) \gg i$. Moreover, $ix([v]_i)$ and $\min d([v]_i^-)$ can only change in connection with calls Visit($[v]_i$), so we conclude that (4) and (5) will remain satisfied until the next such call. This completes the proof that Algorithm C is correct. $\diamond$

6. TOWARDS A LINEAR TIME ALGORITHM

In this section, we present the ingredients of a linear time SSSP algorithm.

6.1 The component tree

Define the *the component tree* $\mathcal{T}$ representing the topological structure of the component hierarchy, skipping all nodes $[v]_i = [v]_{i-1}$. Thus, the leaves of $\mathcal{T}$ are the singleton components $[v]_0 = \{v\}$, $v \in V$. The internal nodes are the components $[v]_i$, $i > 0$, $[v]_{i-1} \subset [v]_i$. The root in $\mathcal{T}$ is the node $[v]_r = G$ with r minimized. The parent of a node $[v]_i$ is its nearest degree ≥ 2 ancestor in the component hierarchy. Since $\mathcal{T}$ have no degree one nodes, the number of nodes is $\leq 2n-1$. In Section 7 we show how to construct $\mathcal{T}$ in time $O(m)$. Given $\mathcal{T}$, it is straightforward to modify our implementation of Visit in Algorithm C so that it recurses within $\mathcal{T}$, thus skipping the components in the component hierarchy that are not in $\mathcal{T}$. In the rest of this paper, when we talk about children or parents, it is understood that we refer to $\mathcal{T}$ rather than to the component hierarchy. A min-child $[w]_h$ of $[v]_i$ is minimizing $\min D([w]_h^-) \gg i-1$. Thus a component of $\mathcal{T}$ is minimal if and only if it is minimal in the component hierarchy to $\mathcal{T}$.

6.2 A linear sized bucket structure

We say a component $[v]_i \in \mathcal{T}$ is *visited* the first time Visit($[v]_i$) is called. Note that if a component is visited, then so are all its ancestors in $\mathcal{T}$. The idea now is that for each visited component $[v]_i$, we will bucket the children $[w]_h$ according to $\min D([w]_h^-) \gg i-1$. That is, $[w]_h$ is found in a bucket denoted $B([v]_i, \min D([w]_h^-) \gg i-1)$. With $ix([v]_i) = \min D([v]_i^-) \gg i-1$ as in Algorithm C, the minimal children of $[v]_i$ are then readily found in $B([v]_i, ix([v]_i))$.

Concerning the bucketing of a visited component $[v]_i$, we can again use the index $ix([v]_i)$, for if $[v]_i$ has parent $[v]_j$ in $\mathcal{T}$, $[v]_i$ belongs in $B([v]_j, ix([v]_i) \gg j-i) = B([v]_j, \min D([v]_i^-) \gg j-1)$. The bucketing of unvisited children of visited components is deferred till later. In the rest of this subsection, the point is to show that we can efficiently embed all "relevant" buckets from $B(\cdot,\cdot)$ into one bucket array A with $O(m)$ entries.

LEMMA 17. *If* $[w]_h$ *is a minimal child of* v_i, $\min d([v]_i) \gg i-1 \leq \min D([w]_h^-) \gg i-1 \leq \max d([v]_i) \gg i-1$.

PROOF. By Lemma 8, $\min D([w]_h^-) = \min d([w]_h^-)$, and by definition of minimality, $[w]_h^-$ is a non-empty subset of $[v]_i$. □

Let $ix_0([v]_i)$ denote $\min d([v]_i) \gg i-1$. We are going to compute some $ix_\infty \geq \max d([v]_i) \gg i-1$. Then by Lemma 17, any minimal child of $[v]_i$ should be found in $B([v]_i, ix_0([v]_i)...ix_\infty([v]_i))$. Conversely, if $\min D([w]_h^-) \gg i-1 \notin [ix_0([v]_i)...ix_\infty([v]_i)]$, we know $[w]_h$ is not minimal, and hence it is not necessary to have $[w]_h$ placed in $B([v]_i,\cdot)$. We therefore define that $B([v]_i, q)$ is *relevant* if and only if $ix_0([v]_i) \leq q \leq ix_\infty([v]_i)$.

Note that the diameter of $[v]_i$ is bounded by $\sum_{e\in[v]_i} \ell(e)$. This immediately implies $\max d([v]_i) \leq \min d([v]_i) + \sum_{e\in[v]_i} \ell(e)$. Define $\Delta([v]_i) = \lceil \sum_{e\in[v]_i} \ell(e)/2^{i-1} \rceil$ and $ix_\infty([v]_i) = ix_0([v]_i) + \Delta([v]_i)$. Then $\max d([v]_i) \gg i-1 \leq ix_\infty([v]_i)$, as desired.

LEMMA 18. *The total number of relevant buckets is* $< 4m + 4n$.

PROOF. In connection with $[v]_i$, we have $\Delta([v]_i) + 1 \leq 2 + \sum_{e\in[v]_i} \ell(e)/2^{i-1}$ relevant buckets. Since $\mathcal{T}$ is a rooted tree with n leaves, where all internal nodes have at least two children, the number of nodes $[v]_i$ in $\mathcal{T}$ is $\leq 2n-1$. Thus, the total number of relevant buckets is at most

$$\sum_{[v]_i\in\mathcal{T}} (2 + \sum_{e\in[v]_i} \ell(e)/2^{i-1}) < 4n + \sum_{[v]_i\in\mathcal{T},\ e\in[v]_i} \ell(e)/2^{i-1} = 4n + \sum_{e\in E} \sum_{[v]_i \ni e} \ell(e)/2^{i-1}.$$

Now, consider any edge $e = (u,w) \in E$. Set $j = \lfloor \log_2 \ell(e) \rfloor + 1$. Then $e \in [v]_i \iff i \geq j \wedge [v]_i = [u]_i$. Since $\ell(e) < 2^j$, we get

$$\sum_{[v]_i \ni e} \ell(e)/2^{i-1} < \sum_{i\geq j} 2^j/2^{i-1} < 4.$$

Thus, the total number of relevant buckets $< 4m + 4n$. □

We will now show how to efficiently embed the relevant buckets of $B(\cdot,\cdot)$ into a single bucket array A with index set $\{0,\ldots,N\}$ where $N = O(m)$ is the total number of relevant buckets.

The Δ-values, or in fact something better, will be found in connection with the construction of $\mathcal{T}$ in Section 7. Also, the value of $\min d([v]_i^-)$ will turn out to be available when we first visit $[v]_i$. Hence both $ix_0([v]_i)$ and $ix_\infty([v]_i)$ can be identified as soon as we start visiting $[v]_i$.

For each component $[v]_i \in \mathcal{T}$, let $N([v]_i)$ denote $\sum_{[w]_j<[v]_i} (\Delta([v]_j)+1)$. Here $<$ is an arbitrary total ordering of the components in $\mathcal{T}$, say, postorder. The prefix sum $N(\cdot)$ is trivially computed in time $O(n)$ as soon as we have computed $\Delta(\cdot)$. Now, for any $[v]_i \in \mathcal{T}$, $x \in \mathbb{N}_0$, if $B([v]_i, x)$ is relevant, i.e. if $x \in \{ix_0([v]_i),\ldots,ix_\infty([v]_i)\}$, we identify $B([v]_i, x)$ with $A(x - ix_0([v]_i) + N([v]_i))$; otherwise, the contents of $B([v]_i, x)$ is deferred to a "waste" bucket $A(N)$. In conclusion, the bucket structure $B(\cdot,\cdot)$ is implemented in linear time and space.

6.3 Bucketing unvisited children

Let $\mathcal{U}$ denote the unvisited subforest of $\mathcal{T}$. An unvisited component $[v]_i$ is a child of a visited component if and only if $[v]_i$ is a root of a tree in $\mathcal{U}$. In Section 8, we will present a data structure that for the changing set of roots $[v]_i$ in $\mathcal{U}$, maintains the changing values $\min D([v]_i^-)$ in linear total time. Assuming the above data structure, the rest of this subsection presents the pseudo-code needed to maintain that every unvisited child $[w]_h$ of a visited component $[v]_j$ is correctly bucketed in $B([v]_i, \min D([w]_h) \gg i-1)$.

When a component $[v]_i$ is first visited, all its children need to be bucketed for the first time:

Algorithm D. Expand($[v]_i$), $i > 0$, assumes that Visit($[v]_i$) has just been called for the first time. It buckets the children of $[v]_i$ in $B([v]_i, \cdot)$. Further, it initiates $ix_0([v]_i)$ and $ix_\infty([v]_i)$.

D.1. $ix_0([v]_i) \leftarrow \min D([v]_i^-) \gg i-1$

D.2. $ix_\infty([v]_i) \leftarrow ix_0([v]_i) + \Delta([v]_i)$

D.3. for $q = ix_0([v]_i)$ to $ix_\infty([v]_i)$, $B([v]_i, q) \leftarrow \emptyset$.

D.4. delete $[v]_i$ from $\mathcal{U}$, turning the children $[w]_h$ of $[v]_i$ into roots in $\mathcal{U}$.

D.5. for all children $[w]_h$ of $[v]_i$,

D.5.1. bucket $[w]_h$ in $B([v]_i, \min D([w]_h^-) \gg i-1)$.

When a vertex v is visited, we may decrement the D-values of some of its neighbors. Accordingly, we may need to re-bucket the unvisited roots of these neighbors.

Algorithm E. Visit(v) where $[v]_0$ is minimal and all ancestors of $[v]_0$ in $\mathcal{T}$ are expanded, visits v and restores the the bucketing of unvisited children of visited components.

E.1. $S \leftarrow S \cup \{v\}$

E.2. for all $(v, w) \in E$, if $D(v) + \ell(v, w) < D(w)$ then

E.2.1. let $[w]_h$ be the unvisited root of $[w]_0$ in $\mathcal{U}$ and let $[w]_i$ be the visited parent of $[w]_h$ in $\mathcal{T}$

E.2.2. decrease $D(w)$ to $D(v) + \ell(v, w)$; if this decreases $\min D([w]_h^-) \gg i-1$ then

E.2.2.1. move $[w]_h$ to $B([w]_i, \min D([w]_h) \gg i-1)$

6.4 An efficient visiting algorithm

We are now ready to present an efficient bucketing based implementation of Visit($[v]_i$). The point in the component tree $\mathcal{T}$ is that it allows us to skip components in the component hierarchy that have only one child. Consequently, if $[v]_j$ is the parent of $[v]_i$ in $\mathcal{T}$, we may have $j > i+1$. Our goal now is to visit all vertices $w \in [v]_i^-$ with $d(w) \gg j-1$ equal to the call time value of $\min D([v]_i^-) \gg j-1$.

As in Algorithm C, we will maintain an index $ix([v]_i)$ which essentially equals $\min D([v]_i^-) \gg i-1$. Then min-children $[w]_h$ of $[v]_i$ are then readily available in $B([v]_i, ix([v]_i))$. By definition of $\mathcal{T}$, $[v]_i = [v]_{j-1}$. Hence, $ix([v]_i) = \min D([v]_i^-) \gg i-1$ implies $ix([v]_i) \gg j-i = \min D([v]_{j-1}^-) \gg j-1$. Thus, as in Algorithm C, we can use Lemma 12 to argue that $[v]_i = [v]_{j-1}$ remains minimal until $ix([v]_i) \gg j-i$

is increased. Until this increase, the minimality of $[v]_i$ implies minimality of the min-children $[w]_h$ of $[v]_i$ in $B([v]_i, ix([v]_i))$. Recursively, we then visit $[w]_h$, thus visiting all vertices $x \in [w]_h$ with $d(x) \gg i-1 = \min D([w]_h^-) \gg i-1 = \min D([v]_i^-) \gg i-1 = ix([v]_i)$. Since $ix([v]_i) \gg j-i$ has not increased, $d(x) \gg j-1 = ix([v]_i) \gg j-i$ is still the call time value of $\min D([v]_i^-) \gg j-1$, as desired.

For the bucketing of unvisited children of visited components, we make appropriate calls to Algorithm E and D. For the bucketing of a visited component $[v]_i$ with parent $[v]_j$, we note that $[v]_i$ belongs in bucket $B([v]_j, ix([v]_i) \gg j-i)$. Hence, we just need to update the positioning of $[v]_i$ in $B([v]_j, \cdot)$ when $ix([v]_i) \gg j-i$ changes. The above ideas are implemented in the following pseudo-code:

Algorithm F. Visit($[v]_i$) assumes that $[v]_i$ is minimal. Let $[v]_j$ be the parent of $[v]_i$ in $\mathcal{T}$—if $[v]_i$ is the root of $\mathcal{T}$, technically, we just set $j = i+1$. Then Visit($[v]_i$) visits all $w \in [v]_i^-$ with $d(w) \gg j-1$ equal to the call time value of $\min D([v]_i^-) \gg j-1$. Moreover, Visit($[v]_i$) maintains a correct bucketing of $[v]_i$ in $B([v]_j, \cdot)$.

F.1. if $i = 0$,
F.1.1. Visit(v) (Algorithm E)
F.1.2. remove $[v]_i$ from $B([v]_j, \cdot)$
F.1.3. return

F.2. if $[v]_i$ has not been visited previously,
F.2.1. Expand($[v]_i$) (Algorithm D)
F.2.2. $ix([v]_i) \leftarrow ix_0([v]_i)$

F.3. repeat until $[v]_i^- = \emptyset$ or $ix([v]_i) \gg j-i$ is increased:
F.3.1. while $B([v]_i, ix([v]_i)) \neq \emptyset$,
F.3.1.1. let $[w]_h \in B([v]_i, ix([v]_i))$
F.3.1.2. Visit($[w]_h, i$) (Algorithm F)
F.3.2. increment $ix([v]_i)$ by one

F.4. if $[v]_i^- \neq \emptyset$, move $[v]_i$ to $B([v]_j, ix([v]_i) \gg j-i)$

F.5. if $[v]_i^- = \emptyset$ and $[v]_i$ is not the root of $\mathcal{T}$, remove $[v]_i$ from $B([v]_j, \cdot)$

Correctness. We will now argue combined correctness of Algorithms D, E, and F. Parts of the proof just mimic the the correctness proof of Algorithm C. However, the bucketing, introduces an added complexity because the bucketing done by one call, depends on bucketing of other calls. In order to avoid cyclic arguments, we will need to be very precise in stating the exact responsibilities of each call Visit($[v]_i$). We will use the terminology that a call Visit($[v]_i$) *maintains* some property P *call-wise* if P is part of our call time assumptions, and P will be satisfied by the end of the call provided that all call-time assumptions were satisfied when the call was made. Further, Visit($[v]_i$) *maintains* P *step-wise* if P is guaranteed to be satisfied before and after every step of the algorithm. Here one step may be a subcall as in F.3.1.2. Clearly step-wise maintenance implies call-wise maintenance. The point in the step-wise conditions is that when we study possible violations of any step, we may assume that all the step-wise conditions are satisfied just before the start of the step.

We are now ready to specify *correct behavior* of a call Visit($[v]_i$). If $[v]_i$ is not the root of $\mathcal{T}$, let $[v]_j$ the the parent of $[v]_i$. Otherwise set $j = \omega + 1$.

(i) We assume that $[v]_i$ is minimal when the call is made.

(ii) Step-wise, we maintain that every vertex v visited so far during the course of the algorithm was minimal when visited.

(iii) The vertices visited by the call Visit($[v]_i$) are exactly the vertices $w \in [v]_i^-$ with $d(w) \gg (j-1)$ equal to the call time value of $\min D([v]_i^-) \gg (j-1)$.

(iv) Call-wise, correct bucketing of $[v]_i$ is maintained

(v) Step-wise, correct bucketing is maintained of all visited components strictly descending from $[v]_i$.

(vi) Step-wise, correct bucketing is maintained of all — also non-descending — unvisited children of visited components.

(vii) If $i > 0$, the following invariants apply from the first call Visit($[v]_i$) when $ix([v]_i)$ is first assigned in step F.2.2, and until $[v]_i^- \neq \emptyset$.

$$ix([v]_i) \gg j - i = \min d([v]_i^-) \gg j - 1 = \min D([v]_i^-) \gg j - 1 \qquad (6)$$

$$ix([v]_i) \leq \min d([v]_i^-) \gg i - 1 \qquad (7)$$

The invariant (6) is maintained call-wise and invariant (7) is maintained step-wise.

(viii) Step-wise, Visit($[v]_i$) maintains (6) and (7) for all visited components strictly descending from $[v]_i$ in $\mathcal{T}$.

We are going to prove correctness of all calls Visit($[v]_i$) by induction on i. However, before doing so, note

LEMMA 19. *Correct behavior of the top-call Visit($[v]_\omega$) to Algorithm F in step B.3 implies correctness of the SSSP algorithm (Algorithm B).*

PROOF. By definition, $[v]_\omega$ is identical to the root of $\mathcal{T}$. Further, $[v]_\omega$ is minimal by definition, so (i) is satisfied when Visit($[v]_\omega$) is called in step B.3. Also, since we have not visited any components when the call is made, all properties to be maintained by (ii)–(viii) are trivially satisfied at call time. Hence, by (ii), all vertices are minimal when visited and by (iii), we visit all vertices w with $d(w) \gg \omega = \min D([v]_\omega^-) \gg \omega$. However, any integer x in a word have $x \gg \omega = 0$. and we have assumed that all distances calculated fit in one word, so we conclude that all vertices are visited. □

As promised, we now proceed to prove the correctness of the calls Visit($[v]_i$) by induction on i. For each step independently, we will prove that all step-wise conditions are satisfied after the step given that they were all satisfied before the step. It then follows that there can never be a first violation of a step-wise conditions. As an example, we may assume that (ii) is satisfied before any step considered, hence that all vertices visited so far were minimal when visited. This means that the lemmatas we have proved in the previous sections apply to the situation immediately before the step. For example, if $[v]_0$ is minimal before a step, by Lemma 8, $D(v) = d(v)$ before that step.

LEMMA 20. *If $i = 0$, the call Visit($[v]_i$) behaves correctly.*

PROOF. First note that (v), (vii), and (viii) do not apply for $i = 0$. We are going to perform steps F.1.1–F.1.3, where step F.1.1 visits v via Algorithm E.

The call time assumption from (i) that $[v]_i = [v]_0$ is immediately implies that (ii) is maintained. Now Algorithm E makes sure that if $\min D([w]_h^-)$ is decreased for some unvisited child $[w]_h$ of a visited component, then $[w]_h$ is rebucketed accordingly. Hence (vi) is maintained.

By (i), $[v]_0$ was minimal when the call was made. Hence by Lemma 8, $d(v) = D(v) = \min D([v]_i^-)$ before the visit of v. Since further v was the only vertex in $[v]_i^-$, we conclude that (iii) is satisfied.

Finally, we need to make sure that (iv) is maintained call-wise. After v has been visited in step F.1.1, $[v]_i^- = \emptyset$, so $[v]_i$ should be removed from $B([v]_j, \cdot)$. This is exactly what happens in the subsequent step F.1.2, and hence (iv) is restored before we return from the call in step F.1.3. □

The rest of the proof is devoted to proving the correct behavior of Visit($[v]_i$) with $i > 0$. Inductively, we assume correct behavior of all subcalls Visit($[w]_h$) made in step F.3.1.2.

LEMMA 21. *First time we reach the repeat-loop F.3, we have not misbehaved yet, and (6) is satisfied.*

PROOF. If $[v]_i$ has been visited before, the repeat-loop F.3 is the first thing to be executed, so we cannot have misbehaved yet. Also, we cannot have violated (6) which was a call-time assumption by (vii).

If $[v]_i$ was not visited before, we have to consider the effect of steps F.2.1–F.2.2. Step F.2.1 calls Algorithm D which formally visits $[v]_i$ by removing it from $\mathcal{U}$. This makes all children of $[v]_i$ new unvisited children of a visited component. Thereafter, Algorithm D correctly buckets these unvisited children of $[v]_i$, so by the end of step F.2.1, we have restored correct bucketing of all unvisited children of visited components. Hence (vi) is maintained, and clearly step F.2.1 cannot cause any other violations.

Now, step F.2.2 can only affect (vii). Since this is the first visit to $[v]_i$, (vii) does not require that (7) is satisfied until after step F.2.2. Also, for (vii), (6) does not need to be satisfied till the end of the call. Nevertheless, as stated in the claim, we will prove that (6) is satisfied when step F.2.2 is completed.

Note that as its first step, Algorithm D assigns $\min D([v]_i^-) \gg i-1$ to $ix_0([v]_i)$, and because no visits are performed since, we still have $ix_0([v]_i) = \min D([v]_i^-) \gg i-1$ when come to step F.2.2. Consequently, the effect of step F.2.2 is to set $ix([v]_i) = \min D([v]_i^-) \gg i-1$. Moreover, since no visits have been made since the start of the call, $[v]_i$ is still minimal, and hence, by Lemma 8, $\min D([v]_i^-) = \min d([v]_i^-)$. In conclusion, $ix([v]_i) = \min d([v]_i^-) \gg i-1 = \min D([v]_i^-) \gg i-1$, implying that (6) and (7) are both satisfied after step F.2.2. □

LEMMA 22. *If $\min D([v]_i) \gg j-1$ has not increased when we start a subcall Visit($[w]_h$) in step F.3.1.2, all call time assumptions of Visit($[w]_h$) are satisfied, and further $ix([v]_i) = \min D([w]_h^-) \gg i-1 = \min D([v]_i^-) \gg i-1$.*

PROOF. First, to satisfy (i) for Visit($[w]_h$), we need to show that $[w]_h$ is minimal. Now, by (i) applied to Visit($[v]_i$), $[v]_i$ was minimal when Visit($[v]_i$) started, and since $\min D([v]_i) \gg j-1$ has not increased, by Lemma 12, $[v]_i$ is still minimal at the time of the subcall Visit($[w]_h$). Thus, we argue minimality of $[w]_h$ by showing that it is a min-child of $[v]_i$.

Now, step F.3.1.2 is only entered if $[w]_h$ is found in bucket $B([v]_i, ix([v]_i))$, and by (iv) and (v) for Visit$([v]_i)$, we may assume that all children of $[v]_i$ are correctly bucketed when step F.3.1.2 is entered. Hence $\min D([w]_h^-) \gg i-1 = ix([v]_i)$. On the other hand,

$$\min D([w]_h^-) \gg i-1 \geq \min D([v]_i^-) \gg i-1 \geq d([v]_i^-) \gg i-1 \geq ix([v]_i).$$

The above three inequalities follow from $[w]_h \subseteq [v]_i$, D's domination of d, and (7), respectively. In conclusion,

$$ix([v]_i) = \min D([w]_h^-) \gg i-1 = \min D([v]_i^-) \gg i-1,$$

as desired. The last equality is the definition of $[w]_h$ being a min-child of $[v]_i$, so we conclude that $[w]_h$ is minimal as required by (i) for the subcall Visit$([w]_h)$.

Now, the properties of (ii), (v), (vi) and (viii) are step-wise, so we may assume that they hold for Visit$([v]_i)$ when the subcall Visit$([w]_h)$ starts. However, the properties of (ii) and (vi) are equivalent for Visit$([v]_i)$ and Visit$([w]_h)$. Also, since $[v]_i$ is parent of $[w]_h$, property (v) for Visit$([v]_i)$ implies both (iv) and (v) for Visit$([w]_h)$, and property (viii) for Visit$([v]_i)$ implies both (vii) and (viii) for Visit$([w]_h)$. Since (iii) has no call-time assumption, we now conclude that all call-time assumptions are satisfied for Visit$([w]_h)$. $\square$

LEMMA 23. *If* $\min D([v]_i) \gg j-1$ *has not increased when we start a subcall Visit*$([w]_h)$ *in step F.3.1.2, step F.3.1.2 behaves correctly.*

PROOF. From Lemma 22, we know that all call-time assumptions of Visit$([w]_h)$ where satisfied, and by induction, Visit$([w]_h)$ may be assumed to behave correctly. Still, we need to show that correct behavior of Visit$([w]_h)$ does not make Visit$([v]_i)$ misbehave.

First note that the conditions of (ii) and (vi) have exactly the same interpretation for Visit$([v]_i)$ and Visit$([w]_h)$, so Visit$([w]_h)$ maintains (ii) and (vi) for Visit$([v]_i)$.

Concerning (iii), by induction we know that Visit$([w]_h)$ visits exactly the vertices $u \in [w]_h^-$ with $d(u) \gg i-1 = \min D([w]_h^-) \gg i-1$ when the subcall was made, and by Lemma 22, at that time $\min D([w]_h^-) \gg i-1 = \min D([v]_i^-) \gg i-1$. Since $j \geq i$, we conclude that $d(u) \gg j-1 = \min D([v]_i^-) \gg j-1$ when Visit$([w]_h)$ was called. However, by assumption, $\min D([v]_i^-) \gg j-1$ has not increased since Visit$([v]_i)$ was first called, so our visits have not violated (iii).

Concerning (v) and (viii), note that (iv), (v), (vii) and (viii) of Visit$([w]_h)$ imply that (v) and (viii) of Visit$([v]_i)$ are maintained for all components equal to or descending from $[w]_h$. To complete the proof that (v) and (viii) are maintained, we consider an arbitrary visited component $[x]_g$ descending from $[v]_i$ but not equal to or descending from $[w]_h$. Note that Visit$([w]_h)$ only visits components descending from $[w]_h$, so it does not visit $[x]_g$. Consequently, $[x]_g$ is not rebucketed during Visit$([w]_h)$, and $ix([x]_g)$ is not changed.

Since Visit$([w]_h)$ does not visit $[x]_g$, $[x]_g$ was visited before the subcall Visit$([w]_h)$. Hence, by (viii), (6) and (7) are both satisfied for $[x]_g$ before the subcall Visit$([w]_h)$. Since $\min d([x]_g^-)$ can only change if vertices from $[x]_g$ are visited, we immediately conclude that (7) is preserved.

Before the subcall Visit$([w]_h)$, by (6), we had $\min D([x]_g^-) \gg j-1 = \min d([x]_g^-) \gg j-1$. Now, D dominates d, $\min d([x]_g^-) \gg j-1$ cannot decrease, and $\min D([x]^-_g) \gg j-1$ can only increase if a vertex $[x]_g$ is visited. Since $\text{Visit}([w]_h)$ does not visit vertices in $[x]_g$, we conclude that $\min D([x]^-_g) \gg j-1$ and $\min d([x]^-_g) \gg j-1$ remain equal and unchanged. The fact that $\min D([x]^-_g) \gg j-1$ does not change, implies that our bucketing of $[x]_g$ remains correct, so (v) follows. Also, since $ix([x]_g)$ is not changed, we conclude that (6) remain satisfied, completing the proof of (viii).

The final step-wise condition to be preserved is (6) for $[v]_i$. However, $\text{Visit}([w]_h)$ does not change $ix([v]_i)$, and $\min d([v]^-_i)$ can only increase, so (6) is preserved. □

LEMMA 24. *If* $\min D([v]^-_i) \gg i-1 = \min d([v]^-_i) \gg i-1$, *step F.3.2 behaves correctly.*

PROOF. Clearly, the only step-wise condition that could be affected by the increment of $ix([v]_i)$ in step F.3.2 is (7) in (vii).

We will now study closer the situation just before entering step F.3.2. Step F.3.2 is only entered if the test of the while loop F.3.1 has just failed, so there is no child $[w]_h$ of $[v]_i$ in $B([v]_i, ix([v]_i))$. However, by (v) and (vi), the children of $[v]_i$ may be assumed correctly bucketed, so we conclude that there is no child $[w]_h$ of $[v]_i$ with $\min D([w]^-_h) \gg i-1 = ix([v]_i)$. Since $\min D([v]^-_i) \gg i-1$ equals the minimum of $\min D([w]^-_h) \gg i-1$ over all children $[w]_h$ of $[v]_i$, it follows that $\min D([v]^-_i) \gg i-1 \neq ix([v]_i)$. By assumption, $\min D([v]^-_i) \gg i-1 = \min d([v]^-_i) \gg i-1$. Thus $\min d([v]^-_i) \gg i-1 \neq ix([v]_i)$. On the other hand, before entering step F.3.2, by (7), $ix([v]_i) \leq \min d([v]^-_i) \gg i-1$, which combined with the above inequality implies $ix([v]_i) < \min d([v]^-_i) \gg i-1$. Consequently, the increase of $ix([v]_i)$ by one in step F.3.2 cannot violate $ix([v]_i) \leq \min d([v]^-_i) \gg i-1$. We therefore conclude that (7) is preserved. □

LEMMA 25. *As long as* $\min D([v]^-_i) \gg j-1$ *does not increase, the repeat-loop F.3 behaves correctly but does not terminate.*

PROOF. Lemma 23 states that step F.3.1.2 behaves, and hence that the while loop F.3.1 behaves. As long as $\min D([v]^-_i) \gg i$ has not increased, Lemma 12, states that $[v]_i$ remains minimal. Therefore, by Lemma 8, we have $\min D([v]^-_i) = \min d([v]^-_i)$. Hence, Lemma 24 ascertains correct behavior of step F.3.2. It follows that the repeat loop F.3 behaves correctly as long as $\min D([v]^-_i) \gg j-1$ does not increase.

To prove non-termination, note that leaving the repeat-loop requires either $[v]^-_i = \emptyset$ or that $ix([v]_i)$ has increased. We want to show that $\min D([v]^-_i)$ must have increased in either case. Clearly this is true if $[v]^-_i = \emptyset$ for then $\min D([v]^-_i)$ has increased to ∞.

Consider an increase in $ix([v]_i)$. Initially, by Lemma 21, we had (6) satisfied, that is, initially, $ix([v]_i) \gg j-i = \min d([v]^-_i) \gg j-1 = \min D([v]^-_i) \gg j-1$. On the other hand, (7) is preserved step-wise, so $ix([v]_i) \leq \min d([v]^-_i) \gg i-1 \leq \min D([v]^-_i) \gg i-1$. Thus, $ix([v]_i) \gg j-1$ cannot have been increased without $\min D([v]^-_i) \gg j-1$ being increased. □

LEMMA 26. *When* $\min D([v]^-_i) \gg j-1$ *increases, the algorithm terminates correctly.*

PROOF. Clearly, $\min D([v]^-_i) \gg j-1$ can only increase during some subcall $\text{Visit}([w]_h)$ in step F.3.1.2, referred to as the *increasing subcall.* As the increase takes place during the increasing subcall, Lemma 23 still provides correctness of the increasing subcall itself. Immediately after the subcall, by (v) and (vi), all children of $[v]_i$ are correctly bucketed, so $B([v]_i, ix)$ is empty unless $ix \geq \min D([v]_i^-)$.

We will now argue that the repeat loop F.3 does not perform any further subcalls $\text{Visit}([w']_{h'})$ in step F.3.1.2. Clearly the increments to $ix([v]_i)$ in step F.3.2 cannot change our correct bucketing, so passing the test of the while loop F.3.1 requires that $ix([v]_i) \geq \min D([v]_i^-) \gg i-1$. However, by Lemma 21, we had $ix([v]_i) \gg j-i = \min D([v]_i^-) \gg j-1$ when we first reached the repeat loop F.3, which was before $\min D([v]_i^-) \gg j-1$ was increased. Thus, we cannot pass the test of the while loop F.3.1 unless $ix([v]_i) \gg j-1$ is increased. However, if step F.3.2 increases $ix([v]_i) \gg j-1$, we leave the repeat loop F.3 before returning to the while loop F.3.1.

Since no further subcalls $\text{Visit}([w']_{h'})$ are made after the increasing subcall, by Lemma 23, all iterations of step F.3.1.2 behave correctly. In particular, these steps respect (iii), so all vertices u visited during the execution of $\text{Visit}([v]_i)$ have $d(u) \gg j-1$ equal to the call-time value of $\min D([v]_i^-) \gg j-1$. In order to complete the proof of (iii), we need to show that we visit all vertices $u \in [v]_i$ with $d(u) \gg j-1$ equal to the call-time value of $\min D([v]_i^-) \gg j-1$.

Suppose the increasing call empties $[v]_i$. Then (vii) no longer applies, so the subsequent increase of $ix([v]_i)$ in step F.3.2 is trivially correct. Afterwards, $\text{Visit}([v]_i)$ will finalize by removing $[v]_i$ from $B([v]_j, \cdot)$, hence preserving (v). Since $[v]_i$ is emptied, we have trivially visited all vertices $u \in [v]_i$ with $d(u) \gg j-1$ equal to the call-time value of $\min D([v]_i^-) \gg j-1$. Hence $\text{Visit}([v]_i)$ satisfies (iii). Since (vii) no longer applies, we conclude that the call $\text{Visit}([v]_i)$ behaves correctly if the increasing subcall empties $[v]_i$.

Now suppose that the increasing subcall does not empty $[v]_i$. Let $u \in [w]_h$ be the vertex whose visit affected the first increase in $\min D([v]_i^-) \gg j-1$. By Lemma 11, the increase of $\min D([v]_i^-) \gg j-1$ is by one, and by Lemma 13, $\min d([v]_i^-) \gg j-1$ is increased with $\min D([v]_i^-) \gg j-1$. Consequently, after the visit of u, we have visited all vertices $x \in [v]_i$ with $d(x) \gg j-1$ equal to the call-time value of $\min D([v]_i^-) \gg j-1$, so (iii) is satisfied for the call $\text{Visit}([v]_i)$.

We leave the repeat loop F.3 as soon as $ix([v]_i) \gg j-1$ is increased. Since the increasing subcall increases $\min d([v]_i^-) \gg j-1$, it follows that the subsequent increases to $ix([v]_i)$ cannot violate (7). Hence all iterations of step F.3.2 behave correctly. Also, when we leave the repeat loop, $ix([v]_i) \gg j-1$ has been increased by one like $\min d([v]_i^-) \gg j-1$ and $\min D([v]_i^-) \gg j-1$, so (6) is restored, meaning (viii) is maintained by $\text{Visit}([v]_i)$. From (6), we further conclude that $[v]_i$ is correctly rebucketed in step F.4, so (iv) is maintained. Since all step-wise conditions have already been proved maintained, we conclude that $\text{Visit}([v]_i)$ behaves correctly. □

Lemmatas 20, 21, 25, and 26 immediately imply correct behavior of Algorithm F, including the subroutines Algorithm E and D. ◇

Proposition 27. *Assume that the component tree $\mathcal{T}$ has been computed (cf. Section 7), and assume an oracle that dynamically maintains the changing value of $\min D([v]_i^-)$ for each root $[v]_i$ in $\mathcal{U}$ in linear total time (cf. Section 8). Then no more than $O(m)$ time and space is needed to solve the SSSP problem.*

PROOF. Except from the bucketing, the space bound is trivial. Lemma 18 says that we have only $O(m)$ relevant buckets, so from our linear implementation of these buckets in the bucket array A, we conclude that only $O(m)$ space is needed.

Clearly, the total time spent in Visit(v) (Algorithm E) is $O(m)$. The time spent in Expand($[v]_i$) (Algorithm D) is proportional to the number of relevant buckets in $B([v]_i, \cdot)$, so by Lemma 18, the total time spent in Expand is $O(m)$.

Having accounted for the time spent in Visit(v) (Algorithm E), the time spent in Visit($[v]_i$) (Algorithm F) is amortized over increases in $ix([v]_i)$, or emptying of $[v]_i^-$. Since $B([v]_i, ix([v]_i))$ is always a relevant bucket, by Lemma 18, there can by at most $O(m)$ increases to $ix([v]_i)$. Also, each $[v]_i^-$ is emptied only once, so the latter event can only happen $O(n)$ times.

By the correctness of Visit($[v]_i$), either $[v]_i^-$ is emptied, or $ix([v]_i) \gg j - i = \min D([v]_i^-) \gg j - 1$ is increased. The latter implies that $ix([v]_i)$ is increased, so we conclude that the call Visit($[v]_i$) is payed for in either case.

Each iteration of the repeat-loop, step F.3, is trivially paid for, either by an increase in $ix([v]_i)$, or by emptying $[v]_i^-$. Finally, each iteration of the the while-loop, step F.3.1, is paid for by the call Visit($[w]_h$). $\square$

7. THE COMPONENT TREE

In this section, we will present a linear time and space construction for the component tree $\mathcal{T}$ defined in the previous section. Recall that on level i, we want all edges of weight $< 2^i$. Thus, we are only interested in the position of the most significant bit (msb) of the weights, i.e. $msb(x) = \lfloor \log_2 x \rfloor$. Although msb is not always directly available, it may be obtained by three standard AC^0 operations by first converting the integer x into a double floating point, and then extract the exponent by two shifts. Alternatively, msb may be coded by a constant number of multiplications, as described in [Fredman and Willard 1993].

Our first step is to construct a minimum spanning tree M. This is done deterministically in linear time as described by Fredman and Willard in [Fredman and Willard 1994]. As for G_i, set $M_i = (V, \{e \in M \mid \ell(e) < 2^i\})$. Also, let $[v]_i^M$ denote the component of v in M_i. Clearly $[v]_i^M$ is a spanning tree of $[v]_i$, so vertex-wise, the components of M_i coincide with those of G_i. Also, since $[v]_i^M$ is a spanning tree of $[v]_i$,

$$\sum_{e\in[v]_i} \ell(e) \geq \sum_{e\in[v]_i^M} \ell(e) \geq diameter([v]_i^M) \geq diameter([v]_i).$$

Hence $\sum_{e\in[v]_i^M} \ell(e)$ gives us a better upper bound on the diameter of $[v]_i$ than $\sum_{e\in[v]_i} \ell(e)$. We can therefore redefine Δ from Section 6 as $\Delta([v]_i) = \lceil \sum_{e\in[v]_i^M} \ell(e)/2^{i-1} \rceil$.

Note that since M has only $n-1$ edges, Lemma 18 states that we have only $4(n-1) + 4n < 8n$ relevant buckets. Consequently, most of the bounds in the proof of Proposition 27 are reduced from $O(m)$ to $O(n)$. More precisely, we are still spending $O(m)$ total time in Visit(v) (Algorithm E), but excluding these calls, the total time spent in Expand($[v]_i$) (Algorithm D) and Visit($[v]_i$) (Algorithm D) is reduced from $O(m)$ to $O(n)$.

The main motivation for working with M instead of with G is that M is a tree, and Gabow and Tarjan [Gabow and Tarjan 1985] have shown that we can preprocess a tree in linear time and space, so as to support union-find operations at constant cost per operation. More precisely, we operate on a dynamic subforest $S \subseteq M$, starting with S consisting of singleton vertices. Each component of S has a *canonical* vertex, and $\mathit{find}(v)$ returns the canonical vertex from the component of S that v belongs to. Hence $\mathit{find}(v) = \mathit{find}(w)$ if and only if v and w are in the same component of S. For an edge $(v, w) \in M$, $\mathit{union}(v, w)$ adds (v, w) to S. The canonical vertex of the new united component is assumed to be $\mathit{find}(v)$ or $\mathit{find}(w)$ from before $\mathit{union}(v, w)$ was called.

Now let $e_1, \ldots, e_{n-1}$ be the edges of M sorted according to $\mathit{msb}(\ell(e_i))$. Note that $\mathit{msb}(\ell(e_i)) < \log_2 w$. Thus, if $\log w = O(n)$, such a sequence is produced in linear time by simple bucketing. Otherwise, $\log w = O(w/(\log n \log\log n))$, and then we can sort in linear time by packed merging [Albers and Hagerup 1997; Andersson et al. 1995].

Defining $\mathit{msb}(\ell(e_0)) = -1$ and $\mathit{msb}(\ell(e_n)) = \omega$, note that

$$\mathit{msb}(\ell(e_i)) < \mathit{msb}(\ell(e_{i+1})) \iff M_{\mathit{msb}(\ell(e_i))+1} = M_{\mathit{msb}(\ell(e_{i+1}))} = (V, \{e_1, \ldots, e_i\}).$$

Thus, sequentially, for $i = 1, \ldots, n-1$, we will call $\mathit{union}(e_i)$, and if $\mathit{msb}(\ell(e_i)) < \mathit{msb}(\ell(e_{i+1}))$, we will collect all the new components of S for $\mathcal{T}$. In order to compute the Δ-values, for a component with canonical element v, we store $s(v)$ equal to the sum of the weights of the edges spanning it. In order to efficiently identify new components of $\mathcal{T}$ and the parent pointers to them, we maintain the set X of old canonical elements for the components united since last components were collected. The desired algorithm is now straightforward.

Algorithm G. Constructs $\mathcal{T}$ and $\Delta(\cdot)$ from the minimum spanning tree M. A leaf $[v]_0$ of $\mathcal{T}$ is identified with v and the internal components are identified with an initial segment of the natural numbers. If c identifies $[v]_i \in \mathcal{T}$, $\wp(c)$ denotes the identifier of the parent of $[v]_i$ in $\mathcal{T}$, and $\Delta(c) = \lceil \sum_{e \in [v]_i^M} / 2^{i-1} \rceil$.

G.1. for all $v \in V$,
G.1.1. $c(v) \leftarrow v$
G.1.2. $\Delta(v) \leftarrow 0$
G.1.3. $s(v) \leftarrow 0$

G.2. $c \leftarrow 0$; $X \leftarrow \emptyset$

G.3. for $i \leftarrow 1$ to $n-1$,
G.3.1. let $(v, w) = e_i$ where e_i is the ith edge from the minimum spanning tree M.
G.3.2. $X \leftarrow X \cup \{\mathit{find}(v), \mathit{find}(w)\}$
G.3.3. $s \leftarrow s(\mathit{find}(v)) + s(\mathit{find}(w)) + \ell(v, w)$
G.3.4. $\mathit{union}(v, w)$
G.3.5. $s(\mathit{find}(v)) \leftarrow s$
G.3.6. if $\mathit{msb}(\ell(e_i)) < \mathit{msb}(\ell(e_{i+1}))$,
G.3.6.1. $X' \leftarrow \{\mathit{find}(v) | v \in X\}$— X' = canonical elements of new components of $\mathcal{T}$.
G.3.6.2. for all $v \in X'$,
G.3.6.2.1. $c \leftarrow c + 1$
G.3.6.2.2. $c'(v) \leftarrow c$

G.3.6.3. for all $v \in X$, $\wp(c(v)) \leftarrow c'(\mathit{find}(v))$
G.3.6.4. for all $v \in X'$,
G.3.6.4.1. $c(v) \leftarrow c'(v)$
G.3.6.4.2. $\Delta(c(v)) \leftarrow \lceil s(v)/2^{msb(\ell(e_i))} \rceil$
G.3.6.5. $X \leftarrow \emptyset$

G.4. $\wp(c) \leftarrow c$

In conclusion we have obtained,

PROPOSITION 28. *The component tree $\mathcal{T}$ is computed in $O(m)$ time and space.*

8. THE UNVISITED DATA STRUCTURE

In this section, we will show that for the changing set of roots $[v]_i$ in the unvisited part $\mathcal{U}$ of the component tree $\mathcal{T}$, we can maintain the changing values $\min D([v]_i^-)$ in linear total time. This was the last unjustified assumption of Proposition 27. Hence the results of this section will allow us to conclude with a linear time SSSP algorithm.

First note that since each root $[v]_i$ of $\mathcal{U}$ is unvisited, $[v]^- = [v]_i$. We identify each vertex v of G with with the leaf $[v]_0$ of $\mathcal{T}$. The operations we want to support in amortized constant time are: (1) if $D(v)$ is decreased for some vertex v with unvisited root $[v]_i$, update $\min D([v]_i)$, and (2) if an unvisited root $[v]_i$ is visited, each child $[w]_h$ of $[v]_i$ in $\mathcal{T}$ becomes a new root in $\mathcal{U}$, so we need to find $\min D([w]_h^-)$. We will reduce this problem to a slightly cleaner data structure problem.

Let $v_1, \ldots, v_n$ be an ordering of the vertices of G induced by an arbitrary ordering of the children of each internal node of $\mathcal{T}$. Now, for each root $[v]_i$ in $\mathcal{U}$, the vertices of $[v]_i$ are the leaves under $[v]_i$, and they form a connected segment $v_j, \ldots, v_l$ of $v_1, \ldots, v_n$. We want to maintain $\min D([v]_i) = \min_{j \le k \le l} D(v_k)$. When we remove a root from $\mathcal{U}$, we split its segment into the segments of the subtrees. Thus we are studying a dynamic partitioning of $v_1, \ldots, v_n$ into connected segments, where for each segment, we want to know the minimal D-value. When we start, $v_1, \ldots, v_n$ forms one segment and $D(v_i) = \infty$ for all i. We may now repeatedly (1) *decrease* the D-value of some v_i, or (2) *split* a segment. After each operation we need to up-date the minimum D-values of all segments. Each split is assumed to be a split in two, for we can always implement splits into > 2 pieces by several splits in two. Note that for a sequence of length n, we can have at most $n-1$ splits in two.

Gabow has shown that we can accommodate the $n-1$ splits and m decreases in $O(m\alpha(m,n))$ total time [Gabow 1985, §4]. However, here we present an $O(m+n)$ solution. Our improvement is based on atomic heaps [Fredman and Willard 1994]. Whereas Gabow's solution works on a pointer machine and only compares weights, the atomic heaps use the full power of the RAM. As our first step, we show

LEMMA 29. *For a sequence of length n, we can accommodate $\le n-1$ splits and m decreases in $O(m + n\log n)$ time.*

PROOF. First we make a balanced binary sub-division of $v_1, \ldots, v_n$ into intervals. That is, the top-interval is $v_1, \ldots, v_n$ and an interval $v_i, \ldots, v_j$, $j > i$, has the two children $v_i, \ldots, v_{\lfloor (i+j)/2 \rfloor}$ and $v_{\lfloor (i+j)/2 \rfloor + 1}, \ldots, v_j$.

An interval is said to be *broken* when it is not contained in a segment, and any segment is the concatenation of at most $2\log_2 n$ maximal unbroken intervals.

In the process, each vertex has a pointer to the maximal unbroken interval it belongs to, and each maximal unbroken interval has a pointer to the segment it belongs to. For each segment *and* for each maximal unbroken interval, we maintain the minimal D-value. Thus, when a D-value is decreased, if it goes below the current minimal D-value of the maximal unbroken interval containing it, or of the segment containing it, we decrease this minima accordingly, in constant time.

When a segment is split, we may break an interval, creating at most $2\log_2 n$ new maximal unbroken intervals. For each of these disjoint intervals, we visit all its vertices in order to find the minimal D-values, and to restore the pointers from these vertices to the new smaller maximal unbroken intervals containing them. Since each vertex is contained in $\log_2 n$ intervals, the total cost of this process is $O(n \log n)$. Next for each of the two new segments, we find the minimal D-value as the minimum of the minimum D-values of the at most $2\log_2 n$ maximal unbroken intervals they are concatenated from. This takes $O(\log n)$ time per split, hence $O(n \log n)$ total time. □

In order to get down to linear total cost, we will make a reduction to Fredman and Willard's atomic heaps [Fredman and Willard 1994]. Let $T(m, n, s)$ denote the total cost of accommodating m decreases and $n-1$ splits, starting with a sequence of length n that has already been divided into initial segments of length at most s. From Lemma 29, we get

Lemma 30. *$T(m, n, s) = O(m + n \log s)$.*

Proof. We simply apply Lemma 29 to each initial segment. Let n_i denote the length of initial segment i and let m_i denote the number of splits of initial segment i. Then $\sum_i n_i = n$ and $\sum_i m_i = m$, and further, $\forall i, n_i \leq s$. If we now apply Lemma 29 to each initial segment, the total cost becomes $O(\sum_i (m_i + n_i \log n_i)) = O(m + \sum_i (n_i \log s)) = O(m + n \log s)$. □

Lemma 31. *$T(m, n, s) \leq O(m) + T(m, n, \log s) + T(m, n/\log s, s/\log s)$.*

Proof. We view our sequence $v_1, \ldots, v_n$ as divided into $n/\log s$ intervals of $\log s$ consecutive vertices (here, for simplicity, we ignore rounding problems). The splits of $v_1, \ldots, v_n$ into segments further break the intervals into interval segments. Since interval segments are contained in intervals, their initial length is at most $\log s$, and hence we can maintain the minimal D-values of the interval segments in $T(m, n, \log s)$ total time.

We will maintain a sequence $w_1, \ldots, w_{n/\log s}$ of super vertices, with super vertex w_i representing interval i. We define a super segment to be a maximal segment of super vertices not representing any broken intervals. Thus, a split in interval i translates into a super split between w_{i-1} and w_i, and a super split between w_i and w_{i+1}. A split between interval i and $i+1$ becomes a super split between w_i and w_{i+1}. If interval i is unbroken, $D(w_i)$ maintains the minimal D-value of interval i. Note that the minimal D-values of unbroken intervals is already maintained by our maintenance of the minimal D-value for each interval segment. If interval i is broken, $D(w_i)$ is not going to be used, and it can just keep its value. Now, an original segment of $v_1, \ldots, v_n$ of length s contains at most $\lfloor s/\log s \rfloor$ unbroken intervals, so we can maintain the minimal D-value of the super segments in $T(m, n/\log s, s/\log s)$ total time.

Given that we maintain the minimal D-value for each interval segment and for each super segment, we can compute the minimal D-values for each segment $v_k, ..., v_l$ of $v_1, ..., v_n$ as follows. Let i and j be the intervals containing v_k and v_l. If $i = j$, $v_k, ..., v_l$ is an interval segment in interval i, so the desired D-value is directly available. If $i \neq j$, $v_k, ..., v_l$ is composed of the last interval segment of interval i, a maximal sequence of unbroken intervals $i+1, ..., l-1$, with a corresponding super sequence $w_{i+1}, ..., w_{l-1}$, and the rightmost interval segment of interval j. Note, however, that if v_k is the first vertex in interval i, interval i is unbroken, and the super sequence contains w_i. A symmetric case occurs if v_l is the last vertex in interval j. In either case, the minimal D-value is found in constant time as the minimum of at most 2 minimal D-values of interval segments and one minimal D-value of a super segment. □

Lemma 32. *For every integer $i \geq 0$, $T(m,n,n) \leq O(im) + T(m,n,\log^{(i)} n)$. Here $\log^{(0)} n = n$ and $\log^{(j+1)} n = \log(\log^{(j)} n)$.*

Proof. Let c be an upper bound on the constants hidden in the O-notation of Lemma 30 and 31. By induction, we will prove the lemma by showing that

$$T(m,n,n) \leq 3cim + T(m,n,\log^{(i)} n).$$

For $i = 0$, the we are just stating that $T(m,n,n) \leq T(m,n,n)$, which is vacuously true. If $i > 0$, by induction, $T(m,n,n) \leq 3c(i-1)m + T(m,n,\log^{(i-1)} n)$. By Lemma 31, $T(m,n,\log^{(i-1)} n) \leq cm + T(m,n,\log^{(i)} n)) + T(m, n/\log^{(i)} n, \log^{(i-1)} n/\log^{(i)} n)$. By Lemma 30, $T(m, n/\log^{(i)} n, \log^{(i-1)} n/\log^{(i)} n) = c(m + (n/\log^{(i)} n)\log(\log^{(i-1)} n/\log^{(i)} n)) < c(m+n) \leq 2cm$. Adding up, we get $T(m,n,n) \leq 3c(i-1)m + 3cm + T(m,n,\log^{(i)} n) = 3cim + T(m,n,\log^{(i)} n)$, as desired. □

From the construction of atomic heaps in [Fredman and Willard 1994], we have:

Lemma 33. *Given $O(n)$ preprocessing time and space for construction of tables, we can maintain a family $\{S_i\}$ of word-sized integers multisets, each of size at most $O(\sqrt[4]{\log n})$, so that each of the following operations can be done in constant time: insert x in S_i, delete x from S_i, and find the rank of x in S_i, returning the number of elements of S_i that are strictly smaller than x. The total space is $O(n + \sum_i |S_i|)$.*

Lemma 34. *Given $O(n)$ preprocessing time and space for construction of tables, we can maintain a family $\{A_i\}$ of arrays $A_i : \{0, \ldots, s-1\} \to \{0 \ldots, 2^b - 1\}$, $s = O(\sqrt[4]{\log n})$, so that each of the following operation can be done in constant time: assign x to $A_i[j]$ and given i, k, l, find $\min_{l \leq j \leq k} A_i[j]$. Initially, we assume $A_i[j] = \infty$ for all i, j. The total space is $O(n + \sum_i |A_i|)$.*

It should be noted that a proof of Lemma 34 is already implicit in [Willard 1992] in a data structure supporting orthogonal range queries in $O(n \log n / \log\log n)$ time. Nevertheless, for completeness, we present a simple direct proof. Like the proof in [Willard 1992], our proof is based on Lemma 33.

Proof. We use Lemma 33 with S_i being the multiset of elements in A_i. Thus, whenever we change $A_i[j]$, we delete the old value of $A_i[j]$ and insert the new value in S_i. Further, we maintain a function $\sigma_i : \{0, \ldots, s-1\} \to \{0, \ldots, s-1\}$, so that

$\sigma_i(j)$ is the rank of $A_i[j]$ in S_i. Here ties are broken arbitrarily. Now σ_i is stored as a sequence of s $(\log s)$-bit pieces, where the jth piece is the binary representation of $\sigma(j-1)$. Thus, the total bit-length of σ_i is $s \log s = O(\sqrt[4]{\log n} \log\log n) = o(\log n)$. Since $\log n \leq \omega$, this implies that σ_i fits in one register.

By Lemma 33, when we assign x to $A_i[j]$ by asking for the rank of x in $S_i[j]$, we get a new rank r for x in $A_i[j]$. To update σ_i, we make a general transition table Σ, that as entry takes a $\sigma : \{0, \ldots, s-1\} \rightarrow \{0, \ldots, s-1\}$ and $j, r \in \{0, \ldots, s-1\}$. In $\Sigma(\sigma, j, r)$, we store σ' such that $\sigma'(j) = r$, $\sigma'(h) = \sigma(h) + (\sigma(j) - r)/|\sigma(j) - r|$ if $\min\{r, \sigma(j)\} < \sigma(h) < \max\{r, \sigma(j)\}$, and $\sigma'(h) = \sigma(h)$ in all other cases. There are $s^s s^2$ entries to Σ and each takes one word and is computed in time $O(s)$. Since $s = O(\sqrt[4]{\log n})$, if follows that Σ is constructed in $o(n)$ time and space. Using Σ, we up-date σ_i by setting it to $\Sigma[\sigma_i, j, r]$.

To complete the construction, we construct another table Ψ that given $\sigma : \{0, \ldots, s-1\} \rightarrow \{0, \ldots, s-1\}$ and $l, k \in \{0, \ldots, s-1\}$ returns the $j \in \{k, \ldots, l\}$ minimizing $\sigma(j)$. Like Σ, the table Ψ is easily constructed in $o(n)$ time and space. Now, $\min_{l \leq j \leq k} A_i[j]$ is found in constant time as $\Psi[\sigma_i, k, l]$. □

LEMMA 35. *$T(m, n, \sqrt[4]{\log n}) = O(m)$.*

PROOF. Divide $v_1, \ldots, v_n$ into intervals of length $\sqrt[4]{\log n}$. We maintain the D-values for each interval as described in Lemma 34. Now any segment of length $\leq \sqrt[4]{\log n}$ is contained in at most two consecutive intervals, and hence the minimum D-value of any such segment, is found in constant time. □

PROPOSITION 36. *$T(m, n, n) = O(m)$.*

PROOF. Applying Lemma 32 with $i = 2$, $T(m, n, n) \leq O(m) + T(m, n, \log\log n)$. By definition, T is non-increasing in its third argument, so $T(m, n, \log\log n) \leq T(m, n, \sqrt[4]{\log n})$. Now the result follows directly from Lemma 35. □

9. CONCLUSION

Combining Propositions 27, 28, and 36, we have now proved

THEOREM 37. *There is a deterministic linear time and linear space algorithm for the single source shortest path problem for undirected graphs with positive integer weights.*

An objection to our linear time and space algorithm could be that it uses Fredman and Willard's atomic heaps [Fredman and Willard 1994]. As stated in [Fredman and Willard 1994], the atomic heaps are only defined for $n > 2^{12^{20}}$. Further atomic heaps use multiplication, which is not an AC^0 operation. As pointed out in [Andersson et al. 1996], the use of non-AC^0 operations is not inherent. Multiplication may be replaced by some simple selection and copying operations in AC^0 that are just missing in standard instruction sets. These tailor-made instructions would also dramatically reduce the constants. Nevertheless, on todays computers, it is relevant to see how well we can do without the atomic heaps.

The main tool in our new undirected single source shortest path algorithm is the use of buckets. We will now discuss how well we can do if we restrict ourselves to "elementary algorithms" that run on a pointer machine except for the bucketing, and use only standard AC^0 operations. In particular, we will avoid fancy tabulation and bit-fiddling as done in atomic heaps.

Our first problem is in the minimum spanning tree computation in Section 7, which is taken from [Fredman and Willard 1994] and is based on atomic heaps. For our algorithm, it satisfies with a spanning tree that is minimal in the graph where each weight x is replaced by $msb(x) = \lfloor \log_2 x \rfloor$ (recall that msb is found with three standard AC^0 operations: first convert to a double, and then extract the exponent with two shifts). Using simple bucketing, we can trivially sort the edges according to their msb-weights in time $O(\log C + m)$ time, where C is the maximal edge weight. Having presorted msb-weights, using Kruskal's algorithm [Kruskal 1956] with Tarjan's union-find algorithm [Tarjan 1975], an msb-minimum spanning tree is found in time $O(\alpha(m, n)m)$ time. Aiming at the $O(\alpha(m, n)m)$ time bound, we can also replace the tabulation based union-find from [Gabow and Tarjan 1985] with Tarjan's union-find in the construction of $\mathcal{T}$ and $\Delta(\cdot)$. This can even be done on the fly while constructing the spanning tree with Kruskal's algorithm.

The use of atomic heaps in Section 8 is avoided if we instead use Gabow's $O(\alpha(m, n)m)$ solution [Gabow 1985, §4] which works directly on a pointer machine. Adding up, we conclude that we have an elementary algorithm for the SSSP problem with running time $O(\log C + \alpha(m, n)m)$.

The above $\log C$ term stems from having to traverse $msb(C)$ buckets in a bucket sort of the msb-weights. Since C fits in one word, $msb(C)$ is bounded by the word length ω. As pointed out in Section 7, if $\omega \geq m$, the sorting of the msb-weights can be completed in $O(m)$ time by packed merging [Albers and Hagerup 1997; Andersson et al. 1995]. Having $\omega \geq n$ is not really relevant on todays computers as $\omega \leq 64$. However, packed merging essentially uses the RAM as a kind of vector processor with shifts, and it would likely be relevant in case of future processors with huge word lengths. Further, packed merging only uses standard AC^0 operations, so we conclude

THEOREM 38. *On a RAM with standard AC^0 instructions, there is a deterministic $O(\alpha(m, n)m)$ time and linear space space algorithm for the single source shortest path problem for undirected graphs with positive integer weights.*

In conclusion, a deterministic linear time and linear space algorithm has been presented for the undirected single source shortest paths problem with positive integer weights. This theoretically optimal algorithm is not in itself suitable for implementations, but there are implementable variants of it that should work well both in theory and in practice.

ACKNOWLEDGMENTS

I would like to thank the referees from *J.ACM.* for some unusually thorough and useful comments. Also, I would like to thank Stephen Alstrup for valuable discussions on the material of Section 8, and particularly, for pointing me to [Gabow 1985].

APPENDIX

A. FLOATING POINT WEIGHTS

In a standard imperative programming language, we have two types of numbers: integers and floating point numbers. As theoretical computer scientists, we tend to assume that the numbers we deal with are integers with the standard representation. However, in many applications, floating point numbers are at least as likely. It is therefore natural to ask if there is a linear time algorithm for the SSSP problem with floating point weights.

The integer algorithm presented in this paper does not work immediately for floating point weights. Interestingly, this contrasts any algorithm based on improving the priority queue in Dijkstra's algorithm, for we do not need to tell a priority queue if the binary strings it works on represent integers or floating points — the ordering is the same.

In this appendix, we modify our integer SSSP algorithm to work for floating point weights in linear time, except that we cannot in linear time sort the weights x according to $msb(x) = \lfloor \log_2 x \rfloor$, as needed for our construction of the component tree $\mathcal{T}$ in Section 7. The problem is that for a floating point number x, $\lfloor \log_2 x \rfloor$ is the exponent of x, and the exponent is itself an integer, so sorting the weight exponents is a general integer sorting problem, which we do not know how to do in linear time. However, our modified floating point SSSP algorithm does work in linear time if the weight exponents are presorted.

Our floating point SSSP algorithm is combined with a linear time reduction of Thorup [Thorup 1998], reducing any undirected SSSP problem with positive floating point weights to a series of independent floating point SSSP problems of linear total size, and with presorted weight exponents. Since this appendix solves the latter problems in linear time, we will be able to conclude that we have a linear time algorithm for the undirected SSSP problem with positive floating point weights.

A.1 The floating point SSSP problem

A *floating point*, or just a *float*, is a pair $x = (a, b)$, where a is an integer represented the usual way, and b is a bit string $b_1 \cdots b_\wp$. Then x represents the real number $2^a(1 + \sum_{i=1}^{\wp} b_i/2^i)$. Thus $a = \lfloor \log_2 x \rfloor$. We call a the *exponent*, denoted $\mathtt{expo}(x)$, and b the *mantissa*, denoted $\mathtt{mant}(x)$. Often we will identify a float with the real number it represents. Both a and b are assumed to fit in a constant number of words. The number $\wp = \Theta(\omega)$ is fixed throughout a given computation and is referred to as the *precision*. Even though $\wp$ is fixed throughout a given computation, it is not a universal constant, for both $\wp$ and ω are $\Omega(\log n)$. As mentioned in the introduction, according to the IEEE floating point standard, we get the correct ordering of floats by perceiving them as integers. This is obtained by first having the sign-bit of m, then e, and finally m without the sign-bit.

When *two floats x and y are added*, we round the result to the nearest floating point number, rounding up if there is a tie. Alternatively, we may always round down to nearest float, which is theoretically cleaner. The techniques presented here work in either case. We use $\oplus$ to denote floating point addition. It should be noted that floating point addition is not associative, so we define the length of a path from s to some vertex v as the floating point sum of the weights added up starting from s. More specifically, if the path is $P = (v_0, v_1, \ldots, v_l)$, $s = v_0$, $v_l = v$, then the length $\ell(P)$ of P is

$$((((\ell(v_0, v_1) \oplus \ell(v_1, v_2)) \oplus \ell(v_2, v_3)) \cdots) \oplus \ell(v_{l-1}, v_l)).$$

Thus we get the exactly the same distances as those found by Dijkstra's algorithm. Since $\oplus$ is not associative, our decision to add up weights from the start of P could give less precision than a careful ordering of the additions. Here by precision we refer to the ideal case of adding all numbers with unlimited precision, and first round to a float at the very end. However, as discussed in [Thorup 1998], the maximal relative error of adding n numbers is at most $2^{-\wp+\log_2 n}$. Hence, in theory, we can circumvent the rounding errors by simulating an extra $\log_2 n$ bits of precision, and this does not affect the asymptotic running time. In practice, according to the IEEE standard format, with long floats, we have $\wp = 52$. Hence the relative error is at most $2^{-52+\log_2 n}$, which is normally OK.

We will assume that $\wp \geq \omega - 1$, implying that a floats can represent any integers represented in a word precisely.

A.2 Dealing with floats

First, we assume that all weight exponents are at least $\wp$, implying that all the weights are integers. If this is not the case, let $\wp - \alpha$ be the minimal exponent. We then add α to all weight exponents. This has the effect of multiplying all weights, and hence all path lengths by 2^α. We then run our SSSP algorithm, and finally, we subtract α from all the computed distances, to get the distances according to the original weights.

Since all our floating point weights represent integers, our component hierarchy is well-defined. We will argue that our algorithm behaves correctly on floats despite the rounding. We will still have $x \gg i$ denoting $\lfloor x/2^i \rfloor$, but for floats, we cannot achieve this by a simple shift operation. To compute $y = x/2^i$ for a float, we simply subtract i from the exponent of x, and preserve the mantissa. To find $z = \lfloor y \rfloor = x \gg i$, we need to drop any fractional part. If $\texttt{expo}(y) \geq \wp$, $z = y$, but if $\texttt{expo}(y) < \wp$, we need to zero the last $\delta = \wp - \texttt{expo}(y)$ bits of the mantissa. This is done by setting $\texttt{mant}(z) = \delta \ll (\texttt{mant}(y) \gg \delta)$. Sine $\texttt{mant}(y)$ is an integer, $\texttt{mant}(z)$ is computed from $\texttt{mant}(y)$ by two simple integer shifts. The exponent of z is the same as that of y.

The only other algorithmic change needed is that the Δ-values may need to be doubled. Thereafter, our algorithm for integer SSSP works directly for floating point weights with presorted weight exponents. However, parts of the correctness proofs need to be changed substantially. Below, we will describe the changes, section by section. First, however, we present two simple lemmatas, describing the relevant properties of floating point addition. They hold no matter whether floating point addition rounds to nearest float, or rounds down to nearest float.

LEMMA 39. *Let x and y be floats. If $(x \oplus 2^i) \gg i = x \gg i$ and $x \gg i+1 \geq y \gg i+1$, then $x \geq y$.*

PROOF. Since $x \gg i+1 > y \gg i+1$ implies $x > y$ directly, we may assume $x \gg i+1 = y \gg i+1$. Now, if $(x \oplus 2^i) \gg i = x \gg i$, we must have $\texttt{expo}(x) > i + \wp$.

But $\mathtt{expo}(x) > i + \wp$ implies that $x \gg i+1 = x/2^i$. On the other hand, since $x \gg i+1 = y \gg i+1$, $\mathtt{expo}(y) = \mathtt{expo}(x) > i + \wp$, so $y \gg i+1 = y/2^i$. Hence $x \gg i+1 = y \gg i+1$ implies $x = y$. $\square$

LEMMA 40. *Let x and l be floats. Then $x \oplus l \leq x + 2^{\mathtt{expo}(l)+1}$.*

Using the above two lemmatas, we will now present the changes for floats, section by section. The non-trivial changes are all for Section 4.

A.3 Section 2

Lemma 1 and 2 are still valid, but we need a little extra care in the proofs. The shortest path P to v needs to be a *prefix shortest path* meaning a shortest path all of whose prefixes are shortest paths. When we dealt with integers, all shortest paths were trivially prefix shortest paths. However, with floats, there may be a shortest path P' to v with an interior vertex u' such that the prefix of P' to u' is not a shortest path. The point is that extra length of a prefix can disappear in subsequent rounding, and hence not affect the overall shortest path length.

By picking P as a prefix shortest path, we ensure that if u is the first vertex in P outside S, then $D(u) = d(u)$. The remaining parts of the proofs of Lemma 1 and 2 are unchanged. It should be noted that in later proofs, say, the proof of Lemma 41, we will need to reason about shortest paths that may not be prefix shortest paths.

It remains to argue that to any vertex v, we can find a prefix shortest path P. We do this by induction on the distance to v. Consider any shortest path P to v. If P is not a prefix shortest path, let Q be the longest prefix which is not a shortest path and let u be the end vertex of Q. Then $d(u) < \ell(Q) \leq \ell(P) = d(v)$. Hence, inductively, we can find a prefix shortest path Q' to u. Since $\ell(Q') = d(u) < \ell(Q)$, we get a prefix shortest path to v if we replace Q by Q' in P.

A.4 Section 3

This section is only illustrates some basic ideas for avoiding the shorting bottleneck, and it is not needed for a formal correctness proof.

A.5 Section 4

The first lemma that causes problems is Lemma 7, which is simply false with floats. However, it is true if we require that $[v]_i$ is or has been minimal. Below, we first settle the case where $[v]_i$ is minimal. This case suffices for the proof of Lemma 8. Second, we use Lemma 8 in settling the case where $[v]_i$ has been minimal.

LEMMA 41. *Suppose $v \notin S$ and $[v]_i$ is minimal. If there is no shortest path to v where the first vertex outside S is in $[v]_i$, $d(v) \gg i > \min D([v]^-_{i+1}) \gg i$.*

PROOF. Among all shortest paths to v, pick one P so that the first vertex u outside S is in $[v]_k$ with k minimized. By assumption of the lemma, $k > i$. Also, $D(u)$ is a lower bound on the length of the part of P from s to u. We prove the result by induction on $\omega - i$. Thus assume the statement is true for all strictly larger values of i.

If $u \notin [v]_{i+1}$, we have $i + 1 < \omega$ and the minimality of $[v]_i$ implies minimality of $[v]_{i+1}$. Hence, by induction, $d(v) \gg i+1 > \min D([v]^-_{i+2}) \gg i+1$. By minimality of $[v]_{i+1}$, $\min D([v]^-_{i+2}) \gg i+1 = \min D([v]^-_{i+1}) \gg i+1$. Thus, $d(v) \gg i+1 > \min D([v]^-_{i+1}) \gg i+1$, implying $d(v) \gg i > \min D([v]^-_{i+1}) \gg i$.

If $u \in [v]^-_{i+1}$, $D(u) \gg i \geq \min D([v]^-_{i+1}) \gg i$. Moreover, since $u \notin [v]_i$, we know there is an edge e of length $\ell(e) \geq 2^i$ on the part of P between u and v. Hence $d(v) \geq D(u) \oplus 2^i$. If $(D(u) \oplus 2^i) \gg i > D(u) \gg i$, we are done since $D(u) \geq \min D([v]^-_{i+1})$. Hence suppose that $(D(u) \oplus 2^i) \gg i = D(u) \gg i$.

Since $[v]_i$ is minimal, there is a vertex $w \in [v]^-_i$ with $D(w) \gg i = \min D([v]^-_{i+1}) \gg i \leq D(u) \gg i$. Hence, by Lemma 39, $D(w) \leq D(u)$. By definition of S there is a path Q from s to w of length $D(w)$ where w is only vertex outside S in Q. Let Q' by a path from s over w to v whose first part to w is Q, and where the remainder is contained in $[v]_i$.

The length D of Q' is found by first setting $D = D(w)$, and then add the weights of the remaining edges of Q' from $[v]_i$, one by one. In each of these additions, we set $D \leftarrow D \oplus l$ where $l < 2^i$ since l is the weight of an edge in $[v]_i$. We claim that D can never become bigger than $D(u)$. Clearly this is true initially when we set $D = D(w) \leq D(u)$. Inductively, we may assume that $D \leq D(u)$ before every assignment $D \leftarrow D \oplus l$. Since $D \leq D(u)$ and $l \leq 2^i$, $(D \oplus l) \gg i \leq (D(u) \oplus 2^i) \gg i \leq D(u) \gg i$. Hence, by Lemma 39, $D \oplus l \leq D(u)$, as desired.

Since $D \leq D(u) \leq d(v)$, we conclude that Q' is a shortest path from to v. However, $w \in [v]_i$ is the first vertex outside S on Q', contradicting that there was no shortest path to v whose first vertex was outside S was in $[v]_i$. □

Now Lemma 8 is proved exactly as in Section 4, except that we apply Lemma 41 instead of Lemma 7.

LEMMA 42. *Suppose $v \notin S$, $[v]_{i+1}$ is minimal, and $[v]_i$ has been minimal. If there is no shortest path to v where the first vertex outside S is in $[v]_i$, $d(v) \gg i > \min D([v]^-_{i+1}) \gg i$.*

PROOF. The proof is the same as that of Lemma 41 except for two cases. Thus, let u and P be as defined in that proof.

First, if $u \notin [v]_{i+1}$, we get $d(v) \gg i+1 > \min D([v]^-_{i+2}) \gg i+1$ not by induction, but by direct application of Lemma 41 to $[v]_{i+1}$. The rest of this case follows the proof of Lemma 41.

Now, if $u \in [v]^-_{i+1}$, as in the proof of Lemma 41, we conclude that the lemma follows unless $(D(u) \oplus 2^i) \gg i = D(u) \gg i$.

Consider the situation when $[v]_i$ was last minimal. By Lemma 8, there was a vertex $w \in [v]^-_i$ with $D(w) \gg i = \min d([v]^-_{i+1}) \gg i$. By definition of S there is a path Q from s to w of length $D(w)$ where w is only vertex outside S in Q. Note that $D(w)$ may change, but the length of Q is fixed, and we denote it $\ell(Q)$.

Since $\min d([v]^-_i)$ is non-decreasing, in the current situation, $\min d([v]^-_{i+1}) \geq \ell(Q)$. Moreover, $u \in [v]^-_{i+1}$, so $D(u) \geq \min D([v]^-_{i+1}) \geq \min d([v]^-_{i+1})$. Hence, $D(u) \gg i \geq \ell(Q) \gg i$, so $D(u) \geq \ell(Q)$ by Lemma 39.

As in the proof of Lemma 41, let Q' by a path from s over w to v whose first part to w is Q, and where the remainder is contained in $[v]_i$. The length D of Q' is may be found by first setting $D \leftarrow \ell(Q)$, and then add the weights of the remaining edges of Q' from $[v]_i$, one by one. In each of these additions, we set $D \leftarrow D \oplus l$ where $l < 2^i$. As in the proof of Lemma 41, we conclude that D can never increase beyond $D(u)$, hence that Q' contradicts that there was no shortest path to v whose first vertex was outside S was in $[v]_i$. $\square$

Using Lemma 42, we can now prove a slightly weaker version of Lemma 9.

LEMMA 43. *If $v \notin S$ and $[v]_i$ is not minimal but $[v]_{i+1}$ is minimal and $[v]_i$ has been minimal, then* $\min d([v]_i^-) \gg i > \min D([v]_{i+1}^-) \gg i$.

PROOF. Consider any $w \in [v]_i^-$ and let u be the first vertex outside S on a shortest path from s to w. If $u \notin [v]_i$, the result follows directly from Lemma 42. However, if $u \in [v]_i$, we have $d(w) \geq D(u) \geq \min D([v]_i^-)$. Moreover, the non-minimality of $[v]_i$ implies $\min D([v]_i^-) \gg i > \min D([v]_{i+1}^-) \gg i$. Hence, we conclude that $d(w) \gg i > \min D([v]_{i+1}^-) \gg i$ for all $w \in [v]_i^-$, as desired. $\square$

A.6 Section 5

We can prove all the results of Section 5, simply replacing all applications of Lemma 9 with application of Lemma 43. This is valid because $[v]_i$ has been minimal in all applications of Lemma 9 in Section 5. One more change, however, is the last line

$$\langle \min D([v]_i^-) \gg i \rangle^a \leq \langle D(x) \gg i \rangle^a \leq (d(u) + \ell(u,x)) \gg i \leq \langle \min D([v]_i^-) \gg i \rangle^b + 1.$$

of the proof of Lemma 11. There we used that $l < 2^i$, implied $(x + l) \gg i \leq (x \gg i) + 1$. However, by Lemma 40, we have $(x \oplus l) \gg i \leq (x + 2^i) \gg i \leq (x \gg i) + 1$, so we can simply replace $+$ by $\oplus$ in the last line, and it still be valid.

A.7 Section 6

Our only problem is that the distance between $\min d([v]_i)$ and $\max d([v]_i)$ may be increased due sums being rounded up. By Lemma 40, however, we can anticipate rounding up by setting $\Delta([v]_i) = \lceil \sum_{e \in [v]_i} 2^{\mathtt{expo}(\ell(e))+1-i} \rceil$. This, can only double the size of the bucket structure, and hence not affect our asymptotic bound. Another minor problem is that when we compute a bucket index, we should convert it to integer representation. However, any bucket index is $O(m)$, so the conversion to integer will not cause an overflow.

A.8 Section 7

The only change in this section is that the weights x are already presorted according to their exponents $msb(x) = \lfloor \log_2 x \rfloor$.

A.9 Section 8

For the data structure in this section, we are only interested in the ordering of certain distances found. Hence, as for priority queues, we can just view the bit strings of these floating point distances as representing integers, and hence we do not need any algorithmic changes.

A.10 Conclusion

We have shown that our SSSP algorithm for integers can be adapted for floats, given that the weight exponents are presorted as described in [Thorup 1998]. Thus, we have a linear time and space algorithm for the undirected SSSP problem with positive floating point weights.

REFERENCES

AHUJA, R. K., MELHORN, K., ORLIN, J. B., AND TARJAN, R. E. 1990. Faster algorithms for the shortest path problem. *J. ACM 37*, 213–223.

ALBERS, S. AND HAGERUP, T. 1997. Improved parallel integer sorting without concurrent writing. *Inf. Contr. 136*, 25–51.

ANDERSSON, A., HAGERUP, T., NILSSON, S., AND RAMAN, R. 1995. Sorting in linear time? In *Proc. 27th STOC* (1995), pp. 427–436.

ANDERSSON, A., MILTERSEN, P. B., AND THORUP, M. 1996. Fusion trees can be implemented with AC^0 instructions only. Technical Report BRICS-TR-96-30, Aarhus. To appear in *Theor. Comp. Sc.*

CHERKASSKY, B. V., GOLDBERG, A. V., AND SILVERSTEIN, C. 1997. Buckets, heaps, lists, and monotone priority queues. In *Proc. 8th SODA* (1997), pp. 83–92.

DIJKSTRA, E. W. 1959. A note on two problems in connection with graphs. *Numer. Math. 1*, 269–271.

DINIC, E. A. 1978. Finding shortest paths in a network. In Y. POPKOV AND B. SHMULYIAN Eds., *Transportation Modeling Systems*, pp. 36–44. Institute for System Studies, Moscow.

FREDMAN, M. L. AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 596–615.

FREDMAN, M. L. AND WILLARD, D. E. 1993. Surpassing the information theoretic bound with fusion trees. *J. Comp. Syst. Sc. 47*, 424–436.

FREDMAN, M. L. AND WILLARD, D. E. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comp. Syst. Sc. 48*, 1994, 533–551.

GABOW, H. N. 1985. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th FOCS* (1985), pp. 90–100.

GABOW, H. N. AND TARJAN, R. E. 1985. A linear-time algorithm for a special case of disjoint set union. *J. Comp. Syst. Sc. 30*, 209–221.

HENZINGER, M. R., KLEIN, P., RAO, S., AND SUBRAMANIAN, S. 1997. Faster shortest-path algorithms for planar graphs. *J. Comp. Syst. Sc. 53*, 2–23.

KRUSKAL, J. B. 1956. On the shortest spanning subtree of a graph and the traviling salesman proble. *Proc. AMS 7*, 48–50.

MELHORN, K. AND NÄHLER, S. 1990. Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space. *Inf. Proc. Lett. 35*, 183–189.

RAMAN, R. 1996. Priority queues: small monotone, and trans-dichotomous. In *Proc. 4th ESA, LNCS 1136* (1996), pp. 121–137.

RAMAN, R. 1997. Recent results on the single-source shortest paths problem. *SICACT News 28*, 81–87.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 215–225.

THORUP, M. 1996. On RAM priority queues. In *Proc. 7th SODA* (1996), pp. 59–67.

THORUP, M. 1998. Floats, integers, and single source shortest paths. In *Proc. 15th STACS, LNCS 1373* (1998), pp. 14–24.

VAN EMDE BOAS, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Lett. 6*, 80–82.

VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. 1977. Design and implementation of an efficient priority queue. *Math. Syst. Th. 10*, 99–127.

WILLARD, D. E. 1992. Applications of the fusion tree method to computational geometry and searching. In *Proc. 3rd SODA* (1992), pp. 386–395.

WILLIAMS, J. W. J. 1964. Heapsort. *Comm. ACM 7*, 347–348.