Undirected Single Source Shortest Paths in Linear Time*

Mikkel Thorup
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk, http://www.diku.dk/~mthorup

Abstract

The single source shortest paths problem (SSSP) is one of the classic problems in algorithmic graph theory: given a 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 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 integer weights. The algorithm avoids the sorting bottle-neck by building a hierechical bucketing structure, identifying vertex pairs that may be visited in any order.

1 Introduction

Let $G = (V, E)$, $|V| = n$, $|E| = m$, be an undirected connected graph with an integer edge weight function $\ell : E \to \mathbb{N}$ and a distinguished source vertex $s \in V$. If $(v, w) \notin E$, $\ell(v, w) = \infty$. *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 determinstic linear time and linear space algorithm for undirected SSSP with integer weights. So far a linear time SSSP algorithm has only been known for planar graphs [KRRS97].

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 w. Addresses are themselves contained in words, so $w \geq \log n$. 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 mulitiplication, 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 each fit in one word so that the input and output size is $O(m)$; otherwise the output size may be assymptotically 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.

In contrast to the RAM, we have the pointer machine model, disallowing address arithmetic, and hence bucketting which is essential to our algorithm. Also, we have the comparison based

*A preliminary version is to be presented at FOCS'97

model where weights may only be compared. For SSSP, however, we also need to add weights along paths. Of all the algorithms described below, it is only those from [Dij59, Wil64, FT87] that work in any of these restricted models.

Since 1959, all theoretical developments in SSSP for general directed or undirected graphs have been based on Dijkstra's algorithm [Dij59]. 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 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 of 1964, we get $O(m \log n)$ time [Wil64]. Fredman and Tarjan's Fibinacci heaps [FT87] 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, that like this paper assume integer weights stored in words, we get an $O(m\sqrt{\log n})$ randomized bound [FW93]. Their later atomic heaps give an $O(m + n \log n / \log\log n)$ bound [FW94]. 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 [Tho96]. These bounds are randomized assuming that we want linear space. Later, Raman has obtained an $O(m + n\sqrt{\log n \log\log n})$ bound in deterministic linear space using standard AC^0 instructions only [Ram96].

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 [MN90, vBoa77, vBKZ77], 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 $O(n\sqrt{\log C} + m)$ [AMOT90]. Recently, this has been improved by Cherkassky, Goldberg, and Silverstein to $O(n\sqrt[3]{\log C \log\log C} + m)$ expected time [CGS97]. Raman has observed that the construction of Cherkassky et.al. actually gives randomized expected bounds of $O(n(\log C)^{1/4+\varepsilon} + m)$ and $O(n(\log\log n)^{1/3+\varepsilon} + m)$ [Ram97].

As observed in [FT87], 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 [Tho96]. 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 [Din78] 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 bucketting 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.

It should be noted that our algorithm does *not* work for floating point weights. This contrasts any Dijkstra's implementation, for we do not need to tell the priorty queue if the binary strings it works on represent integers or floating points — the ordering is the same. A paper is in preparation, however, showing that a linear time oracle for undirected integer SSSP gives a linear time algorithm for the case of floating points.

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 Dijksta's algorithm. In particular, concerning S and D, $\forall v \in S : D(v) = d(v)$ and $\forall v \in V \setminus S : D(v) = \min_{u \in S}\{d(u) + \ell(u, v)\}$

We will let b denote the word length. We will write $\lfloor x/2^i \rfloor$ as $x \downarrow i$ to emphasize that it is calculated simply by shifting out the i least significant bits. Note that $x \leq y \Rightarrow x \downarrow i \leq y \downarrow i$ while $x < y \Leftarrow x \downarrow i < y \downarrow i$.

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 $\min \emptyset = \infty$

3 The component hierachy

From Dijkstra's algorithm, we know that $D(v) = d(v)$ if $v \in V \setminus S$ minimizes $D(v)$. In this section, we will identify a more flexible condition for $D(v) = d(v)$. Thus, we will show how to avoid the sorting bottle-neck inherent in Dijkstra's algorithm.

The *component hierachy* is now defined as follows. By G_i we denote the subgraph of G whose edge set is the edges e from G with $\ell(e) < 2^i$. Thus $G_b = G$ while G_0 consists of singleton vertices. On level i in the component hierachy, we have the componenets (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 $w \in [v]_i$.

Observation 1 *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 lenght $\geq 2^i$. ■

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 $\min(D([v]_i^-) \downarrow i = \min(D([v]_{i+1}^-) \downarrow i$, and that $[v]_i$ is *minimal* if for $j = i, \ldots, b-1$, $[v]_j$ is a min-child of $[v]_{j+1}$.

Below, in Corollary 4, 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, $\forall i : \min D([v]_i^-) \downarrow i = D([v]_{i+1}^-) \downarrow i = D(v) \downarrow i$, 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 hings, 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 5, where it will be used to limit the size of an underlying bucketting structure.

Lemma 2 *If $[v]_i$ is minimal and $j \geq i$, then* $\min D([v]_i^-) \downarrow j - 1 = \min D([v]_j^-) \downarrow j - 1$.

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

Lemma 3 *Let $[v]_{i+1}$ be minimal and u be the first vertex outside S on a shortest path from s to v. If $u \notin [v]_i$, $d(v) \downarrow i > \min D([v]_{i+1}^-) \downarrow i$; otherwise $d(v) \geq \min D([v]_i^-)$.*

Proof: Since u is the first vertex outside S on a shortest path from s to v, $D(u) = d(u)$ and $d(v) = d(u) + dist(u, v)$. If $u \in [v]_i$, $d(v) \geq d(u) = D(u) \geq \min D([v]_i^-)$.

We prove the statement of the lemma for $u \notin [v]_i$ by induction on $b - i$. In the base case with $i = b$, $[v]_b = G$, so u cannot be outside $[v]_i$, and the statement is vacuously true. Thus, assume $i < b$.

If $u \notin [v]_{i+1}$, by induction, $d(v) \downarrow i + 1 > \min D([v]^-_{i+2}) \downarrow i + 1$. By minimality of $[v]_{i+1}$, $\min D([v]^-_{i+2}) \downarrow i + 1 = \min D([v]^-_{i+1}) \downarrow i + 1$. Thus, $d(v) \downarrow i + 1 > \min D([v]^-_{i+1}) \downarrow i + 1$, implying $d(v) \downarrow i > \min D([v]^-_{i+1}) \downarrow i$.

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

Since $D(v) \geq d(v)$ for all v, we get some immediate corollaries:

Corollary 4 *If $[v]_i$ is minimal, $\min D([v]^-_i) = \min d([v]^-_i)$. In particular, $D(v) = d(v)$ if $[v]_0 = \{v\}$ is minimal.*

The above corollary gives us our basic condition for visiting vertices, moving them to S. Our algorithms will need one more corollary of Lemma 3:

Corollary 5 *If $[v]_i$ is not minimal but $[v]_{i+1}$ is minimal, then $\min d([v]^-_i) \downarrow i > \min D([v]^-_{i+1}) \downarrow i$.*

4 Visiting minimal vertices

In this section, we will discuss the basic dynamics of visting vertices v with $[v]_0$ minimal. First we show a series of lemmas culminating in Lemma 11, stating that if $[v]_i$ has once been minimal, $\min D([v]^-_i) \downarrow i = \min d([v]^-_i) \downarrow i$ in all future. Based on this, we will present our first concrete SSSP algorithm.

Definition 6 *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 $(v, w) \in E$.*

The following facts now follow immediately from Corollary 4:

Fact 7 *When we visit a vertex v, $D(v) = d(v)$.*

Fact 8 *For all $[v]_i$, $\max d([v]_i \setminus [v]^-_i) \downarrow i \leq \min d([v]^-_i) \downarrow i$.*

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. Note that whatever vertex we visit,

$$\langle \min d([v]^-_i) \rangle^a \geq \langle \min d([v]^-_i) \rangle^b \tag{1}$$

Also, since $\forall u : D(u) \geq d(u)$,

$$\langle \min D([v]^-_i) \rangle^b = \langle \min d([v]^-_i) \rangle^b \;\Rightarrow\; \langle \min D([v]^-_i) \rangle^a \geq \langle \min D([v]^-_i) \rangle^b \tag{2}$$

Lemma 9 *Suppose $\min D([v]^-_i) \downarrow i = \min d([v]^-_i) \downarrow i$ and visiting some vertex $w \in V \setminus S$ changes $\min D([v]^-_i) \downarrow i$. Then $w \in [v]_i$ and if $[v]^-_i$ is not emptied, the change in $\min D([v]^-_i) \downarrow i$ is an increase by one.*

Proof: We are studying the event of visiting the vertex w. Since $\langle \min D([v]_i^-)\rangle^b = \langle \min d([v]_i^-)\rangle^b$, by (2), $\langle \min D([v]_i^-)\rangle^a \geq \langle \min D([v]_i^-)\rangle^b$. By assumption, $\langle \min D([v]_i^-)\rangle^a \neq \langle \min D([v]_i^-)\rangle^b$, so $\langle \min D([v]_i^-)\rangle^a > \langle \min D([v]_i^-)\rangle^b$. 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

$$d(u) \downarrow i \leq \min\langle D([v]_i^-\rangle^b) \downarrow i. \tag{3}$$

If $u \notin \langle [v]_i^-\rangle^b$, (3) follows directly from Fact 8. Otherwise $u = w$. By Corollary 4 and Lemma 2, the minimality of $[u]_0 = [w]_0$ implied, $d(u) \downarrow i = D(u) \downarrow i = \langle \min D([v]_i^-) \downarrow i\rangle^b$. Thus (3) follows.

Based on (3), since $\ell(u, x) < 2^i$, we conclude

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

■

In connection with Lemma 9, 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 10 *If $[v]_i$ is minimal, it remains minimal until either $[v]_i^- = \emptyset$ or $\min D([v]_i^-) \downarrow i$ is increased. In the latter case, $\min d([v]_i^-) \downarrow i$ is also increased.*

Proof: Suppose $[v]_i$ is minimal, but visiting some vertex w stops $[v]_i$ from being minimal. There is then a minimal value $j \geq i$ and a vertex u such that $[v]_{j+1} = [u]_{j+1}$ and $[u]_j$ is minimal after the event of visiting w. If $\langle [v]_i^-\rangle^a \neq \emptyset$, the result is trivial, so we assume that $\langle [v]_i^-\rangle^a \neq \emptyset$.

Before the visit to w, $[v]_i$ was minimal, so $\langle \min D([v]_i^-) \downarrow j\rangle^b = \langle \min D([v]_{j+1}^-) \downarrow j\rangle^b$ by Lemma 2. Also, $[v]_{j+1}$ was minimal, so by Corollary 4 and (2), $\langle \min D([v]_{j+1}^-)\rangle^a \geq \langle \min D([v]_{j+1}^-)\rangle^b$.

After the visit, since $[v]_{j+1}$ is minimal and $[v]_j$ is not a min-child of $[v]_{j+1}$, by Corollary 5, $\langle \min d([v]_j^-) \downarrow j\rangle^a > \langle \min D([v]_{j+1}^-) \downarrow j\rangle^a$. Thus

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

■

Lemma 11 *If $[v]_i$ has once been minimal, in all future,*

$$\min D([v]_i^-) \downarrow i = \min d([v]_i^-) \downarrow i. \tag{4}$$

Proof: First time $[v]_i$ turns minimal, (4) gets satisfied by Corollary 4. Now, suppose (4) is satisfied before visiting some vertex w. Since $\forall u : D(u) \geq d(u)$, (4) can only be violated by an increase in $\min D([v]_i^-)$. If $\min D([v]_i^-)$ is increased, by Lemma 9, $w \in [v]_i$ 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, (4) follows from Corollary 4. Also, if $[v]_i^-$ is emptied, (4) follows with $\min D([v]_i^-) \downarrow i = \min d([v]_i^-) \downarrow i = \infty$. If $[v]_i$ becomes non-minimal and $[v]_i^-$ is not emptied, by Lemma 10, $\min d([v]_i^-) \downarrow i$ is also increased. Since $\min d([v]_i^-) \downarrow i \leq \min D([v]_i^-) \downarrow i$ and $\min D([v]_i^-) \downarrow i$ was increased by one, we conclude that (4) is restored. ■

We are now ready to derive a first algorithm for the undirected single source shortest paths problem. The algorithm is so far inefficient, but it presents some of the basic ideas in how we intend to visit the vertices in a linear time algorithm.

Algorithm A: **SSSP**, given $G = (V, E)$ with weight function ℓ and distinguished vertex s, outputs D with $D(v) = d(v) = dist(s, v)$ for all $v \in V$.

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

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

A.3. Visit($[s]_b$)

A.4. return D

Algorithm B: **Visit**($[v]_i$), assuming that $[v]_i$ is mininal, it visits all $w \in [v]_i^-$ with $d(w) \downarrow i$ equal to the value of $\min D([v]_i^-) \downarrow i$ when the call is made.

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

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

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

B.3.1. while $\exists$ child $[w]_{i-1}$ of $[v]_i : \min D([w]_{i-1}^-) \downarrow i - 1 = ix([v]_i)$,

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

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

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

Correctness: We now prove that Algorithm B is correct, that is, if $[v]_i$ is minimimal, Visit($[v]_i$) visits all $w \in [v]_i^-$ with $d(w) \downarrow i$ equal to the value of $\min D([v]_i^-) \downarrow i$ when the call is made. The proof is by induction on i. For the base case with $i = 0$, we just visit v in Step B.1, which is trivially correct. Thus, we may assume that $i > 0$. Inductively, if a call Visit($[w]_{i-1}$) (step B.3.1.2) is made with $[w]_i$ minimal, we may assume that it correctly visits all $u \in [w]_{i-1}^-$ with $d(u) \downarrow i - 1$ equal to the value of $\min D([u]_{i-1}^-) \downarrow i - 1$ when the call is made. We will prove the following invariants for when $[v]_i^- \neq \emptyset$:

$$ix([v]_i) \downarrow 1 = \min D([v]_i^-) \downarrow i = \min d([v]_i^-) \downarrow i \tag{5}$$

$$ix([v]_i) \leq \min d([v]_i^-) \downarrow i - 1 \tag{6}$$

When $ix([v]_i)$ is first assigned in step B.2, it is assigned $\min D([v]_i^-) \downarrow i - 1$. Also, at that time, $[v]_i$ is minimal, so $\min D([v]_i^-) = \min d([v]_i^-)$ by Corollary 4. Thus $ix([v]_i) = \min D([v]_i^-) \downarrow i - 1 = \min d([v]_i^-) \downarrow i - 1$, implying both (5) and (6). Now, assume (5) and (6) both holds at the beginning of an iteration of the repeat-loop B.3.

Claim A If $\min D([v]_i^-) \downarrow i$ has not increased, $[v]_i$ remains minimal and (5) and (6) remain true.

Proof: By Lemma 10 and Corollary 4, $[v]_i$ is remains minimal with $\min D([v]_i^-) = \min d([v]_i^-)$. Then, by (2), $\min D([v]_i^-)$ is non-decreasing, so a violation of (6) should be due to an increase in $ix([v]_i)$. However, $ix([v]_i)$ is only increased in step B.3.2, which is only entered if $\forall [w]_{i-1} \subseteq [v]_i : \min D([w]_{i-1}^-) \downarrow i - 1 \neq ix([v]_i)$. Before the increase, by (6), $ix([v]_i) \leq \min D([v]_i^-) \downarrow i - 1 = \min_{[w]_{i-1} \subseteq [v]_i} \min D([w]_{i-1}^-) \downarrow i - 1$. Hence $ix([v]_i) < \min D([v]_i^-) \downarrow i - 1 = \min d([v]_i^-) \downarrow i - 1$, so the increase by one cannot violate (6). Moreover, since $\min D([v]_i^-) \downarrow i$ is not increased and $\min D([v]_i^-) = \min d([v]_i^-)$, (6) implies (5). □

Claim B If a call Visit($[w]_{i-1}^-$) (step B.3.1.2) is made before $\min D([v]_i^-) \downarrow i$ is increased, all vertices u visited have $d(u) \downarrow i$ equal to the origianal value of $\min D([v]_i^-) \downarrow i$ (as required for visits within Visit($[v]_i$)).

PROOF: When Visit$([w]^-_{i-1})$ is called, Claim A applies and $ix([v]_i) = \min D([w]^-_{i-1}) \downarrow i-1 \geq \min D([v]^-_i) \downarrow i-1 \geq \min d([v]^-_i) \downarrow i-1$. Then, by (6), $\min D([w]^-_{i-1}) \downarrow i-1 = \min D([v]^-_i) \downarrow i-1$, so $[w]_i$ inherits the minimality of $[v]_i$. Hence, by induction, Visit$([w]^-_{i-1})$ correctly visits the vertices $u \in [w]^-_i$ with $d(u) \downarrow i-1$ equal to the value of $\min D([w]^-_{i-1}) \downarrow i-1$ at the time of the call. However, at the time of the call, $\min D([w]^-_{i-1}) \downarrow i-1 = ix([v]_i)$ and by (5), $ix([v]_i) \downarrow 1 = \min D([v]^-_i) \downarrow i$, so $d(u) \downarrow i = \min D([v]^-_i) \downarrow i$. □

CLAIM C $\min D([v]^-_i) \downarrow i$ has increased when the repeat-loop B.3 terminates.

PROOF: If $\min D([v]^-_i) \downarrow i$ did not increase, (6) holds by Claim A, and (6) implies $ix([v]_i) \downarrow 1 \leq \min D([v]^-_i) \downarrow i$. Initially we have equallity by (5). However, the repeat-loop can only terminate if $ix([v]_i) \downarrow 1$ increases or $[v]^-_i$ becomes empty, setting $\min D([v]^-_i) \downarrow i = \infty$. □

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

By Lemma 11, $\min d([v]^-_i) \downarrow i$ increases with $\min D([v]^-_i) \downarrow i$. Hence by Claim B, Visit$([w]_i)$ will visit no more vertices. Moreover, it implies that we have visited all vertices $u \in [v]_i$ with $d(u) \downarrow i$ equal to the original value of $\min D([v]^-_i)$, so we have now visited exactly the required vertices.

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

Since no more vertices are visited after the increase of $\min D([v]^-_i) \downarrow i$, by Lemma 9, the increase is by one. Thus, we conclude that all of $ix([v]_i)$, $D([v]^-_i) \downarrow i$, and $\min d([v]^-_i) \downarrow i$ are increased by one, restoring the equalities of (5). Since, $ix([v]_i)$ now has the smallest value such that $ix([v]_i) \downarrow 1 = \min d([v]_i) \downarrow i$, we conclude that (6) is also satisfied.

By Lemma 11, in all future $\min d([v]^-_i) \downarrow i = \min D([v]^-_i) \downarrow i$. Moreover, $ix([v]_i)$ and $\min d([v]^-_i)$ can only change in connection with calls Visit$([v]_i)$, so we conclude that (5) and (6) will remain satisfied untill the next such call. ■

5 Towards a linear time algorithm

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

The component tree

Define the *the component tree* $\mathcal{T}$ representing the topological structure of the component hierachy, 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 hierachy. Since $\mathcal{T}$ have no degree one nodes, the number of nodes is $\leq 2n-1$. In Section 6 we show how to construct $\mathcal{T}$ in time $O(m)$. Given $\mathcal{T}$, it is straightforward to modify Algorithm B so that it recurses within $\mathcal{T}$. 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 hierachy. A min-child $[w]_h$ of $[v]_i$ is minimizing $\min D([w]^-_h) \downarrow i-1$. Thus minimality of components is inherited from the component hierachy to $\mathcal{T}$.

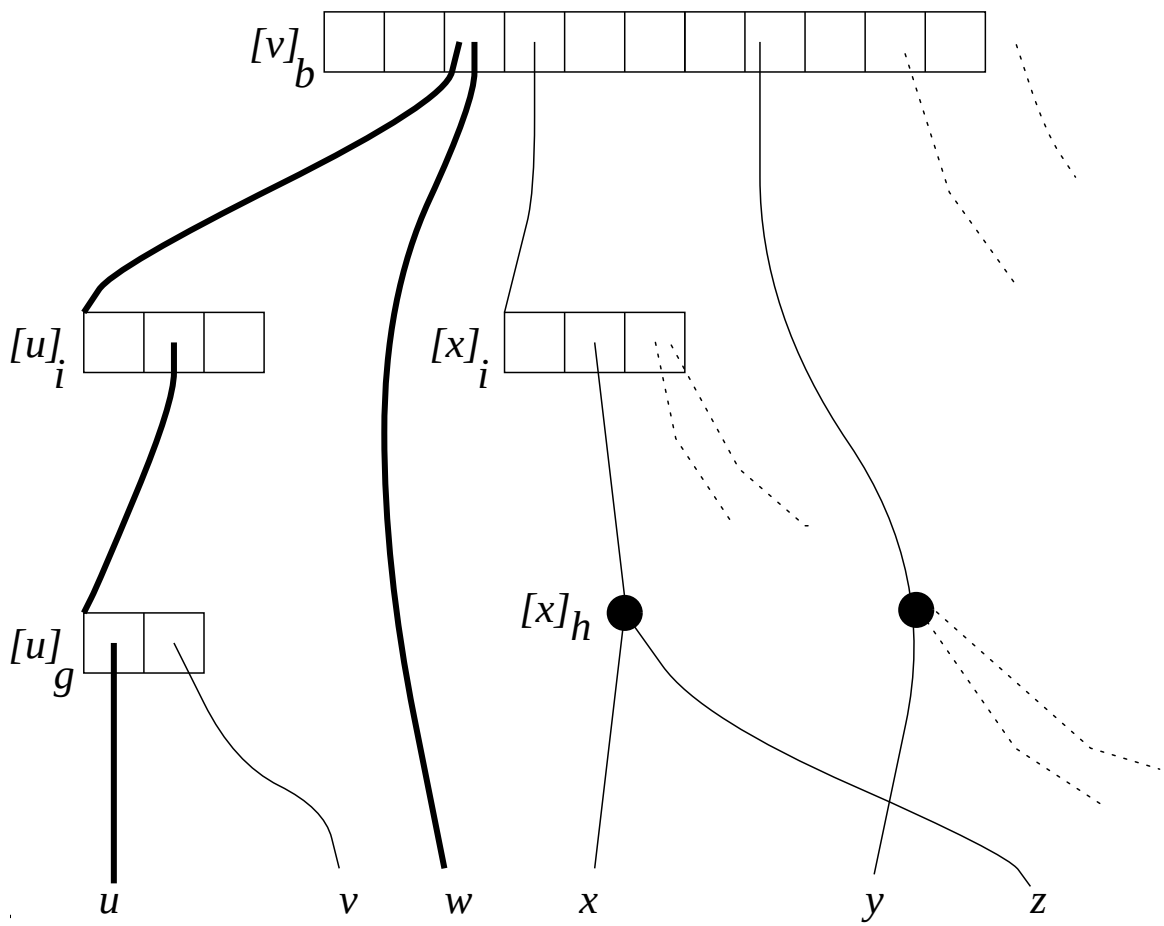

Figure 5.1: Bucketting of components

A linear sized bucketing structure

The idea now is that for each visited node $[v]_i$, we will bucket the children $[w]_h$ according to $\min D([w]_h^-) \downarrow i-1$. That is, $[w]_h$ is found in $B([v]_i, \min D([w]_h^-) \downarrow i-1)$. With $ix([v]_i) = \min D([v]_i^-) \downarrow i-1$ as in Algorithm B, the minimal children of $[v]_i$ are readily found in $B([v]_i, ix([v]_i))$. The bucketing is illustrated in Figure 5.1.

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

Let $ix_0([v]_i)$ denote $\min d([v]_i) \downarrow i-1$. Then $ix_0([v]_i)$ is also giving a lower bound on all possible values of $D(w) \downarrow i-1$ for $w \in [v]$, and hence $ix_0([v]_i)$ is the minimal relevant index for $B([v]_i, \cdot)$.

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 $ix_\infty([v]_i) = ix_0 + \Delta([v]_i)$. Then

$$\max d([v]_i) \downarrow i-1 \leq ix_\infty([v]_i) \tag{7}$$

If $[w]_h$ is minimal and child of $[v]_i$, by Corollary 4, $\min D([w]_h^-) \downarrow i-1 = \min d([w]_h^-) \downarrow i-1 \leq ix_\infty([v]_i)$. For this reason, for each visited component $[v]_i \in \mathcal{T}$, it is only for

$q = ix_0([v]_i), \ldots, ix_\infty([v]_i)$ that the bucket $B([v]_i, q)$ is called *relevant*.

Lemma 12 *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. Thus, the total number of relevant buckets is $\leq 4n + \sum_{[v]_i \in \mathcal{T},\ e \in [v]_i} \ell(e)/2^{i-1}$.

For any edge e, there is a least bucket $[v]_i$ such that $e \in [w]_j \iff j \geq i \wedge [w]_j = [v]_j$. Since $\ell(e) < 2^i$,

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

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

We will now show how to efficiently embed the 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 6. 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, a 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 $A(x + N([v]_i) - ix_0([v]_i))$; otherwise, the contents of $B([v]_i, x)$ is deferred to the "waste" bucket $A(N)$. Each bucket is represented by a doubly linked list, so that elements can be inserted and deleted in constant time. In conclussion, the bucket structure $B(\cdot,\cdot)$ is constructed in linear time and space, and elements can be inserted and deleted in constant time.

Bucketing unvisited children

Let $\mathcal{U}$ denote the unvisited part of $\mathcal{T}$. Supposing $\mathcal{U} \neq \mathcal{T}$, 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 7, we will show that for all roots $[v]_i$ in $\mathcal{U}$, we can maintain $\min D([v]_i)$. Note that if $[v]_i$ has not been visited $[v]_i^- = [v]_i$. Thus, if $[v]_i$ is unvisited and has visited parent $[v]_j$, it belongs in $B([v]_j, \min D([v]_i) \downarrow j-1)$. When a component is first visited, all its children need to be bucketed:

Algorithm C: **Expand**$([v]_i)$, assuming that $[v]_i$, $i > 0$, is an unvisited and mininal child of a visited component in $\mathcal{T}$, it places all children of $[v]_i$ in $B([v]_i, \cdot)$.

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

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

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

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

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

C.5.1. place $[w]_h$ in $B([v]_i, \min D([w]_h) \downarrow i-1)$.

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 D: **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.

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

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

D.2.1. let $[w]_i$ be the root of $[w]_0$ in $\mathcal{U}$ and let $[w]_j$ be the parent of $[w]_i$ in $\mathcal{T}$.

D.2.2. decrease $D(w)$ to $D(v) + \ell(v, w)$.

D.2.3. if this decreases $\min D([w]_i) \downarrow (j-1)$ then

D.2.3.1. move $[w]_i$ to $B([w]_j, \min D([w]_i) \downarrow (j-1))$

An efficient visiting algorithm

We are now ready to present an efficient version of Algorithm B. Besides the calls to the above to Algorithms C and D, the following two changes should be noted. First, the parent of $[v]_i$ may not be $[v]_{i+1}$ but $[v]_j$ for some $j > i+1$. For this reason, j has to be passed as a parameter. Second, the bucket position of $[v]_i$ is updated as we change $ix([v]_i)$.

Algorithm E: **Visit**$([v]_i, j)$, assuming that $[v]_i^- \neq \emptyset$ and that $[v]_i$ is mininal and a child of $[v]_j$ in $\mathcal{T}$, it visits all $w \in [v]_i^-$ with $d(w) \downarrow j-1$ equal to the value of $\min D([v]_i^-) \downarrow j-1$ when the call is made. Updates the ix-values and bucketing of all components equal to or descending from $[v]_i$.

E.1. if $i = 0$, Visit(v) and return

E.2. if $[v]_i$ has not been visited previously,

E.2.1. Expand$([v]_i)$

E.2.2. $ix([v]_i) \leftarrow ix_0([v]_i)$

E.3. repeat until $[v]_i^- = \emptyset$ or $ix([v]_i) \downarrow (j-i)$ is increased:

E.3.1. while $B([v]_i, ix([v]_i)) \neq \emptyset$,

E.3.1.1. let $[w]_h \in B([v]_i, ix([v]_i))$

E.3.1.2. Visit$([w]_h, i)$

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

E.4. if $[v]_i^- = \emptyset$, delete $[v]_i$ from $B(\cdot, \cdot)$

E.5. if $[v]_i^- \neq \emptyset$, move $[v]_i$ to $B([v]_j, ix([v]_i) \downarrow (j-i))$

Correctness: Since $[v]_i = [v]_{j-1}$, Lemma 10 states that $[v]_i$ will remain minimal until $\min D([v]_i^-) \downarrow j-1$ increases. Thus, except for the bucketing, the proof of correctness is a straightforward modification of the correctness proof for Algorithms B. In particular, the invariants (5) and (6) still hold. By (5), $ix([v]_i) \downarrow 1 = \min D([v]_i^-) \downarrow i$, and in step E.5, we rebucket $[v]_i$ whenever $ix([v]_i) \downarrow (j-i) = \min D([v]_i^-) \downarrow j-1$ is changed. Hence we conclude that $[v]_i$ is correctly bucketed before and after each call Visit$([v]_i, j)$. This completes the proof of correctness for Algorithm E. ■

Proposition 13 *Given the component tree $\mathcal{T}$, and given that we for each root $[v]_i$ in $\mathcal{U}$ can maintain $\min D([v]_i)$, the SSSP problem is solved in $O(m)$ time and space.*

Proof: Except from the bucketing, the space bound is trivial. Lemma 12 saying 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 D) is $O(m)$. The time spent in Expand($[v]_i$) (Algorithm C) is proportional to the number of relevant buckets in $B([v]_i, \cdot)$, so by Lemma 12, the total time spent in Expand is $O(m)$.

The complexity of Visit($[v]_i$) (Algorithm E) 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 12, 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 $ix([v]_i) \downarrow j - i = \min D([v]_i^-) \downarrow j - 1$ 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 E.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 E.3.1, is paid for by the call Visit($[w]_h$). ■

6 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 two standard AC^0 operations by first converting the integer x into a double floating point, and then extract the exponent by shifts. Alternatively, *msb* may be coded by a constant number of multiplications, as described in [FW93].

Our first step is to construct a minimum spanning tree M. This is done deterministically in linear time as described in [FW94]. 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 $[v]_i$,

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

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 5 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 12 gives that we now have only $4(n-1)+4n \leq 8n$ relevant buckets. Consequently, many (but not all) of the bounds in the proof of Proposition 13 are reduced from $O(m)$ to $O(n)$.

The main advantage of working with M instead of with G is that in linear time and space, we can preprocess M so that union-find over components of M can be supported at constant cost per operation [GT85]: we operate on a dynamic subforest $S \subseteq M$, starting with S consisting of singleton vertices. For an edge $e \in M$, *union*(e) adds e to S, and *find*(v) returns a canonical vertex from the component of S that v belongs to.

Now let $e_1, \ldots, e_{n-1}$ be the edges of M sorted according to $msb(\ell(e_i))$. Note that $msb(\ell(e_i)) < \log_2 w$. Thus, if $\log w = O(n)$, such a sequence is produced by simple bucketting. Otherwise, $\log w = O(w/(\log n \log\log n))$, and then we can sort in linear time by packed merging [AH92, AHNR95].

Defining $msb(\ell(e_0)) = -1$ and $msb(\ell(e_n)) = b$, note that

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

Thus, sequentially, for $i = 1, \ldots, n-1$, we will call *union*(e_i), and if $msb(\ell(e_i)) < msb(\ell(e_{i+1}))$, we will collect all the new components 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 F: Constructs $\mathcal{T}$ and $\Delta(\cdot)$. A leaf $[v]_0$ of $\mathcal{T}$ is identified with v and the internal components are identified with an intial 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 $\Delta(c) = \lceil \sum_{e \in [v]_i^M} / 2^{i-1} \rceil$.

F.1. for all $v \in V$,

F.1.1. $c(v) \leftarrow v$

F.1.2. $\Delta(v) \leftarrow 0$

F.1.3. $s(v) \leftarrow 0$

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

F.3. for $i \leftarrow 1$ to $n-1$,

F.3.1. let $(v, w) = e_i$

F.3.2. $X \leftarrow X \cup \{\mathit{find}(v), \mathit{find}(w)\}$

F.3.3. $s \leftarrow s(\mathit{find}(v)) + s(\mathit{find}(w)) + \ell(v, w)$

F.3.4. $\mathit{union}(v, w)$

F.3.5. $s(\mathit{find}(v)) \leftarrow s$

F.3.6. if $\mathit{msb}(\ell(e_i)) < \mathit{msb}(\ell(e_{i+1}))$,

F.3.6.1. $X' \leftarrow \{\mathit{find}(v) | v \in X\}$ — X' = canonical elements of new components of $\mathcal{T}$.

F.3.6.2. for all $v \in X'$,

F.3.6.2.1. $c \leftarrow c + 1$

F.3.6.2.2. $c'(v) \leftarrow c$

F.3.6.3. for all $v \in X$, $\wp(c(v)) \leftarrow c'(\mathit{find}(v))$

F.3.6.4. for all $v \in X'$,

F.3.6.4.1. $c(v) \leftarrow c'(v)$

F.3.6.4.2. $\Delta(c(v)) \leftarrow \lceil s(v)/2^{\mathit{msb}(\ell(e_i))} \rceil$

F.3.6.5. $X \leftarrow \emptyset$

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

In conclussion,

Proposition 14 *The component tree $\mathcal{T}$ is computed in $O(m)$ time and space.*

7 The unvisited data structure

As in Section 5, let $\mathcal{U}$ be the unvisited sub-forest $\mathcal{U}$ of the component tree $\mathcal{T}$. For each root $[v]_i$ in $\mathcal{U}$, we wish to maintain $\min D([v]_i)$. Also, if $[v]_i$ is visited, each child $[w]_h$ of $[v]_i$ in $\mathcal{T}$ is becomming a new root in $\mathcal{U}$, so we need to find $\min D([w]_h)$.

We will formulate the problem in a more clean data structure way. Let $v_1, \ldots, v_n$ be an ordered of the vertices corresponding to an arbitrary ordering of $\mathcal{T}$. Thus, each tree in $\mathcal{U}$ corresponds to a segment $v_i, \ldots, v_k$ for which we want to know $\min_{i \le j \le k} D(v_j)$. When we remove a root from $\mathcal{U}$ we split its segment into the segments of the subtrees. In conclussion we are studing 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 *split* a segment or *change* the D-value of some v_i. After each operation we need to up-date the minimum D-values of the segments.

In this section we will show how to perform $\leq n-1$ splits and m changes in $O(m)$ time, thus showing that we can maintain $\min D([v]_i)$ for all roots in $\mathcal{U}$ in $O(m)$ total time. As a first step, we show

Lemma 15 *We can accomodate $\leq n-1$ splits and m changes in $O(n \log n + m)$ 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 $2 \log n$ 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 changed, we may have to update the minimal D-value of the maximal unbroken interval and the segment containing it. This takes 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 the vertices, in order to find the minimal D-values, and to restore the pointers from the vertices to the maximal unbroken intervals containing them. Since each vertex is contained in $\log n$ intervals, the amortized 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 $2 \log n$ 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 [FW94]. Let $T(m, n, s)$ denote the cost of accomodating m changes and $n-1$ splits, starting with a sequence of length n that has already been divided into segments of size at most s. By Lemma 15, $T(m, n, n) = O(n \log n + m)$. In fact, noting that we only need the $\log s$ bottom levels of the interval structure, the construction of the proof actually gives $T(m, n, s) = O(n \log s + m)$. This improvement is, however, not necessary for our reduction.

Lemma 16 $T(m, n, s) = O(m) + T(m, n, \log s) + T(m, 2n/\log s, 2s/\log s)$.

Proof: Besides the original splits, we introduce a split at every $\log s$ vertex, thus splitting $v_1, \ldots, v_n$ into pieces of size $\log s$. Maintaining the minimum for the segments within these pieces is done in $T(m, n, \log s)$ total time.

We will maintain a sequence $w_1, \ldots, w_{2n/\log s}$ of super vertices derived from the pieces as follows. To piece i correspond w_{2i-1} and w_{2i}. If piece i is not broken, $D(w_{2i-1}) = D(w_{2i})$ is the minimal D-value of piece i. However, if piece i is broken, $D(w_{2i-1})$ is the minimal D-value of the leftmost segment of piece i, and $D(w_{2i})$ is the minimal D-value of rightmost segment. The D-values of the super vertices are trivially maintained in total time $O(m)$, and the minimum D-values of the segments of super vertices are maintained in $T(m, 2n/\log s, 2s/\log s)$ total time. Now the result follows since any segment of the orginal sequence, is either within a segment of a piece, or a segment of super vertices. ■

Applying Lemma 15 and Lemma 16 twice, we get

Corollary 17 $T(m, n, n) = O(m) + T(m, n, \log \log n)$.

From the construction of atomic heaps in [FW94], we have:

Lemma 18 ([FW94]) *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 stricly smaller than x. The total space is $O(n + \sum_i |S_i|)$.*

Lemma 19 *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 $\min_{l \le j \le k} A_i[j]$. Initially, we assume $A_i[j] = \infty$ for all i, j. The total space is $O(n + \sum_i |A_i|)$.*

Proof: We use Lemma 18 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 pieces 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 $\log n \le b$, this implies that σ_i fits in one register.

By Lemma 18, 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 $\sigma : \{0, \ldots, s-1\} \to \{0, \ldots, s-1\}$ 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 $\sigma : \{0, \ldots, s-1\} \to \{0, \ldots, s-1\}$ 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, $\min_{l \le j \le k} A_i[j]$ is found in constant time as $\Psi[\sigma_i, k, l]$. ■

Corollary 20 $T(m, n, \sqrt[4]{\log n}) = O(m)$.

Proof: Divide $v_1, \ldots, v_n$ to pieces of lenght $\sqrt[4]{\log n}$. We maintain the D-values for each piece as described in Lemma 19. Now any segment of lenght $\le \sqrt[4]{\log n}$ is contained in at most two pieces, and hence the minimum D-value of any such segment, is found in constant time. ■

Combining Corollaries 17 and 20, we get

Proposition 21 $T(m, n, n) = O(m)$.

8 Conclussion

Combining Propositions 13, 14, and 21, we have now proved

Theorem 22 *There is a deterministic linear time and linear space algorithm for the single source shortest path problem for undirected weighted graphs.*

It should be mentioned that our algorithm uses multiplication implicitely in its calls to Fredman and Willard's atomic heaps [FW94]. Multiplication is not an AC^0-operation, but as shown in [AMT96], the use of non-AC^0-operations is not inherent. Multiplication may be replaced by some simple selection and copying functions in AC^0 that are just missing in standard instruction sets.

The above algorithm is quite simple, except for the use of atomic heaps, which, as stated in [FW94], require $n > 2^{12^{20}}$. For the sake of implementations, we suggest some simple alternatives.

First consider the minimum spanning tree computation in Section 6, which is taken from [FW94] and is based on atomic heaps. Note that it satisfies with a spanning tree that is minimal in the graph where each weight x is replaced by $msb(x)$ (recall that msb is found in two fast AC^0-operations: first convert to a double, and then extract the exponent). Let C be the maximal weight. Since C fits in one word, $msb(C)$ is bounded by the word length b (≤ 128?). Assume the typical case that $n \geq msb(C)$. Then, by simple bucketting, in linear time and space, we can sort the edges of G according to the msb-weights. Afterwards, using Kruskal's algorithm [Kru56], an msb-minimum spanning tree is found in time $O(\alpha(m,n)m) = O(m + n \log^* n)$ time applying Tarjan's standard union-find. Note that the construction of $\mathcal{T}$ and $\Delta(\cdot)$ may be done on the fly, without invoking the special case union-find from [GT85].

Concerning the use of atomic heaps in the previous section, recall that the construction of the proof of Lemma 15 actually gives $T(m,n,s) = O(m + n \log s)$. Together with Corollary 17, this gives $T(m,n,n) = O(m + n \log\log\log n)$. In conclussion, we have an implementable, $O(\log C + m + n \log\log\log n)$-algorithm for SSSP. This time bound is easily improved, but likely at the expence of a more complicated algorithm with larger constants.

Thus, a deterministic linear time and linear space algorithm has been presented for the single source shortest path problem on undirected weighted graphs. This theoretically optimal algorithm is not in itself suitable for implementations, but there are simple variants of it that should work well in both theory in practice.

References

[AMOT90] R.K. Ahuja, K. Melhorn, J.B. Orlin, and R.E. Tarjan, Faster algorithms for the shortest path problem, *J. ACM* **37** (1990) 213–223.

[AH92] S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *in Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 463–472, 1992.

[AHNR95] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proc. 27th ACM Symposium on Theory of Computing (STOC)*, pages 427–436, 1995.

[AMT96] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. BRICS-TR-96-30, Aarhus, 1996.

[Dij59] E.W. Dijkstra, A note on two problems in connection with graphs, *Numer. Math.* **1** (1959), 269–271.

[Din78] E.A. Dinic, Economical Algorithms for Finding Shortest Paths in a Network, In *Transportation Modeling Systems*, Y.S. Popkov and B.L. Shmulyian (eds), Institute for System Studies, Moscow, 1978, 36–44.

[CGS97] B.V. Cherkassky, A.V. Goldberg, and C. Silverstein, Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, 1997.

[GT85] H.N. Gabow and R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union. *J. Comp. Syst. Sc.* 30:209–221, 1985.

[FT87] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987) 596–615.

[FW93] M.L. Fredman and D.E. Willard, Surpassing the information theoretic bound with fusion trees. *J. Comp. Syst. Sc.* 47:424–436, 1993.

[FW94] M.L. Fredman and D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comp. Syst. Sc.* **48** (1994) 533–551.

[KRRS97] M.R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster Shortest-Path Algorithms for Planar Graphs. *J. Comp. Syst. Sc.* **53** (1997) 2–23. See also STOC'94.

[Kru56] J.B. Kruskal, On the shortest spanning subtree of a graph and the traviling salesman problem. *Proc. AMS* **7** (1956), 48–50.

[MN90] K. Melhorn and S. Nähler, Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space, *Inf. Proc. Lett.* **35,** 4 (1990), 183–189.

[Ram96] R. Raman. Priority queues: small monotone, and trans-dichotomous. *Proc. ESA'96, LNCS 1136*, 1996, 121–137.

[Ram97] R. Raman. Recent results on the single-source shortest paths problem. *SICACT News* **28**, 2 (1997), 81–87.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM* 22:215-225, 1975.

[Tho96] M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

[vBoa77] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inf. Proc. Lett.* **6** (1977), 80–82.

[vBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Syst. Th.* **10** (1977), 99–127.

[Wil64] J.W.J. Williams, Heapsort, *Comm. ACM* **7**, 5 (1964), 347–348.