Technical Report DIKU-TR-97/4
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

January 1997

Undirected Single Source Shortest Path in Linear Time

Mikkel Thorup

Undirected Single Source Shortest Path 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`

January 1997

Abstract

A deterministic linear time and linear space algorithm is presented for the undirected single source shortest path problem.

1 Introduction

Let $G = (V, E)$, $|V| = n$, $|E| = m$, be an undirected connected graph with a weight function $\ell : E \to \mathbb{N}_0$ and a distinguished vertex $s \in V$. *The single source shortest path problem (SSSP)* is for every vertex v to find the distance from $d(v) = dist(s, v)$ from s to v. This is one of the classic problems in algorithmic graph theory. In this paper we present an determinstic linear time and linear space algorithm for undirected SSSP. We assume that the weights are intergers each stored in one word.

Since 1959, all developments in SSSP have been based in 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 $\forall v \in S : D(v) = d(v)$. Initially, $S = \{s\}$ and $D(s) = d(s) = 0$. For all $v \in V \setminus S$, we maintain that $D(v) = \min_{u \in S, (u,v) \in E}\{d(u) + \ell(u, v)\}$. In each round of the algorithm we *visit* a vertex $v \in V \setminus S$ minimizing $D(v)$. As proved by Dijksta, $D(v) = d(v)$, 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 $D(v) + \ell(v, w)$. Dijksta's algorithm finishes when $S = V$, returning $D(\cdot) = d(\cdot)$.

The complexity of Dijksta's algorithm is determined by the $n - 1$ times we find the vertex $v \in V \setminus S$ minimizing $D(v)$ and the at most m times we decrement some $D(w)$. All subsequent developments in SSSP have been based 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. 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 algorithmn in a comparison based model, since it visits the vertices in sorted order. Using Fredman and Willard's fusion trees, that like this paper assumes integer weights stored in words, we get an $O(m\sqrt{\log n})$ randomized bound [FW93].

[*]Technical Report DIKU-TR-97/4, Department of Computer Science, University of Copenhagen.

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. Latter, Raman has obtained an $O(m + n\sqrt{\log n \log \log n})$ bound in deterministic linear space using standard AC^0 instructions only.

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 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. Particularly for the SSSP application, Ahuja, Melhorn, and Tarjan has found a priority queue giving running time $O(n\sqrt{\log C} + m)$. Recently this has been improved by Cherkassky, Goldberg, and Silverstein to $O(n(\log C)^{1/3+\varepsilon} + m)$ [CGS97].

As observed in [FT87], implementing Dijksta'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 Dijksta'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.

Finally, a note should be made on our assumption that the weights are integers, each stored in one word. If the weights are not integers but floating point, the problem is less well-defined since adding up edge-weights may conflict with the available precision. This contrast the situation with sorting or, say, minimum spanning tree where we do not need to add up weights. For those problems, one can simply exploit that the order of floating points is given from the representation by interpreting them as integers. Also, for any given computiner, a word is understood to be the maximal entity that can be dealt with in one time step. Our algorithm is easily modified so as to allow for weights stored in multiple words. The complexity then becomes linear in the total number of words.

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. The point in this work is to find a different more efficient visiting sequence, still preserving that $D(v) = d(v)$ when a vertex v is visited.

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. If f is a function on the elements from a set X, we let $f(X)$ denote $\{f(x) | x \in X\}$.

3 The component hierachy

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 hierachy, we have the componenets of G_i. The component of v on level i is denoted $[v]_i$. By $[v]_i^-$ we will denote $[v]_i \setminus S$, noting that $[v]_i^-$ may not be connected.

Observation 1 *If $[v]_i \neq [w]_i$ and $D(w) \downarrow i \leq D(v) \downarrow i$ then $D(w) - D(v) < dist(v, w)$.*

Proof: Since $[v]_i \neq [w]_i$, any path from v to w contains an edge of lenght $\geq 2^i$ and hence the $dist(v,w) \geq 2^i$. However, $D(w) \downarrow i \leq D(v) \downarrow i$ implies $D(w) - D(v) < 2^i$. ■

For $i < w$, we call $[v]_i$ a child of $[v]_{i+1}$, and say that $[v]_i$ is a *min-child* of $[v]_{i+1}$ if $\min(D([v]_i^-) = \min(D([v]_{i+1}^-)$. We say that $[v]_i$ is *minimal* if for $j = i, \ldots, w-1$, $[v]_j$ is a min-child of $[v]_{j+1}$. Note that minimality depends on S.

Lemma 2 *Let $[v]_{i+1}$ be minimal and u be the first vertex outside S on a 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 $u \notin [v]_i$, $dist(u,v) \geq 2^i$. 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 3 *If $[v]_i$ is minimal, $\min D([v]_i^-) = \min d([v]_i^-)$. In particular, if $D(v) = \min D([v]_i^-)$, $D(v) = d(v)$. Also, if $i = 0$, $[v]_i = \{v\}$ and $D(v) = d(v)$.*

Corollary 4 *If $[v]_i, i < w$ is not minimal but $[v]_{i+1}$ is minimal, then*

$$\min d([v]_i^-) \downarrow i > \min D([v]_{i+1}^-) \downarrow i.$$

In particular, if $\min D([v]_i^-) \downarrow i = (\min D([v]_{i+1}^-) \downarrow i) + 1$, $\min D([v]_i^-) \downarrow i = \min d([v]_i^-) \downarrow i$.

4 Visiting minimal vertices

Definition 5 *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 3:

Fact 6 *When we visit a vertex v, $D(v) = d(v)$.*

Fact 7 *For all $[v]_i$, $\max d([v]_i \setminus [v]_i^-) \downarrow i \leq \min d([v]_i^-) \downarrow i$.*

Lemma 8 *If $[v]_i$ is minimal and visiting some vertex $w \in V \setminus S$ implies that $D([v]_i^-)$ changes, then $w \in [v]_i$ and the change is an increase by one.*

Proof: By Corollary 3, before the visit to w, $\min D([v]_i^-) \downarrow i = \min d([v]_i^-)$. Since D dominates d and d does not change, this means that $\min D([v]_i^-) \downarrow i$ cannot decrease. Moreover, since the D-values are never increased, an increase of $\min D([v]_i^-) \downarrow i$ means that $[v]_i^-$ is decreased, i.e. that $w \in [v]_i^-$ before the visit. Let $[v]_i^-$ refer to the value before the visit to w, i.e. $w \in [v]_i^-$. Since $[v]_i$ is connected, if $[v]_i^- \setminus \{w\}$ is non-empty, there must be an edge (u, x) in $[v]_i$ where $u \in [v]_i^-$ and $x \in [v]_i^- \setminus \{w\}$. By definition of $[v]_i$, $\ell(u, x) < 2^i$. Moreover, by Fact 7, $d(u) \downarrow i \leq \min D([v]_i^-) \downarrow i$. Thus

$$\min D([v]_i^- \setminus \{w\}) \downarrow i \leq D(x) \downarrow i \leq (\min D([v]_i^-) \downarrow i) + 1.$$

■

In connection with Lemma 8, 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 9 *If $[v]_i$ is minimal, it remains minimal until either $[v]_i^- = \emptyset$ or $\min D([v]_i^-) \downarrow i$ is increased.*

Proof: If $[v]_i$ is minimal, but visiting some vertex w stops $[v]_i$ from being minimal, then there is 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 visit to w. Before the visit to w, $[v]_i$ was minimal. Hence $[v]_j$ was minimal, so $\min D([v]_j) \downarrow j \leq \min D([v]_{j+1}) \downarrow j$. By Lemma 3, the minimality of $[v]_{j+1}$ implies that $\min D([v]_{j+1}) = \min d([v]_{j+1})$. Hence $\min D([v]_{j+1}) \downarrow j$ cannot decrease. Thus, if $\min D([v]_i^-) \downarrow i$ is neither increased, nor emptied, we must have $\min D([v]_{j+1}) \downarrow j = D([v]_j) \downarrow i$, contradicting that $[v]_j$ stopped being minimal. ■

We are now ready to derive a first algorithm for the undirected single source shortest path problem. The algorithm is so far inefficient, but it illustrates 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 $D(v) = d(v) = \mathit{dist}(s, v)$ for all $v \in V$.

```
A.1.  for all $v$, $D(v) \leftarrow \infty$
A.2.  $D(s) \leftarrow 0$, $S \leftarrow \{s\}$
A.3.  for all $(s, v) \in E$, $D(v) \leftarrow \ell(s, v)$.
A.4.  Visit($[s]_b$, 0)
A.5.  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. After the call, either $[v]_i^- = \emptyset$ or $\min D([v]_i^-) \downarrow i = d([v]_i^-) \downarrow i$ is increased by one.

```
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:      $\triangleright$ $ix([v]_i) = \min D([v]_i^-) \downarrow i - 1$
B.3.1.     while $\exists [w]_{i-1} \subseteq [v]_i : \min D([w]_{i-1}^-) \downarrow i - 1 = ix([v]_i)$,
B.3.1.1.        let $[w]_{i-1} \subseteq [v]_i \ \wedge\ \min D([w]^-_{i-1}) \downarrow i-1 = ix([v]_i)$
B.3.1.2.        Visit($[w]_{i-1}$)
B.3.2.      increment $ix([v]_i)$ by one
```

In the remainder of this section, we will prove the correctness of Algorithms B, and hence of Algorithm A. The proof is by induction on i. If $i = 0$, $[v]_i = [v]^-_i = \{v\}$, and the visit to v in Step B.1 is correct since v is $[v]_i$ is minimal by assumption. Afterwards, $[v]^-_i = \emptyset$. Thus, we may assume that $i > 0$. To prove correctness, we will study the following invariant:

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

When $ix([v]_i)$ is first assigned in step B.2, it is assigned $D([v]_i) \downarrow i-1$. Also, at that time, $[v]_i$ is minimal, so $\min D([v]_i) = \min d([v]_i)$ by Lemma 3. Thus (1) holds after this assignment.

Note that $\min D([v]_i) \downarrow i-1 = \min d([v]_i) \downarrow i-1$ implies that $\min D([v]_i) \downarrow i-1$ cannot decrease. On the other hand, $\min D([v]_i)$ can only increase in connection with visits to vertices in $[v]_i$. Thus, a violotion of (1) is either due to a visit within $[v]_i$ or to a change in $ix([v]_i)$. The first possible violation is hence in connection with the call Visit($[w]_{i-1}$). In particular, we may assume that (1) is true before the call. Then, since $ix([v]_i) \downarrow 1$ has not been increased, by Lemma 9, $[v]_i$ is still minimal. Also, by definition $[w]_{i-1}$ is a min-child, so $[w]_{i-1}$ is minimal. Thus, by induction, we may assume that the call Visit($[w]_{i-1}$) is correct. However, after the call, if the condition of the while-loop is still satisfied, it says that $\min D([v]_i)$ did not increase, so (1) still holds. If the condition fails, we know that $\min D([v]_i)$ did increase. By Lemma 8, first time in the call Visit($[w]_{i-1}$) where we increase $\min D([v]_i) \downarrow i$, we increase it by one. However, this implies that $D([w]_{i-1}) \downarrow i-1 \geq D([v]_i) \downarrow i-1$ is also increased, but then, by induction, there are no more visits to made by Visit($[w]_{i-1}$). Hence we conclude that $D([v]_i) \downarrow i$ is increased by exactly one, but that means that (1) is restored in step B.3.2.

5 Towards a linear time algorithm

We will now give an overiew of a linear time algorithm algorithm SSSP. 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 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 the component tree $\mathcal{T}$.

The idea now is that for each visited node $[v]_i$, we will bucket the children $[w]_h$ according $\min D([w]^-_h) \downarrow i-1$. Having done so, we can identify the min-children in constant time. In Section 7, we will show how to do the bucketing for unvisited children of visited components, and later in this section, we will show how to do it for visited children of visited components.

As a main point, we will show that the total number of relevant buckets is $O(m)$. When a node is first to be visited, it is a child of a visited node, so we know $\min D([v]^-_i) \downarrow i-1$. Also, since $[v]_i$ is has not yet been visited $[v]_i = [v]_i^-$. Finally, when $[v]_i$ is about to be visited, it is minimal, implying $\min D([v]_i^-) = \min d([v]_i^-)$, that is, $\min D([v]_i^-) \downarrow i{-}1 = \min d([v]_i) \downarrow i{-}1$.

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 $ix_0([v]_i) = \min d([v]_i)$ and $\Delta([v]_i) = \lceil \sum_{e \in [v]_i} \ell(e)/2^{i-1} \rceil$. Then

$$\max d([v]_i) \downarrow i-1 \leq (ix_0([v]_i) \downarrow i-1) + \Delta([v]_i) \tag{2}$$

Now, for each $[v]_i \in \mathcal{T}$ and each $q \leq \Delta([v]_i)$, we allocate a bucket $B([v]_i, q)$ for the children $[w]_h$ of $[v]_i$ with $\min D([w]_h^-) \downarrow (i-1) = ix_0([v]_i) + q$.

Lemma 10 *The total number of buckets is $O(m)$.*

Proof: In connection with $[v]_i$, we have $\Delta([v]_i) + 1 \leq 2 + \sum_{e \in [v]_i} \ell(e)/2^{i-1}$ buckets. Thus, the total number of buckets is $\leq 4n + \sum_{[v]_i \in \mathcal{T},\ e \in [v]_i} \ell(e)/2^{i-1}$.

If $e \in [v]_i$, $\ell(e) < 2^i$. Hence, for any e,

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

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

Note that the values of $\Delta([v]_i)$ are easily found in time $O(m)$ in a bottom-up traversal of $\mathcal{T}$. Thus, in time $O(m)$, we can initialize the whole bucket structure, embedding it in one array of pointers to the initially empty doubly linked list of children going into the bucket — having the list doubly linked allows us to take children out of buckets, when they are to be moved. We are now ready to present the final visiting procedure. Relative to the previous Algorithm B, it is generalized to handle that that if a child $[w]_h$ of $[v]_i$ may not have $h = i-1$. This generalization is straightforward. Moreover, it is modified so as to take care of the bucketing of visited children of visited components.

Algorithm C: **Visit**$([v]_i, j)$, assuming 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. After the call, either $[v]_i^- = \emptyset$ or $\min D([v]_i^-) \downarrow j-1 = d([v]_i^-) \downarrow j-1$ is increased by one.

```
C.1.  if $i = 0$, visit $v$ and return
C.2.  if $[v]_i$ has not been visited previously,
C.2.1.      $ix_0([v]_i) \leftarrow \min D([v]_i) \downarrow i-1$
C.2.2.      for $q = 0$ to $\Delta([v]_i)$, $B([v]_i, q) \leftarrow \emptyset$.
C.2.3.      for all children $[w]_h$ of $[v]_i$,
C.2.3.1.        if $\min D([w]_h) \downarrow i-1 \leq ix_0([v]_i) + \Delta([v]_i)$,
C.2.3.1.1.          add $[w]_h$ to $B([v]_i, (\min D([w]_h) \downarrow i-1) - ix_0([v]_i))$.
C.2.4.      $ix([v]_i) \leftarrow ix_0([v]_i)$
C.3.  repeat until $[v]_i^- = \emptyset$ or $ix([v]_i) \downarrow (j-i)$ is increased: $\triangleright$ $ix([v]_i) = (\min D([v]_i^-) \downarrow j-1)$
```

```
C.3.1.      while $B([v]_i, ix([v]_i) - ix_0([v]_i)) \neq \emptyset$,
C.3.1.1.        let $[w]_h \in B([v]_i, ix([v]_i) - ix_0([v]_i))$
C.3.1.2.        Visit$([w]_h, i)$
C.3.1.3.        delete $[w]_h$ from $B([v]_i, ix([v]_i) - ix_0([v]_i))$
C.3.1.4.        if $[w]_h^- \neq \emptyset$, add $[w]_h$ to $C([v]_i, ix([v]_i) - ix_0([v]_i) + 1)$
C.3.2.      increment $ix([v]_i)$ by one
```

Given the correctness of Algorithms B, to see the correctness of Algorithm C, our only problem is to show that it correctly buckets the visited children of visited components. Algorithm C is only responsible for the bucketing of a child $[w]_h$ of $[v]_i$ from after $[w]_h$ has been visited, that is, we may assume that $[w]_h$ is in the correct bucket of $[v]_i$ just before the visit. However, just before the visit, $[w]_h$ is minimal, so by Corollary 3, $\min D([w]_h^-) = \min d([w]_h^-)$. In particular,

$$\min D([w]_h^-) \downarrow (i-1) = \min d([w]_h^-) \downarrow (i-1) \tag{3}$$

We shall refer to $\min D([w]_h^-) \downarrow (i-1)$ as the *bucket index* since it uniquely determines the correct bucket for $[w]_h$; namely $B([v]_i, (\min D([w]_h) \downarrow i-1) - ix_0([v]_i))$.

We claim that (3) is maintained after $[v]_i$ has first been visited. As in the proof of (1), note that (3) implies that the bucket index $\min D([w]_h^-) \downarrow (i-1)$ cannot decrease, and that (3) can only be violated if the bucket index increases. At the same time, the bucket index $\min D([w]_h^-) \downarrow (i-1)$ can only increase in connection with visits to vertices in $[w]_h^-$, i.e. in Step C.3.1.2. By induction, the call Visit$([w]_h)$ is correct, so if $[w]_h^-$ is not emptied, $\min D([w]_h^-) \downarrow i-1 = d([w]_h^-) \downarrow i-1$ is increased by one. Thus, moving $[w]_h$ up one bucket is correct.

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 intestested 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 to a double, and then extract the exponent. Alternatively, *msb* may be coded by a constant number of multiplications, as described in [FW93].

1. Construct a minimum spanning tree T deterministically in linear time as described in [FW94]. Clearly, for all i, the components of G_i coincide with those in T_i.

2. In linear time and space, we preprocess T so that union-find over components of T can be supported in constant time per operation [GT85]: we operate on a $S \subseteq T$, starting with S consisting of singleton vertices. A union operation adds an edge from T to S, and find(v) returns a canonical vertex from the component of S that v belongs to.

3. Construct a sequence $e_1, \ldots, e_{n-1}$ of the edges of T sorted so that $msb(\ell(e_i)) < msb(\ell(e_{i+1}))$. Note that $msb(\ell(e_i)) < \log_2 w$. Thus, if $\log w = O(n)$, such a sequence may be 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].

After the above preliminary steps, we can construct the component tree $\mathcal{T}$ as follows.

Algorithm D:

```
D.1.        $i \leftarrow -1$
D.2.        $X \leftarrow V$
D.3.        for $j \leftarrow 1$ to $n-1$,
D.3.1.          if $msb(\ell(e_j)) > i$,
D.3.1.1.            $X' \leftarrow \{find(v) | v \in X\}$
D.3.1.2.            for all $v \in X'$,
D.3.1.2.1.              $P(c(v)) \leftarrow c$
D.3.1.2.2.              $c \leftarrow c + 1$
D.3.1.3.            for all $v \in X$,
D.3.1.3.1.              $P(c(v)) \leftarrow P(c(find(v)))$
D.3.1.4.            $X \leftarrow \emptyset$
D.3.2.          $i \leftarrow msb(\ell(e_j))$
D.3.3.          let $(v, w) = e_i$
D.3.4.          $X \leftarrow X \cup \{find(v), find(w)\}$
D.3.5.          $union(v, w)$
```

7 The unvisited data structure

Let $\mathcal{U}$ be the unvisited sub-forsest $\mathcal{U}$ of the component tree $\mathcal{T}$. Thus a component $[v]_i$ of $\mathcal{T}$ is in $\mathcal{U}$ if and only if $[v]_i^- = [v]_i$.

For each root $[v]_i$ in $\mathcal{U}$ we which 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 some 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 the 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 roots in $\mathcal{U}$ accordingly.

In this section we will show how to perform $\le n-1$ 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 11 *We can accomodate $\le n-1$ splits and m changes in $O(n \log n + m)$ time.*

Proof: First we make a balanced binary sub-division $v_1, \ldots, v_n$ of 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 an $O(n \log n)$ total time. ■

In order to get down to linear total cost, we will make a reduction to Fredman and Willard's atomic priority queues [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 11, $T(m,n,n) = O(n\log n + m)$. 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 12 $T(m,n,s) = O(m) + T(m,n,\log s) + T(m, 2n/\log s, 2s/\log s)$.

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 11 and Lemma 12 twice, we get

Corollary 13 $T(m,n,n) = O(m) + T(m,n,\log\log n)$.

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

Lemma 14 ([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 operation 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 15 *Given $O(n)$ preprocessing time and space for construction of tables, we can maintain a family $\{A_i\}$ of arrays $A_i : \{0, \dots, s-1\} \to \{0 \dots, 2^b - 1\}$, $s = O(\sqrt[4]{\log n})$, 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 $O(n + \sum_i |A_i|)$.*

Proof: We use Lemma 14 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 $\sigma_i : \{0, \dots, s-1\} \to \{0, \dots, s-1\}$, 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 $\log n \leq b$, this implies that σ_i fits in one register.

By Lemma 14, 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 $\sigma : \{0, \dots, s-1\} \to \{0, \dots, s-1\}$ and $j, r \in \{0, \dots, s-1\}$. In $\Sigma(\sigma, j, r)$, we store σ' such that $\sigma'(j) = r$, $\sigma'(h) = \sigma(h) + 1$ 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 $\sigma : \{0, \dots, s-1\} \to \{0, \dots, s-1\}$ and $l, k \in \{0, \dots, s-1\}$ returns the $j \in \{k, \dots, l\}$ 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 $\Psi[\sigma_i, k, l]$. ■

Corollary 16 *$T(m, n, \sqrt[4]{\log n}) = O(m)$.*

Proof: Divide $v_1, \dots, v_n$ to pieces of lenght $\sqrt[4]{\log n}$. We maintain the D-values for each piece as described in Lemma 15. Now any segment of lenght $\leq \sqrt[4]{\log n}$ is contained in at most two pieces, and hence the minimum D-value of any such segment, is found in constant time. ■

Combining Corollaries 13 and 16, we get

Proposition 17 *$T(m, n, n) = O(m)$.*

8 Concluding remarks

For completeness, we round up by presenting the pseudo-code for visiting a vertex v:

Algorithm E: visit v

```
E.1.    $S \leftarrow S \cup \{v\}$
E.2.    for all $(v, w) \in E$, if $D(v) + \ell(v, w) < D(w)$,
E.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 $\mathcal{T}$.
E.2.2.      change $D(w)$ to $D(v) + \ell(v, w)$.
```

```
E.2.3.      if this decreases $\min D([w]_i^-) \downarrow (j-1)$ and $\min D([w]_i^-) \downarrow (j-1) \leq ix_0([w]_j) + \Delta([w]_j)$,
E.2.3.1.        move $[w]_i$ to $B([w]_j, \min D([w]_i^-) \downarrow (j-1) - ix_0([w]_j))$
```

In conclussion,

Theorem 18 *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 $n \geq msb(C)$. We can then run Prim's minimum spanning tree algorithm [Pri57], growing the minimum spanning tree from a single vertex, letting the priority queue have a bucket for each msb-weight, and a bit-map telling which buckets are non-empty. Using msb on the bit-map, we can immediately identify the least non-empty bucket. Thus, we have a very simple deterministic $O(\log C + m)$ time and space minimum spanning tree algorithm for msb-weights.

Concerning the use of atomic heaps in the previous section, recall that the construction of the proof of Lemma 11 actually gives $T(m, n, s) = O(m + n \log s)$. Together with Corollary 13, 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.

[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.

[Kru56] J.B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.* 7:48–50, 1956.

[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.

[Pri57] R.C. Prim, Shortest connection networks and some generalizatoins. *Bell System Technical Journal* **36** (1957), 1389–1401.

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

[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.