Undirected Single-Source Shortest Paths in Linear Time

An unusual way of solving the SSSP Problem

Cornelius Diekmann

September 28, 2010

Abstract

The solution by Mikkel Thorup presented in [Tho99], [Tho97] for the Single Source Shortes Path (SSSP) graph problem is reviewed and evaluated. In contrast to Thorup's original work, this paper puts more emphasis on the unusual way of visiting vertices instead of the conclusion that the SSSP problem can be solved in linear time.

Contents

1 Introduction

According to Thorup [Tho97], many solutions for the SSSP problem exist, but all algorithms are based on Dijkstra's way of visiting vertices, thus no algorithm has achieved linear time so far. This paper presents a different visiting approach from which a linear time algorithm can be derived.

2 Preliminaries and terminology

Let $G = (V, E)$ be an undirected graph with positive integer edge weights. Let $l : E \to \mathbb{N}$ be the weight function for G and $s \in V$ a distinguished vertex. l is supposed to only work on Integers. By denoting the length of an Integer in Bit as $sizeof(Integer)$, $2^{sizeof(Integer)} - 1$ is the maximum value of l. We assume s to be the source for the SSSP-problem and G to be connected. For convenience, if $(v, w) \notin E$, let $l(v, w) = \infty$. For $d : V \to \mathbb{N}$, $d(v)$ is the shortest distance from s to v. $D(v)$, with the same signature as $d(v)$, represents a super distance for $d(v)$. $D(v)$ is the shortest distance from s to v currently known in the ongoing step, so $D(v)$ is an upper limit for the absolute shortest distance $d(v)$.

Referencing to Dijkstra's Algorithm, let $S \subseteq V$ be the set of vertices already visited. At the beginning, $D(v)$ is initialized with $l(s, v)$ for all $v \in V \setminus \{s\}$, defaulting to ∞ for the most vertices, $S = \{s\}$, $D(s) = d(s) = 0$.

Algorithm 1 initalize(G, l, s)

Require: $G = (V, E)$ is an undirected graph with positive integer edge weights defined by l.
Require: $s \in V$.
  $S := \{s\}$
  $D(s) := 0$
  $d(s) := 0$
  for all $v \in V \setminus S$ **do**
    $D(v) := l(s, v)$
  end for

Visiting a vertex $v \in V \setminus S$ results in setting $D(w)$ for all $(v, w) \in E, w \notin S$ to $\min\{D(w), D(v) + l(v, w)\}$ and moving v to S. v can only be visited if $D(v) = d(v)$.

Algorithm 2 visit(v)

Require: $D(v) = d(v) \wedge v \in V \setminus S$
  for all $(v, w) \in E, w \notin S$ **do**
    $D(w) := \min\{D(w), D(v) + l(v, w)\}$
    $S := S \cup \{v\}$
  end for

The SSSP-Algorithm terminates with $S = V$, returning the minimum distance $d(v)$ for all $v \in V$.

Thus, the following invariant holds in every step $\forall v \in S : D(v) = d(v)$. Lemma 2.1 and 2.2 directly result from the way we visit each vertex.

Lemma 2.1. *If $v \in V \setminus S$ minimizes $D(v)$, then $D(v) = d(v)$. [Tho99]*

Proof. $D(v)$ can only hold a value less than ∞. If v can be connected with a vertex in S, thus minimizing $D(v)$, means finding the shortest Path from v to s. If there is any vertex u

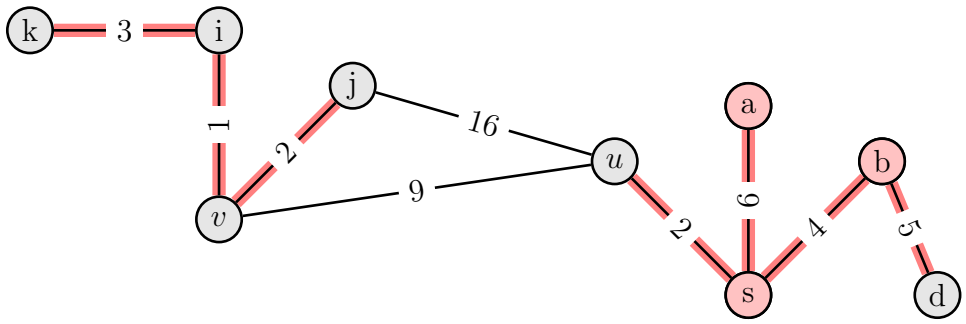

Figure 1: Example: $\delta = 8$, $S = \{s, a, b\}$, red highlighted edges describe the two components.

on a shortest path from s to v which can be visited, thus requiring $D(u) = d(u)$, it results $D(v) \overset{\text{super distance}}{\geq} d(v) \overset{\text{on a shortest path}}{\geq} d(u) = D(u) \overset{\text{minimum}}{\geq} D(v)$.

□

Lemma 2.2. $\min D(V \setminus S) = \min d(V \setminus S)$ *is nondecreasing. [Tho99]*

Proof. Looking closely at Algorithm 2, D is only decreased and visiting requires $D(v) = d(v)$. The complete proof is shown in the proof of correctness of Dijkstra's Algorithm. □

By combining all previous lemmata, Dijkstra's Algorithm can be derived. However, our aim is not to use Dijkstra's Algorithm, but the previous lemmata will prove to be useful, when handled with care.

2.1 The shift operator

The shift operator ($\gg i$) is defined like in the programming language C. $x \gg i$ means shifting the binary representation of x i digits to the right. $x \gg i = \lfloor \frac{x}{2^i} \rfloor$. This operation corresponds to the fast and cheap x86 assembly operation *shr*.

Lemma 2.3. $x \leq y \;\Rightarrow\; x \gg i \leq y \gg i$

Lemma 2.4. $x \gg i < y \gg i \;\Rightarrow\; x < y$

3 Towards an alternative visiting algorithm

Lemma 3.1. *Suppose the vertex set V divides into disjoint subsets $V_1, \ldots, V_k$ and that all edges between the subsets have length at least δ. Further suppose for some i, $v \in V_i \setminus S$ that $D(v) = \min D(V_i \setminus S) \leq \min D(V \setminus S) + \delta$. Then $d(v) = D(v)$. [Tho99]*

Proof. To imagine lemma 3.1, suppose u to be the first vertex outside S on a shortest path from s to v. If $u \in V_i \setminus S$ then $D(v) \geq d(v) \geq d(u)$. The way we visit each vertex and the fact that u is one of the vertices directly outside S, concludes $d(u) = D(u)$. However, as $D(v)$ is the minimum in $V_i \setminus S$, $D(u) \geq D(v)$. Putting the inequality together $d(v) = D(v)$. If $u \notin V_i$, every edge between the component of u and V_i has at least a weight of δ. This scenario is demonstrated in Figure 1.

If $u \notin V_i$, every edge between the component of u and V_i has at least a weight of δ. Again, $D(v) \geq d(v)$. A component border has to be passed to (possibly indirectly) connect u and v, thus $d(v) \geq d(u) + \delta = D(u) + \delta$. However, the minimal vertex concerning D in the whole graph might still be less than u, in combination with the assumption, $D(u) + \delta \geq \min D(V \setminus S) + \delta \overset{\text{assumption}}{\geq} \min(V_i \setminus S) \overset{\text{assumption}}{=} D(v)$. Thus $d(v) = D(v)$. □

Lemma 3.1 gives us a basic idea of how a vertex can be visited in a different way than Dijkstra's Algorithm but still preserving the $d(v) = D(v)$ constraint. In the next section, we will describe how to break up the graph into disjoint subsets suitable for a linear time algorithm.

3.1 Component hierarchy

The graph is divided into components, forming a component hierarchy. $G_i = (V_i, E_i)$ denotes the subgraph of G with $\forall e \in E_i : l(e) < 2^i$. We consider the connected components of G_i. The component on level i containing v is denoted by $[v]_i$, similar to the equivalent classes notation. The component hierarchy can be imagined like a tree, thus all $[w]_{i-1}$ with $[w]_i = [v]_i$ are called children of $[v]_i$.

Examples:

- G_0 only consists of singleton vertices.
- $G_{sizeof(Integer)}$ represents the whole graph.
- If $[v]_i \neq [w]_i \Rightarrow dist(v, w) \geq 2^i$, since any path from v to w must pass a component border with an edge of weight of at least 2^i.
- Considering Figure 1, the red highlighted edges describe the components of G_3: All highlighted edges carry a weight less than 8. $[k]_3 = [i]_3 = [v]_3 = [j]_3 \neq [u]_3 = [s]_3 = [a]_3 = [b]_3 = [d]_3$

$[v]_i^-$ is an abbreviation for $[v]_i \setminus S$. Thus $[v]_i^-$ are all vertices on level i in the component $[v]_i$ which have not been visited. $[v]_i^-$ is not necessarily connected and depends on S!

Examples related to Figure 1:

- The red highlighted vertices are in S, thus $[u]_3^- = \{u, d\}$ which is not connected.
- $[v]_3^- = \{k, i, v, j\}$

Definition 3.2. *$[v]_i$ is a min-child of $[v]_{i+1}$ if* $\min(D([v]_i^-)) \gg i \;=\; \min(D([v]_{i+1}^-)) \gg i$

Definition 3.3. *$[v]_i$ is minimal if $[v]_i^- \neq \emptyset$ and $\forall j \geq i : [v]_j$ is a min-child of $[v]_{j+1}$*

The definition of minimality directly implies the following two lemmata.

Lemma 3.4. *If $v \notin S, [v]_i$ is minimal, and $i < j \leq sizeof(Integer)$,* $\min D([v]_i^-) \gg j - 1 \;=\; \min D([v]_j^-) \gg j - 1$.

Lemma 3.5. *Suppose $v \notin S$ and there is a shortest path to v where the first vertex u outside S is in $[v]_i$. Then* $d(v) \geq \min D([v]_i^-)$.

3.1.1 Example

Figure 2 provides a small example about min-childs and minimality. Let $S = \{s\}$ fixed because minimality depends on S.
$[s]_3 = \{s, a, w, v\}$
$[s]_3^- = \{a, w, v\}$
$[s]_3$ is minimal by definition. $[s]_3 = [v]_3$, thus $[v]_3$ minimal too.
$[v]_2 = \{s, v\}$
$[v]_2^- = \{v\}$
$\min D([v]_2^-) = \min D(\{v\}) = D(v)$
$\min D([v]_3^-) = \min D(\{v, a, w\}) = D(v)$
$\min D([v]_2^-) \gg 2 = D(v) \gg 2 = 0$ and $\min D([v]_3^-) \gg 2 = D(v) \gg 2 = 0$, which directly implies: $[v]_2$ is a min-child of $[v]_3$, thus $[v]_2$ is minimal. $[v]_1 = [v]_0$, hence $[v]_0$ is minimal.

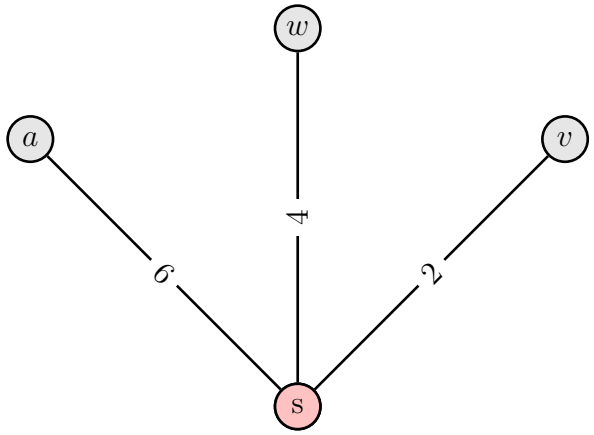

Figure 2: min-child example: $S = \{s\}$

3.2 Visiting a vertex

A vertex v can only be visited, if $v \in V \setminus S$ minimizes $D(v)$, thus $D(v)$ is the shortest distance from s to v, see lemma 2.1. Proving that the minimality of $[v]_0$ implies $D(v) = d(v)$, we can construct an algorithm which visits minimal vertices, thus solving the SSSP problem. If we can prove the following lemma 3.6, this condition directly derives from it.

Lemma 3.6. *If $v \notin S$ and $[v]_i$ is minimal,* $\min D([v]_i^-) = \min d([v]_i^-)$.

Proof. $\forall w \in V : D(w) \geq d(w)$, which implies that $\min D([v]_i^-) \geq \min d([v]_i^-)$. Let $x \in [v]_i^-$, when we imagine the graph, we see that $d(x) \geq \min D([v]_i^-)$. The resulting equation is $d(x) \geq \min D([v]_i^-) \geq \min d([v]_i^-)$ for all x, especially for the x which minimizes $d([v]_i^-)$, which delivers the result $\min D([v]_i^-) = \min d([v]_i^-)$.

Only $d(x) \geq \min D([v]_i^-)$ is left to show:
Let u be the first vertex outside S on a shortest path to x. On the one hand assume $u \in [v]_i$. $D(u)$ is a lower bound for the length of the path from s to x, thus $d(x) \geq D(u)$ and $D(u) \geq \min D([v]_i^-)$ since $u \in [v]_i^-$. On the other hand assume $u \notin [v]_i$. $[v]_i$ is minimal, thus $\min(D([v]_i^-)) \gg i = \min(D([v]_{i+j}^-)) \gg i$, for all $j \in \mathbb{N}_0$. When starting with $j = 0$ and incrementing j until $u \in [v]_{i+j}^-$, the path from s to x over u must at least contain one edge of weight $\geq 2^i$. Note that $2^i \gg i = 1$. However, $\min(D([v]_{i+j}^-)) \gg i$ remains constant when incrementing j and $u \notin [v]_i$ results in $d(x) \geq \min D([v]_i^-)$. □

Lemma 3.7. *If $v \notin S$ and v minimizes $D(v)$, $[v]_0$ is minimal.*

Proof. $\forall i : \min D([v]_i^-) \gg i = \min D([v]_{i+1}^-) \gg i = D(v) \gg i$, so $[v]_0$ is minimal. □

4 The first algorithm

Let S, D, d be initialized by Algorithm 1.

Algorithm 3 and 4 provide a solution to the SSSP-Problem, however not in linear time, but they describe an approach of visiting vertices which differs from Dijkstra. With the right data structures, Algorithm 3 can be transformed into a linear time and space algorithm. When recalling that $[s]_{sizeof(Integer)}$ represents the whole graph and $[s]_{sizeof(Integer)}$ is minimal by definition, no further words need to be said about Algorithm 4.

Algorithm 3 Visit($[v]_i$)

Require: $[v]_i$ is minimal
  if $i = 0$ **then**
    visit v {This visit refers to Algorithm 2}
    return
  end if
  if $[v]_i$ has not been visited previously **then**
    $x_i([v]_i) := \min D([v]_i^-) \gg i - 1$
  end if
  repeat
    while $\exists$ child $[w]_{i-1}$ of $[v]_i$ such that $\min D([w]_{i-1}^-) \gg i - 1 = x_i([v]_i)$ **do**
      assert($\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i$)
      assert($[w]_{i-1}$ is minimal)
      Visit($[w]_{i-1}$)
    end while
    $x_i([v]_i) := x_i([v]_i) + 1$
  until $[v]_i^- = \emptyset \vee x_i([v]_i) \gg 1$ is increased

Algorithm 4 SSSP(G, l, s)

  initialize(G, l, s)
  $w = sizeof$(Integer)
  for $i \in [0, w]$ **do**
    declare globally x_i {just define this variable, it will be used by the Visit Algorithm}
  end for
  Visit($[s]_w$)
  return D

4.1 About Algorithm 3

In Algorithm 3, the termination condition $i = 0$ must be reasonable according to section 3.2. The two assertions in the inner while loop describe the intuition of that loop. As a matter of fact[1], $\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i$ is preserved constant throughout the whole while loop, thus the while loop only visits all $w \in [v]_i^-$ with $d(w) \gg i = \min D([v]_i^-) \gg i$.

The termination conditions of the repeat until loop only holds if $[v]_i$ is no longer minimal. If $[v]_i^- = \emptyset$, $[v]_i$ is not minimal by definition. $x_i([v]_i) \gg 1$ is equal to $\min D([v]_i^-) \gg i$. The following lemma explains the repeat until loop condition.

Lemma 4.1. *If $[v]_i$ is minimal, it remains minimal until $\min D([v]_i^-) \gg i$ is increased, in which case $\min d([v]_i^-) \gg i$ is also increased.*

Proof. Assuming $[v]_i$ is minimal and visiting $w \in [v]_i^-$ stops $[v]_i$ from being minimal. If $[v]_i^-$ was $\{w = v\}$, after visiting w, $\min D([v]_i^-) = \min d([v]_i^-) = \infty$ and after the visit $[v]_i^- = \emptyset$. In a more generic scenario, before the visit to w, $\min d([v]_i^-)$ was equal to $\min D([v]_i^-)$ according to lemma 3.6. Now we pick the smallest possible $j \geq i \in \mathbb{N}$ such that $[v]_j$ is minimal after the visit of w. If $[v]_i \neq \emptyset$, this j must exist, in the worst case it could be $sizeof(Integer)$. Before the visit $\min D([v]_i^-) \gg j \leq \min D([v]_j^-) \gg j$. The way we have chosen j, $\underbrace{\min D([v]_j^-) \gg j}_{before} \leq \underbrace{\min D([v]_j^-) \gg j}_{after}$. However as j was chosen at the minimality border $\underbrace{\min D([v]_j^-) \gg j}_{after} < \underbrace{\min d([v]_{j-1}^-) \gg j}_{after}$. After the visit, it also holds that $\min D([v]_i^-) \gg j \geq \min d([v]_i^-) \gg j \geq \min d([v]_{j-1}^-) \gg j$. Putting it all together, we conclude

$$\underbrace{\min D([v]_i^-) \gg j}_{after} \geq \underbrace{\min d([v]_i^-) \gg j}_{after} \geq$$

$$\underbrace{\min d([v]_{j-1}^-) \gg j}_{after} > \underbrace{\min D([v]_j^-) \gg j}_{after} \geq \underbrace{\min D([v]_j^-) \gg j}_{before} =$$

$$\underbrace{\min D([v]_i^-) \gg j}_{before} = \underbrace{\min d([v]_i^-) \gg j}_{before}$$

With Lemma 2.4, this Lemma is proven.

□

An observing reader has recognized the line in which $x_i([v]_i)$ is increased by exactly one. Lemma 4.2 explains this line and finally explains the benefit of the shift operator throughout the paper.

Lemma 4.2. *Suppose $\min D([v]_i^-) \gg i = \min d([v]_i^-) \gg i$ and that visiting a vertex $w \in V \setminus S$ changes $\min D([v]_i^-) \gg i$. Then $w \in [v]_i$ and if $[v]_i^-$ is not emptied, the change in $\min D([v]_i^-) \gg i$ is an increase by one. [Tho99]*

Proof. When visiting w, $\min D([v]_i^-) \gg i$ is changed and due to the fact that $\min D([v]_i^-) \gg i$ was equal to $\min d([v]_i^-) \gg i$ and d is nondecreasing and a lower bound for D

$$\underbrace{\min D([v]_i^-) \gg i}_{after} > \underbrace{\min D([v]_i^-) \gg i}_{before}$$

[1] see Lemma 13 in [Tho99]

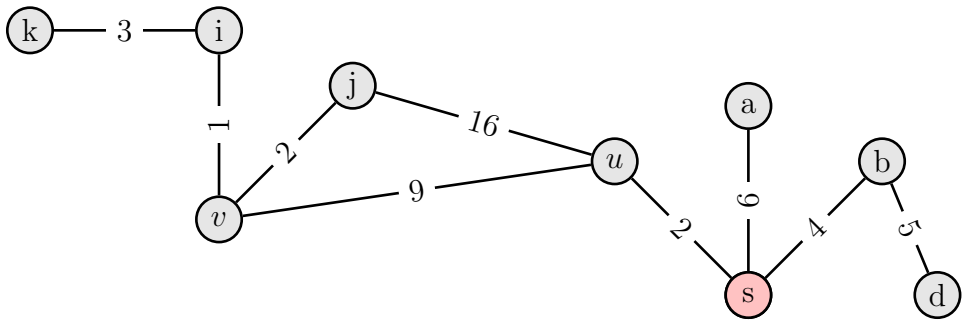

Figure 3: $S = \{s\}$

However, when recalling the way a vertex is visited(Algorithm 2), we conclude that the D values can never increase, thus $[v]_i^-$ must have been decreased and hence w must have been in $[v]_i$.

Considering $[v]_i^-$ after the visit, according to the assumption, $[v]_i^-$ is not empty, and since $[v]_i$ is connected, there must be an edge (u, x) in $[v]_i$ with $u \notin [v]_i^-$ and $x \in [v]_i^-$. This edge must exist, we could for example choose $u = w$ and x one arbitrary vertex of $[v]_i^-$. Assuming $u = w \Rightarrow [u]_0 = [w]_0$ and with Lemma 3.7 $[u]_0$ minimal, $d(u) \gg i = D(u) \gg i$ which equals $\min D([v]_i^-) \gg i$ before the visit. If $u \neq w$, $u \notin [v]_i^-$ before the visit to w, we are free to choose u from S. Since S contains the visited vertices, $d(u) \gg i \leq \min D([v]_i^-) \gg i$ just before the visit [2].

In any case $d(u) \gg i \leq \min D([v]_i^-) \gg i$ before the visit. Since the edge (u, x) is in $[v]_i$, $l(u, x) < 2^i$ and $D(x) \gg i \leq (d(u) + l(u, x)) \gg i \leq (\min D([v]_i^-) + l(u, x)) \gg i$. Thanks to the shifting, $l(u, x)$ can at least produce an overflow of one, thus $(\min D([v]_i^-) + l(u, x))) \gg i \leq (\min D([v]_i^-) \gg i) + 1$. Putting it together

$$\underbrace{\min D([v]_i^-) \gg i}_{after} \leq \underbrace{D(x) \gg i}_{after} \leq \underbrace{(\min D([v]_i^-) \gg i) + 1}_{before}$$

□

5 Example run

Related to Figure 3, the algorithm would start the following:

$S := \{s\}$
$D(s) = 0$, $D(u) = 2$, $D(a) = 6$, $D(b) = 4$, $D(*) = \infty$
$d(s) = 0$, $d(*) = \infty$
Visit($[s]_5$)

Keeping track on the Visit($[s]_5$) call, at fist $x_i([s]_5)$ is is set:
$x_i([s]_5) = \min D([s]_5^-) \gg 5 - 1 = \min\{\underbrace{2}_{u}, \underbrace{6}_{a}, \underbrace{4}_{b}, \underbrace{\infty}_{*}\} \gg 4 = 0$

The repeat-until loop runs until $[s]_5^- = \emptyset \vee x_i([s]_5) \gg 1$ is increased. If the loop terminates with $[s]_5^- = \emptyset$ then $S = V$ and all vertices are visited. According to Lemma 4.1, the second condition cannot apply. The inner while loop is executed as long as the following condition holds true:
$\exists$ child $[w]_4$ of $[s]_5$ such that $\min D([w]_4^-) \gg 4 = x_i([s]_5) = 0$
$[w]_4$ means all components with edge weight < 16, $[w]_4 = \{k, i, v, j, u, a, b, d\}$ and $\min D([w]_4^-) \gg$

[2] c.f. [Tho99] Lemma 10 for an in depth proof of this well conceivable fact. This condition even holds, if we cannot pick u from S

$4 = \min\{2, 6, 4, \infty\} \gg 4 = 0$. thus the while loop will make the following recursive calls.

Visit($[k]_4$)
Visit($[i]_4$)
Visit($[v]_4$)
Visit($[j]_4$)
Visit($[u]_4$)
Visit($[a]_4$)
Visit($[b]_4$)
Visit($[d]_4$)

6 Conclusion

A basic algorithm for visiting vertices has been presented. With this algorithmic structure and some additional data structures, a linear time and linear space algorithm can be created. However, Thorup's resulting algorithm uses data structures which only achieve the linear time characteristics for more than $2^{12^{20}}$ vertices. Thus, the existence of a linear time SSSP algorithm is shown, which however is not usable for practical application. The work towards a different node visiting algorithm has been presented, maybe with some different data structures, the algorithm can be changed to a linear time algorithm with more practical usage.

References

[Tho97] Mikkel Thorup. Undirected Single Source Shortest Path in Linear Time. In *FOCS*, pages 12–21, 1997.

[Tho99] Mikkel Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. ACM*, 46(3):362–394, 1999.