Finding Single-Source Shortest Paths on Planar Graphs with Nonnegative Edge Weights in Linear Time

in the seminar "Algorithms for Planar Graphs"

Nikolas Kirschstein

February 1, 2021

Abstract

The need to find shortest paths in a graph from some fixed source vertex to all other vertices is quite obvious and therefore one of the most important problems in graph theory. For general graphs, the standard way to go is the Dijkstra algorithm. On planar graphs, this approach takes linearithmic time in the number of vertices. However, we present an algorithm published by Henzinger et al. in 1997 [HKRS97] that accomplishes the task in *linear* time on planar graphs.

1 Introduction

Let $G = (V, E)$ be a *directed* planar graph. By convention, we set $n := |V|$ and $m := |E|$. Furthermore, we are given

- nonnegative *edge weights* $\$: E \to \mathbb{R}_{\geq 0}$ and
- an arbitrary but fixed *source vertex* $s \in V$

Our goal is to find a vertex vector $\boldsymbol{d} \in \mathbb{R}^V_{\geq 0}$ with the property that $\boldsymbol{d}[v]$ is the length of a shortest path in G with respect to $\$$ from s to v for all vertices $v \in V$. A well-known technique used for this is called *relaxation*:

Definition 1 (Relaxation)**.** An edge $uv \in E$ is *relaxed* if $\boldsymbol{d}[v] \leq \boldsymbol{d}[u] + \$(uv)$, otherwise we say it is *tense*. To *relax* an edge means updating $\boldsymbol{d}[v] := \boldsymbol{d}[u] + \$(uv)$ if uv is tense.

Dijkstra's Algorithm uses this technique in such a way that every edge is relaxed at most once and afterwards the labels in $\boldsymbol{d}$ are correct. Using priority queues implemented by min-heaps, the algorithm achieves a worst-case running time of

$$\mathcal{O}((n+m)\log(n)) \subseteq \mathcal{O}(n\log(n))$$

on planar graphs.[1] However, the Dijkstra algorithm always performs operations on a priority queue of size $\Theta(n)$, resulting in a queuing cost of $\Theta(\log(n))$. We will see that the asymptotic runtime can be significantly improved by subdividing the graph G into smaller parts and assigning priority queues to each portion of the graph. This way we need to perform operations on multiple priority queues but with far smaller size! Thanks to this, the resulting algorithm will run in time $\mathcal{O}(n)$.

[1] Planarity requires $m \leq 3n - 6$ due to Euler's Polyhedron Formula.

2 Prerequisites

Dividing the input graph randomly will not take us very far. Rather, we need certain properties to hold which are formalized by the concept of an r-*division*.

Definition 2 (r-division)**.** For $r \in \mathbb{N}_{>0}$, an r-*division* of a graph $G = (V, E)$ is a partition of E into disjoint *regions* $E = R_1 \cup R_2 \cup \ldots \cup R_s$ where

(R1) all regions have size $|R_i| \leq r$

(R2) the number of regions is $s \in \mathcal{O}(\frac{m}{r})$

(R3) each region's boundary size is $|\partial_G(R_i)| \in \mathcal{O}(\sqrt{r})$

The *boundary* of a region R consists of all vertices that are incident to edges within R but also connect to edges in at least one other region:

$$\partial_G(R) \coloneqq \{v \in V \mid \exists u, w \in V : uv \in R, vw \in E \setminus R\}.$$

It is important that we are not fooled by the terminology of a "region" into assuming that the regions resulting from an r-division are always connected.

Remark 3 (Connectedness)**.** The regions forming an r-division are *not* required to be connected. However, property (R3) ensures that region boundaries do not become too big, as it incents the regions to have as many inner vertices as possible. Thus, thinking intuitively about the regions as somewhat connected is indeed appropriate.

As shown by Henzinger et al. [HKRS97], running a kind of "distributed Dijkstra" on the regions of a single r-division already improves the running time to $\mathcal{O}(n \log(\log(n)))$. This naturally raises the question of whether dividing the graph further might lead to even better performance and motivates the notion of a recursive division, which is essentially a tree of divisions of G where the root is all of E and every division further down the tree is made up of smaller regions until the leaves only consist of single edges as illustrated in Figure 1.

Definition 4 (Recursive r-division)**.** Let $G = (V, E)$ be a nonempty graph, i.e. $|E| \geq 1$, and $\overrightarrow{r} = (r_0, \ldots, r_p) \in \mathbb{N}^{p+1}$ a vector of increasing integers $1 = r_0 < r_1 < \cdots < r_p = m$. The $\overrightarrow{r}$-*height* of a region $R \subseteq E$ shall be defined as

$$h(R) \coloneqq \min\{i \in \{0, \ldots, p\} \mid |R| \leq r_i\}$$

If $h(R) = 0$, the region R is called *atomic*. A *recursive* $\overrightarrow{r}$-*division* of G is a tree $\mathcal{R}$ with root E, whose children are defined recursively as follows:

- If $p = 0$, the root E has no children. It is atomic as $r_0 = 1 = r_p$.
- If $p > 0$, let $R_1, \ldots, R_s \subseteq E$ form an r_{p-1}-division of G. Then E has s children, namely for each $i = 1, \ldots, s$ some recursive $(r_0, \ldots, r_{h(R_i)-1}, |R_i|)$-division of the subgraph induced by R_i constitutes a child of E.

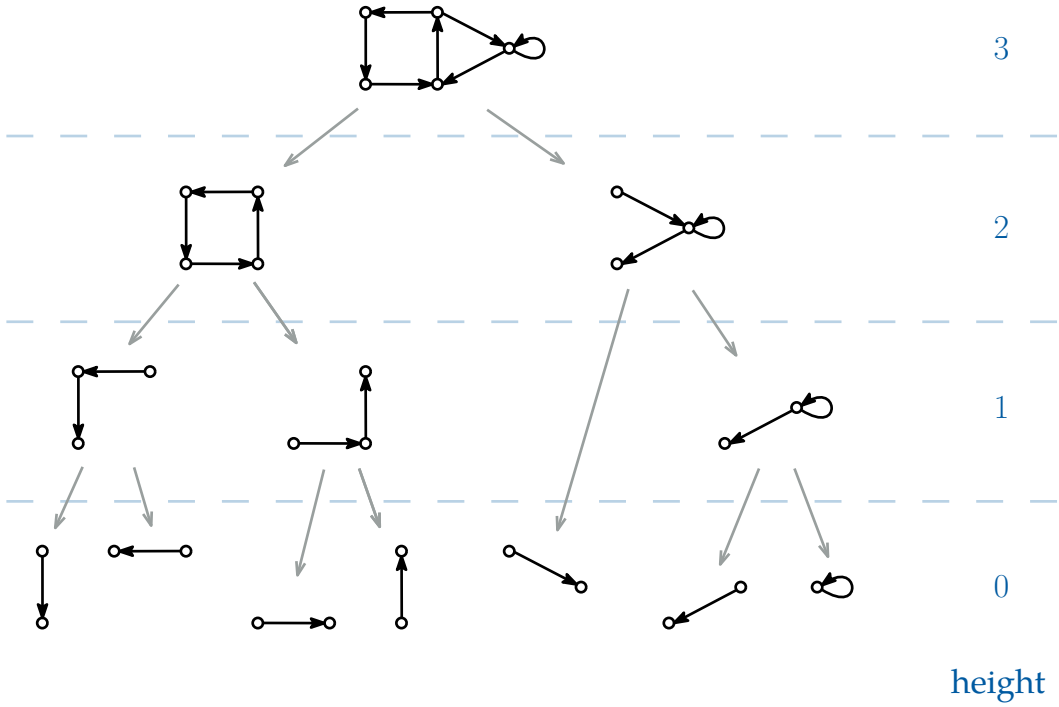

Figure 1: An examplatory recursive $(1, 2, 4, 7)$-division

We will use such a recursive division as the basis for our algorithm. Fortunately, calculating it in a preprocessing step does not affect our linear-time ambitions at all, thanks to the following theorem:

Theorem 5 (Goodrich, 1995)**.** *A recursive $\vec{r}$-division of G can be computed in $\mathcal{O}(n)$ time.*

Proof. See Goodrich [Goo95]. □

Therefore, we can from now on assume that the planar input graph G is already equipped with some recursive $\vec{r}$-division $\mathcal{R}$. Furthermore, we assume without loss of generality that for every vertex $v \in V$ in G we have $\text{indeg}(v) \leq 2$ and $\text{outdeg}(v) \leq 2$. This imposes no restriction, because every vertex violating the requirement can be split into multiple "dummy" vertices connected by a cycle of edges with weight zero like in Figure 2. However, we must be careful not to enlarge G too much, but since it is a planar graph, the average vertex degree can be at most $\frac{2m}{n} \leq \frac{2(3n-6)}{n} = 6 - \frac{12}{n} < 6$ and hence G will only grow by a constant factor through this process in the worst case, rendering it completely irrelevant to the asymptotic running time.

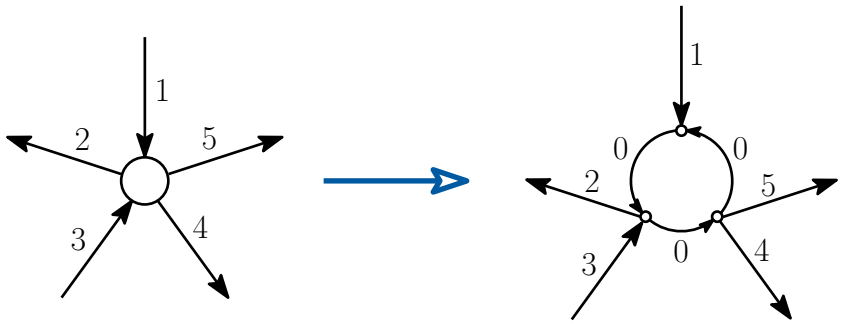

Figure 2: Splitting vertices to achieve small degrees

3 The Algorithm

As mentioned earlier in Section 1, the algorithm is going to depend heavily on priority queues as auxiliary data structure in a similar way that Dijkstra does. We require a priority queue Q to support the following interface:

Operation	Description
Q.`minItem()` $\in \mathcal{R}$	Returns the item in Q with the minimum key.
Q.`minKey()` $\in \mathbb{R}_{\geq 0}$	Returns the key of the Q.`minItem()`.
Q.`setKey`$(x, k) \in \mathbb{B}$	Updates the key of the item $x \in Q$ to $k \in \mathbb{R}_{\geq 0}$. Returns whether the update caused Q.`minKey()` to decrease.

Table 1: Required operations of the priority queue data structure

Albeit, in contrast to Dijkstra's algorithm, we will not use only a single priority queue for the whole graph, but rather grant every region $R \in \mathcal{R}$ of the recursive $\overrightarrow{r}$-division an own priority queue Q_R obeying the following invariants:

(I1) If R is atomic, its queue $Q_R = \{uv\}$ consists of precisely the single edge uv in R. The key of uv in Q_R is either

- finite, in which case it must be equal to $\boldsymbol{d}[u]$, the distance label of the tail, and we say the edge uv is *active*, or
- ∞, in which case uv is called *inactive*.

(I2) If R is nonatomic, $Q_R = \{R_1, \ldots, R_s\}$ contains its immediate subregions $R_i \in \mathcal{R}$. The key of each child region R_i in Q_R must be Q_{R_i}.`minKey()`.

The idea behind these somewhat nested priority queues is that Q_R.`minKey()` represents the minimum over the keys of *all* edges $uv \in R$. Formal evidence for this will be given later by Corollary 12 (Queue Consistency).

With this, we are ready to formulate the algorithm: In essence, it performs a modified version of Dijkstra recursively on the regions of $\mathcal{R}$, though not exhaustively until all edges within a region are relaxed, but only at most for a limited number of iterations. This *attention span* is defined per level of $\mathcal{R}$ by parameters $\alpha_0, \alpha_1 \ldots, \alpha_p$, the values of which will be specified in Section 5.3

The algorithm consists of three procedures. Procedure 1 is the top-level procedure and initializes the graph similarly to Dijkstra, except it also has to take care of the keys in the priority queues. Afterwards it resorts to calling the main procedure on the whole graph until all edges are inactive.

Procedure 2 is responsible for updating some key in a priority queue and informing the parent queue about this event, as it might need to update the child region's key accordingly in order to maintain the invariant (I2).

Procedure 3 does the actual work of relaxing edges within a region for a limited amount $\alpha_{h(R)}$ of iterations, as sketched before. If an invocation of `process` is not able to execute the full $\alpha_{h(R)}$ iterations because there are no more active edges in the region left, we call the invocation *truncated*.

```
/* calculates shortest path lengths                    */
sssp(s ∈ V : source):
1   | foreach v ∈ V :
2   |   | d[v] := ∞
3   | foreach R ∈ ℛ, x ∈ Q_R :
4   |   | Q_R.setKey(x, ∞)
5   | d[s] := 0
6   | foreach sv ∈ E : // outgoing edges
7   |   | update({sv}, sv, 0)
8   | while Q_E.minKey() ≠ ∞ :
9   |   | process(E)
```

Procedure 1: The entry point to the algorithm

```
/* updates the given item with the provided
   key and propagates it upwards                       */
update(R ∈ ℛ : region, x ∈ Q_R : item, k ∈ ℝ_≥0 : key):
1   | if Q_R.setKey(x, k) : // Q_R.minKey() reduced
2   |   | update(R.parent, R, k)
```

Procedure 2: The useful key update helper

```
/* performs some work in the given region
   for
   a limited "attention span"                          */
process(R ∈ ℛ : region):
1   | if R = {uv} ⊆ E : // R is atomic
2   |   | if d[u] + $(uv) < d[v] :
3   |   |   | d[v] := d[u] + $(uv)
4   |   |   | foreach vw ∈ E : // outgoing edges
5   |   |   |   | update({vw}, vw, d[v])
6   |   | Q_R.setKey(uv, ∞) // deactivate edge
7   | else : // R is nonatomic
8   |   | repeat α_h(R) times or until Q_R.minKey() = ∞ :
9   |   |   | R' := Q_R.minItem()
10  |   |   | process(R')
11  |   |   | Q_R.setKey(R', Q_R'.minKey())
```

Procedure 3: The main workhorse

Limiting the number of iterations in `process` is necessary for the following reason: Assume we have relaxed all edges in some region $R \in \mathcal{R}$ and are currently working on another region $R' \in \mathcal{R}$ as we relax an ingoing edge $uv \in E$ of a boundary vertex v between R' and R. Because of line 5, we activate its outgoing edges $vw \in E$, some of which are part of R that we thought to have already finished! Thus, work done within a region is only speculative and we do not want to be biased too much towards that region. The event is so significant that we introduce some new terminology:

Definition 6 (Entry Vertex)**.** A boundary vertex of a region is called an *entry vertex* if it has an outgoing edge into that region. More formally, a vertex $u \in V$ is an entry vertex of a region $R \in \mathcal{R}$ iff $u \in \{v \in \partial_G(R) \mid \exists vw \in R\}$. As the root region E has an empty boundary, we additionally define the source s to be the only entry vertex of E.

If in the situation described above updating the key of $vw \in R$ results in decreasing Q_R.`minKey()`, we refer to this as *foreign intrusion of R via entry vertex v*. Note that the reduction of the distance label of some entry vertex is the *only* way a foreign intrusion can happen. The following two observations will be highly relevant to the analysis.

Lemma 7 (Dumping)**.** *Let $R \in \mathcal{R}$ be a region and v an entry vertex of R. If there are two foreign intrusions of R via v at times $t_1 < t_2$, then Q_R.`minKey()`$|_{t_1} > Q_R$.`minKey()`$|_{t_2}$.*

Proof. A foreign intrusion is only possible through the activation of outgoing edges of an entry vertex v in line 5 of `process`, which because of line 2 however only occurs if $\boldsymbol{d}[v]$ decreases in the first place, so it is smaller at time t_2 than at t_1. □

Remark 8 (Foreign Intrusion)**.** Let $R \in \mathcal{R}$ be a region. If Q_R.`minKey()` decreases at any point throughout the algorithm, this is due to a foreign intrusion into R.

Proof. The only point where Q_R.`minKey()` can decrease is line 1 of `update` through updating an edge $vw \in R$. If no foreign intrusion were responsible for this, then v would be no entry vertex of R and thus the edge uv, the relaxation of which was responsible for the call to `update`, would belong to R. Since uv has been selected in line 10 of `process`, at that time Q_R.`minKey()` $= \boldsymbol{d}[u]$, as will be shown in Corollary 12, and therefore

$$Q_R\texttt{.getKey}(vw) = \boldsymbol{d}[v] = \boldsymbol{d}[u] + \$(uv) \geq Q_R\texttt{.minKey()}$$

which clearly is a contradiction. ↯ □

The direct implication of this insight is that a chain of recursive `update` calls in line 5 on regions $R_0 \subseteq R_1 \subseteq R_2 \subseteq \cdots \in \mathcal{R}$ ends as soon as the region in question contains the edge uv having originally issued these calls, which we depict in Figure 3.

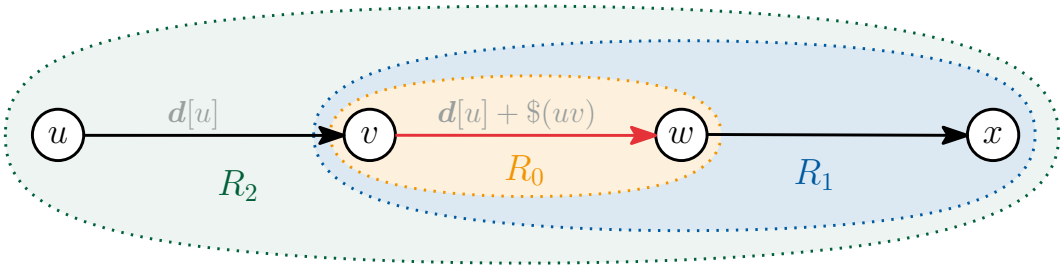

Figure 3: A foreign intrusion stops being one higher up in the recursive division tree

4 Correctness

Let $\delta : V \times V \to \mathbb{R}_{\geq 0}$ be the true shortest path lengths in G. To show that the algorithm works correctly, we need three properties to hold at termination:

(C1) $\boldsymbol{d}[s] = 0$

(C2) $\forall v \in V : \boldsymbol{d}[v] \geq \delta(s, v)$

(C3) $\forall uv \in E : \boldsymbol{d}[v] \leq \boldsymbol{d}[u] + \$(uv)$

If these are satisfied, then $\boldsymbol{d}[v] = \delta(s, v)$ for all $v \in V$. That the first two properties are satisfied is pretty clear from the algorithm. The third one will be proven in two steps: First we show that, if an edge is inactive, then it is relaxed, and second that in the end all edges are inactive, which yields (C3). The first step is resolved quickly:

Lemma 9 (Relaxation)**.** *Let $uv \in E$ be an edge. Then*

$$uv \text{ is inactive} \Rightarrow uv \text{ is relaxed}$$

Proof. This is trivially true initially, as labels and keys are ∞. Just before the first call to process in line 9 of `sssp` this still holds as only the outgoing edges of s are unrelaxed, but also active thanks to the `update` call in line 7 of `sssp`. Later on the algorithm only deactivates an edge $uv \in E$ in line 6 of `process` just after having made sure it is relaxed.

On the other hand, a relaxed edge $vw \in E$ can only become tense if the label $\boldsymbol{d}[v]$ of the tail v decreases in line 3 of `process`. However, immediately afterwards in line 5 of `process` an `update` is performed for each outgoing edge, including vw, leading it to be activated again by line 1 of `update`. □

Now we turn to the second step, for which we prior need to finally prove that the two invariants (I1) and (I2) the algorithm relies on are maintained.

Lemma 10 (Key Invariant)**.** *The key k of an active edge $uv \in E$ satisfies $k = \boldsymbol{d}[u] < \infty$.*

Proof. Initially all edges are inactive. An edge $vw \in E$ becomes only active due to line 1 of `update`, namely by setting its key precisely to the finite value $\boldsymbol{d}[v]$ if we consider the calls to `update` in line 7 of `sssp` and line 5 of `process`. □

Lemma 11 (Queue Invariant)**.** *For any non-root region $R \in \mathcal{R}$ that is not an ancestor of the currently* `processed` *region,*

$$Q_{R.\texttt{parent}}.\texttt{getKey}(R) = Q_R.\texttt{minKey}()$$

Proof. Initially, this holds trivially as all keys are ∞. Whenever the minimum key of a queue is decreased in line 1 of `update`, the subsequent recursive call on the parent guarantees that the equality is preserved. Furthermore, when an edge is deactivated in line 6 of `process`, this is accounted for in line 11 of `process`, which in turn accounts for its own update inductively on higher levels. □

With these in tow, we can now show that the priority queues are indeed consistently behaving in the way we have expected them to all the time. This is particularly relevant when deciding which region to select next in line 10 of `process`.

Corollary 12 (Queue Consistency)**.** *For any region $R \in \mathcal{R}$ that is not an ancestor of the currently* `processed` *region,*

$$Q_R\texttt{.minKey()} = \min\{\boldsymbol{d}[u] : uv \in R \textit{ active}\}$$

Proof. Induction on the height of R:

- If $h(R) = 0$, then $R = \{uv\} \subseteq E$ is atomic.
  - If uv is inactive, then $Q_R\texttt{.minKey()} = \infty = \min \emptyset$ trivially.
  - If uv is active, then $Q_R\texttt{.minKey()} = \boldsymbol{d}[u] = \min\{\boldsymbol{d}[u]\}$ by Lemma 10.
- If $h(R) > 0$, then R has children $R_1, R_2, \ldots, R_s \in \mathcal{R}$ and

$$\begin{aligned} Q_R\texttt{.minKey()} &= \min_i\{Q_R\texttt{.getKey(}R_i\texttt{)}\} \\ &= \min_i\{Q_{R_i}\texttt{.minKey()}\} \\ &= \min_i\{\min\{\boldsymbol{d}[u] : uv \in R_i \text{ active}\}\} \\ &= \min\{\boldsymbol{d}[u] : uv \in R \text{ active}\} \end{aligned}$$

  using Lemma 11 (Queue Invariant) and the induction hypothesis. □

Now that we have proven the parts of our two-step strategy separately, we simply put it all together to show property (C3). This concludes the correctness proof.

Corollary 13. *At termination, all edges are relaxed.*

Proof. The algorithm terminates when the condition in line 8 of `sssp` is false, which means according to Corollary 12 (Queue Consistency) that

$$\infty = Q_E\texttt{.minKey()} = \min\{\boldsymbol{d}[u] : uv \in E \text{ active}\}$$

which yields using Lemma 10 (Key Invariant) that there are no active edges, so all edges are inactive, and therefore by Lemma 9 (Relaxation) all edges are relaxed. □

5 Complexity

Being reassured that the algorithm indeed is working correctly, we now turn towards determining its time complexity. After all, we have promised a linear running time in Section 1, so we need to substantiate this claim now.

5.1 Accounting Scheme

For the purpose of analyzing the running time of the algorithm, we augment the two core procedures `process` and `update` to track the total computational cost throughout execution. The underlying idea is to eventually charge all arising worst-case queuing time to *accounts* (R, v) where $R \in \mathcal{R}$ is a region and v an entry vertex of R.

Procedure 4, the modified version of `update`, has two additional tasks: The first one is to accumulate the cost incurred by a call of `update`, including potential descendant invocations. This is done by estimating $\log |Q_R|$ in line 0.5 as worst-case queueing cost for line 1 and, if executed, adding the cost returned by the recursive call in line 2. This accumulated cost is then returned in line 2.5 to provide it to the caller.

The second task of the modified `update` is just as important: Keeping track of foreign intrusions! For this, we add an additional array `entry` indexed by regions $R \in \mathcal{R}$. Whenever the reduction of the distance label $\boldsymbol{d}[v]$ in line 3 of `process` causes Q_R.`minKey()` to decrease, `entry`$[R]$ is set to v in line 1.5, as we know by Remark 8 (Foreign Intrusion) that in this case v must be an entry vertex of R. Furthermore, the only way for Q_R.`minKey()` to become finite is via foreign intrusion, wherefore we are guaranteed that at any time where Q_R.`minKey()` is finite, the entry `entry`$[R]$ is well-defined. In particular, it is the entry vertex of the last foreign intrusion into R.

```
update(R ∈ ℛ : region, x ∈ Q_R : item, k ∈ ℝ_≥0 : key,
       v ∈ V : potential entry vertex of R):
0.5   cost := log |Q_R|
1     if Q_R.setKey(x, k) : // Q_R.minKey() reduced
1.5       entry[R] := v
2         cost += update(R.parent, R, k, v)
2.5   return cost
```

Procedure 4: The modified `update` code tracking cost and foreign intrusions

While having an eye on its queueing operations as well, procedure 5, the modified version of `process`, is essentially responsible for bookkeeping of cost amounts that can be thought of as *debt obligations*. These travel up and down the tree of recursive `process` invocations until at some point being charged to an account. In order to accomplish this, an invocation of the modified `process` receives an extra argument `debt` that is meant to be a portion of the debt of ancestor invocations and will be referred to as *inherited* debt. Henceforth, let $R \in \mathcal{R}$ be the region of the invocation.

If R is atomic, the `debt` is incremented in line 6.5 by 1 for the `setKey` operation on the singleton priority queue in line 6 and, if executed, for each `update` call in line 5 increased by the cost it returns.

If R is nonatomic, the invocation will share its `debt` among the children, of which it expects to have $\alpha_{h(R)}$. Thus, in line 10 each child invocation receives in *its* arguments

- a $\frac{1}{\alpha_{h(R)}}$ fraction of the `debt` inherited by the parent plus
- a cost of $\log |Q_R|$ for selecting the child's region in line 9.

Using the variable `credit`, the parent invocation tracks the amount of debt passed on to its children via line 9.5. If the invocation calls $\alpha_{h(R)}$ children, we have `credit` $=$ `debt`. Otherwise, the invocation is truncated and some debt remains.

Debts are not only passed down in the tree of `process` invocations, but also bubble upwards through return values. The parent invocation combines

- the accumulated debt received from its child invocations (`upDebt`) and
- the amount of inherited debt not covered by a credit (`debt` $-$ `credit`)

to determine the updated debt in line 11.1 after the recursive subcalls. This ensures that the balance is correct again also in the case of a truncated execution. Hence we know that, for the last part of the modified procedure, `debt` contains the true final debt of the current invocation.

```
process(R ∈ ℛ : region, debt ∈ ℕ : inherited debt):
1   if R = {uv} ⊆ E : // R is atomic
2       if d[u] + $(uv) < d[v] :
3           d[v] := d[u] + $(uv)
4           foreach vw ∈ E : // outgoing edges
5               debt += update({vw}, vw, d[v], v)
6       Q_R.setKey(uv, ∞) // deactivate edge
6.5     debt += 1
7   else : // R is nonatomic
7.3     credit := 0 // debt passed to children
7.7     upDebt := 0 // debt received from children
8       repeat α_h(R) times or until Q_R.minKey() = ∞ :
9           R' := Q_R.minItem()
9.5         credit += debt / α_h(R)
10          upDebt += process(R', debt / α_h(R) + log |Q_R|)
11          Q_R.setKey(R', Q_R'.minKey())
11.1    debt := upDebt + (debt − credit)
11.2    if Q_R.minKey() will decrease in the future :
11.3        return debt
11.4    else : // this invocation is stable
11.5        pay off debt from account (R, entry[R])
11.6        return 0
```

Procedure 5: The modified `process` code managing debt obligations

To describe the behavior at the end of procedure 5, we need to differentiate `process` invocations based on a key property that we discuss next. For invocations A and B of `process` we write $A \leq B$ if both process the same region and either $A = B$ or the invocation A occurs before B, in which case we also write $A < B$.

Definition 14 (Stability)**.** Let A be an invocation of `process` on a region $R \in \mathcal{R}$. Define:

- $\triangleright(A)$ is the value of Q_R.`minKey()` just before the *start* of A.
- $\square(A)$ is the value of Q_R.`minKey()` just after the *end* of A.

We call A *stable* if for all later invocations $B > A$ we have $\triangleright(A) \leq \triangleright(B)$.

This notion of stability now determines whether the invocation simply returns the updated debt to *its* parent in line 11.3 or rather pays off the debt itself by withdrawing from the account $(R, \texttt{entry}[R])$ in line 11.5. Ensuring that this payoff does not get out of hand is the concern of the next section.

5.2 Bounding Payoff

Our main goal for this section is to bound the usage of the accounts computational cost is charged to during the modified algorithm. This will be rather technical and not required for understanding the main argumentation, so feel free to skip this part and only take note of the central result, which is Theorem 17 (Payoff).

First, we prove an analogon to the fact that Dijkstra's algorithm assigns vertex labels in nondecreasing order. It is clear that this cannot hold for our algorithm as a whole, because foreign intrusions are a violation of monotonous assignment. However, it is at least true for the execution of `process` on a particular region.

Lemma 15 (Rising Keys)**.** *For any invocation A of* `process` *with children $A_1, A_2, \ldots, A_s$,*

$$\triangleright(A) \leq \triangleright(A_1) \leq \triangleright(A_2) \leq \cdots \leq \triangleright(A_s) \leq \square(A)$$

In addition, every key assigned during A is no less than $\triangleright(A)$.

Proof. Induction on the height of R.

- If $h(R) = 0$, then A processes an atomic region $\{uv\} \subseteq E$. Hence, $\triangleright(A) = \boldsymbol{d}[u]$ and $\square(A) = \infty$. The only key assigned is $\boldsymbol{d}[u] + \$(uv) \geq \triangleright(A)$.
- If $h(R) > 0$, then A processes a nonatomic region $R \in \mathcal{R}$. For $i = 1, 2, \ldots, s$ let k_i be the value of Q_R.`minKey()` when A_i is invoked and k_{s+1} its value when A_s returns. Line 10 of `process` ensures $k_i = \triangleright(A_i)$ for $i \leq s$. Using the inductive hypothesis, each key assigned during A_i is at least $\triangleright(A_i)$. Thus we have $k_i \leq k_{i+1}$ by Corollary 12 (Queue Consistency) and altogether

  $$\triangleright(A) = k_1 \leq k_2 \leq \cdots \leq k_s \leq k_{s+1} = \square(A)$$

  Observing also line 11 of `process`, keys assigned during A are at least $\triangleright(A)$. □

Therewith, we can show the following helper lemma which appears rather specific, because it is the immediate groundwork for proving Theorem 17.

Lemma 16 (Children Stability)**.** *Let $A < B$ be invocations of* `process` *on $R \in \mathcal{R}$ such that*

(1) *no foreign intrusion of R occurs between A and B*

(2) *the later invocation B is stable*

Then every child of A is stable.

Proof. Let A' be a child of A and C' an invocation with $A' < C'$. We need to show:

$$\triangleright(A') \leq \triangleright(C')$$

For this, we consider the parent invocation C of C':

- If $C = A$, then $\triangleright(A') \leq \triangleright(C')$ by Lemma 15 (Rising Keys).
- If $C > A$, then $\triangleright(A') \leq \square(A)$ and also $\triangleright(C) \leq \triangleright(C')$ by Lemma 15 (Rising Keys). Thus, it suffices to show that $\square(A) \leq \triangleright(C)$.
  - If $C \leq B$, then no foreign intrusion of R occurs between A and C, wherefore we get $\square(A) \leq \triangleright(C)$ by Remark 8 (Foreign Intrusion).
  - If $C > B$, then clearly $\square(A) \leq \triangleright(B)$ by Remark 8 (Foreign Intrusion) and $\triangleright(B) \leq \triangleright(C)$ by the stability of B, yielding $\square(A) \leq \triangleright(C)$. □

Here is our desired statement on how often accounts are overdrawn: At most once! It cannot be overstated how crucial this circumstance will be for estimating the overall running time later on.

Theorem 17 (Payoff)**.** *Let $R \in \mathcal{R}$ be a region and v an entry vertex of R. The account (R, v) is charged with a* positive *amount* at most once*.*

Proof. If (R, v) is never charged to, this trivially holds. Otherwise let A be the *earliest* invocation paying off a *positive* amount from (R, v). Then R must be the region processed by A, vertex v must be the value of $\texttt{entry}[A]$ at termination time of A and, most importantly, A must be stable. Let t_1 be the last time before A where $\texttt{entry}[R]$ was set.

Assume for a contradiction that there exists an invocation $B > A$ that also charges a positive amount to (R, v). Then v must also be the value of $\texttt{entry}[R]$ at termination time of B and B must be stable. Let t_2 be the last time before B that $\texttt{entry}[R]$ was set.

- If $t_2 > t_1$, then by Lemma 7 (Dumping)

  $$Q_R.\texttt{minKey()}|_{t_2} < Q_R.\texttt{minKey()}|_{t_1} \leq \triangleright(A)$$

  although A is stable. $\lightning$
- If $t_2 = t_1$, then there are no foreign intrusions between A and B. This means by Lemma 16 (Children Stability) that every child of A is stable and returns zero debt in line 11.6 of `process`, so $\texttt{upDebt} = 0$ at termination time of A.
  - If A is truncated, then $\square(A) = \infty$ and by Remark 8 thus $\triangleright(B) = \infty$. $\lightning$
  - If A is not truncated, then $\texttt{debt} = \texttt{credit}$ in line 11.1 of `process` and A pays off zero in line 11.5, which is not positive. $\lightning$

Therefore, no such $B > A$ exists. □

5.3 Parameter Juggling

The total computational cost depends on two factors:

- The region size limits $\overrightarrow{r} = (r_0, \ldots, r_p)$ of the recursive $\overrightarrow{r}$-division.
- The attention spans $\alpha_1, \ldots, \alpha_p$ for the levels of $\mathcal{R}$.

For now, define the latter in terms of the former:

$$\alpha_i \coloneqq \frac{4\log(r_{i+1})}{3\log(r_i)}, \tag{1}$$

where we set $r_{p+1} \coloneqq 16^{r_p{}^{\frac{1}{6}}}$, the choice of which will become apparent later on. With this, we can more concretely describe debt in the modified `process` procedure:

Lemma 18 (Inherited Debt)**.** *A height-i* `process` *invocation inherits at most $4\log(r_{i+1})$ debt.*

Proof. Reverse induction on i.

- If $i = p$, then the invocation inherits no debt.
- If $i < p$, then the invocation inherits no more than $4\log(r_{i+1})$ debt by the inductive hypothesis and in line 10 calls each child with a debt of at most

  $$\tfrac{4\log(r_{i+1})}{\alpha_i} + \log(r_i) = 3\log(r_i) + \log(r_i) = 4\log(r_i)$$

  using the definition of α_i in (1). □

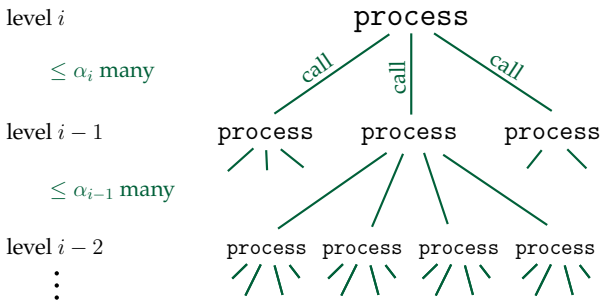

Figure 4: Combinatorially counting descendant invocations

To estimate the overall debt, we need to also bound the number of child invocations of a given invocation. Here is where the attention spans play a key role: Consider an invocation of `process` at height i. It can have at most α_i child invocations at height $i-1$, exactly α_i if it is not truncated. Every child can in turn call at most α_{i-1} further children itself, each of which has at most α_{i-2} children and so on. This combinatorial argument is illustrated in Figure 4 and leads to the definition of the following shorthand, signifying the maximum number of height-j descendant invocations that any height-i invocation of `process` can possibly have:

$$\beta_{ij} \coloneqq \begin{cases} \alpha_i \alpha_{i-1} \dots \alpha_{j+1} & \text{if } i \geq j \\ 0 & \text{otherwise} \end{cases}$$

Putting together all we found until here, a first expression for the worst-case amount of overall debt emerging throughout the modified algorithm is possible.

Lemma 19 (Preliminary `process` Debt)**.** *The total debt incurred by* `process` *invocations is*

$$\mathcal{O}\Big(\sum_{i=0}^{p} \frac{m}{\sqrt{r_i}} \sum_{j \leq i} \beta_{ij} \cdot 4 \log(r_{j+1})\Big) \tag{2}$$

Proof. First, we make sure that all debt incurred by `process` invocations is eventually charged to accounts. The region of a root `process` call is all of E, where no foreign intrusions can occur into. Therefore, every invocation of height p is stable and pays off any potentially remaining debt.

Next, consider some account (R, v) where R is a height-i region and v a boundary vertex of R. By Theorem 17 (Payoff), this account is paid off a positive amount with at most once. Let A be the corresponding invocation. The withdrawn debt consists only of the debt returned by child invocations of A and so we need to take into account all descendant invocations of A, whereof there are at most β_{ij} many. Each such descendant at height j inherits at most $4 \log(r_{j+1})$ debt by Lemma 18 (Inherited Debt) and hence the maximum debt paid off with (R, v) is given by

$$\sum_{j \leq i} \beta_{ij} \cdot 4 \log(r_{j+1})$$

Finally, we simply need to count all accounts, in which we are aided by the properties of r-divisions. We know by property (R2) that there are $\mathcal{O}(\frac{r_{i+1}}{r_i}) \subseteq \mathcal{O}(\frac{m}{r_i})$ regions of height i, each of which has by (R3) a boundary size in $\mathcal{O}(\sqrt{r_i})$, yielding a total boundary size at height i of at most $\mathcal{O}(\frac{m}{r_i}) \cdot \mathcal{O}(\sqrt{r_i}) = \mathcal{O}(\frac{m}{\sqrt{r_i}})$, which constitutes an upper bound on the number of entry vertices on level i. Summing over all levels gives us (2). □

A similar result can be achieved for the procedure `update`, the proof of which is rather involved and not shown here. Instead, we refer to the literature.

Lemma 20 (Preliminary `update` Debt)**.** *The total debt incurred by* `update` *invocations is*

$$\mathcal{O}\Big(\sum_{i=0}^{p}\frac{m}{\sqrt{r_i}}\beta_{i0}\sum_{k\leq i+1}\log(r_k)+\sum_{j=0}^{p}2\frac{m}{\sqrt{r_j}}\sum_{i<j}\sqrt{r_i}\beta_{i0}\sum_{k\leq j+1}\log(r_k)\Big)$$

Proof. See Klein and Mozes [KM11]. □

An end is in sight! The only unknowns left are the region sizes $\overrightarrow{r}=(r_0,r_1,\ldots,r_p)$ that we are at this point finally going to define recursively as

- $r_0 \coloneqq 1$
- $r_{j+1} \coloneqq 16^{r_j^{\frac{1}{6}}}$

Intuitively speaking, from the leaves upwards regions shall grow exponentially. Having resolved this last missing piece, we are able to finalize our debt estimations. Again, we will only prove the result for `process` and cite that for `update`.

Lemma 21 (Final `process` Debt)**.** *The total debt incurred by invocations of* `process` *is* $\mathcal{O}(m)$*.*

Proof. It is not too difficult to show that $r_j^{\frac{1}{6}} \geq 1.78^j$ for $j \geq 7$. The underlying analytical calculations are carried out by Klein and Mozes [KM11], so we take it as a given here. Using this, we continue bounding expression (2) for `process`:

$$\begin{aligned}
\mathcal{O}\Big(\sum_{i=0}^{p}\frac{m}{\sqrt{r_i}}\sum_{j\leq i}\beta_{ij}\cdot 4\log(r_{j+1})\Big) &\overset{\text{resolve }\beta_{ij}}{=} \mathcal{O}\Big(\sum_{i=0}^{p}\frac{m}{\sqrt{r_i}}\sum_{j\leq i}\Big(\frac{4}{3}\Big)^{i-j}\frac{\log(r_{i+1})}{\log(r_{j+1})}\cdot 4\log(r_{j+1})\Big)\\
&= \mathcal{O}\Big(4m\sum_{i=0}^{p}r_i^{-\frac{1}{2}}\sum_{j\leq i}\Big(\frac{4}{3}\Big)^{i-j}\log(r_{i+1})\Big)\\
&\subseteq \mathcal{O}\Big(4m\sum_{i=0}^{p}r_i^{-\frac{1}{2}}(i+1)\Big(\frac{4}{3}\Big)^{i}\log(r_{i+1})\Big)\\
&= \mathcal{O}\Big(8pm\sum_{i=0}^{p}r_i^{-\frac{1}{2}}\Big(\frac{4}{3}\Big)^{i}4r_i^{\frac{1}{6}}\Big)\\
&= \mathcal{O}\Big(32pm\sum_{i=0}^{p}r_i^{-\frac{1}{3}}\Big(\frac{4}{3}\Big)^{i}\Big)\\
&\overset{r_j^{\frac{1}{6}}\geq 1.78^j}{\subseteq} \mathcal{O}\Big(32pm\sum_{i=0}^{p}\frac{\left(\frac{4}{3}\right)^{i}}{1.78^{2i}}\Big)\\
&\subseteq \mathcal{O}\Big(32pm\sum_{i=0}^{\infty}0.5^{i}\Big)\\
&\overset{\text{geom. }\Sigma}{=} \mathcal{O}(64pm)=\mathcal{O}(m)
\end{aligned}$$

□

Lemma 22 (Final `update` Debt)**.** *The total debt incurred by invocations of* `update` *is* $\mathcal{O}(m)$*.*

Proof. See Klein and Mozes [KM11]. □

This concludes the time complexity analysis. Inferring that the overall running time is linear constitutes merely a formality, but let us deservedly state it anyway.

Theorem 23. *The algorithm* `sssp` *runs in* $\mathcal{O}(n)$ *time.*

Proof. As remarked earlier, the input preprocessing steps can be executed in $\mathcal{O}(n)$ time. The total computational cost without preprocessing is

$$\mathcal{O}(m) + \mathcal{O}(m) = \mathcal{O}(2m) = \mathcal{O}(m) \subseteq \mathcal{O}(3n - 6) = \mathcal{O}(n)$$

and therefore linear in the size of the planar input graph G. □

6 Summary

By successively subdividing the planar input graph into smaller and smaller regions in a clever way, we have managed to build an algorithm that finds single-source shortest paths on the graph in linear time. This is possible because most of the priority queue operations involved are performed on rather small queues. The algorithm shows that the SSSP problem on planar graphs can be solved with an asymptotically optimal time complexity, a significant theoretical result!

However, since the algorithm was published in 1997 there has apparently not been a single real implementation. The suspicion arises that it is not practicable despite its linear running time for the following reason: We simply took it as a given that the preprocessing step for computing a recursive $\overrightarrow{r}$-division is possible in time $\mathcal{O}(n)$, however this itself already involves heavy machinery incurring a high implicit constant in front of the n. Moreover, our algorithm is not exactly economical in the amount of priority queues it handles and does not in any way optimize their use like established versions of the Dijkstra algorithm do, which increases the implicit constant even further.

We therefore conjecture that it takes an impractically large input graph for our algorithm with $\mathcal{O}(n)$ time and big constant to outperform a highly-optimized Dijkstra variant with $\mathcal{O}(n \log(n))$ time but extremely small constant. That nevertheless does not change the theoretical value of this algorithm and the neat concepts it utilizes.

References

[Goo95] Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374 – 389, 1995.

[HKRS97] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3 – 23, 1997.

[KM11] Philip Klein and Shay Mozes. Shortest paths with nonnegative lengths. In *Optimization Algorithms for Planar Graphs*, pages 61 – 75. `http://www.planarity.org`, 2011.