Chapter 6

Shortest paths with nonnegative lengths

In this chapter, we give a linear-time algorithm for computing single-source shortest paths in a planar graph with nonnegative lengths. The algorithm uses recursive divisions (discussed in Section 5.10). We start with some basic concepts about shortest paths in arbitrary graphs.

6.1 Shortest-path basics: path-length property and relaxed and tense darts

Let G be a directed graph with dart-lengths given by a dart-vector $\boldsymbol{c}$. We say a vertex vector $\boldsymbol{d}$ has the *path-length property* with respect to $\boldsymbol{c}$ if, for each vertex v, $\boldsymbol{d}[v]$ is the length (with respect to $\boldsymbol{c}$ of some s-to-v path (not necessarily the shortest).

We say a dart vw is *relaxed* with respect $\boldsymbol{c}$ and the vertex vector $\boldsymbol{d}$ if $\boldsymbol{d}[w] \leq \boldsymbol{d}[v] + \text{length}(vw)$. An unrelaxed dart is said to be *tense*.

Let s be a vertex. If $\boldsymbol{d}[s] = 0$ and $\boldsymbol{d}$ satisfies the path-length property and every arc is relaxed then, for each vertex v, $\boldsymbol{d}[v]$ is the length of the *shortest* s-to-v path.

A basic step in several shortest-path algorithms is called *relaxing* a dart. Suppose $\boldsymbol{d}$ is a vertex vector with the path-length property and dart vw is tense. Relaxing vw consists of executing

$$\boldsymbol{d}[w] := \text{length}(vw) + \boldsymbol{d}[v]$$

after which vw is relaxed and $\boldsymbol{d}$ still has the path-length property.

For the case of nonnegative lengths, Dijkstra's algorithm [**?**] generates an ordering of darts to relax so that each dart is relaxed at most once, after which all dart are relaxed. If the algorithm uses a priority queue as suggested by

Johnson [**?**], the running time is $O(m \log n)$ where m is the number of arcs (we assume $m \geq n - 1$ since otherwise the graph is disconnected).

6.2 Using a division in computing shortest-path distances

The shortest-path algorithm operates on a graph equipped with a recursive division. The algorithm runs quickly because most queue operations are performed on smaller queues.

The algorithm has a limited "attention span." It chooses a region, then performs a number of steps of Dijkstra's algorithm (the number depends on the height of the region), then abandons that region until later. Thus it skips around between the regions.

To provide intuition, we briefly describe a simplified version of the algorithm. The simplified version runs in $O(n \log \log n)$ time. Divide the graph into $O(n/\log^4 n)$ regions of size $O(\log^4 n)$ with boundaries of size $O(\log^2 n)$. We associate a status, *active* or *inactive*, with each edge. Initialize by deactivating all edges and setting all node labels $\boldsymbol{d}[v]$ to infinity. Then set the label of the source to 0, and activate its outgoing edges. Now repeat the following three steps:

Step 1: Select the region containing the lowest-labeled node that has active outgoing edges in the region.

Step 2: Repeat $\log n$ times:

Step 2A: Select the lowest-labeled node v in the current region that has active outgoing edges in the region. Relax and deactivate all its outgoing edges vw in that region. For each of the other endpoints w of these edges, if relaxing the edge vw resulted in decreasing the label of w, then activate the outgoing edges of w.

Note that applying Dijkstra's algorithm to the region would entail repeating Step 2A as many times as there are nodes in the region. Every node would be selected exactly once. We cannot afford that many executions of Step 2A, since a single region is likely to be selected more than once in Step 1. In contrast to Dijkstra's algorithm, when a node is selected in Step 2A, its current label may not be the correct shortest-path distance to that node; its label may later be decreased, and it may be selected again. Since the work done within a region during a single execution of Step 2 is speculative, we don't want to do too much work. On the other hand, we don't want to execute Step 1 too many times. In the analysis of this algorithm, we show how to "charge" an execution of Step 1 to the $\log n$ iterations of Step 2A.

There is an additional detail. It may be that Step 2A cannot be repeated $\log n$ iterations because after fewer than $\log n$ times there are no active outgoing edges left in the region. In this case, we say the execution of Step 2 is *truncated*.

Since we cannot charge a truncated execution of Step 2 to $\log n$ iterations of Step 2A, we need another way to bound the number of such executions. It turns out that handling this "detail" is in fact the crux of the analysis. One might think that after a region R underwent one such truncated execution, since all its edges were inactive, the same region would never again be selected in Step 1. However, relax-steps on edges in another region R' might decrease the label on a node w on the boundary between R and R', which would result in w's outgoing edges being activated. If w happens to have outgoing edges within R, since these edges become active, R will at some later point be selected once again in Step 1.

This problem points the way to its solution. If R "wakes up" again in this way, we can charge the subsequent truncated execution involving R to the operation of updating the label on the boundary node w. The analysis makes use of the fact that there are relatively few boundary nodes to bound the truncated executions. Indeed, this is where we use the fact that the regions have small boundaries.

6.2.1 The algorithm

We assume without loss of generality that the input graph G is directed, that each node has at most two incoming and two outgoing edges, and that there is a finite-length path from s to each node. We assume that the graph is equipped with a recursive division.

The algorithm maintains a vertex vector $\boldsymbol{d}$ with the path-length property.

For each region R of the recursive division of G, the algorithm maintains a priority queue $Q(R)$. If R is nonatomic, the items stored in $Q(R)$ are the immediate subregions of R. For an atomic region $R(uv)$, $Q(R(uv))$ consists of only one item, the single arc uv contained in $R(uv)$; in this case, the key associated with the arc is either infinity or the label $\boldsymbol{d}[u]$ of the tail of the arc.

The algorithm is intended to ensure that for any region R, the minimum key in the queue $Q(R)$ is the minimum distance label $\boldsymbol{d}[v]$ over all arcs vw in R that need to be processed. We make this precise in Lemma 6.3.3. This idea of maintaining priority queues for nested sets is not new, and has been used, e.g. in finding the k^{th} smallest element in a heap [. Frederickson min-heap .].

We assume the priority queue data structure supports the operations

- *minItem*(Q), which returns the item in Q with the minimum key,
- *minKey*(Q), which returns the key associated with *minItem*(Q)
- *updateKey*(Q, x, k), which updates the key of x to k (x must be an item of Q) and returns a boolean indicating whether the update caused *minKey*(Q) to decrease.

We indicate an item is inactive by setting its key to infinity. Items go from inactive to active and back many times during the algorithm. We never delete items from the queue. This convention is conceptually convenient because it

avoids the issue of having to decide, for a given newly active item, into which of the many priority queues it should be re-inserted.

The algorithm uses parameters α_i to specify an "attention span" for each level of the recursive division. We will specify their values in Section 6.5. The algorithm consists of the following two procedures.

```
def PROCESS(R)
        pre: R is a region.
1       if R contains a single edge uv,
2         if d[v] > d[u] + length(uv),
3           d[v] := d[u] + length(uv)
4           for each outgoing edge vw of v, call UPDATE(R(vw), vw, d[v]).
5         updateKey(Q(R), uv, ∞).
6       else (R is nonatomic)
7         repeat α_height(R) times or until minKey(Q(R)) is infinity:
8             R' := minItem(Q(R))
9             PROCESS(R')
10            updateKey(Q(R), R', minKey(Q(R'))).
```

```
def UPDATE(R, x, k)
        : pre: R is a region, x is an item of Q(R), and k is a key value.
1         updateKey(Q(R), x, k)
2         if the updateKey operation reduced the value of minKey(Q(R)) then
3           UPDATE(parent(R), R, k).
```

To compute shortest paths from a source s, proceed as follows. Initialize all vertex-labels and keys to infinity. Then assign the label $\boldsymbol{d}[s] := 0$, and for each outgoing arc sv, call UPDATE$(R(sv), sv, 0)$. Then repeatedly call PROCESS(R_G), where R_G is the region consisting of all of G, until the call results in $minKey(Q(R_G))$ being infinity. Since the vertex-labels $\boldsymbol{d}$ are only updated by steps in which arcs are relaxed, the vertex-labels satisfy the path-length property throughout the algorithm's execution. In the next section, we show that, when the algorithm terminates, all arcs are relaxed. It follows that the vertex-labels give distances from s.

We define an *entry vertex* of a region R as follows. The only entry vertex of the region R_G is s itself. For any other region R, v is an entry vertex if v is a boundary vertex of R such that some arc vw belongs to R.

When the algorithm processes a region R, it finds shorter paths to some of the vertices of R and so reduces their labels. Suppose one such vertex v is a boundary vertex of R. The result is that the shorter path to v can lead to shorter paths to arcs in a neighboring region R' for which v is an entry vertex. In order to preserve the property that the minimum key of $Q(R')$ reflects the labels of vertices of R', therefore, the algorithm might need to update $Q(R')$. Updating

priority queues of neighboring regions is handled by the UPDATE proedure. The reduction of $minKey(Q(R'))$ (which can *only* occur as a result of the reduction of the label of an entry vertex v) is a highly significant event for the analysis. We refer to such an event as a *foreign intrusion of region R' via entry vertex v.*

Lemma 6.2.1. *Let R be a region. Suppose there are two foreign intrusions of R via v, one at time t_1 and one at time t_2, where $t_1 < t_2$. Then $minKey(Q(R))$ is greater at time t_1 than at time t_2.*

Proof. $\mathbf{d}[v]$ must get smaller for the second intrusion to count. □

6.3 Correctness

We say that an arc uv is *active* if the key of uv in $Q(R(uv))$ is finite. To prove that every arc is relaxed at termination, we show that (a) if an arc is inactive, then it is relaxed, and (b) at termination all arcs are inactive.

Lemma 6.3.1. *If an arc uv is inactive then it is relaxed (except during Line 4).*

Proof. The lemma holds just before the first call to PROCESS since at that point every node but s has label infinity, and every outgoing arc of s is active. The algorithm only deactivates an arc uv, i.e., uv is assigned a key of ∞ in Line 5, just after the arc is relaxed.

An arc vw could become tense when the labels of its endpoints change. Note that labels never go up. The label of v might go down in Line 4, but in the same step the algorithm calls $\text{UPDATE}(R(vw), vw, \mathbf{d}[v])$ for each outgoing arc vw of v. In Line 1 of UPDATE, the key of vw is updated to a finite value, so vw is again active. □

Lemma 6.3.2. *The key of an active arc vw is $\mathbf{d}[v]$ (except during Line 4).*

Proof. Initially all labels and keys are ∞. Whenever a label $\mathbf{d}[v]$ is assigned a value k (either in the initialization, where $v = s$, or in Line 4), $\text{UPDATE}(R(vw), vw, k)$ is called for each outgoing arc vw. The first step of $\text{UPDATE}(R, v, k)$ is to update the key of vw to k. □

Next we show that the queues are, in a sense, consistent. The region of an invocation A of PROCESS is simply the region that was the argument to that invocation of PROCESS. The most recent invocation of PROCESS that has not yet returned is called the *current invocation*, and that invocation's region is called the *current region*.

Lemma 6.3.3. *For any region R that is not an ancestor of the current region, the key associated with R in $Q(parent(R))$ is the min key of $Q(R)$.*

Proof. At the very beginning of the algorithm, all keys are infinity. Thus in this case the lemma holds trivially. Every time the minimum key of some queue $Q(R)$ is changed in Line 1 of UPDATE, a recursive call to UPDATE in Line 2 ensures that the key associated with R in $Q(\text{parent}(R))$ is updated.

We must also consider the moment when a new region becomes the current region. This happens upon invocation of the procedure PROCESS, and upon return from PROCESS.

- When PROCESS(R) is newly invoked, the new current region R is a child of the old current region, so Lemma 6.3.3 applies to even fewer regions than before; hence we know it continues to hold.
- When PROCESS(R') returns in step 9, the parent R of R' becomes current. Hence at that point Lemma 6.3.3 applies to R'. Note, however, that immediately after the call to PROCESS(R'), the calling invocation updates the key of R' in $Q(R)$ to the value $\mathit{minKey}(Q(R'))$.

□

Corollary 6.3.4. *For any region R that is not an ancestor of the current region,*

$$\mathit{minKey}(Q(R)) = \min\{\boldsymbol{d}[v] : vw \text{ is an active arc contained in } R\} \tag{6.1}$$

Proof. By induction on the height of R, using Lemma 6.3.3. □

The algorithm terminates when $Q(R_G)$ becomes infinite. At this point, according to Corollary 6.3.4, G contains no active arc, so, by Lemma 6.3.1, all arcs are relaxed. Since the vertex-labels satisfy the path-length property, it follows that they are shortest-path distances. We have proved the algorithm is correct.

6.4 The Dijkstra-like property of the algorithm

Because of the recursive structure of PROCESS, each initial invocation PROCESS(R_G) is the root of a tree of invocations of PROCESS and UPDATE, each with an associated region R. Parents, children, ancestors, and descendants of invocations are defined in the usual way.

In this section, we give a lemma that is useful in the running-time analysis. This lemma is a consequence of the nonnegativity of the lengths; it is analogous to the fact that in Dijkstra's algorithm the labels are assigned in nondecreasing order.

For an invocation A of PROCESS on region R, we define start(A) and end(A) to be the values of $\mathit{minKey}(Q(R))$ just before the invocation starts and just after the invocation ends, respectively.

Lemma 6.4.1. *For any invocation A, and for the children $A_1, \ldots, A_p$ of A,*

$$\mathit{start}(A) \le \mathit{start}(A_1) \le \mathit{start}(A_2) \le \cdots \le \mathit{start}(A_p) \le \mathit{end}(A). \tag{6.2}$$

Moreover, every key assigned during A is at least $\mathit{start}(A)$.

Proof. The proof is by induction on the height of A. If A's height is 0 then it has an atomic region $R(uv)$. In this case the start key is the value of $\boldsymbol{d}[u]$ at the beginning of the invocation. The end key is infinity. The only finite key assigned (in UPDATE) is $\boldsymbol{d}[u]+\text{length}(uv)$, which is at least $\boldsymbol{d}[u]$ by nonnegativity of edge-lengths. There are no children.

Suppose A's height is more than 0. In this case it invokes a series $A_1, \ldots, A_p$ of children. Let R be the region of A. For $i = 1, \ldots, p$, let k_i be the value of $minKey(Q(R))$ at the time A_i is invoked, and let k_{p+1} be its value at the end of A_p. Line 8 of the algorithm ensures $k_i = \text{start}(A_i)$ for $i \leq p$. By the inductive hypothesis, every key assigned by A_i is at least $\text{start}(A_i)$, so $k_{i+1} \geq k_i$. Putting these inequalities together, we obtain

$$k_1 \leq k_2 \leq \cdots \leq k_{p+1}$$

Note that $k_1 = \text{start}(A)$ and $k_{p+1} = \text{end}(A)$. Thus (6.2) holds and every key assigned during A is at least $\text{start}(A)$. □

6.5 Accounting for costs

Now we begin the running-time analysis. The time required is dominated by the time for priority-queue operations. Lines 8 and 10 of PROCESS involve operations on the priority queue $Q(R)$. We charge a total cost of $\log |Q(R)|$ for these two steps. Similarly, we charge a cost of $\log |Q(R)|$ for Line 3 of UPDATE. Line 5 performs an operation on a priority queue of size one, so we only charge one for that operation. Our goal is to show that the total cost is linear.

To help in the analysis, we give versions of the procedures UPDATE and PROCESS that have been modified to keep track of the costs and to keep track of foreign intrusions. Note that the modifications are purely an expository device for the purpose of analysis; the modified procedures are not intended to be actually executed, and in fact one step of the modified version of PROCESS cannot be executed since it requires knowledge of the future!

Amounts of cost are passed around by UPDATE and PROCESS via return values and arguments. We think of these amounts as *debt obligations*. These debt obligations travel up and down the forest of invocations, and are eventually charged by invocations of PROCESS to pairs (R, v) where R is a region and v is an entry vertex of R.

The modified version of UPDATE, given below, handles cost in a simple way: each invocation returns the total cost incurred by it and its proper descendants (in Line 3 if there are proper descendants and in Line 3a if not).

The modified UPDATE has another job to do as well: it must keep track of foreign intrusions. There is a table entry$[\cdot]$ indexed by regions R. Whenever the reduction of the vertex-label $\boldsymbol{d}[v]$ causes $minKey(Q(R))$ to decrease, entry$[R]$ is set to v in step 2a. Initially the entries in the table are undefined. However, for any region R, the only way that $minKey(Q(R))$ can become finite is by an intrusion. We are therefore guaranteed that, at any time at which $minKey(Q(R))$ is finite, entry$[R]$ is an entry vertex of R.

```
def UPDATE(R, x, k, v)
      pre: R is a region, x is an item of Q(R), k is a key value,
           and v is a boundary vertex of R.
1     updateKey(Q(R), x, k, v)
2     if the updateKey operation reduced the value of minKey(Q(R)) then
2a      entry[R] := v
3       return log |Q(R)| + UPDATE(parent(R), R, k, v).
3a    else return log |Q(R)|
```

The cost of the invocation UPDATE(R, x, k, v) is $\log |Q(R)|$ because of the *updateKey* operation in step 1.

We now turn to the modified PROCESS procedure. An invocation takes an additional argument, debt, which is a portion of the cost incurred by ancestor invocations. We refer to this amount as the debt *inherited* by the invocation.

Let R be the invocation's region. If R is atomic, the invocation's debt is increased by the cost incurred by calls to UPDATE, and then by 1 to account for the *updateKey* operation on the single-element priority queue.

If R is not atomic, the invocation will have some children, so can pass some of its debt down to its children. Since the parent invocation expects to have $\alpha_{\text{height}(R)}$ children, it passes on to each child

- a $1/\alpha_{\text{height}(R)}$ fraction of the parent's inherited debt, plus
- the $\log |Q(R)|$ cost the parent incurred in selecting the child's region from the priority queue.

The parent uses a variable *credit* to keep track of the amount of its inherited debt that it has successfully passed down to its children. If the parent has $\alpha_{\text{height}(R)}$ children, the parent's total credit equals its inherited debt. If not, we say the parent invocation is *truncated*.

Lemma 6.5.1. *A nontruncated invocation sends to its children all the debt it inherits or incurs.*

Debts also move *up the tree.* The parent invocation receives some debt from each child. The parent adds together

- the debt received from its children (upDebt) and
- the amount of inherited debt for which it has not received a credit (zero unless the parent is truncated)

to get the total amount the parent owes. The parent then either passes that aggregate debt to *its* parent or pays off the debt itself by withdrawing from an account, the account associated with the pair (R, v) where v is the value of entry$[R]$ at the time of the invocation (step 10d).

Because of Lemma 6.5.1, as debt moves up the tree, no new debt is added by nontruncated invocations of PROCESS.

```
def PROCESS(R, debt)
        pre: R is a region.
1       if R contains a single edge uv,
2         if d[v] > d[u] + length(uv),
3           d[v] := d[u] + length(uv)
4           for each outgoing edge vw of v, debt+ = UPDATE(R(vw), vw, d[v], v).
5           updateKey(Q(R), uv, ∞).
5a          debt+ = 1
6       else (R is nonatomic)
6a        upDebt:= 0, credit:=0
7         repeat α_height(R) times or until minKey(Q(R)) is infinity:
8             R' := minItem(Q(R))
8a            credit+ = debt/α_height(R)
9             upDebt+ = PROCESS(R', debt/α_height(R) + log |Q(R)|)
10            updateKey(Q(R), R', minKey(Q(R'))).
10a       debt+ = upDebt − credit
10b     if minKey(Q(R)) will decrease in the future,
10b       return debt
10c     else (this invocation is stable)
10d       pay off debt by withdrawing from account of (R, entry[R])
10e       return 0
```

We say an invocation A of PROCESS is *stable* if, for every invocation $B > A$, the start key of B is at least the start key of A. If an invocation is stable, it pays off the debt by withdrawing the necessary amount from the account associated with $(R, \text{entry}[R])$. If not, it passes the debt up to its parent. The following theorem is proved in Section 6.6

Theorem 6.5.2 (Payoff Theorem)**.** *For each region R and entry vertex v of R, the account (R, v) is used to pay off a positive amount at most once.*

Any invocation whose region is the whole graph is stable because there are no foreign intrusions of that region. Therefore such an invocation never tries to pass any debt to its nonexistent parent. We are therefore guaranteed that all costs incurred by the algorithm are eventually charged to accounts.

The total computational cost depends on the parameters $\bar{r} = (r_0, r_1, r_2, \ldots)$ of the recursive $\bar{r}$-division and on the parameters $\alpha_0, \alpha_1, \ldots$ that govern the number of iterations per invocation of PROCESS. For now, we define the latter parameters in terms of the former parameters; later we define the former. Define $\alpha_i = \frac{4 \log r_{i+1}}{3 \log r_i}$.

Lemma 6.5.3. *Each invocation at height i inherits at most $4 \log r_{i+1}$ debt.*

Proof. By reverse induction on i. Each top-level invocation inherits no debt. Suppose the lemma holds for i, and consider a height-i invocation. By the inductive hypothesis, it inherits at most $4 \log r_{i+1}$ debt, so it passes down to

each child a debt of at most $\frac{4\log r_{i+1}}{\alpha_i} + \log r_i$. The choice of α_i ensures that this is $4\log r_i$. □

Define $\beta_{ij} = \alpha_i\alpha_{i-1}\ldots\alpha_{j+1}$ for $i > j$. β_{ii} is defined to be 1, and for $i < j$, we define β_{ij} to be zero.)

Lemma 6.5.4. *A* PROCESS *invocation of height i has at most β_{ij} descendant* PROCESS *invocations of height j.*

Debt incurred by the algorithm by a step of PROCESS is called *process debt.*

Lemma 6.5.5. *For each height-i region R and boundary vertex x of R, the amount of process debt payed off by the account (R, x) is at most $\sum_{j\le i}\beta_{ij}4\log r_{j+1}$.*

Proof. By the Payoff Theorem, this account is used at most once. Let A be the invocation of PROCESS that withdraws the payoff from that account. Each dollar of process debt paid off by A was sent back to A from some descendant invocation who inherited or incurred that dollar of debt. Thus, to account for the total amount of process debt paid off by A, we consider each of its descendant invocations. By Lemma 6.5.4, A has β_{ij} descendants of height j, and each such descendant inherited or incurred a debt of at most $4\log r_{j+1}$ dollars, by Lemma 6.5.3. □

Cost incurred by the algorithm by a step of UPDATE is called *update debt.* The event (in Line 3 of PROCESS) of reducing a vertex v's label $\boldsymbol{d}[v]$ initiates a chain of calls to UPDATE in Line 4 for each outgoing arc vw. We say the debt incurred is *on behalf of* v.

Lemma 6.5.6. *If* UPDATE *is called on the parent of region R during an invocation A of* PROCESS *then R is not the region of A.*

Proof. Let R_A be the region of A. Recall that start(A) is the value of *minKey*$(Q(R_A))$ just before A begins. By Lemma 6.4.1, every key assigned during A is at least start(A).

Consider a chain of UPDATE calls initiated by the reduction of the label of vertex v. By the condition in Line 2 of PROCESS and the condition in Line 2 of UPDATE, in order for UPDATE to be called on the parent of R, that label must have been less than the value of *minKey*$(Q(R))$. This shows $R \neq R_A$. □

For a vertex v, define

$$\text{height}(v) = \max\{j \,:\, v \text{ is a boundary vertex of a height-}j\text{ region}\}$$

Corollary 6.5.7. *A chain of calls to* UPDATE *initiated by the reduction of the label of v has total cost at most $\sum_{k\le height(v)+1}\log r_k$.*

Proof. Let A_0 be the invocation of Process during which the initial call to UPDATE was made, and let $R(uv)$ be the (atomic) region of A_0. Consider the chain of calls to UPDATE, and let

$$R(vw)=R_0, R_1, \ldots, R_p$$

be the corresponding regions. Note that height$(R_j) = j$. The cost of call j is $\log |Q(R_j)|$, which is at most $\log r_j$. Since R_{p-1} contains vw but (by Lemma 6.5.6) does not contain uv, v is a boundary vertex of R_{p-1}, so $p \leq \text{height}(v) + 1$. □

6.6 The Payoff Theorem

In this section, we prove the Payoff Theorem, repeated here for convenience:

> **Payoff Theorem** (Theorem 6.5.2): For each region R and entry vertex v of R, the account (R, v) is used to pay off a positive amount at most once.

In this section, when we speak of an *invocation*, we mean an invocation of PROCESS. We define the partial order $\leq$ on the set of invocations of PROCESS as follows: $A \leq B$ if A and B have the same region, and A occurs no later than B. We say in this case that A is a *predecessor* of B. We write $A < B$ if $A \leq B$ and $A \neq B$.

As we remarked earlier, the only way *minKey*$(Q(R))$ can decrease is if there is a foreign intrusion into R. We restate this as follows.

Lemma 6.6.1. *Let B be an invocation with region R. Suppose that between time t and the time B starts, there are no foreign intrusions of R. Then start(B) is at least the value of minKey$(Q(R))$ at time t.*

Lemma 6.6.2. *Suppose $A < B$ are two invocations such that no foreign intrusion occurs between A and B and such that B is stable. Then every child of A is stable.*

Proof. Let A' be a child of A, and let C' be any invocation such that $A' < C'$. If we can prove

$$\text{start}(A') \leq \text{start}(C') \tag{6.3}$$

then it will follow that A' is stable. Let C be the parent invocation of C' (so the region of C' is R). If $C = A$, then (6.3) follows from Lemma 6.4.1.

Assume therefore that $C > A$. It follows from Lemma 6.4.1 that start$(A') \leq$ end(A) and that start$(C) \leq$ start(C'), so it suffices to show end$(A) \leq$ start(C). There are two cases.

- *Case 1: $C \leq B$* . In this case, end$(A) \leq$ start(C) by Lemma 6.6.1.
- *Case 2: $C > B$.* In this case, end$(A) \leq$ start(B) follows by Lemma 6.6.1, and start$(B) \leq$ start(C) follows by the stability of B, so end$(A) \leq$ start(C).

□

Now we can prove the Payoff Theorem, which states that each pair (R, v) is charged a positive amount at most once.

If (R, v) is never charged to, we are done. Otherwise, let A be the earliest invocation that pays off a positive amount from the account (R, v) in step 10d. Then R is the region of A, v is the value of entry$[R]$ at the time of A, and A is stable. Let t_1 be the time when entry$[R]$ was last set before A.

Assume for a contradiction that there is an invocation B such that $A < B$ and such that B also charges to (R, v). Then v is the value of entry$[R]$ at the time of B, and B is stable. Let t_2 be the time when entry$[R]$ was last set before B. If $t_2 > t_1$ then, by Lemma 6.2.1, $minKey(Q(R))$ at time t_2 was less than that at time t_1, which in turn was no more than start(A) by Lemma 6.2.1, contradicting the stability of A.

We conclude that no foreign intrusion of R occurs between A and B. By Lemma 6.6.2, therefore, every child of A is stable. It follows from step 10e that every child of A returns a zero debt, so in invocation A the value of upDebt is zero. Assume for a contradiction that A's credit does not cover its own inherited debt. Then A must be a truncated invocation, so end$(A) = \infty$. By Lemma 6.6.1, start$(B) = \infty$, a contradiction. Therefore, A pays off zero, a contradiction. This completes the proof of the Payoff Theorem.

6.7 Analysis

Let c_1, c_2 be the constants such that an r-division of an m-arc graph has at most $c_1 m/r$ regions, each having at most $c_2\sqrt{r}$ boundary vertices.

Lemma 6.7.1. *Let m be the number of edges of the input graph. For any nonnegative integer i, there are at most $c_1 c_2 m/\sqrt{r_i}$ pairs (R, x) where R is a height-i region and x is an entry vertex of R.*

Combining this lemma with Lemma 6.5.5, we obtain

Corollary 6.7.2. *The total process debt is at most*

$$c_1 c_2 \sum_i \frac{m}{\sqrt{r_i}} \sum_{j \le i} \beta_{ij} 4 \log r_{j+1} \tag{6.4}$$

Lemma 6.7.3. *Let i, j be nonnegative integers, and let R be a region of height i. The total amount of update debt incurred on behalf of vertices of height at most j and paid off from accounts $\{(R, x) : x \text{ an entry vertex of } R\}$ is at most*

$$c_2\sqrt{r_i}\beta_{i0} \sum_{k=0}^{j+1} \log r_k$$

Proof. The number of entry vertices x of R is at most $c_2\sqrt{r_i}$. For each, the Payoff Theorem ensures that all the debt paid off from account (R, x) comes from descendants of a single invocation A of PROCESS. The number of height-0 descendants of A is β_{i0}. For each such level-0 descendant A_0, if the corresponding update debt is on behalf of a vertex of height at most j then by Corollary 6.5.7 the cost is at most $\sum_{k=0}^{j+1} \log r_k$. □

Lemma 6.7.4. *The total update debt is at most*

$$\sum_i c_1 c_2 \frac{m}{r_i} \sqrt{r_i} \beta_{i0} \sum_{k=0}^{i+1} \log r_k \tag{6.5}$$

$$+ \quad \sum_j c_1 \frac{m}{\sqrt{r_j}} 2 \sum_{i<j} c_2 \sqrt{r_i} \beta_{i0} \sum_{k=0}^{j+1} \log r_k \tag{6.6}$$

Proof. To each unit of update debt, we associate two integers: i is the height of the region R such that the debt is paid off from an account (R, x), and j is the height of the vertex v on whose behalf the debt was incurred. If $i \geq j$, we refer to the debt as *type 1* debt, and if $i < j$, we refer to it as *type 2* debt.

First we bound the type-1 debt. For each integer i, there are at most $c_1 \frac{m}{r_i}$ regions R of height i. By Lemma 6.7.3, the total type-1 debt is therefore

$$\sum_i c_1 c_2 \frac{m}{r_i} \sqrt{r_i} \beta_{i0} \sum_{k=0}^{i+1} r_k$$

Now we bound the type-2 debt. For each integer j, the number of regions of height j is at most $c_1 \frac{m}{r_j}$ and each has at most $c_2 \sqrt{r_j}$ entry vertices, so the total number of vertices of height j is at most $c_1 c_2 \frac{m}{\sqrt{r_j}}$. For each such vertex v, there are at most 2 incoming darts uv. Any height-0 invocation of PROCESS that reduced v's label involved one of these two darts. For each such dart uv, for each integer $i < j$, there is exactly one height-i region R that includes uv. By Lemma 6.7.3, the total type-2 debt is therefore

$$\sum_j c_1 \frac{m}{\sqrt{r_j}} 2 \sum_{i<j} c_2 \sqrt{r_i} \beta_{i0} \sum_{k=0}^{j+1} \log r_k$$

□

6.8 Parameters

In this section, we show that there is a way to choose the parameters $r_0, r_1, r_2, \ldots$ so that the total process cost and the total update cost are $O(m)$.

Problem 6.8.1. *For $r_0 = 1, r_1 = \log^4 m$, and $r_2 = m$, show that the total cost is $O(m \log \log m)$.*

We define $r_0, r_1, \ldots$ inductively by $r_0 = 1$ and $r_{j+1} = 16^{r_j^{1/6}}$. This defines an increasing sequence such that

$$\log r_{j+1} = 4 r_j^{1/6} \tag{6.7}$$

so

$$\log^2 r_{j+1} = 16 r_j^{1/3} \tag{6.8}$$

Lemma 6.8.2. $r_j^{1/6} \geq 1.78^j$ *for* $j \geq 7$.

Proof. Define $u_j = \log r_j^{1/6}$. Then $\log r_{j+1} = 4 \cdot 2^{u_j}$ and

$$u_{j+1} = \log r_{j+1}^{1/6} = \frac{1}{6} \log r_{j+1} = \frac{1}{6} 4 \cdot 2^{u_j} = \frac{2}{3} 2^{u_j}$$

A simple induction shows that $u_j \geq .838j$ for $j \geq 7$. Since $r_j^{1/6} = 2^{u_j}$, this implies the lemma.. □

Lemma 6.8.3. *The process-debt (6.4) is* $O(m)$.

Proof.

$$\begin{aligned}
&\sum_i \frac{m}{\sqrt{r_i}} \sum_{j \leq i} \beta_{ij} 4 \log r_{j+1} \\
&= \sum_i \frac{m}{\sqrt{r_i}} \sum_{j \leq i} (4/3)^{i-j} \frac{\log r_{i+1}}{\log r_{j+1}} 4 \log r_{j+1} \text{ by definition of } \beta_{ij} \\
&= 4m \sum_i r_i^{-1/2} \sum_{j \leq i} (4/3)^{i-j} \log r_{i+1} \\
&\leq 4m \sum_i r_i^{-1/2} c (4/3)^i \log r_{i+1} \text{ for a constant } c \\
&\leq 4cm \sum_i r_i^{-1/2} (4/3)^i 4 r_i^{1/6} \\
&= 16cm \sum_i r_i^{-1/3} (4/3)^i
\end{aligned}$$

which is $O(m)$ by Lemma 6.8.2. □

Lemma 6.8.4. *The update-debt is* $O(m)$.

Proof. First we show that $\sum_j r_j^{-1/2} (4/3)^j \log^2 r_{j+1}$ is bounded by a constant c. (The recurrence relation for r_j was chosen to make this true.)

$$\begin{aligned}
\sum_j r_j^{-1/2} (4/3)^j \log^2 r_{j+1} &\leq \sum_i r_j^{-1/2} (4/3)^j 16 r_j^{1/3} \text{ by (6.8)} \\
&= \sum_i r_j^{-1/6} (4/3)^j \\
&= \sum_i 1.78^{-j} (4/3)^j \text{ by Lemma 6.8.2}
\end{aligned}$$

which is bounded by a constant c.

We also use the fact that $\sum_{k=0}^{i+1} \log r_k \leq c' \log r_{i+1}$ for a constant c'.

Next we bound the type-1 debt (6.5).

$$\begin{aligned}
&\sum_i c_1 c_2 \frac{m}{r_i}\sqrt{r_i}\beta_{i0}\sum_{k=0}^{i+1}\log r_k \\
&\quad\le\quad c_1 c_2 m \sum_i r_i^{-1/2}\sqrt{r_i}\beta_{i0}c'\log r_{i+1} \\
&\quad=\quad \sum_i c_1 c_2 c' m r_i^{-1/2}\sqrt{r_i}(4/3)^i\frac{\log r_{i+1}}{\log r_1}\log r_{i+1} \text{ by definition of } \beta_{i0} \\
&\quad=\quad \frac{c_1 c_2 c' m}{\log r_1}\sum_i \frac{1}{\sqrt{r_i}}(4/3)^i \log^2 r_{i+1} \\
&\quad=\quad \frac{c_1 c_2 c' m}{\log r_1} c
\end{aligned}$$

Now we bound the type-2 debt (6.6).

$$\begin{aligned}
&\sum_j c_1\frac{m}{\sqrt{r_j}}2\sum_{i<j} c_2\sqrt{r_i}\beta_{i0}\sum_{k=0}^{j+1}\log r_k \\
&\quad\le\quad \sum_j c_1\frac{m}{\sqrt{r_j}}2\sum_{i<j} c_2\sqrt{r_i}\beta_{i0}\ c'\log r_{j+1} \\
&\quad=\quad 2c_1c_2c'm\sum_j r_j^{-1/2}\log r_{j+1}\sum_{i<j}\sqrt{r_i}\ (4/3)^i\frac{\log r_{i+1}}{\log r_1} \text{ by definition of } \beta_{i0} \\
&\quad\le\quad \frac{2c_1c_2c'm}{\log r_1}\sum_j r_j^{-1/2}\log r_{j+1}\ c''\ r_{j-1}^{1/2}(4/3)^{j-1}\log r_j \text{ for a constant } c'' \\
&\quad\le\quad \frac{2c_1c_2c'c''m}{\log r_1}\sum_j r_j^{-1/2}\log r_{j+1}\ r_{j-1}^{1/2}r_{j-1}^{1/6}(4r_{j-1}^{1/6}) \text{ by (6.7) and Lemma 6.8.2} \\
&\quad\le\quad \frac{8c_1c_2c'c''m}{\log r_1}\sum_j r_j^{-1/2}\log r_{j+1}\ r_{j-1} \\
&\quad\le\quad \frac{8c_1c_2c'c''m}{\log r_1}\sum_j r_j^{-1/2}\log^2 r_{j+1} \\
&\quad\le\quad \frac{8c_1c_2c'c''m}{\log r_1} c
\end{aligned}$$

□

Theorem 6.8.5. *The shortest-path algorithm runs in $O(m)$ time.*

6.9 History

Frederickson [Frederickson, 1987] gave the first shortest-path algorithm for planar graphs that is faster than the one for general graphs. His algorithm runs in

$O(n\sqrt{\log n})$ time. His algorithm used an r-division (a concept he pioneered) to ensure that most priority-queue operations involved small queues.

Building on these ideas, Henzinger, Klein, Rao, and Subramanian [Henzinger et al., 1997] gave the linear-time algorithm presented here.