From Boston to San Francisco: A Survey of Shortest Paths Algorithms in Planar Graphs

Stuart Baker, Ömer Cerrahoğlu, Sebastian Claici

December 6, 2014

Abstract

1 Introduction

2 Background

We call an edge uv relaxed if $d(v) \leq d(u) + c(u, v)$. We call the assignment

$$d(v) \leftarrow \min\{d(v), d(u) + c(u, v)\}$$

the relaxation of vertex v. We know that the labels give a correct shortest path distances if the shortest-path conditions are satisfied:

- $d(s) = 0$,
- every label $d(v)$ is an upper bound on the $s - v$ distance,
- every edge is relaxed.

3 Single source shortest paths with nonnegative edge weights

3.1 Simple algorithm

For a planar graph with nonnegative edge weights, Dijkstra's algorithm runs in $O(n \log n)$ as $m \leq 3n - 6$. It is possible to improve this to $O(n)$. To get there, recall that and r-division of a planar graph is a partition of the graph into $\Theta(n/r)$ regions of size $O(r)$ with boundary size $O(\sqrt{r})$. An r-division of a planar graph can be computed in linear time.

A simple $O(n\sqrt{\log n \log\log n})$ emerges quite beautifully just from the r-division if we set $r = \frac{\log n}{\log\log n}$. The algorithm follows a divide-and-conquer approach in which each region is processed first, followed by a clean-up phase where the results are merged.

Algorithm 1 Shortest paths in each region R

```
for all Regions R do
    for all Boundary nodes v ∈ R do
        Compute SSSP from v in R
        Store (u, v) distances for any two boundary nodes u, v
    end for
end for
```

The first step is to compute the single-source shortest paths for each boundary node in each region R (algorithm 1). We can now replace each region R by a complete graph on R's boundary nodes with shortest paths distances between any two nodes. Call this auxiliary graph G'. The second phase of the algorithm is to compute the SSSP from s in G'. This gives the true shortest paths from s to all the boundary nodes. Finally, we must tidy up by finding the distances from s to the nodes inside each region (algorithm 2).

Algorithm 2 Clean up: shortest paths from s to inside of each region R

```
for all Regions R do
    for all Boundary nodes v ∈ R do
        Set d(v) = d_{G'}(s, v)
        Compute SSSP from v in R
    end for
end for
```

To analyze the algorithm, we will need a few pieces of information:

- Total number of boundary nodes is $O(\sqrt{r})O(n/r) = O(n/\sqrt{r})$.
- Number of nodes in G' is $O(n/r)O(\sqrt{r}) = O(n/\sqrt{r})$.
- Number of edges in G' is $O(n/r)O(r) = O(n)$.

Using $r = \frac{\log n}{\log \log n}$, the first phase is bounded above by

$$O\left(n\frac{\sqrt{\log\log n}}{\sqrt{\log n}}\log n\right) = O(n\sqrt{\log n \log\log n}),$$

the second phase is an SSSP in a size $O(n\frac{\sqrt{\log\log n}}{\sqrt{\log n}})$ graph, and thus also $O(n\sqrt{\log n \log\log n})$, while the tidying up is a series of SSSPs in each of the regions, and has the same bound as the first phase—$O(n\sqrt{\log n \log\log n})$. The total time bound ends up $O(n\sqrt{\log n \log\log n})$.

3.2 Recursion

The key idea that can improve the running time to linear is to go deeper inside the recursive world. Instead of just using an r-division of the graph, we recursively subdivide each region until we reach edges which are atomic regions.

The simple algorithm shown above is an improvement over Dijkstra's algorithm, but one can do better. In fact, we can achieve linear time by recursively subdividing the graph. Without loss of generality, assume the graph is directed, and that each node has at most two incoming and two outgoing edges. We call a region atomic if it contains only one edge uv. A nonatomic region will have as children subregions that are contained within it.

For each region R, we maintain a priority queue $Q(R)$ that stores the subregions of R if R is nonatomic, or the single arc uv is R is atomic. The algorithm ensures that for every region R, the minimum element of $Q(R)$ is the minimum label $d(v)$ over all edges vw in R that remain to be processed.

Algorithm 3 Process region

```
1: procedure PROCESS
2:     if R contains only uv then
3:         if d(v) > d(u) + c(u, v) then
4:             d(v) ← d(u) + c(u, v)
5:             for each outgoing edge vw of v, call UPDATE(R(vw), vw, d(v))
6:         end if
7:         Q(R).updateKey(uv, ∞)
8:     else
9:         repeat
10:            R' ← Q(R).getMin()
11:            PROCESS(()R')
12:            Q(R).updateKey(R', Q(R').minKey())
13:        until Q(R).minKey() is infinity or if repeated α_{h(R)} times
14:    end if
15: end procedure
```

Algorithm 4 Update region

```
1: procedure UPDATE(R, x, k)
2:     Q(R).updateKey(x, k)
3:     if updateKey reduced the value of Q(R).minKey() then
4:         UPDATE(parent(R), R, k)
5:     end if
6: end procedure
```

Algorithm 5 Faster SSSP

```
1: Find recursive subdivision R(G), R(P_i), ..., R(uv)
2: Allocate queue Q for each region
3: d(v) ← ∞, ∀v
4: d(s) ← 0
5: for all sv ∈ E(G) do
6:     UPDATE(R(sv), sv, 0)
7: end for
8: while Q(R(G)).minKey() < ∞ do
9:     PROCESS(R(G))
10: end while
```

The procedures are used in algorithm 3.2.

By playing around with the number of levels, number of nodes per region per level, and the α_i, we can achieve linear time. To give an intuition, we present here a $O(n \log\log n)$ algorithm and briefly comment on how to change it to get rid of the $\log\log n$ factor.

3.2.1 Correctness

Recall from Dijkstra's algorithm that three properties imply correctness:

1. Initialization: $d(s) = 0$.
2. Minimum length property: $d(v)$ is an upper bound on the s to v distance
3. Edges are relaxed: $d(v) \leq d(u) + c(u, v), \forall uv \in E$

The first property is true at the start of the algorithm, and remains true throughout. The following lemma establishes the second property:

Lemma 3.1. *For each node v, $d(v)$ is an upper bound on the distance from s to v throughout the algorithm.*

Proof. Initially, all labels except $d(s)$ are infinity. The labels only get changed in line 4 of PROCESS, and assuming inductively that the old labels $d(v)$ and $d(u)$ are upper bounds on

the distance to u and v, it follows that the new labels will also be upper bounds. □

Similarly, the following three lemmas establish the third property.

Lemma 3.2. *If an edge uv is inactive then it is relaxed.*

Proof. The lemma holds before the first call to PROCESS as every node but s has label infinity and outgoing edges from s are active. Edges are deactivated only in line 7 of PROCESS which occurs only after the edge has been relaxed.

Note that it is possible that an edge becomes unrelaxed after changes to the labels of its endpoints. This can occur for a call to UPDATE, but line 2 of UPDATE changes the key of the edge, making it active again. □

Lemma 3.3. *The key of an active edge uv is $d(u)$ (except during lines $3-6$ of* PROCESS*).*

Proof. Whenever a label $d(u)$ is assigned a value k, UPDATE$(R(uv), uv, k)$ is called for each outgoing edge uv, and the key of uv is updated to k. □

Lemma 3.4. *For any region R that is not an ancestor of the current region, the key associated with R in $Q(parent(R))$ is the minimum key of $Q(R)$.*

Proof. Whenever the minimum key of a queue $Q(R)$ is changed in line 2 of UPDATE, the recursive call in line 4 ensures that the key associated with R in the parent of R is also changed. □

From lemma 3.4, the following corollary follows:

Corollary 3.4.1. *For any region R that is not an ancestor of the current region,*

$$Q(R).minKey() = \min\{d(v) | uv \text{ is a pending edge contained in } R\}$$

When the algorithm terminates, the minimum key of the priority queue associated with the entire graph will be infinity, and by corollary 3.4.1 all edges will be relaxed.

4 Single source shortest paths with arbitrary edge weights

5 Multiple source shortest paths

6 Extensions to higher genus