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

Stuart Baker, Ömer Cerrahoğlu, Sebastian Claici

December 5, 2014

Abstract

1 Introduction

2 Background

3 Planar Graph Separators

3.1 Overview

For large planar graphs, a common approach to solving a specific problem is to take a divide-and-conquer approach. By dividing the problem into two or more smaller chunks and solving each subproblem, calculations that are difficult on a large scale can still be done. The core problem in divide-and-conquer problems is finding a way to divide the problem space into smaller spaces and recurse until you reach a subspace that is small enough to allow for effective computation. The results at the lowest level are then rolled back up through the recursion to give the final answer.

In order to tackle planar graphs, there are several important relations that we rely on. If C is some closed curve in the plane, by removing C you can divided the plane into exactly two connected regions: the inside region and the outside region. Additionally, Kuratowski's theorem states that a graph is planar if and only if it does not have any generalized subgraphs that are either a complete graph of five nodes or a complete bipartite graph of two sets of three nodes. From Kuratowski's theorem we know that we can shrink any edge of a planar

graph to a single vertex and preserve the planarity. Expanding this idea, we can shrink any subgraph of a planar graph to a single vertex and still have a planar graph.

3.2 Fundamental Cycle Separators

On method for dividing a planar graph into smaller set was developed by Lipton and Tarhan called the fundamental cycle separator. Lipton and Tarjan demonstrated that for any planar graph $G = (V, E)$ on $n = |V|$ vertices, and for any weight function $w : V \to \mathbb{R}^+$, it is possible to partition the nodes in the graph into three sections $A, B, C, \subseteq V$ with the following qualities.

- $w(A), w(B) \leq \alpha \cdot w(V)$ for some $\alpha \in (0, 1)$
- There are no edges between any node in A ($a \in A$) and any other node in B ($b \in B$), ($A \times B \cap E = \emptyset$)
- The size of the separator S is small, $|S| \leq f(n)$, specifically $\frac{2}{3}$

Fro a root vertex r, you can build a spanning tree of the graph G that has depth d. You can then define T^* as the dual tree of the triangulated version of G. From this tree, every non-tree edge e defines a fundamental cycle $C(e)$. Since the depth of T is at most d, we know that $|C(e)| \leq 2d + 1$. However, because the diameter of G may be large, we want to reduce it so that we can constrain $|S| \leq \sqrt{n}$. Central to developing this better partition is the ability to divide the planar graph into levels. Given some n-vertex planar graph G with nonnegative vertex costs, it is possible to partition the vertices of the graph based on their distance from a vertex v. One method for finding this partitioning is to run a breadth-first search from v. Given the partitioning, we can define $L(l)$ as the number of vertices on the level l which is the distance from v. The levels range from 0 to r where r is the maximum distance from v to any vertex in the graph. For the algorithm to work, an additional, empty, level must be added at $r + 1$.

The algorithm is as follows:

1. Find the most costly component in the graph and run a depth-first search from this graph. This calculates the level of each vertex in the graph in provides the level values ($L(l)$) for every l. For the maximum depth r in the level tree, add an additional level at $r + 1$ that contains no vertices. This can be performed in $\mathcal{O}(n)$.
2. Find the level i_0 that contains the median vertex. This is the level where $\sum_{i \leq i_0} |L_i(v)| \geq \frac{n}{2}$ and $\sum_{i \geq i_0} |L_i(v)| \geq \frac{n}{2}$. This can be performed in $\mathcal{O}(n)$.
3. Find the levels $i_- \leq i_0 \leq i_+$. This can be performed in $\mathcal{O}(n)$.
    - Start from the median vertex containing level i_0 and increase i_+ as well as decrease i_- until $|L_{i_-}|, |L_{i_+}| \leq \sqrt{n}$

- Because each section can only contain half of the vertices, we can use the counting argument to state that $|i_0 - i_-|, |i_+ - i_0| \leq \frac{\sqrt{n}}{2}$
- At this point we have the separator $|L_{i_-} \cup L_{i_+}| \leq 2\sqrt{n}$, which we can return if some grouping of $L_{<i_-}$, $L_{>i_+}$, and L_{i_-,i_+} is balanced.

4. From a condensed graph G'
   - Delete or contract all edges in $L_{\geq i_+}$
   - Contract all edges in $L_{\leq i_-}$ to form a super-vertex v that is connected to all vertices $u \in L_{i_-+1}$
5. From the condensed graph G', create a fundamental cycle separator
   - Build a breadth-first tree from G' from v. This tree will have a depth $|i_+ - i_-| \leq \sqrt{n}$
   - Triangulate the BFS tree.
   - Apply the fundamental cycle separator lemma presented above.
6. Using this separator, return A and B as some combination of int(C), ext(C), $L_{<i_-}$, and $L_{>i_+}$. S can be returned as a combination of L_{i_-}, L_{i_+}, and C.

3.3 Miller's Algorithm

3.4 Recursive Segmentation

The graph segmentation algorithms presented above can be used to recursively break a graph apart for a divide-and-conquer approach to solving problems. Specifically, the graph can be divided into $\Theta\left(\frac{n}{r}\right)$ regions, each of which have $\mathcal{O}(r)$ vertices and a total of $\mathcal{O}\left(\frac{n}{\sqrt{r}}\right)$ boundary vertices. Miller takes this definition a step further and defines a *suitable* r-division of a planar graph as the r-division that satisfies two characteristics:

1. Each boundary vertex is contained in at most three regions
2. Any region that is not connected consists of connected components, all of which share boundary vertices with exactly the same set of either one or two connected regions.

Starting with the initial graph G, all of the vertices are in the interior region. A separator algorithm can then be applied to the graph with all of the vertex weights set to $\frac{1}{n}$. This will produce three sets: A, B, and C. From these two sets, we can infer two regions that have the vertex sets $A_1 \subseteq A \cup C$ and $A_2 \subseteq B \cup C$. These two vertex sets will have sizes $\alpha n + \mathcal{O}\sqrt{n}$ and $(1-\alpha)n + \mathcal{O}\sqrt{n}$ with $\frac{1}{3} \leq \alpha \leq \frac{2}{3}$. To continue the process, the separator algorithm can be recursively applied to any region that has more than r vertices. The total runtime of this algorithm is $\mathcal{O}\left(n \log\left(\frac{n}{r}\right)\right)$.

However, to ensure that the two qualities that Miller enumerated hold, additional steps must be made. After applying the planar separator algorithm, there are three sets of vertices: A, B, and C. If we say that C' is the set of vertices in C not adjacent to any vertex in $A \cup B$, the we can define $C'' = C - C'$. Next, we need to identify the connected components $A_1, A_3, \ldots, A_q$ in $A \cup B \cup C'$. Using this information, we can remove any vertex v from C'' and insert it into A_i if that vertex is adjacent to a vertex in A_i, but not adjacent to a vertex in A_j for $i \neq j$. This will ensure that a boundary vertex will be in at most three subgraphs. However, this does not ensure that there are at most $\Theta\left(\frac{n}{r}\right)$ connected subgraphs. To ensure that there are no more than $\Theta\left(\frac{n}{r}\right)$ connected subgraphs, we can apply a greedy approach. Sweep through the set of connected regions, join together any two neighboring regions that each have less than $\frac{r}{2}$ vertices.

4 Single source shortest paths

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.

4.1 Nonnegative edge weights

4.1.1 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.

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

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
```

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 $O(n\sqrt{\log n \log\log n})$.

4.1.2 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

```
procedure PROCESS
    if R contains only uv then
        if d(v) > d(u) + c(u, v) then
            d(v) ← d(u) + c(u, v)
            for each outgoing edge vw of v, call UPDATE(R(vw), vw, d(v))
        end if
        Q(R).updateKey(uv, ∞)
    else
        repeat
            R' ← Q(R).getMin()
            PROCESS(()R')
            Q(R).updateKey(R', Q(R').getMinKey())
        until Q(R).getMinKey() is infinity or if repeated α_{h(R)} times
    end if
end procedure
```

Algorithm 4 Update region

```
procedure UPDATE(R, x, k)
    Q(R).updateKey(x, k)
    if updateKey reduced the value of Q(R).getMinKey() then
        UPDATE(()parent(R), R, k)
    end if
end procedure
```

The procedures are used in algorithm 4.1.2.

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.

Algorithm 5 Faster SSSP

Find recursive subdivision $R(G), R(P_i), \ldots, R(uv)$
Allocate queue Q for each region
$d(v) \leftarrow \infty, \forall v$
$d(s) \leftarrow 0$
for all $sv \in E(G)$ **do**
    UPDATE$(R(sv), sv, 0)$
end for
while $Q(R(G)).getMinKey() < \infty$ **do**
    PROCESS$(R(G))$
end while

4.2 Arbitrary edge weights

5 Multiple source shortest paths

6 Extensions to higher genus

References