Computational Geometry: Searching Planar Subdivisions

Don Sheehy

April 14, 2010

1 Searching Planar Subdivisions

Suppose you have a map, and you have the GPS coordinates of a point on this map. **How do you find out what country that points lies in?**

There are many contexts in which this problem arises. Consider the case where the map is represented on a display and the query point is given as $(mouseX, mouseY)$. Here you want to know what country was clicked.

This problem also comes up often as a subproblem of other geometric problems. In fact, we've seen this already in class. Recall that the running time of the random incremental Delaunay triangulation was dominated by the cost of finding the triangle containing each query point. The data structure we use in that case was called the history DAG. Today, we'll see a variant of the history DAG that works for a more general class of subdivisions known as polygonal decompositions.

Definition 1.1. *A* polygonal decomposition *is a planar cell complex in which all faces are polygons.*

2 History DAGS

Recall that a history DAG is an ordered collection of planar decompositions with edges between cells in level i and cells in level $i + 1$ if they overlap. The decomposition in the top layer of the DAG is very simple, it has only constant complexity and the subsequent levels increase in complexity as we go down. There are two important properties that we want for the history DAG to give us a nice search data structure.

1. The outdegree of the cells should be $O(1)$.

2. The depth of the DAG should be $O(\log n)$.

If we achieve both then we can process queries in $O(\log n)$ time.

The algorithm we will present today is known as Kirkpatrick's algorithm. It can be constructed in $O(n)$ time and space. It supports queries in $O(\log n)$ time.

Before we move on, we should just be extra clear why this differs from the history DAG from the random incremental Delaunay triangulation algorithm. There are two main reasons.

1. We don't know the queries in advance.
2. We care about worst-case performance rather than average case performance for queries.

3 The Algorithm

3.1 Restricting ourselves to Triangulations

We can divide the cells in the input map into triangles so that without loss of generality and only a constant factor increase in complexity, the problem is the same.[1] We will assume that we only have to deal with triangulations as input.

3.2 Working backwards

In the case of Delaunay triangulation, it was clear how to construct it one vertex at a time. We simply added a new vertex and flipped our way to Delaunay. For an arbitrary triangulation, it makes more sense to think of this process backwards.

That is, given a triangulation, remove a vertex and then fill in the gap left behind with more triangles. We will refer to the interior of the shape left after removing vertex v as $cavity(v)$.

Let δ_v denote the degree of v in the triangulation. The triangles filling in $cavity(v)$ will each be a new node in the history DAG. There will be at most $\delta_v - 2$ of them because that is the maximum number of triangles in a polygon with δ_v sides. Also, the degree of these new nodes in the history DAG will be at most δ_v.

So, in order to achieve our goals of keeping the history DAG small and the degree low, we need to remove vertices of low degree. Euler's formula guarantees that there are many low degree vertices. We'll see later how to use this bound explicitly.

3.3 Exploiting independence

Observation 3.1. *If u and v are non-adjacent then $cavity(u) \cap cavity(v) = \emptyset$.*

[1]How to do this is an interesting problem in its own right. There exists a complicated deterministic algorithm and at least one comprehensible randomized algorithm that run in linear time. We will assume this for now.

This observation means that the degree of a node in the DAG at level i depends only on the degree of a single vertex in level $i+1$. If we can find an independent set of low degree vertices, then we can remove them all at once and none of the DAG edges will interact. We can think about this as removing several independent vertices in parallel.

Planar graphs have large independent sets. The 4-color Theorem implies that there is an independent set of size at least $|V|/4$. This is a bit stronger than we need. Instead, we'll use a simple algorithm to find a constant fraction of the vertices that are independent and have low-degree.

3.4 Building the DAG

The construction of the DAG is quite simple. For each level, we will find an independent set of low degree vertices of size at least $\frac{|V|}{24}$. We remove these vertices, triangulate the cavities and add link from the new triangles to the old ones. We stop when the subdivision has $O(1)$ complexity. We just need to fill in the details about how to find such an independent set, why it even exists, and what kind of guarantees we have about the degree and the depth.

4 Analysis

Lemma 4.1. *In a plane triangulation, at least $\frac{|V|}{2}$ vertices have degree less than* 12.

Proof. Suppose for contradiction that $\frac{|V|}{2}$ vertices have degree more than 12. Then,

$$|E| = \frac{1}{2}\sum_{v\in V}\deg(v) \geq \frac{1}{2}\left(\frac{|V|}{2}12\right) = 3|V| > |E|.$$

Note that the last inequality follows from Euler's formula.

We can also note that this is just Markov's inequality applied to the degree distribution of planar graphs, where it is known that the average degree is less than 6. $\square$

Lemma 4.2. *In a plane triangulation, we can find $\frac{|V|}{24}$ independent vertices of degree less than* 12 *in linear time.*

Proof. We will use a simple greedy algorithm. At first, all vertices are unmarked. For each vertex, if it has not been marked and it has degree less than 12, add it to the set and mark its neighbors. It a vertex is marked or has high degree then do nothing. Clearly, this will result in an independent set because each time we add a vertex, we eliminate the possibility of adding any adjacent vertices. It is also clear from the construction that all of the vertices chosen have degree less than 12. By the preceding Lemma, there are $\frac{|V|}{2}$ vertices of sufficiently low degree and each time we add one, we mark at most 11 other vertices. Thus, we will be able to find at least $\frac{|V|}{24}$ such vertices. $\square$

Theorem 4.1. *A query takes $O(\log n)$ time.*

Proof. Let n_i be the number of vertices at level i of the DAG. We eventually get to level 1 where $n_1 = O(1)$. Each level grows by $\frac{24}{23}$ until the size reaches $O(n)$. Thus the total depth is $\log_{24/23} O(n) = O(\log n)$. □

Theorem 4.2. *The preprocessing time and space is $O(n)$.*

Proof. Given a planar subdivision we can decompose it into triangles in linear time. (Recall that this step is non-trivial) We have already shown that each level of the tree increases in size by $\frac{24}{23}$. So, it will suffice to prove that the time required to build a level of the tree is linear in the number of vertices at that layer. There are three steps in each in each level.

1. Finding an independent set of low degree vertices. [$O(n)$ by Lemma 4.2.]
2. Retriangulating the cavities. [$O(n)$ because each cavity has constant size and there are $O(n)$ of them.]
3. Adding the edges of the history DAG. [$O(n)$ because each new triangle has constant degree and there are $O(n)$ of them.]

It follows that the total time (and consequently the space) is $O(n)$.

□

The preceding Lemma is a little surprising. Think about the similar problem in one dimension, which might be implemented using a binary tree. We know that it is not generally possible to build a balanced binary search tree in linear time from an arbitrary set of vertices in $O(n)$ time. So, again we ask: **What is different?**

The difference is that the input has a lot of structure. We are given a subdivision, not just a collection of edges. We will explore the relationship between this problem and the one dimensional search problem soon (Section 5).

5 What does this look like in 1D

We can now drop back down to one dimension to see how this data structure relates to the quintessential one dimensional search data structure, the binary tree. The one dimensional input is just a decomposition of the line, specified by an ordered list of real numbers. A query is a real number.

In the 1D setting, the DAG is actually a tree. Actually it's just our normal vanilla binary search tree. This is a natural way to build a balanced binary search tree from an ordered list. We choose an independent set, say every other number and pull them up to the next level of the tree. We repeat $O(\log n)$ times and we get a binary tree. The total cost is $O(n)$.

Here's a question you might want to ponder a bit in your free time:

In binary search trees, we do flips to balance the tree. What does it mean to do a flip in the history DAG?

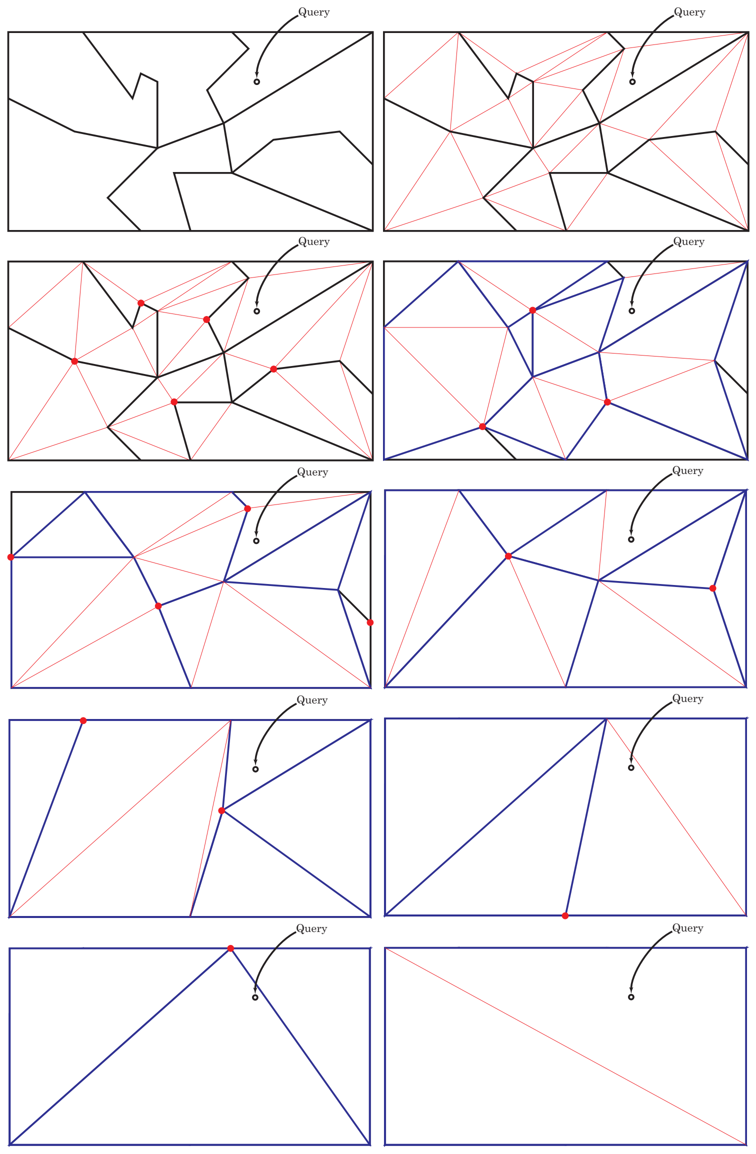

Figure 1: A sample run of the iterated decomposition to build the history DAG.