Lecture 3 (September 15, 2022)

Scribe: Joe Poremba

Announcements:

- You should send Will an email to chat about the project. You should start a conversation early, even if you don't have concrete ideas right now.
- Homework 1 will be coming out soon.
- Will's notes will be made available to the class (but we will still be doing "scribes").
- If you are interested in implementation of computational geometry, check out the book "Computational Geometry in C" by O'Rourke.

Today we cover a bit more about partitioning polygons into nice pieces:

1. The "best" triangulation of a simple polygon
2. The "best" convex partition of a simple polygon

Part 1: "Best" Triangulation

What do we mean by best? Some possibilites:

- As equilateral as possible. A good idea, but later...
- Minimize the total chord length. Like you have a really dull pair of scissors and want to cut as little as possible. This is what we are doing today.

Dynamic Programming Algorithm

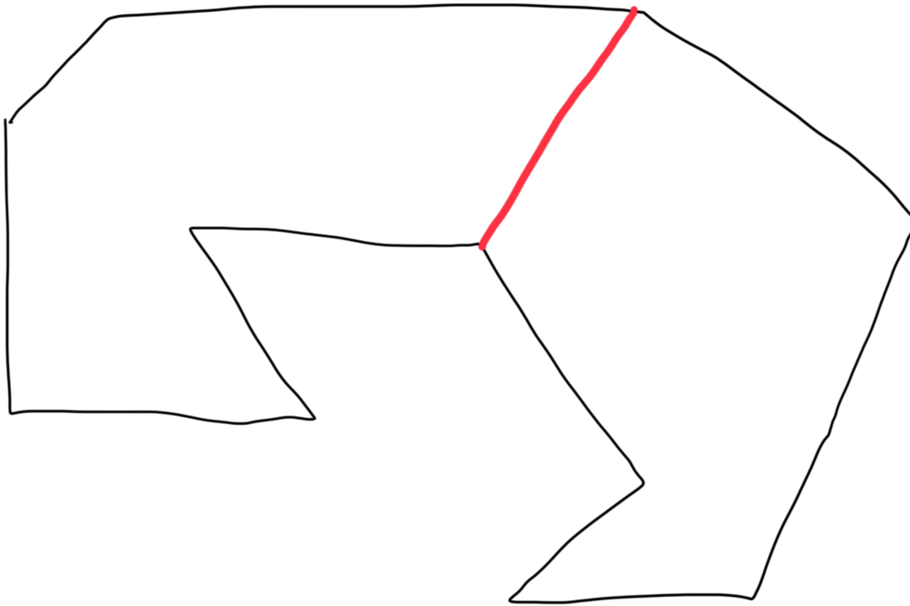
We will solve this by *dynamic programming*. Recall, the idea of dynamic programming is:

- Break a big problem into smaller subproblems
- Solve problems from smallest to largest
- Somewhat like recursion, but instead we process subproblems iteratively, storing results in a table and looking them up

The key to dynamic programming is how to break down the problem and re-compose it.

How are we going to break down the best triangulation problem? The first idea we probably have is to consider adding a chord to the polygon. For all the ways we might add a chord, we look at the subproblems resulting from the smaller polygons.

created with craft



However, we have to be very careful about how we do this.

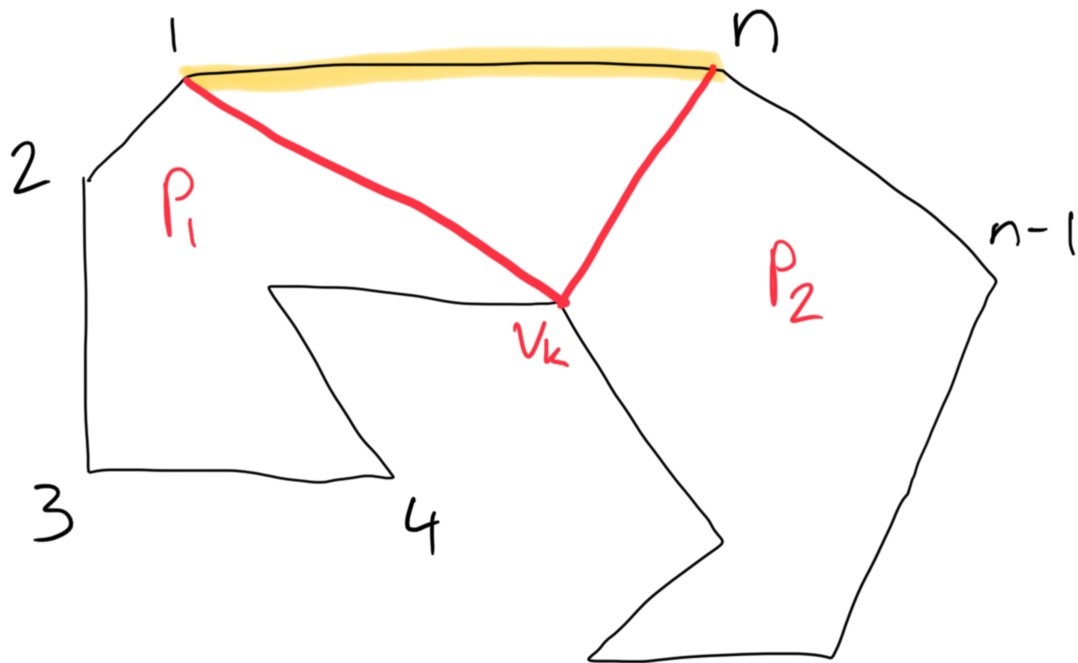
- Suppose for example, we just added an arbitrary chord to our polygon.
- There are roughly n^2 ways to do this. This is not great, but it is polynomial, so it is not immediately terrible.
- The issue comes when we consider the *total number* of subproblems. If we allow arbitrary chords to be placed, what is the total set of subproblems we have to iterate over?
- We might end up with subpolygons with arbitrary sets of vertices (well, not exactly, since some of them may be illegal, but we could get close to arbitrary vertex sets). This means there is an exponential amount of subproblems to iterate over in our dynamic programming table. This is not good.

This gives us a general lesson about dynamic programming: *you need to exercise some control over the subproblems you generate*.

So what do we do? We narrow our focus. The idea of our algorithm is as follows.

- List the vertices of the polygon (in order around the outside) as $v_1,...,v_n$
- Consider the edge v_1v_n. In the best triangulation, this edge must appear in some triangle, along with a third vertex v_k. We split into subproblems using canditates for v_k.

created with craft



Why is this better?

- The subpolygons we generate will have vertex sets that are contiguous subsequences of the form v_1,...,v_k and v_k,...,v_n.
- As we continue subdividing into smaller subproblems, always based on finding the third vertex of the triangle with side $v_i v_j$, where i, j are the minimum and maximum index respectively of the subpolygon, the vertex sets will still always be contiguous subsequences.
- So, we can uniquely define a subproblem based on its *start vertex* v_i and *end vertex* v_j. Therefore there are only $O(n^2)$ subproblems in the whole table!

Formally, our algorithm is as follows.

1. For each pair of vertices v_i, v_j define $|v_i v_j|_P$ to be the length of the chord from v_i to v_j, or $+\infty$ if the chord illegally crosses the polygon.
2. The Bellman equation (fancy-pants term for the recursive dynamic programming relationship) for the subproblems is defined as follows:

$$\mathrm{OPT}(v_i, v_j) = \min_{i<k<j} |v_i v_k|_P + |v_j v_k|_P + \mathrm{OPT}(v_i, v_k) + \mathrm{OPT}(v_k, v_j)$$

Runtime

- We can compute the lengths $|v_i v_j|_P$ in advance in $O(n)$ time for each pair (we have to do a collision check with each edge), for a total of $O(n^3)$ time for all pairs.
- There are $O(n^2)$ subproblems to iterate through.
- To solve $\mathrm{OPT}(v_i, v_j)$, we have to look up (in constant time) the solutions to k subproblems. $k \leq n$, so the runtime is $O(n)$ per subproblem, for a total of $O(n^3)$ over all subproblems (and this is tight since k will be at least $\Omega(n)$ for at least $\Omega(n^2)$ of the subproblems).
- So the total time is $O(n^3)$.

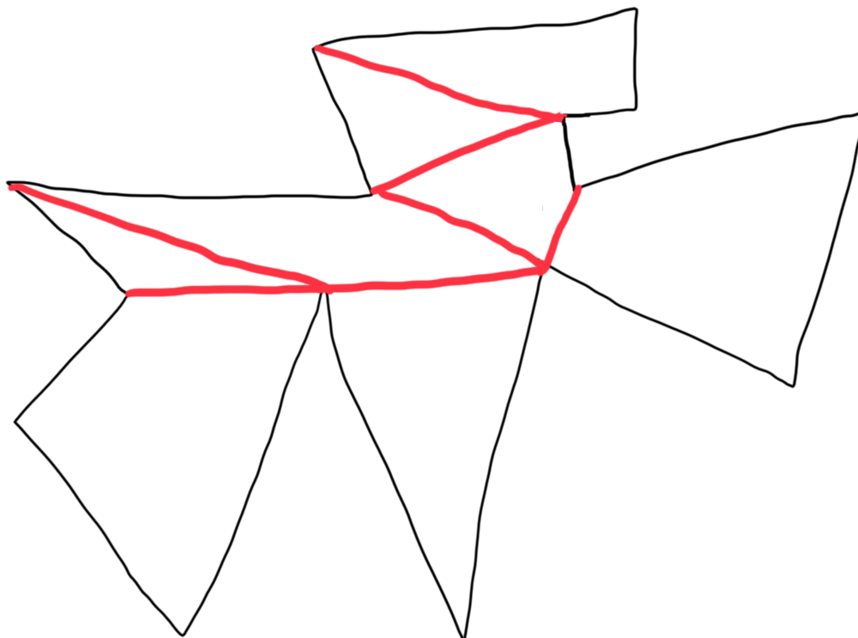
created with craft

A Brief Remark on Complexity

- The length calculation requires taking square roots, and the algorithm ultimately compares sums of square roots. To do this computation, we need real numbers (it is an open question whether we can use bounded bit precision).
- As such, we do not typically use a bit model for computational geometry. Alternatives:
  - *Real RAM model*. The registers hold real numbers
  - *Algebraic decision tree model*. A decision tree model is like we often use in sorting, where you have operations that can compare whether $a > b$. In the algebraic version, a and b can be arbitrary algebraic expressions, including square roots.
- We will talk about these issues more later in the course.

Part 2: "Best" Convex Partition

The goal for this problem is to chop a simple polygon into pieces. The pieces need not be triangles, but must be convex. "Best" here means simply to minimize the number of pieces.



Results:

- A $O(n^3 \log n)$ dynamic programming algorithm by M. Keil (1985). It is very tricky, but is a very good paper.
- We will talk about a 4-approximation that runs in time $O(n \log n)$ by Hertel and Mehlhorn (1983).

Approximation Algorithm

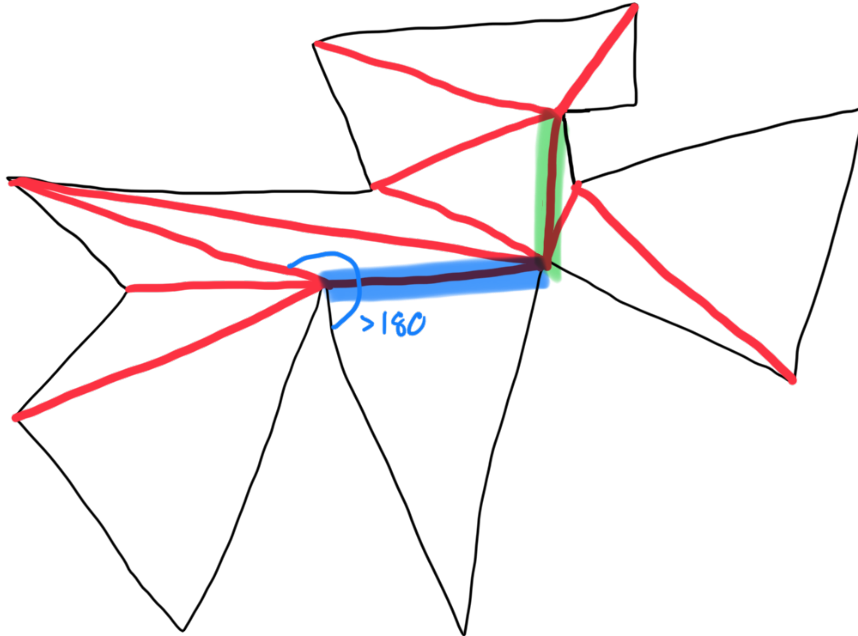
The approximation algorithm is fairly straightforward:

1. Triangulate P.

created with craft

2. One at a time (but in any order), remove any chord that leaves only convex pieces behind. Terminate when you can remove no more chords.

For example, in this picture, you could remove the green chord, but not the blue one. Removing the blue chord creates a *reflex angle* (greater than 180 degrees).



Analysis

We use the usual tactic for approximation algorithms:

1. Find an upper bound for our algorithm's solution.
2. Find a lower bound for arbitrary solutions (including the optimal one) that relates to some parameters of our upper bound.

For any partition Q, let $\mathrm{chords}(Q)$ be the number of chords, and let $\mathrm{pieces}(Q)$ be the number of convex pieces.

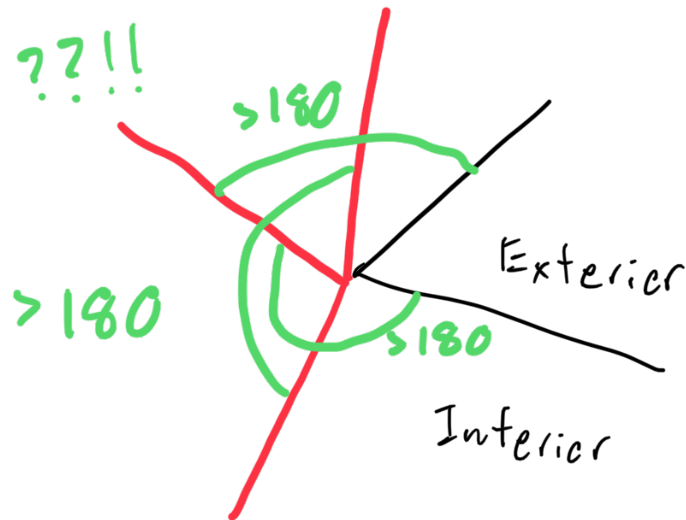
> Observation: For any partition, $\mathrm{chords}(Q) = \mathrm{pieces}(Q) - 1$.

Let r be the number of reflex vertices in the polygon P (before any partitioning).

> Claim: $\mathrm{chords}(\mathrm{ALG}) \leq 2r$. Hence, $\mathrm{pieces}(\mathrm{ALG}) \leq 2r + 1$ by the observation.

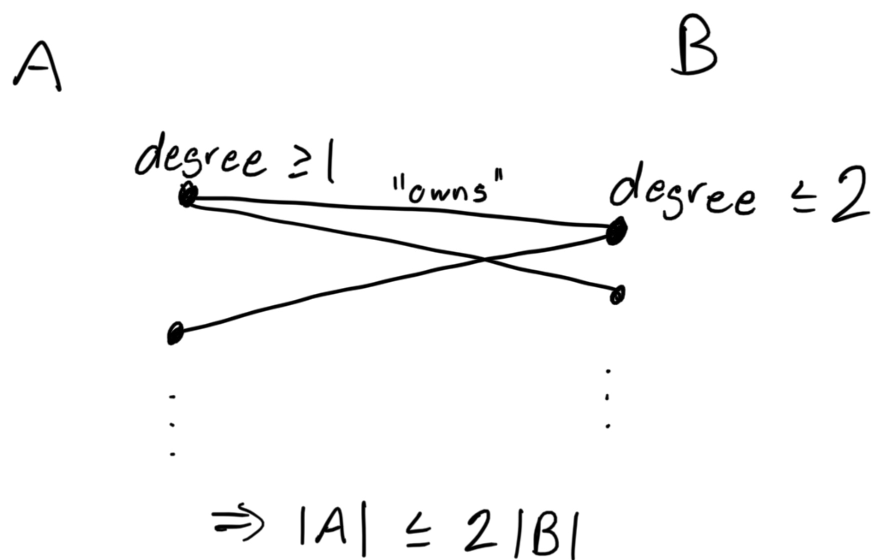
> Proof:
>
> - If a chord survives our algorithm, then if we try to remove it, it must leave a reflex angle at ≥ 1 of its endpoints. Say a chord "owns" any such vertex where this would happen.
> - Every reflex vertex in P can be owned by at most 2 chords (otherwise the total angle would be greater than 360 degrees).

created with craft



- So the number of chords is at most $2r$. $\square$

- Joe's note: sometimes these counting arguments trip me up, but I find it helps to draw a bipartite graph to make sense of it (one set being the chords, the other set being the reflex vertices, the edges being the "owns" relationship).



Claim: For any partition Q, $\text{chords}(Q) \geq \lceil r/2 \rceil$. Therefore, $\text{pieces}(Q) \geq \lceil r/2 \rceil + 1$.

Proof:

- For any partition, every reflex vertex must touch a chord.
- One chord can touch ≤ 2 reflex vertices.
- Therefore the number of chords is at least $\lceil r/2 \rceil$. $\square$

The two claims together (using $Q = \text{OPT}$) gives us the desired approximation ratio.

created with craft

created with craft