CS91T: Computational Geometry **Fall 2023**

Homework 4

David Yang and Nick Fettig

1. **If a simple n-gon $\mathcal{P}$ has been triangulated by a set $\mathcal{D}$ of $n-3$ diagonals, then the stabbing number of this triangulation is the maximum, over all segments s in $\mathcal{P}$'s interior, of the number of diagonals that s intersects.**

   Design an $O(n)$-time algorithm that, given any convex n-gon $\mathcal{P}$, lists the diagonals of a triangulation that has stabbing number $O(\log n)$. Prove correctness and running time.

   Assume we are given $\mathcal{P}$, consisting of the vertices of the n-gon in clockwise order, as well as the number of vertices, n.

Algorithm LowestStabbingNumber($\mathcal{P}, n$)

```
Set diagonals := empty list, will hold line segments
Set SN := 2                                   // SN = Skip Number
Set CV := 1                                   // CV = Current Vertex
for i ∈ {1 ... n − 3} do
    if (CV + SN) > n + 1 then
        Double SN
        Set CV := 1
    end if
    Append the segment P_CV P_(CV+SN)%n to diagonals
    Set CV := CV + SN
end for
return diagonals
```

   Algorithm Explanation: Our algorithm "cycles" through the polygon clockwise, always starting at the same arbitrary point, which we will denote $\mathcal{P}_1$. In each cycle, we connect vertices that are separated by double the corresponding distance[1] between vertices in the previous cycle. In our first cycle, we start by drawing diagonals between vertices that are distance 2 away from each other (so skipping every other point) and in the next cycle, we draw diagonals between vertices that are distance $2 \cdot 2 = 4$ away from each other. When we reach the end of a cycle, as in the next point added would result in a crossing, we reset the starting point to $\mathcal{P}_1$. We continue this process until we have exactly $n-3$ diagonals, which is the number of diagonals in any triangulation. The main idea is that every cycle creates a new polygon made by the diagonals of the previous cycle, and we perform this

[1] distance here does not correspond to Euclidean distance but rather the number of vertices between two given points, if we traverse clockwise along the boundary of $\mathcal{P}$

same process until the entire n-gon has been triangulated. Furthermore, note that this process does not create any crossings between diagonals, and thus yields a valid triangulation of $\mathcal{P}$. We claim that each cycle increases the stabbing number by at most 2, and so if we have $O(\log n)$ cycles, we will have created a triangulation with a stabbing number of $O(\log n)$.

Runtime Analysis: Lines 1-3 consist of pre-processing. We initialize the three variables used in the algorithm: *diagonals* will hold the triangulation's diagonals as line segments, which will eventually be returned. SN is the skip number, or the number of vertices skipped in each cycle (see above), and CV is the current vertex which just keeps track of where we are in each cycle. Each initialization takes $O(1)$-time.

Lines 4-11 represent the bulk of the algorithm. We iterate $n-3$ times as there will be exactly $n-3$ diagonals in a triangulation of a convex n-gon. In lines 5-8, we ensure that we are still in the same cycle by checking if the next point passes $\mathcal{P}_1$, as this would result in a crossing. If we need to end a cycle, we double the skipping number and reset CV, the current vertex, to 1. Otherwise, we know we are in the same cycle and we can add the diagonal from the current vertex to the next to *diagonals*. We also update the current vertex to be the end (tail) of the diagonal we just made. This check and each of these operations take $O(1)$-time.

We can summarize the run-time as follows:

$$\underbrace{O(1)}_{\text{Pre-processing}} + \underbrace{O(n-3)}_{\text{Loop}} \cdot \underbrace{(O(1)+O(1)+O(1))}_{\text{Loop operations}} = O(n)$$

as desired.

Proof of Correctness: We will begin by proving that this algorithm always yields a valid triangulation for any convex n-gon. The triangulation of any simple n-gon will always consist of $n-3$ diagonals.

Thus, since our algorithm creates $n-3$ diagonals (we create one diagonal for each of the $n-3$ iterations through the for loop) for any polygon, and avoids crosses, our algorithm will always create a valid triangulation of $\mathcal{P}$.

We will also show that the stabbing number increases by a constant, 2, for every cycle. Observe that any cycle creates a convex polygon, called P, with $\lceil c/2 \rceil$ vertices (where c represents the number of vertices of the previous cycle's polygon). Because P is contained by $\mathcal{P}$, the stabbing line (the line that produces the stabbing number) is allowed to pass through the outside of the boundary of P. In order for the stabbing line to have more than 2 intersections with P, there must exist a "hole" within the convex hull of P, which contradicts the convexity of P guaranteed by our construction. Therefore, every cycle adds at most 2 "stabs" to the stabbing number.

Finally, it remains to show that the stabbing number of $\mathcal{P}$ is $O(\log n)$. Clearly, since each cycles adds at most two to the stabbing number of $\mathcal{P}$, it suffices to show that the number of

cycles is also $O(\log n)$. Clearly, each cycle by construction adds half of the remaining diagonals needed to complete the triangulation. Therefore, since our triangulation yields $n-3$ diagonals, we require $O(\log(n-3)) = O(\log n)$ cycles.

Thus, our algorithm creates a triangulation with a stabbing number $O(\log n)$, as desired.

2. **A polygon with holes is a polygon with one or more polygons removed from its interior. More formally, it is any plane set that can be written in the form**

$$\mathcal{P} \setminus \left(\bigcup_{i=1}^{h} \text{int}(\mathcal{H}_i) \right),$$

where $\mathcal{P}$ is a simple polygon and $\mathcal{H}_1, \ldots, \mathcal{H}_i \subseteq \text{int}(\mathcal{P})$ are disjoint simple polygons. Here, $\setminus$ denotes set subtraction and int denotes the interior of a set. Use Euler's formula to prove that every polygon with h holes and n total vertices can be triangulated into $n + 2h - 2$ triangles.

Solution. Consider the triangulated polygon with n total vertices and h holes and let G be the graph formed by placing a vertex at each vertex of the triangulated polygon, an edge for each edge of the triangulated polygon (including its holes). By construction, G is planar since there are no edge crossings in a triangulation.

We know that v, the number of vertices of G is simply n since there are n total vertices in the triangulated polygon. Let t denote the number of triangles in the triangulated n-gon.

To count the number of edges of G, we consider the n edges consisting of the boundary of our polygon with holes (which include the boundaries of the holes) and the remaining edges separately. Since there are t triangles in the triangulation and each triangle has 3 edges, we have $3t$ total "edges" to consider. However, n of these "edges" are accounted for by the boundary of our polygon with holes, and for the $3t - n$ non-boundary "edges", we've overcounted by a factor of 2 – since each non-boundary edge appears in two triangles. Thus, G has

$$e = n + \frac{3t - n}{2} = \frac{3t + n}{2} \text{ edges}$$

Finally, note that the number of faces of G is $t + h + 1$, since each of the t triangles and h holes counts as a face, in addition to the face outside the polygon.

Since G is planar, we know by Euler's Formula that $v - e + f = 2$. Thus, we must have that

$$n - \frac{3t + n}{2} + (t + h + 1) = 2.$$

Simplifying the left hand side, we find that

$$\frac{n}{2} - \frac{t}{2} + h + 1 = 2.$$

Adding $\frac{t}{2} - 2$ to both sides and multiplying both sides by 2, we find that

$$t = 2\left(\frac{n}{2} + h - 1\right) = n + 2h - 2$$

and so we find that every polygon with h holes and n total vertices can be triangulated into $n + 2h - 2$ triangles, as desired. ■

3. **Using the dual graph of the triangulation, design an $O(n \log n)-$ time algorithm that splits any simple n-gon into two simple polygons of similar size, in the sense that each has at most $\lfloor \frac{2n}{3} \rfloor + 2$ vertices. Prove correctness and running time.**

Assume we are given a simple n-gon $\mathcal{P}$ that is to be split. As a further abstraction, assume we are given TRIANGULATE and DUALGRAPH, which give us a list of diagonals that triangulate $\mathcal{P}$ and the dual graph of the corresponding triangulation, respectively.

For the following algorithm, we assume that each vertex in the dual graph is an object that contains information about its "size" (number of descendants).

Algorithm SPLITPOLYGON($\mathcal{P}$, n)

1: Set *triangulation* := TRIANGULATE($\mathcal{P}$, n) *// returns list of triangulated diagonals*
2: Set *dualGraph* := DUALGRAPH(*triangulation*)
3: *root* := arbitrary vertex in *DualGraph*
4: Run GETSIZE(*root*)[2] and store size associated with each vertex in each vertex object
5: **while** $root.size > \lfloor \frac{2n}{3} \rfloor$ **do**
6: **if** $root.left.size > root.right.size$ **then**
7: Set $root := root.left$
8: **else**
9: Set $root := root.right$
10: **end if**
11: **end while**
12: **return** the corresponding diagonal in *triangulation* that joins *root* and MAX($root.left, root.right$) on basis of their size

Algorithm Explanation: We assume we are given the triangulation and dual graph (lines 1-2). The main idea of our algorithm is to traverse the dual graph, looking for a "balanced" node that "divides" the dual graph and consequently $\mathcal{P}$ into two polygons of similar size. To accomplish this, we pick an arbitrary node on the dual graph as the root and traverse the tree to find the sizes of each node about the chosen root (accomplished in GETSIZE, using a recursive DFS approach). We traverse the tree again, following the largest subtrees recursively until we have found a node of with size between $\lfloor \frac{n}{3} \rfloor$ and $\lfloor \frac{2n}{3} \rfloor$. To find the corresponding split on the n-gon, we simply look for the diagonal in *triangulation* that intersects the segment between this node and its neighbor in the dual graph.

Runtime Analysis: Lines 1-2 are assumed to be given. Finding each of these results will take $O(n \log n)$ time, regardless. Choosing an arbitrary route (line 3) will be $O(1)$ - the node in *DualGraph* we choose as the root does not matter. The GETSIZE function on line 4 will run in $O(n)$ as we will recursively run a DFS algorithm that goes through the n vertices, storing their associated sizes in the appropriate objects. The checks in lines 6 to 11 take constant time, and there are at most $O(n)$ checks. Consequently, the overall runtime is $O(n)$

[2]Here, GETSIZE is a recursive DFS algorithm that finds the "size" of each vertex, where the "size" is the number of children of that vertex

(if we assume the dual graph and triangulation are given to us), or $O(n \log n)$ if we need to determine the triangulation and dual graphs of $\mathcal{P}$.

Proof of Correctness: By construction, our algorithm will only conclude if there is some vertex in the dual graph with appropriate size; splitting $\mathcal{P}$ by the appropriate diagonal corresponding to this vertex will consequently yield two simple polygons of similar size.

It remains to show that this algorithm concludes; namely, that such a vertex exists, which implies that our algorithm will find that appropriate split. Suppose for the sake of contradiction that such a vertex does not exist; equivalently, every vertex in the dual graph has "size" greater than $\lceil \frac{2n}{3} \rceil$ or less than $\lfloor \frac{n}{3} \rfloor$. Assume without loss of generality that our dual graph is ordered such that the vertex with largest size is the original root vertex. Since the sizes of our vertices decrease as we traverse the dual graph tree, there must be some vertex v' with size greater than $\lceil \frac{2n}{3} \rceil$. Note that by construction, in the next iteration, our algorithm will consider the larger subtree, i.e. the child of v' with larger size. By the Pigeonhole Principle, this node must have degree $\geq \frac{\lceil \frac{2n}{3} \rceil}{2} \geq \lfloor \frac{n}{3} \rfloor$, as one of this node's children must have at least half its size. Thus, we find a vertex in our dual graph with size larger than $\lfloor \frac{n}{3} \rfloor$, contradicting our initial assumption. Consequently, we know that there must be some vertex in our dual graph with the appropriate size, and thus, our algorithm will find the appropriate split, as desired.

4. **Prove that $\lceil n/2 \rceil$ points are sufficient to solve the fortress problem, where given a simple n-gon $\mathcal{P}$, the goal is to find a set Z of points on $\mathcal{P}$'s boundary such that for all points q that are not in $\mathcal{P}$, there is some $z \in Z$ such that the segment $\overline{zq}$ does not intersect $\mathcal{P}$.**

Solution. First, note that covering the exterior of $\mathcal{P}$ can be broken down into covering $\text{CH}(\mathcal{P}) \setminus \mathcal{P}$ (the "pockets" of $\mathcal{P}$) and the exterior of $\text{CH}(\mathcal{P})$. Triangulate these pockets of $\mathcal{P}$.

Next, take any point q outside $\text{CH}(\mathcal{P})$ and connect each vertex of $\text{CH}(\mathcal{P})$ to q. Note that this new graph is still planar; the graph $\text{CH}(\mathcal{P})$ with the triangulated pockets is planar and connecting these segments adds 1 vertex, k edges – where k corresponds to the number of vertices on $\text{CH}(\mathcal{P})$ – and $k-1$ faces, thus preserving Euler's Formula $v - e + f = 2$ and the planarity of the graph. Furthermore, note that these segments create three-sided regions, since each new region is bounded by segments from v to consecutive points in $\text{CH}(\mathcal{P})$. To summarize, connecting these segments partitions the exterior of $\text{CH}(\mathcal{P})$ into three-sided regions, some of which may have curly sides.

To make these three-sided regions triangular, imagine "breaking" the boundary of $\text{CH}(\mathcal{P})$ at one of its vertices a. Create an additional copy of a (denote it as a') and "unroll" $\text{CH}(\mathcal{P}) \setminus \mathcal{P}$ so that the boundary lies on a single line (beginning with a and ending with a') and does not distort any of the individual pockets. This new graph preserves edge relations from the previous graph, has $n+2$ vertices corresponding to the n vertices in $\mathcal{P}$ and the additional points v and a', and has a boundary that is a simple polygon.

Imagine any 3-coloring of this graph. By construction, since each of the regions are triangular and each region's vertices correspond to vertices on $\mathcal{P}'s$ boundary, the vertices of any given color form a valid camera set Z of points on $\mathcal{P}$'s boundary satisfying the fortress problem. Thus, we will appeal to a statement about the 3-coloring of this graph to prove the $\lceil n/2 \rceil$ point sufficiency of the fortress problem.

Consider the color of v. Consider the color of the remaining two colors that has less points associated with it. By the Pigeonhole Principle, at most $\lfloor \frac{n+1}{2} \rfloor$ points are associated with this color, and so if we place our cameras at each vertex associated with this color, we have a collection of at most $\lceil \frac{n}{2} \rceil$ points that satisfies the conditions of the fortress problem. Thus, $\lceil \frac{n}{2} \rceil$ points are sufficient to solve the fortress problem, as desired. ■