1 Summary

2 Art Gallery problem

2.1 Definitions

1. **A simple polygon P** is a region enclosed by a *single* polygonal chain that does not intersect itself. By this definition, we cannot have holes in a simple polygon.

2. **Diagonal** in a polygon P is a line segment that connects two vertices of the polygon P. The line segments must lie on the interior of P, and the diagonals can therefore not "cross corners".

3. **Triangulation** is a decomposition of P into triangles by a maximum set of non-overlapping *diagonals*.

4. **Dual graph** of a triangulated polygon places a node in each triangle, and an edge between each node if the triangles share a diagonal.

5. **Comp shaped polygon** always $\lfloor n/3 \rfloor$ cameras - one for each peak.

6. **Monotone polygon**: A polygon is monotone to line l if there fore every line l' perpendicular to l, the intersection of l' and the polygon is connected. This means all intersections with l' should form a line segment, be empty or only contain a point. **y-monotone** simply means we have $l = y$, and thus the name y-monotone polygon.

7. **Below:** p is below q if $p_y < q_y$ or if $p_y = q_y \wedge p_x > q_x$

8. **Above:** p is below q if $p_y > q_y$ or if $p_y = q_y \wedge p_x < q_x$

9. **Start vertex** v: If the two neighbors are below and interior angle at v is less than π

10. **Split vertex** v**:** if the interior angle is greater than π

11. **End vertex** v: if the two neighbors lie above an the interior angle is below π

12. **Merge vertex** v: if the angle is greater than π

13. **Regular vertex**: one neighbor above the other below

14. **Left chain**: The left boundary of the polygon from the top most vertex to the bottom most vertex.

15. **Right chain**: Analogous to the left chain but on the boundary on the right side.

2.2 Solution to the problem

We can solve the art gallery problem by placing a camera in each of the triangles of in a triangulated polygon. Why? The camera can the points in the interior of the polygon for which there is an open line segment on the interior (not crossing corners) of the polygon connecting the points to the camera. Since there is an open line segment between all points in a triangle and the camera, then this solves the *art gallery problem.*

2.3 Does a triangulation always exist?

Theorem 3.1

Every simple polygon admits a triangulation, and any triangulation of a simple polygon with n vertices consists of exactly n−2 triangles.

Proof

The proof is on induction on n, the number of vertices in the polygon. The base case is for $n = 3$, and this trivially holds since this is a triangle, which is triangulated by $n - 2 = 1$ triangle.

Next, assume $n > 3$ and the induction hypothesis holds for all $m < n$. Since a triangulation consists of a maximum set of non-overlapping diagonals, let us first prove the existence of a diagonal. Let P be a polygon of n vertices. Let us pick the vertex v that is furthest to the left in P, and break ties by choosing the lowest vertex. Let the vertices u, w be the neighbors of v on the border of P (we know there are two neighbors since $n > 3$). If the line segment $\overline{uw}$ is on the interior of P then we have a diagonal (this is by definition a diagonal by the definition). However, if $\overline{uw}$ is not on the interior of the line segment, then let v' be the vertex the furthest from the left from the line segment and let $v' \neq v$. The line segment $\overline{vv'}$ cannot intersect another edge, since this contradicts choosing v' as the vertex furthest to the left of $\overline{uw}$. Since v' is the line segment furthest to the left the segment must also be inside the polygon, so $\overline{vv'}$ is a diagonal.

We have shown that a diagonal exists. For the first case, the polygon is simply split into the triangle Δvuw and the simple polygon of one vertex less (we essentially cut off v). Thereby, by induction the smaller polygon can now be triangulated since we assumed it holds for all $m < n$ and $n - 1 < n$. For the second case, the diagonal splits the simple polygon into two simple polygons P_1 and P_2 of size m_1 and $n - m_1 + 2$ respectively. We get that P_2 has size $n - m_1 + 2$ since, P originally had n vertices and we cut away $P_1$with m vertices, but the diagonal still shares 2 vertices. Since we have that $3 \leq m_1$ (otherwise it would not be a triangulation) then $n - m + 2 \leq n - 1 < n$ and so we can again use induction to show each subpolygon can be triangulated.

In either case however, the diagonal we choose, splits the two polygons into two smaller simple polygons P_1 and P_2 of size m_1 and m_2 respectively. All vertices of P are in either of the two vertices. But the two polygons also share two vertices at the diagonal, and these vertices are counted twice. Hence we

get $m_1 + m_2 = n + 2$ and we have proved that each polygon have $m_i < n$. Each simple polygon can by the induction hypothesis be triangulated by $m_i - 2$ vertices. Using this we get

$$m_1 + m_2 - 4 = n + 2 - 4$$
$$(m_1 - 2) + (m_2 - 2) = n - 2$$

2.4 3-coloring

A three coloring always exists. Can be seen by looking at the dual graph and that is in fact a tree. This shows the art gallery theorem, since we can place all cameras at a particular color. There are n vertices and 3 colors.

Theorem 3.2 (Art Gallery Theorem) For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are occasionally necessary and always sufficient to have every point in the polygon visible from at least one of the cameras.

2.5 Two-ears theorem and algorithm

Based on the proof that a triangulation always exists for a simple polygon, we can just find diagonals, solve the problem recursively and cut of triangles. Takes time $O(n^2)$

3 Y-monotone partitioning

Ideas

We can use top down plane sweep to split a non y-monotone polygon into pieces that are y monotone. Each iteration goes one vertex down. Then we will triangulate each y-monotone piece. The question is how should we determine at each iteration whether the current part of the polygon is y-monotone or we break the invariant? We remove split-and merge vertices as this theorem shows

> **Lemma 3.4:** *A polygon is y-monotone if it has no split vertices or merge vertices.*

- The theorem implies we have y-monotone polygons after removing split and merge vertices
- We create the y-monotone pieces by creating up-diagonals from split vertices and downward diagonals of merge vertices
  - Each diagonal creates a new polygon
  - Once all are removed we have y-monotone polygons

Figure 1: Split vertex

Split vertices

We let the vertices $v_1, v_2, ..., v_n$ be the vertices of the polygons taken by a counterclockwise enumeration along the border. We define an edge to be $e_i = \overline{v_i v_{i+1}}$. The algorithm is a sweeping algorithm, using a top-down sweep line halting for certain events at vertices. We store the vertices in a priority Q stored with priority by y coordinate and breaking ties with the leftmost point on the x-axis. We can now get event points is $O(\lg n)$ time, or presort and get them in $O(n \cdot \lg n)$ time.

Based on the above ideas, we want to add diagonals from a **split vertex** to get y-monotone pieces. Let us say the sweep-line is at the vertex v_i and the edge to the left of v_i is called e_j(note the edges intersects the sweep line) and the edge immediately to the right is called e_k. Now we can connect v_i to the lowest vertex that is between the edges e_j and e_k

Such a vertex might not exist, in which case we can connect v_i to the upper endpoint of e_j or e_k. We define these 3 vertices as the $helper(e_j)$. It is formally defined as the lowest vertex above the sweep-line such that the horizontal segment taken by connecting $helper(e_j)$ and $e_j$is within P. In other words, the helper of an edge is the lowest vertex above the sweep line such that the horizontal line segment from the edge to the vertex is within P.**In short:** *split vertices are handled by connecting v_i to the helper of the edge to the left.*

Merge Vertex

We handle these differently from split vertices. We can only work with vertices above the sweep line, but merge vertexes are handled by connecting merge vertex v_i to the highest vertex below the sweep line. Let e_k be the edge intersecting the sweep line to the left of v_i and let e_j be the right edge. *An observation at this point is that v_i will become the helper of e_j, since it is the lowest vertex with line segment connecting e_j to v_i being inside the polygon.* The goal is to connect v_i to the highest vertex below the sweep line between edges e_j and e_k(this is the opposite of what we did for split vertices). Since we cannot access vertices below the sweep line, we simply save v_i for later. When we reach a new vertex v_m that becomes the helper for e_j, then we make a diagonal from v_m to v_i. On thing to note is that when the new helper v_m is itself a helper, we add the diagonal anyhow, and we get rid of both a split and merge vertex with one diagonal. Not also that there might not appear any new helpers below v_i- this is fixed be just connecting v_i to the lower endpoint of edge e_j .

Data structure

For both types of problematic vertices, we only look at the left edge, since we create diagonals by connecting vertices with their helper of the edge to the left.

With a dynamic binary search tree, T, we can save the edges of intersection the sweep line. By doing this, we can distinguish between left and right edge by left and right leaves. Furthermore, since the algorithm only considers edges to the left of split- and merge edges, we can get by with only storing edges that have the inner of the polygon to the right (these are potential left edges). Each edge also stores its helper. We call the tree the *status* since it stores the status of the sweeping algorithm. In short we store a tree as follows

1. Store edges intersecting the sweep line in the leaves of a binary search tree T
2. Use left and right children to distinguish left- and right edges
3. Only store edges that have the inner of the polygon to the right. These are the edges that can be left edges
4. Store the helper together with the edge
5. Update the tree, called *status*, when moving the sweep line
   (a) Update the helper - we use this for merge vertices

We need to store the y-monotone pieces and the diagonals that splits the polygon into monotone pieces. To store polygon pieces and diagonals found we use a doubly-linked list of edges? When finding diagonals at split-and merge edges, we add this to the doubly linked list.

Start vertex algorithm

For start vertex $v_i$we do as follows

1. Add e_i to T
2. Set $helper(e_i) = v_i$(this is stored together with e_i)

End vertex algorithm

For an end vertex v_i

1. Check if $helper(e_{i-1})$ is a merge vertex (this is the edge coming into v_i)
   (a) Insert diagonal connecting v_i to $helper(e_{i-1})$ in D (this is the case there are no lower vertices below the sweep line)
2. Delete $e_{i-1}$from T

Split vertex algorithm

For split vertex v_i

1. Search T for edge e_j directly to left of v_i(directly to left is defined earlier)
2. Insert diagonal from v_i to $helper(e_j)$ into D (we now know $v_i$is the lowest vertex that has line segment inside P intersecting e_j)
3. $helper(e_j) = v_i$
4. insert e_i into T and set $helper(e_i)$ to v_i. ($e_i$is the edge going down to the right from v_i. So in this case we set the helper of the edge to its upper end-point)

Merge vertex algorithm

1. If the helper to edgee_{i-1}($helper(e_{i-1})$) is a merge vertex (the edge e_{i-1}will be the edge going from the right going into v_i. Now we check if the helper of this edge is a merge vertex and eliminate two merge vertices)
   (a) Insert diagonal between $helper(e_{i-1})$ and $v_i$in D
2. Delete e_{i-1} from T
3. Search T to find edge e_j directly to the left of v_i. The definition of directly to the left is the one described earlier, but in short it is the first edge hit by the sweep line that has a line segment connected to v_i inside the polygon
4. if the helper of e_j is a merge vertex (this is the case where we replace helper of an edge, and we have to check if the old helper is a merge vertex)
   (a) Insert diagonal connecting v_i to $helper(e_j)$ into D
5. $helper(e_j) = v_i$

Alternative is to not handle merge vertices at all. Then, each subpolygon created is flipped around so merge vertices become split vertices, and we run the algorithm again. Alternatively, sweep from upwards.

Regular vertex algorithm

1. When the interior of P lies to the right of v_i
   (a) If helper e_{i-1}(edge coming into v_i) is a merge vertex
      i. Insert diagonal between v_i and $helper(e_{i-1})$
   (b) Delete $e_{i-1}$from T
   (c) Insert $e_i$in T and set $helper(e_i) = v_i$(the helper of the edge is now the upper end-point)

2. Else find edge $e_j$directly to the left(using correct definition of what this means) of v_i
   (a) if the helper $helper(e_j)$ is a merge vertex
      i. Insert diagonal between $v_i$and $helper(e_j)$ into D
   (b) Now we also update the helper, $helper(e_J) = v_i$, since $v_i$is the lowest vertex that has edge $e_j$directly to the left

Degenerate cases

Proof of correctness for each algorithm

Running time

We initialize the priority queue Q by y-coordinate and breaking ties with lowest x-coordinate. The priority queue can be a max heap, so it takes linear time to build. At each iteration of the algorithm we query for the next event, v_i. Each of these queries take $O(\log n)$time. We insert at most two diagonals (in case of merge vertices) into D. Each insert takes constant time $O(1)$, and otherwise do constant time operations, so the time is $O(n \log n)$.

For storage, we store every vertex in Q. Every edge is stored at most once in T. We have more vertices than edges and so we get linear storage $O(n)$.

4 Triangulating a monotone polygon

The algorithm handles the vertices in decreasing order of y-coordinates. We can assume that the polygon is strictly y-monotone such that there are no horizontal edges, and hence no ties in y-coordinates and we precede strictly top down. The algorithm uses a stack S whose purpose is shown below

1. Uses a stack S for vertices encountered
2. Add ass many diagonals from $v_j$to the vertices on the stack
3. S will contain vertices of P that may still need more diagonals
4. The diagonals splits off triangles from P
5. S will contain vertices that have not been split of in a triangle
6. The vertices still in S are on the boundary of the part of P that still needs triangulation
7. S is ordered so that the last encountered, and therefore the lowest visited vertex, is on the top of the stack.
8. *The part of P that lie above the last seen vertex and still needs to be triangulated form an upside down funnel.*

   (a) The funnel has two boundaries
      i. One of them contains just a part of one of the edges of P
      ii. The other con boundary a chain of reflex vertices (vertices with interior angle above 180°.

9. Only the highest vertex (bottom of the stack) is convex - degree above 180 (π radians). This is an invariant of the algorithm that holds after visiting at each vertex.

Now we will look at how the algorithm adds diagonals.

Algorithm

1. The left and right chain of the polygon are sorted into a sequence by decreasing y-coordinate (highest y-coordinate first). Ties are broken by lowest x-coordinate. The sorted sequence is denoted $u_1, ..., u_n$

2. Initialize the empty stack S and push $u_1$and $u_2$onto it (the vertices with the two highest coordinates).

3. for $j = 3...n - 1$ (we skip the lowest vertex in the loop)

   (a) If $u_j$and and the top of the stack are on different chains (left or right chains).
      i. Pop all vertices from S
      ii. Insert a diagonal from $u_j$to each popped vertex except the last one
      iii. Push $u_{j-1}$and u_j to the stack (always have the last two seen vertices on the stack)

   (b) else
      i. Pop one vertex from S (ensures we pop at least one vertex, and there will be at least two on the stack)
      ii. Keep popping vertices from S while the diagonal from $u_j$to the vertex a the top of the stack is inside P, and insert the diagonal into D(of course only the diagonals within P)
      iii. Push the last vertex popped onto S (in case step 2 never popped anything we push back the original vertex)
      iv. Push $u_j$onto S

4. Add diagonals from $u_n$to all vertices on the stack except the first and last vertex (doing so would cross diagonals)

Next some details and notes for the different steps of the algorithm will be described.

1. S contains all the vertices for which we might add more diagonals to. We see this from the algorithm in that we only completely remove vertices in step 3.a.i when we add diagonals to these vertices

2. S contains the vertices we encounter. We see in both branches at step 3.a and 3.b we add the current vertex $u_j$to the stack.

3. The part of P that still needs to be triangulated above $v_j$forms an "upside down funnel" (see image. It is a bit hard to imagine at first, but at some point it makes perfect sense, since we shoot diagonals upwards without crossing a boundary). The funnel consists of two chains

   (a) One chain will consist of reflex vertices. These have degree of at least 180°. It makes sense that these will be part of the polygon above v_j that have not been triangulated yet, since their degree makes it impossible to connect diagonals without breaking the polygon barrier. This is essentially the case 3.b, and line 3.b.ii checks wether we break the polygon barrier by shooting upwards diagonals. However, the diagonals we are able to connect are consecutive (say there are 5 reflex vertices, the first 3 might be far enough away we do not break the polygon barrier, but afterwards the angle to the remaining two i too great to connect diagonals without breaking the barrier). This is reflected in the algorithm that keeps popping vertices in step 3.b.ii. Furthermore, line 3.b.i pops one vertex, and we cannot do anything with this reflex vertex since it is already connected by an edge to v_j. How we check that shooting diagonals between reflex vertices can be done as follows. We want to make a diagonal from $v_j$to v_k(which was popped from the stack). We look at v_j, $v_k$that was just popped from the stack and the previously popped vertex (this can be the initial popped vertex that in line 3.b.i)

   (b) The other chain is just part of a single edge (we might not have reached the lower end point yet since it can be lower than v_j), let us call it e. However, we can say that the lower end point of e bounds the height of the "funnel", since this is from where we eventually shoot up diagonals to the reflex edges in step 3.a of the algorithm. We can also shoot diagonals from the lower end point of e to all vertices of the funnel (which will be all in the stack) except the last one, since this is the upper end point of e and we see this in line 3.a.ii. We also see all these vertices are now popped (including the upper end point of e).

4. The invariant of the funnel is kept after 3.a or 3.b has been executed.

   (a) For case 3.a the invariant is kept as follows: The untriangulated part of P above $v_j$will be bound by the diagonal from $v_j$to the previously on top of the stack vertex (this makes a diagonal that makes it impossible to shoot up more diagonals farther than the previously on

top of the stack vertex), and the edge downward from the previously on top of stack vertex. This is like a funnel again with the pointy end at v_j

(b) We need to argue it for case 3.b as well: Here we try to connect as many reflex vertices as possible and push back the last vertex. This last vertex either had a diagonal added if we do anything in 3.b.ii or when we could not add any diagonals it must have been the neighbor (only do 3.b.i and nothing in 3.b.ii). Then we push $v_j$to the stack - this is still a funnel with a part of the edge and a chain of reflex vertices (this might only be v_j) so we must again have a funnel.