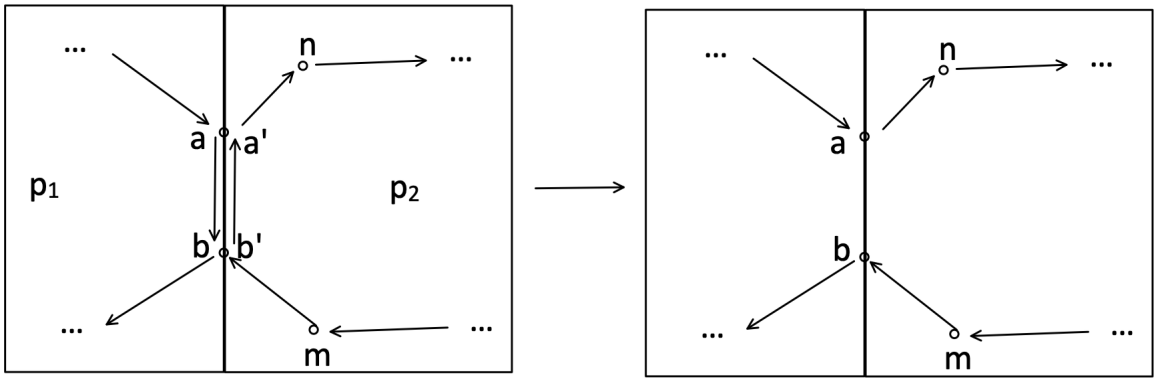
COS 451: Computational Geometry **Professor Chazelle**

Final: Generation of Simple Polygons

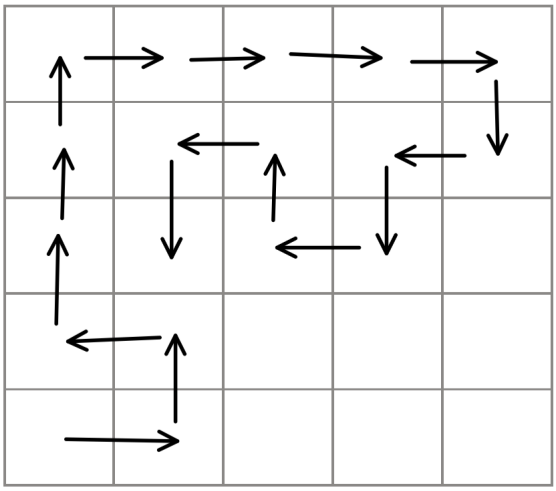
Name: David Weisberg, *NetID:* dw26 Due: 12/23/22

Design

The general idea here is to divide a $[0,1] \times [0,1]$ box into a lattice of regions, where in each region there is a random set of points with a simple polygon in each cell. Then the algorithm randomly traverse the cells, only going up, down, right, or left, and stitches together the mini simple polygons. This algorithm creates simple polygons that are complicated since it is irregular, creates random arrangements within each cell, has the capacity to be winding due to the random cell traversal, and is balanced, in the sense that there is no part of the polygon that is locally convoluted leading to many very small edges, so as to require a zoom.
Specifically, to generate a simple polygon within each cell, the algorithm starts off by choosing a random set of points. Then it computes an upper and lower hull by first lexicographically sorting the points, by x coordinates, and then by y coordinates. Then going in increasing order by x, and then y, linking the points for which their y coordinate is above the line connecting the right and left extremes of the polygon, we get the upper hull. For the points passed over because they are below the diagonal connecting the left and right extremes, those are put into another array and linked in the same way to make the lower hull. Linking together the lower and upper hull, we get a simple polygon within the cell.
To stitch a polygon within one cell to another, during the process of choosing the set of points for the cell, the algorithm also designates some points, in a deterministic way, to be the points in common between the 2 polygons of the 2 cells. For example, if the two cells to be stitched are oriented left-right, then on their boundary, we choose two points that will be part of both simple polygons. Then, the algorithm loads each point in both polygons into a node in a linked list that is connected to the next point in one polygon. To then stitch the two linked lists, in both polygons, the links between the common nodes in the polygon are broken, and then the common node of one polygon is linked to the next node of the other polygon. Concretely, let's call the set of nodes linked together that corresponds to the vertices of the left and right polygon as p_1 and p_2 respectively. Both polygons are oriented clockwise, and thus direct their links in a clockwise direction. Let's call the common nodes $a, b \in p_1$ and $a', b' \in p_2$. For $a = a' > b = b'$, in p_1, they are linked $a \to b$, and in p_2, they are linked $b' \to a'$, because of the clockwise convention. Let's also say the nodes part of p_2 that are after a' and before b' as m and n respectively. Thus, there are the links $a \to b$ in p_1, and $m \to b' \to a' \to n$ in p_2. In order to stitch the polygons together, we break the link $a \to b$, delete the nodes a' and b', and connect $a \to n$ and $m \to b$. This stitching process is depicted in the diagram below.



This is achieved analogously in the up-down direction as well. As described before, we link some random path between the cells, and so this process is used iteratively to link all the consecutive polygons in the consecutive cells. Below is an example path in a 5×5 lattice, where each cell in the path contains a simple polygon that will be stitched to its previous and its successor. Since the path taken is random, it is possible for it to get stuck, and thus not all cells will contribute to the construction of the simple polygon. The start and end cells, will require only one stitching, whereas the cells in between will require two stitchings.



Correctness

To guarantee that this is a simple polygon, it is necessary make sure that every transformation and construction in the process preserves the invariant that it is a simple polygon, maintaining that the there are no holes, that it is a cyclic path, and that it does not cross over itself.
Each mini simple polygon is constructed in the usual way of making an upper and lower hull and merging them, as described in the design. The merge of simple polygons, as described in the design, does not introduce any new holes, since the matrix traversal to form the path of cells never merges back into itself. Thus, all merges are just the elimination of a common edge, as described in the design, yielding a whole simple polygon. It also continues to be a cyclic path, as the design explains, it maintains the cyclic property by linking both sides of both polygons in the correct orientation and order to make sure it's not crossing over, after the elimination of the common edge.
The number of points is also correct, as the number of points per cell are allocated appropriately equivalently, modulo a remainder if the number of points input isn't a perfect square. The construction of the simple polygon also makes sure to allocate 4 or 2 of the merging points as part of the total points per cell. It is also important to take into consideration that 2 points will be deleted after a merge from one of the polygons. All of these details are just a matter of implementation to guarantee the total input points is preserved.

Runtime

This algorithm will run in linear time, $O(N)$, where N is the number of points in the polygon.
The minipolygons in each cell take constant time, as the algorithm designates a constant number of points per cell, so even though the sort is logarithmic, its on a constant, so in total is a constant. Thus, since the number of cells/polygons is $O(N)$, constructing all of the mini polygons is $O(N)$. Merging two cells take constant time, and so too there are $O(N)$ merges, and so this too is linear. The path traversal takes linear time, as the algorithm caps the number of steps taken at $O(N)$, the total number of cells in the matrix, which in any case is almost never reached due to reaching dead ends. Conversions between list and array representations, as well a constant number of list traversals all takes in total $O(N)$. Thus, the entire algorithm is also $O(N)$.

Graphical Evidence

40 points