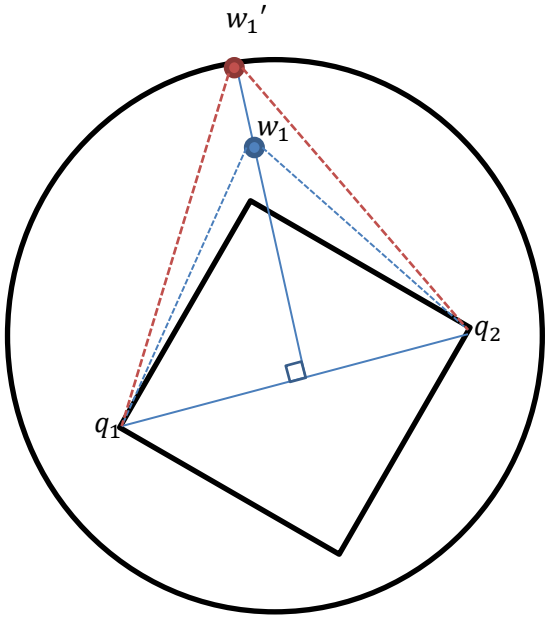
CS623 Winter 2012 \ Extra Credit (Midterm)

Ariel Stolerman

Assignment 1, question 5)

Let P be an n-sided convex polygon contained within a closed circular disk C. Following is a linear time algorithm that either finds two points $w_1, w_2 \in C - P$ such that for each point q on the boundary of P, q is visible to at least one of w_1, w_2, or returns that no such pair exists.

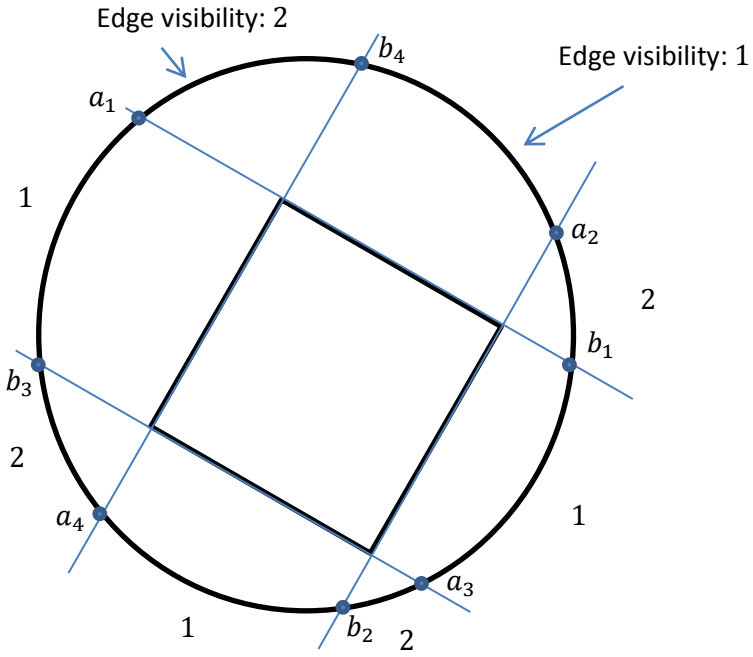
First, we show that if such points exist, they may be assumed to lie on the boundary of C. Assume there exist such two points w_1, w_2. If they are on the boundary of C, we are done. If one of them, say w_1, is not on the boundary of C, then let q_1 be the left-most point on the boundary of P that is visible from w_1 and q_2 – the right-most. Then we know that the boundary of P between q_1 and q_2 is visible from w_1. We can then take the point w_1' which is the intersection of the perpendicular from w_1 to the line q_1q_2 and the boundary of C instead of w_1, as any q on the boundary of P that is visible from w_1 is visible from w_1':



Assume that q is visible from w_1 but not from w_1', then the line $w_1'q$ intersects with P, meaning that at some point it intersects with the boundary of P (between q_1, q_2). But since the triangle $q_1w_1q_2$ is fully contained in $q_1w_1'q_2$, this must mean that either the boundary of P intersects with w_1q or that w_1q is fully contained in P – in both cases, a contradiction to the visibility of q from w_1. Therefore all points visible from w_1 are visible from w_1' which is on the boundary of C, so we can assume that if a pair w_1, w_2 as described above exists, then there must exist some pair w_1', w_2' on the boundary of C with the same visibility.

Now we show how to subdivide the boundary of C in $O(n)$ time into circular arc intervals such that all the point within each interval "see" the same edges of P:

1. "Stretch" all the edges of P to intersect with the boundary of C in clockwise direction, denoting the left points a_i and the right points b_i, for all $i = 1, \dots n$ (such that the direction from a_i to b_i on the boundary of C is clockwise).
2. Flat the boundary of C on a line, such that a_1 is the 0-point, and since it's originally a circle circumference, it's a clock arithmetic such that after the last point a_1 appears again. Since we have n edges in P, we have $2n$ intersection points (that may overlap), thus a division of the boundary of C into at most $2n$ arcs.
3. Note that each point on an arc that is bounded between a_i and b_i (clockwise direction between a_i and b_i) can "see" all points on the corresponding edge of P, and every intersection of k such arcs derives an arc that "sees" all points on all the corresponding k edges of P.



We can now mark each arc with the total number of edges that can be seen from any point on it: starting with a_1 and initial value 0, any arc that goes from a_i to $a_{i+1(mod\ n)}$ or to some b_j is marked with $value{+}{=}\ 1$; any arc that goes from b_i to $b_{i+1(mod\ n)}$ or to some a_j is marked with $value{-}{=}\ 1$. We set the initial value as follows: from a_1 proceed clockwise until the first b_i encountered. Then go counterclockwise from a_1 and count the number of a-values encountered up until a_i, included. This number is to be set as the initial value. If some vertex of P is on the boundary of C, this point will be both a_i and b_j, for some i, j, and it will follow neither increase nor decrease in the value.

Eventually we get a sequence of $\leq 2n$ arcs, each marked with the total number of edges of P it "sees", and the process of creating that sequence takes **linear** time.

Finally, to find the points w_1, w_2 on the boundary of C (or return that no such pair exists) it is sufficient to first find the arc with the largest value (or one of them, if there is more than one), which will have to be between 2 points a_i and b_j (since after b_j the value decreases by 1) – which takes linear time. Denote that arc's value A_1. Then continue clockwise on the

boundary of C until b_i (which will appear before a_j, see explanation for that under the correctness section). Then continue clockwise until reaching a_j and find the arc with the largest value in that sector ($[b_i, a_j]$), denoted A_2. This part takes linear time as well. If $A_1 + A_2 = n$, return:

4. $w_1 \in A_1 - \{the\ boundary\ points\ of\ A_1\}$
5. $w_2 \in A_2 - \{the\ boundary\ points\ of\ A_2\}$

Otherwise return **false**, denoting that no such pair w_1, w_2 exists.

Correctness of the last part:

First, due to the way we draw the line segments $[a_i, b_i]$, which bound the polygon to the right of the vector $a_i \to b_i$, it cannot be that after b_j we will first encounter a_j and only then b_i (advancing clockwise), since then it "breaks" the polygon in half. Therefore b_i will be encountered after b_j before reaching a_j. This also means $[a_i, b_i]$ and $[a_j, b_j]$ intersect.

Second, it is sufficient to take the arc with maximal value (denoted $a_i \to b_j$, clockwise) and its "complement" (i.e. an arc between $b_i \to a_j$, clockwise) since if there is a solution, it means we can split the polygon by crossing some line between 2 of its vertices such that its boundary is split into two "halves" – the right and the left of that line, which can be seen by 2 points on C. The maximum value arc will surely see one of those halves.

Lastly, each point on the arc (a_i, b_j) with the largest value "sees" all segments that can be seen between a_j and b_i, clockwise. Therefore it is sufficient to find another point that will cover all segments seen between b_i and a_j, clockwise. Moreover, the segments seen in the first are by definition of the a's and b's distinct from the segments seen in the second. Thus if the sum of both equals the total number of segments, it is assured that all n segments (= edges of P) are visible by one of the two selected points.

Assignment 2, question 6)

Following is an efficient algorithm that given a convex polygon P with n vertices computes a triangulation that has stabbing number $O(\lg n)$:

1. Let P be represented by the chain $p_1, p_2, \dots, p_n$ and let $k = 2^{\lfloor \lg n \rfloor}$, then for $i = 1, \dots, k$ do:
   1.1. Starting at p_1 and ending at p_k, cross diagonals between consecutive points on the chain (in the direction $p_1 \to p_2 \to \cdots$) jumping over 2^i points every step. For instance, for $i = 1$ cross: $(p_1, p_3), (p_3, p_5), \dots, (p_{k-2}, p_k)$; for $i = 2$ cross: $(p_1, p_5), (p_5, p_9), \dots, (p_{k-4}, p_k)$ and so on.
2. Run the algorithm recursively over the polygon represented by the chain $p_1, p_k, p_{k+1}, \dots, p_n$.

Correctness:

First note that when choosing $k = 2^{\lfloor \lg n \rfloor}$, since we jump in steps of 2^i we are guaranteed that the logarithmic division of the part of the polygon handled at step 1 will be complete, i.e. p_1 will be connected eventually to p_k. Furthermore, no diagonal we cross ever crosses another one: for each i the diagonals never start at a point in between 2 points that were previously connected; and since we expand the jump step by a factor of 2 at each phase of 1.1, no diagonal at phase i between any two points we connect p_a, p_b will ever cross any of the diagonals crossed between points in the range $p_a \to \cdots \to p_b$ (basically that's a proof by induction).

Next we show that we are given a proper triangulation: for any chunk of the polygon that is handled at phase 1, every diagonal we cross between closes a triangle with 2 diagonals that were crossed at the previous phase (and for $i = 1$: closes a triangle with two consecutive edges of the polygon), thus inductively a complete triangulation is formed for that polygon chunk. Since any n can be represented as $n = \sum_{k=1}^{m} 2^{i_k}$, for some $i_1, \dots, i_m$ (i.e. a sum of powers of 2), the recursive algorithm covers the entire polygon P completely (where phase 1 runs on each of the chunks; this can also be more rigorously proven by proper induction on the number of chunks P will be divided into).

As last step we show that any line segment l interior to P crosses at most $O(\lg n)$ diagonals: if l starts at any point on an edge $p_i p_{i+1}$ of P, it may cross at most 1 diagonal that leaves p_i and jumps 2 points, at most 1 diagonal that leaves p_i and jumps 4 points, …, and at most 1 diagonal that starts at p_i and jumps $\lg 2^n = n$ vertices. The same is true for the edge on which the ending point of l lies at. Therefore the maximum number of diagonal l may cross is $O(\lg n)$, as required.

Running time:

The running time can be determined by the largest chunk of the polygon, as the division is by powers of 2 and we know that the largest chunk with $k = 2^{\lfloor \lg n \rfloor}$ vertices, will have at most $k - 1$ vertices in all other chunks altogether. Therefore, for simplicity, we will address the case where $k = n$. The number of diagonals we cross is: $\frac{n}{2}$ diagonals that jump over 1 point, $\frac{n}{2^2}$ diagonals that jump over 3 points, …, $\frac{n}{n} = 1$ diagonal that jumps over all points in between p_1 and p_n. The total number of diagonals is therefore $\sum_{i=1}^{\lg n} \frac{n}{2^i} = n \cdot \sum_{i=1}^{\lg n} \frac{1}{2^i} \xrightarrow[n \to \infty]{} n \cdot 1 \Rightarrow$ asymptotically the total number of diagonals crossed will be n, which determines the total complexity, which is linear: $O(n)$.

The worst-case running time is $O(n)$, as the number of diagonals crossed will always be $n - 3$ anyway (proved in class), and for each diagonal we do a constant number of operations, which is basically crossing it. Calculations on partitioning the polygon into chunks with power of 2 nodes and calculating steps is linear, **given the chain is represented by an array** (where any fetch operation is $O(1)$).

Simulation for 1 chunk of size 8: