Computational Geometry (CS60064)
Homework Set 3

Bratin Mondal - 21CS10016

Question 1

The diameter of a convex polygon is a/the longest line segment contained within it. Given a convex polygon P with n vertices in counterclockwise order, design an $O(n)$-time algorithm to find a diameter of P. Justify its correctness and why its time complexity is $O(n)$.

Solution

Let us assume that the convex polygon is given in the form of a list of vertices in counterclockwise order. Let the vertices be denoted by $p_1, p_2, \ldots, p_n$ where $p_i = (x_i, y_i)$ for $1 \leq i \leq n$.

Description of the Algorithm

The algorithm to compute the diameter efficiently in $O(n)$ time is as follows:

Algorithm 1 Compute the diameter of a convex polygon

$i \leftarrow n$
$j \leftarrow i + 1$
while $\texttt{Area}(p_i, p_{i+1}, p_{j+1}) > \texttt{Area}(p_i, p_{i+1}, p_j)$ **do**
  $j \leftarrow j + 1$
end while
$j_0 \leftarrow j$
Initialize an empty vector *pairs*
while $i \neq j_0$ **do**
  $i \leftarrow i + 1$
  Add (i, j) to *pairs*
  while $\texttt{Area}(p_i, p_{i+1}, p_{j+1}) > \texttt{Area}(p_i, p_{i+1}, p_j)$ **do**
    $j \leftarrow j + 1$
    if $(i, j) \neq (j_0, 1)$ **then**
      Add (i, j) to *pairs*
    end if
  end while
  if $\texttt{Area}(p_i, p_{i+1}, p_{j+1}) = \texttt{Area}(p_i, p_{i+1}, p_j)$ **then**
    if $(i, j) \neq (j_0, n)$ **then**
      Add $(i, j + 1)$ to *pairs*
    end if
  end if
end while
Compute the maximum Euclidean distance between points in *pairs*
Return the pair of points with the maximum distance

The function `Area` computes the area of the triangle formed by three points p_i, p_j, p_k using the determinant formula:

$$\texttt{Area}(p_i, p_j, p_k) = \frac{1}{2} \left| x_i(y_j - y_k) + x_j(y_k - y_i) + x_k(y_i - y_j) \right| \tag{1}$$

The above algorithm efficiently finds all antipodal pairs using the rotating calipers method and then determines the pair with the maximum Euclidean distance, which represents the diameter of the convex polygon.

Correctness of the Algorithm

Consider a vertex p_i of the convex polygon. Suppose we traverse the counterclockwise chain along the boundary of P starting at p_i, until we reach a vertex $q_R^{(i)}$, which is the farthest vertex from $p_{i-1}p_i$. In case of ties, when P has parallel edges, $q_R^{(i)}$ is the first vertex encountered in this traversal. Similarly, we define $q_L^{(i)}$ as the vertex farthest from p_ip_{i+1} in the clockwise traversal of the boundary of P starting at p_i.

We first claim that the chain of vertices between $q_R^{(i)}$ and $q_L^{(i)}$ defines the set $C(p_i)$ of vertices that form an antipodal pair with p_i. To prove this claim, we first derive the condition for a set of vertices to be antipodal.

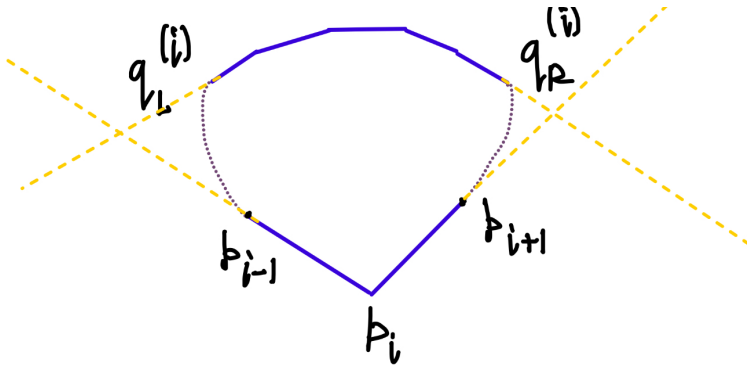

Figure 1: Antipodal pairs for a vertex p_i

Consider two vertices p_i and p_j. Let α_i denote the external angle formed by the line segment $p_{i-1}p_i$ and p_ip_{i+1}, and let α_j denote the external angle formed by the line segment $p_{j-1}p_j$ and p_jp_{j+1}. The pair of vertices p_i and p_j are antipodal if and only if there is a straight line in the intersection of α_i and α_j. Informally, we can say that α_i and α_j should overlap in two opposite directions; just one intersection is not sufficient for a line to pass through.

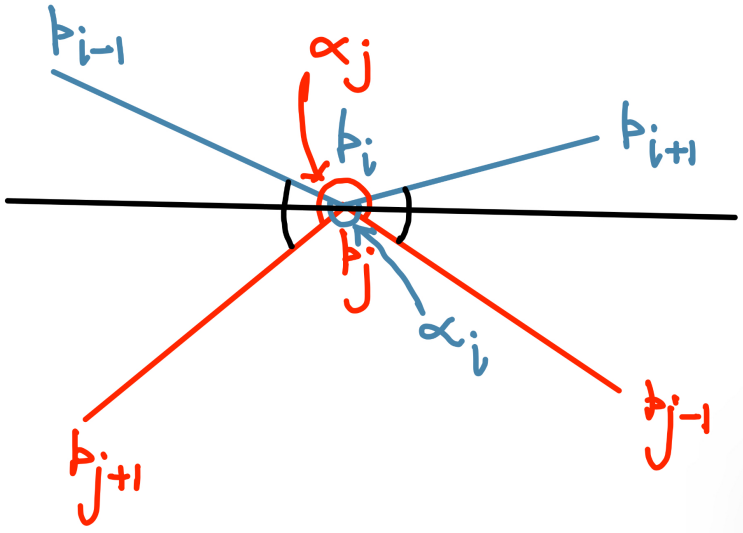

Figure 2: Condition for antipodal pairs

We now prove that during our counterclockwise traversal of the boundary of P starting at p_i, any vertex before $q_R^{(i)}$ or after $q_L^{(i)}$ cannot be part of an antipodal pair with p_i. Consider the vertex p_j encountered just before $q_R^{(i)}$ during the traversal. Through the point $q_R^{(i)}$, draw a line parallel to $p_{i-1}p_i$ and call it $L_{q_R^{(i)}}$. Since $q_R^{(i)}$ is the farthest vertex from $p_{i-1}p_i$, the point p_j must lie on the same side of the line as p_i. Let the segment $q_R^{(i)}p_j$ form an angle α ($\alpha > 0$) with $L_{q_R^{(i)}}$. The same angle α is formed by $q_R^{(i)}p_j$ with any line segment parallel to $p_{i-1}p_i$. Due to this, the intersection of α_i and α_j cannot contain a straight line, since it will only have one intersection. There cannot be another intersection, as α represents the opposite non-overlapping part. Hence,

p_j cannot be part of an antipodal pair with p_i. (As shown in Figure 3, $\alpha_i \cap \alpha_j$ has only one intersection and a straight line cannot pass through it.)

Due to the convex nature of angles, any other vertex before p_j will also not be part of an antipodal pair with p_i. Similarly, we can prove that any vertex after $q_L^{(i)}$ cannot be part of an antipodal pair with p_i. Due to convex nature of $C(p_i)$, the set of vertices between $q_R^{(i)}$ and $q_L^{(i)}$ (inclusive) forms the antipodal pair with p_i.

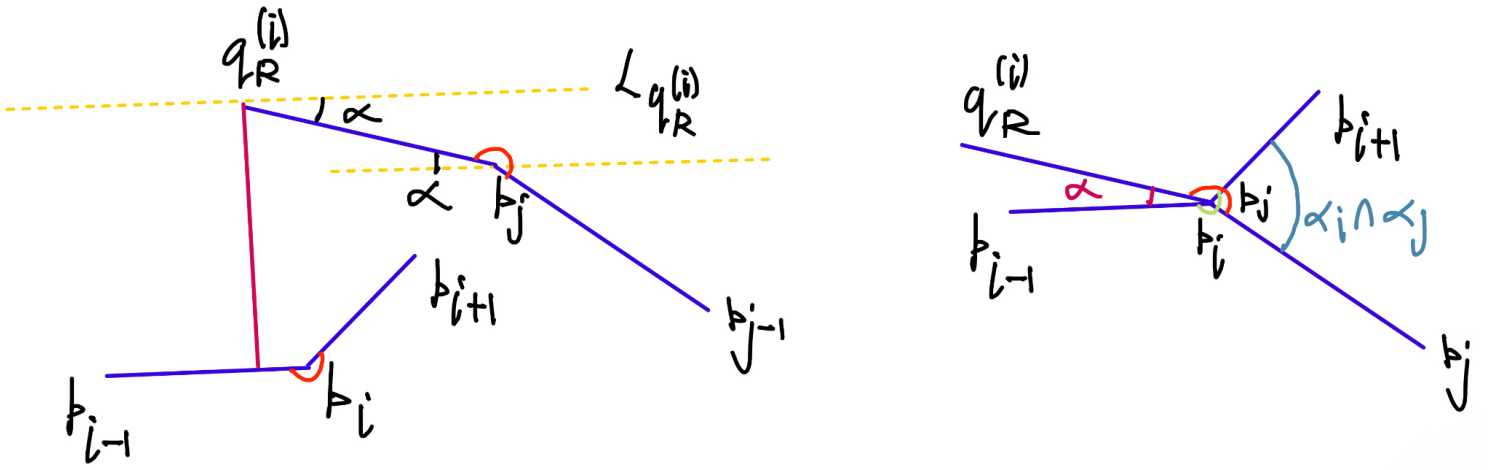

Figure 3: Vertex before $q_R^{(i)}$ cannot be part of the antipodal pair with p_i

The proposed algorithm efficiently finds the antipodal pairs for all vertices of the convex polygon. Lines 1-4 locate the vertex $q_R^{(1)}$. The subsequent while loop which constructs the set $C(p_i)$ uses two pointers i and j to traverse the vertices of the polygon. These two pointers move counterclockwise around P, i advancing from n to $q_R^{(1)}$ and j advancing from $q_R^{(1)}$ to n. An antipodal pair is generated each time any of the pointers advances. Parallel edges which can lead to a point having multiple antipodal pairs are handled by lines 17-21. The first pair generated is p_1, p_{j_0} and the last pair generated is p_{j_0}, p_n. Hence, each antipodal pair is generated exactly once since i advances from 1 to j_0 and j advances from j_0 to n. Finally, the pair of points with the maximum Euclidean distance is returned as the diameter of the convex polygon.

Time Complexity Analysis

Finding j_0 in lines 1–4 takes $O(n)$ time. Since each antipodal pair is generated exactly once as i advances from 1 to j_0 and j advances from j_0 to n, the while loop consists of a total of n moves. Hence, the while loop takes $O(n)$ time.

In the absence of parallel edges, the while loop generates exactly n antipodal pairs. If parallel edges exist, their number is at most $\lfloor n/2 \rfloor$, resulting in a maximum of $3n/2$ antipodal pairs.

The final step of computing the maximum Euclidean distance among the pairs takes $O(n)$ time. Thus, the overall time complexity of the algorithm is $O(n)$.

Question 2

Let P and Q be two convex polygons with m and n vertices, respectively, given in counterclockwise order. Suggest an $O(m \log n)$-time algorithm to check whether P contains Q. Suggest another algorithm that will take time $O(m + n)$. Derive the time complexities of both algorithms.

Solution

Let the vertices of the convex polygon P be denoted by $p_1, p_2, \ldots, p_m$ and the vertices of the convex polygon Q be denoted by $q_1, q_2, \ldots, q_n$.

Part-1

Deriving an algorithm with a time complexity of $O(n \log m)$ is straightforward. For each point in Q, we can use the convex polygon membership query algorithm to check if the point lies inside P, where each query takes

$O(\log m)$ time (as discussed in Homework 1). If all points in Q are inside P, then due to the convex nature of Q, Q is completely inside P. The total time complexity of this algorithm is therefore $O(n \log m)$.

For an algorithm with $O(m \log n)$ time complexity, we will use a different approach where for every edge of P, we check if the vertices of Q lie on the same side as the vertices of P.

Specifically, for each edge of P, we proceed as follows. Let $p_i p_{i+1}$ be the current edge of P. First, we consider the perpendicular direction to the edge $p_i p_{i+1}$, which points towards the interior of the polygon. We then consider a line passing through the midpoint of the edge $p_i p_{i+1}$ in this perpendicular direction. Let this line be denoted by $\vec{p}_\perp$, and the midpoint of the edge $p_i p_{i+1}$ is p_{mid}.

Next, consider the projection of all the vertices of Q onto the line $\vec{p}_\perp$. We note that we do not need to construct all the projection points, but we compute them on the fly as needed. The `projection distance` for a projection point q_j on the line $\vec{p}_\perp$ is defined as the `signed distance` of the projection point from the midpoint p_{mid} of the edge $p_i p_{i+1}$. The `signed distance` is positive if the projection point is on the side of the edge $p_i p_{i+1}$ that is the interior of the polygon and negative otherwise.

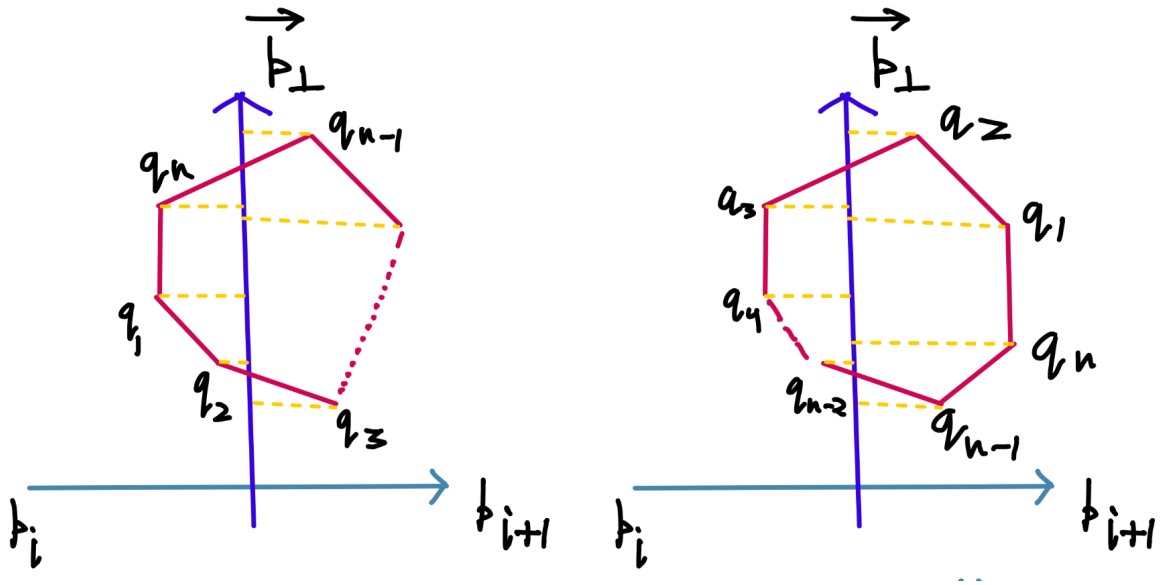

Figure 4: Projection of vertices of Q on the line $\vec{p}_\perp$

We observe that due to the convex nature of Q, the `projection distance` of the vertices of Q on the line $\vec{p}_\perp$ can follow two specific patterns. The first pattern decreases to a minimum, increases to a maximum, and then decreases again. The second pattern increases to a maximum, decreases to a minimum, and then increases again. (Any of the increasing or decreasing parts can have length 0.) The comparison between the `projection distance` of q_1 and q_n will tell us which pattern we are dealing with. If the `projection distance` of q_1 is less than that of q_n, we are in the first pattern; otherwise, we are in the second pattern.

Now, we can perform a binary search to find the minima and maxima of the `projection distance` of the vertices of Q on the line $\vec{p}_\perp$ in $O(\log n)$ time. The binary search algorithm needs to be modified to work in the presence of the two types of increasing/decreasing patterns.

For simplicity, we describe the binary search approach for the first pattern. As shown in Figure 5, there are two decreasing regions labeled C^1_{dec} and C^2_{dec}, and one increasing region labeled C_{inc}. It is clear that we can easily detect when we are in C_{inc} during the binary search and can move left for a minimum and right for a maximum. To distinguish between C^1_{dec} and C^2_{dec}, we compare the `projection distance` of the current element with q_1. If it is greater, we are in C^2_{dec}; otherwise, we are in C^1_{dec}, and we move accordingly for the binary search. The second pattern can be handled similarly.

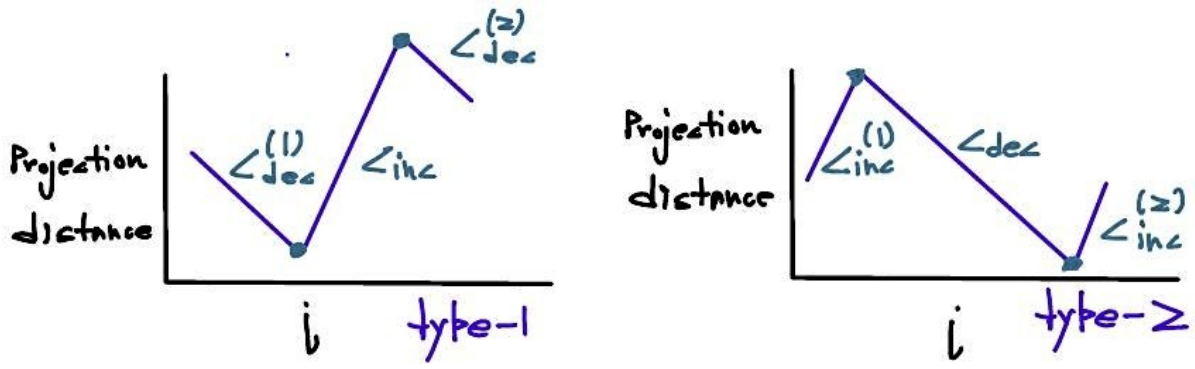

Figure 5: Finding the maxima of the `projection distance` of the vertices of Q on the line $\vec{p}_\perp$

We are interested in finding the minima in this case. Using the approach described above, we find the minima of the `projection distance` of the vertices of Q on the line $\vec{p}_\perp$. If we find a minimum that is less than 0, we can conclude that Q is not completely inside P. (We can include other optimizations, such as immediately concluding that Q is not inside P if we find any `projection distance` less than 0 during the binary search, but this does not improve the time complexity.)

Thus, in the final algorithm, we check for all edges of P. If we find a minimum less than 0 for any edge, we can conclude that Q is not completely inside P. If no such minimum is found, we conclude that Q is completely inside P.

Time Complexity Analysis

For each edge of P, we perform a binary search to find the minima of the `projection distance` of the vertices of Q on the line $\vec{p}_\perp$. We compute the `projection distance` of the vertices of Q on the fly as needed which helps us avoid computing all the projection points. The binary search takes $O(\log n)$ time. Since there are m edges in P, the total time complexity of the algorithm is $O(m \log n)$.

Part-2

To check whether polygon Q is completely inside polygon P in $O(m+n)$ time, we apply the convex polygon intersection algorithm described in [1]. The idea is to construct the intersection of the two polygons and verify if the intersection is equal to Q. The time complexity of this algorithm is $O(m+n)$.

We begin by defining some necessary terms. For a point p_i in a polygon P, the half-plane $hp(p_i)$ is defined as the set of points that lie on the same side as the other points of the polygon with respect to the line passing through vertices p_{i-1} and p_i. The mathematical definition of $hp(p_i)$ is given by:

$$hp(p_i) = \{p \in \mathbb{R}^2 : (p_i - p_{i-1}) \times (p - p_i) \geq 0\}$$

where $\times$ denotes the cross product of two vectors.

The algorithm for finding the intersection of two convex polygons is described in Algorithm 2. The process begins with the first vertices of the two polygons and iterates over the vertices of both polygons. Depending on the relative position of the vertices, the algorithm advances through the vertices and calculates the intersection points. The algorithm terminates after at most $2(|P| + |Q|)$ iterations, after which we can conclude that the polygons do not intersect.

Subsequently, the algorithm checks if one of the polygons is completely inside the other by verifying if all vertices of one polygon lie on the same side of all edges of the other polygon. If one polygon is fully contained within the other, the algorithm outputs the polygon that is completely inside. If neither polygon is fully contained within the other, the algorithm outputs $\varnothing$.

Algorithm 2 Intersection of two convex polygons

$p \leftarrow p_1$, $q \leftarrow q_1$ and `intersection_polygon` $\leftarrow \emptyset$
repeat
  Test if p and q intersect
  if p and q intersect **then**
    if this intersection is the same as the first intersection **then**
      output `intersection_polygon`
    else
      add the intersection point to `intersection_polygon`
      if $p \in hp(q)$ **then**
        inside $\leftarrow$ "P"
      else
        inside $\leftarrow$"Q"
      end if
    end if
  end if
  if $q \times p \geq 0$ **then**
    if $p \in hp(q)$ **then**
      `advance` q
    else
      `advance` p
    end if
  else
    if $q \in hp(p)$ **then**
      `advance` p
    else
      `advance` q
    end if
  end if
until repeat has executed more than $2(|P| + |Q|)$ times
$p \leftarrow p_1$, $q \leftarrow q_1$
if $p \in Q$ (By checking if p lies on the same side of all edges of Q) **then**
  output P
else if $q \in P$ (By checking if q lies on the same side of all edges of P) **then**
  output Q
else
  output $\varnothing$
end if

The `advance` operation is defined as follows for a point p_i in a polygon P:

Algorithm 3 Advance p_i for polygon P

if `inside` = "P" **then**
  add p_i to `intersection_polygon`
end if
$i \leftarrow (i \mod |P|) + 1$

Similarly, the `advance` operation for a point q_i in a polygon Q is defined as follows:

Algorithm 4 Advance q_i for polygon Q

if `inside` = "Q" **then**
  add q_i to `intersection_polygon`
end if
$i \leftarrow (i \mod |Q|) + 1$

Our final algorithm for checking if Q is completely inside P operates in $O(m+n)$ time. It first computes the intersection of the two polygons using Algorithm 2. If the intersection is equal to Q, then Q is completely inside P.

Note that the original algorithm for computing the intersection also checks if one polygon is completely inside the other. However, in cases where Q is inside P and their edges overlap, we still need to find the intersection to verify that Q is completely inside P.

Time Complexity Analysis

The algorithm iterates over the vertices of the two polygons. The repeat loop executes at most $2(|P|+|Q|)$ times. If the two polygons do not intersect, the algorithm terminates after at most $2(|P|+|Q|)$ iterations. In this case, we only need to check if one polygon is completely inside the other, which takes $O(m+n)$ time.

The intersection of the two polygons can have at most $m+n$ vertices. This is because every edge of P can be separated into at most three parts by the edges of Q, but at most one of these parts can be part of the intersection. The same reasoning applies to the edges of Q.

To check if the intersection is equal to Q, we first verify if q_1 exists in the intersection. This can be done in $O(m+n)$ time. Once q_1 is found, we compare the vertices of Q with the vertices of the intersection, which also takes $O(m+n)$ time.

Hence, the total time complexity of the algorithm is $O(m+n)$.

References

[1] J. O'Rourke, C.-B. Chien, T. Olson, and D. Naddor, "A new linear algorithm for intersecting convex polygons," *Computer Graphics and Image Processing*, vol. 19, no. 4, pp. 384–391, 1982, ISSN: 0146-664X. DOI: https://doi.org/10.1016/0146-664X(82)90023-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0146664X82900235.