Computational Geometry Homework 2

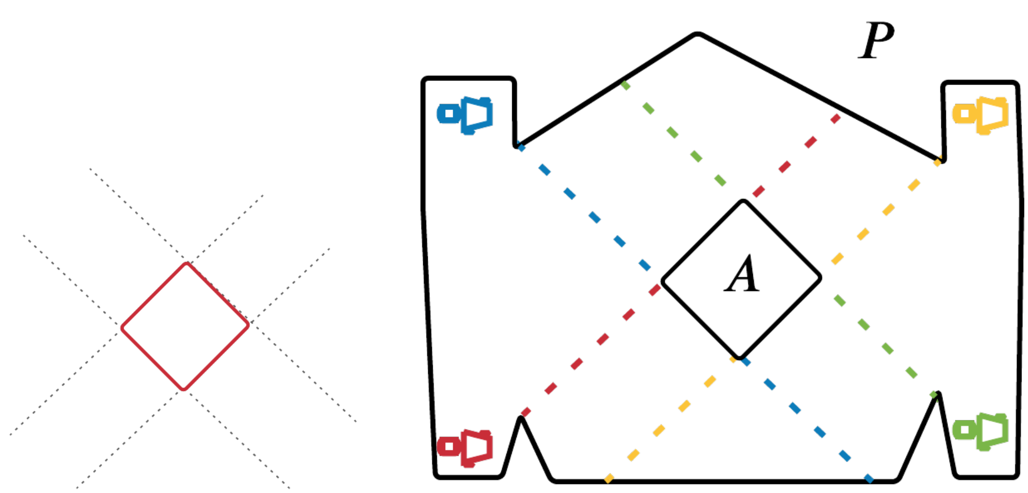
Ian Sinclair
COMP 3705-1

March 30, 2022
Dr. Mario A. Lopez

Problem 1:

Consider the problem of guarding the walls of a simple polygon P. Construct a polygon P and a placement of cameras such that the cameras can see every point on ∂P but not every point in the strict interior of P.

> Consider any simple non self intersecting polygon, A. It is possible to construct a polygon P in which A is fully contained in P, and there is at least one camera placement in P such that all of ∂P is guarded but none of the points in ∂A are guarded.
> This can be done by extending the edge angles of A and ensuring that the sight line of each camera c_i follows the edge angle of each edge in A.

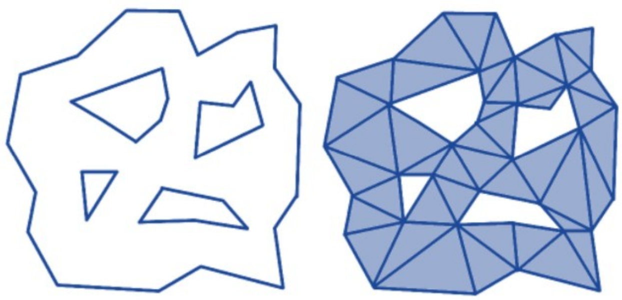


Problem 2:

Consider a polygon P with holes of size n, such that the interior of the polygon is to the left of every edge (this implies that the holes are listed CW). Here, n denotes the total number of vertices including those in the outside boundary of P and in the interior holes.

a) Derive a formula for the number of triangles in a triangulation of P built using diagonals. Justify your formula.

b) Can the triangulation graph of P always be 3-colored? Explain.



A: Derive a formula for the number of triangles in a triangulation of P built using diagonals. Justify your formula.

Let P be a polygon with h holes of size n total vertices. Let T be a triangularization of P with t triangles.
Then because the sum of interior angles in any triangle is $180°$, it follows that the sum of all interior angles across every triangles in T is $\Theta = 180t$.
Next, note that the interior angle of any triangle in T is a portion of either the internal angle between two vertices on the boundary of P,
Or the external angle of two vertices on a hole.
Then, because P is completely covered by T,
it follows that the total sum of interior angles on the boundary of P plus the sum of exterior angles on each hole of P is equal to $180t$ or the sum of angles across every triangle.
Now, partition the vertices of P such that, p_0 is the set of vertices on the boundary of P, then p_i is the set of vertices making the hole h_i on the interior of p_0.
Then it follows that, $|p_0| + \sum_{i \leq h} |p_i| = n$, where $|p_j|$ is the number of vertices in p_j.
Now note, the sum of interior angles of p_0 is $(|p_0| - 2)180°$. And the sum of exterior angles for each of p_i is $(|p_i| + 2)180°$.
Then it follows that,

$$(|p_0| - 2)180° + (|p_1| + 2)180° + (|p_2| + 2)180° + ... + (|p_h| + 2)180° = 180t$$

And so,

$$\begin{aligned} t &= (|p_0| - 2) + (|p_1| + 2) + (|p_2| + 2) + ... + (|p_h| + 2) \\ &= \underbrace{|p_0| + \sum_{i \leq h} |p_i|}_{n} - 2 + \underbrace{2 + 2 + 2 + ... + 2}_{2h} \\ &= n + 2h - 2 \end{aligned}$$

Solution

Therefore, for a polygon P with h holes and n total vertices, a triangularization T has the following number of triangles.

$$t = n + 2h - 2$$

B: Can the triangulation graph of P always be 3-colored? Explain.

Let P be a polygon with triangulation T.
Now let G be the triangulation graph of T.
Then it follows that G is planar, and each vertex $v_i \in G$ is connected to at most two over vertices.
Now proceeding by induction,
consider the hypothesis, that a triangulation graph with n vertices is 3-colored.
Note the basis for a triangulation graph with $n = 3$ vertices is trivial.
Each vertex gets a unique color.
Now consider the inductive step by allowing a triangulation graph G with $n + 1$ vertices.
Then let G' be a subgraph of G with n vertices. Now, because every vertex $v_i \in G$ must be connected to at most 2 other vertices, this property must be conserved in G'.
Then it follows that G' is a triangulization graph.
Additionally, let v_s be the vertex included in G but not in G'.
Then let V be the set of connecting vertices of v_s, so V is either a set of 1 or 2 vertices.
Without loss of generality assume $V = \{v_{s-1}, v_{s+1}\}$ vertices.
Now because G' is a triangulation graph with n vertices, by the inductive hypothesis, G' is 3-colored.
From here consider two cases,

1. v_{s-1}, v_{s+1} have different coloring.

   If v_{s-1}, v_{s+1} have a different color in G', then color v_s with the third option, then G is 3-colored.

2. v_{s-1}, v_{s+1} have the same color in G'.

   Consider G' is a subgraph of G, and so v_{s-1}, v_{s+1} must have only one edge in G'. So v_{s-1}, v_{s+1} are each a leaf.
   So, take v_{s-1}, has one edge to v_{s-2}.
   Then v_{s-2} connects to v_{s-1} and v_{s-3}. Now, because G' is 3-colored, v_{s-2} must have a different color than v_{s-3}.
   Finally, because v_{s-2} is connected to both v_{s-1} and v_{s-3} it must be true that v_{s-1} and v_{s-3} don't share the same color.
   Then redefine the coloring by switching the color of v_{s-1} and v_{s-2}.
   Notice this does not invalidate the 3-coloring of G' and then v_{s-1} and v_{s+1} are different colors.
   So then assign v_s with the third color option, therefore G is 3-colored.

Lastly, if v_s is connected to a single vertex v_{s-1}, then v_{s-1} is a leaf in G', so set v_s to be a different color to v_{s-1} and v_{s-2}. So then G is 3-colored.

Solution

Therefore, any triangulation graph of P has a configuration that is always 3-colored.

Problem 3:

Let P be a polygon of size n. A vertex v of P is a reflex vertex if its internal angle is strictly greater than π.

a) Write and test a robust predicate $Reflex(polygon\ P, int\ i, int\ j)$ that returns true iff the ith vertex of P is a reflex vertex.

b) Write and test a robust predicate $LocallyInterior(polygon\ P, int\ i, int\ j)$ that returns true iff the segment from vertex i to vertex j if interior to P in the neighborhood of vertex i.

A: Write and test a robust predicate $Reflex(polygon\ P, int\ i)$ that returns true iff the ith vertex of P is a reflex vertex.

Consider a polygon P with 2D points, and assume P is given in counter clockwise order and that each vertex $v_i \in P$ is on the boundary ∂P.
Then given a query v_i, v_i is a reflex point if $v_i - v_{i-1}$ and $v_{i+1} - v_i$ make a right turn, or that the cross product of the two vectors will have negative sign. If the two vectors make a left turn, then v_i must be convex. This is an attribute of listing the polygon in counter-clockwise order. And so to ensure the robustness of the algorithm, it may be use full to pre-process P into counter-clockwise order.

```
class vertex() :
        def __init__( self, x : int, y : int ) :
            self.x = x
            self.y = y



def cross( A : vertex, B : vertex, C : vertex ) :
    """
    Parameters
    ----------
    C : vertex
    A : vertex
    B : vertex

    Returns
    -------
    Cross product between 2D vectors (B-A) and (C-B).
    """
    return ( (B.x - A.x) * (C.y - B.y) ) - ( (B.y - A.y) * (C.x - B.x) )

def Reflex( P, i : int ) :
    if len( P ) < 3 :
        return None

    if not ( 0 <= i < len( P ) ) :
        raise("Index i out of bounds of polygon.")

    A = P[i-1]
    B = P[i]
    if i == len( P ) - 1 :
        C = P[0]
    else :
        C = P[i+1]

    det = cross( A, B, C )

    if det < 0 :
        return True
    return False


def driver1() :
    P = [ vertex(20,20),
          vertex(15,30),
          vertex(10,20),
          vertex(10,10),
          vertex(13,10), # reflex point
          vertex(15,0),
          vertex(18,20), #  reflex point
          ]

    print( Reflex( P, 6 ) )

if __name__ == "__main__" :
    driver1()
```

Write and test a robust predicate $LocallyInterior(polygon\ P,\ int\ i,\ int\ j)$ that returns true iff the segment from vertex i to vertex j if interior to P in the neighborhood of vertex i.

Given a polygon P and a line segment between vertices $v_i, v_j \in P$, consider we are interested in whether the line segment is partially contained in ∂P around v_i.
First consider the case that segment (v_i, v_j) intersect the boundary ∂P, then there must be some neighborhood around v_i, where at least one point on the line segment (v_i, v_j) is not contained in P.
Therefore, first check if (v_i, v_j) intersects P. If it does, return False.
Next, allow a partitioning of P, P', where P' contains all vertices between $(v_i \rightarrow v_j)$ including v_i and v_j. Then, if P' has the same orientation of P and the segment (v_i, v_j) doesn't intersect ∂P, then segment (v_i, v_j) must be strictly contained in P. And if P' and P have different orientations, then segment (v_i, v_j) must not be contained in P.

```python
# -*- coding: utf-8 -*-

class vertex() :
    def __init__( self , x : int , y : int ) :
        self.x = x
        self.y = y

    def toString( self ) :
        return str(self.x) + " , " + str(self.y)



def cross( A : vertex , B : vertex , C : vertex ) :
    """
    Parameters
    ----------
    C : vertex
    A : vertex
    B : vertex

    Returns
    -------
    Cross product between 2D vectors (B-A) and (C-B).
    """
    return ( (B.x - A.x) * (C.y - B.y) ) - ( (B.y - A.y) * (C.x - B.x) )


def Orient(A : vertex , B : vertex , C : vertex ) :
    """
    Parameters
    ----------
    q : vertex
    A : vertex
    B : vertex

    Returns
    0 --> A, B and q are collinear
    1 --> Clockwise
    -1 --> Counterclockwise
    """
    Cross_product = cross( A, B, C )
    if Cross_product > 0 :
        return 1
    elif Cross_product < 0 :
        return -1
    return 0;


def onEdge( q : vertex , v1 : vertex , v2 : vertex ) :
    """
    Parameters
    ----------
    q : vertex
    v1 : vertex
    v2 : vertex
    Assume q, v1, v2 are colinear
    Returns
    -------
    bool
        True --> q is on line segment v1v2
        False --> q is not on line segment v1v2.

    """
    if (v1.y >= q.y >= v2.y) or (v2.y >= q.y >= v1.y) :
        if (v1.x >= q.x >= v2.x) or (v2.x >= q.x >= v1.x) :
            return True
    return False
```

```python
def intersect( A1 : vertex, A2 : vertex, B1 : vertex, B2: vertex ) :
    """
    Parameters
    ----------
    A1 : vertex
    A2 : vertex
    B1 : vertex
    B2 : vertex

    Line segment A1 --> A2 and B1 --> B2

    Returns
    -------
    INT
        0 --> if A1 is on line segment of B1 --> B2
        1 --> if line segment A1 --> A2 intersects B1 --> B2
        -1 --> if no intersection.

    """
    #  Orientation of every combination of connecting vectors between
    #   line segments.
    O1 = Orient(A1, A2, B1)
    O2 = Orient(A1, A2, B2)
    O3 = Orient(B1, B2, A1)
    O4 = Orient(B1, B2, A2)

    if (O3 == 0) : #  A1 is colinear with B1,B2
        if onEdge( A1, B1, B2 ) : #  A1 is on line segment B1 --> B2
            return 0
        return -1

    if (O4 == 0) : #  A1 is colinear with B1,B2
        if onEdge( A2, B1, B2 ) : #  A2 is on line segment B1 --> B2
            return 0
        return -1


    if (O1 == 0) and (O2 == 0) :  #  B1 and B2 are colinear with A1 and A2.
        if onEdge( A1, B1, B2 ) : #  A1 is on boundary of B1 -- B2
            return 0
        return -1

    #  General case, if both line segments intersect but none are colinear.
    if ( O1 != 0 and O2 != 0 and O3 != 0 and O4 != 0 ) :
        if (O1 != O2) and (O3 != O4) :
            return 1
    return -1


def LocallyInterior( P, i : int, j : int ) :

    if len( P )  < 3:
        raise("Invalid Polygon, must have more than 3 vertices.")

    if i == j :
        raise("Line segment must be non trivial, i != j")


    if abs( i - j) == 1 :
        return True

    #  check if segment ij intersects boundary of P
    if intersect( P[i], P[0], P[j], P[-1] ) == 1 :
        return False

    for v in range( 0, len(P)-1 ) :
        if (i is not v) and (i is not v+1) :
            if (j is not v) and (j is not v+1) :
                if intersect(P[i],P[v]  , P[j], P[v+1]) == 1 :
                    return False

    # Finds orientation of polygon P, and P'
    P_sub = P[j:] + P[:i] + [P[i]]

    P_sub_orient = 0
    P_orient = 0

    if len( P_sub ) <= 2 :
        print('Trivial')
        return True

```

```
    for v in range( 0 , len( P )-2 ) :
        P_orient += Orient( P[v] , P[v+1], P[v+2] )

    for v in range( 0 , len( P_sub )-2 ) :
        P_sub_orient += Orient( P_sub[v], P_sub[v+1], P_sub[v+2] )

        # Checks if P and P' are different orientations.
    if P_sub_orient < 0 and P_orient >= 0 :
        return False

    if P_sub_orient >= 0 and P_orient < 0 :
        return False

    return True


def driver1() :
    P = [ vertex(20,20),
          vertex(15,30),
          vertex(10,20),
          vertex(10,10),
          vertex(13,10), # reflex point
          vertex(15,0),
          vertex(18,20), #  reflex point
          ]

    print( Reflex( P, 6 ) )



def driver2() :
    P = [ vertex(40,20),
          vertex(40,40),
          vertex(20,40),
          vertex(20,20),
          vertex(30,30),
          ]

    print(LocallyInterior(P, 0, 3 )) #  Should be False
    print(LocallyInterior(P, 1, 3 )) #  Should be True
if __name__ == "__main__" :
    driver1()
    driver2()
```