DSA Assignment 1:Well Formed Polygons

Joe Kurian Eappen
EE13B080

March 28, 2016

1 Introduction

Program was done in C++ with the use of the STL. Vectors were used to facilitate dynamically allocated structures (such as number of sides in a Polygon, number of Polygons,etc.). This could have been done with preallocated memory spaces with the limiting cases in mind as well.

1.1 Data Structures And Special Variables Used

1.1.1 Structures:

1. Point: To store X and Y coordinates of a point where both are non negative integers.
2. Side: Consisting of two Point Structures. Preprocessing was done to store the left coordinate of a particular side in the first Point structure and the right coordinate in the second for all edges.
3. CentroidPoint: To store the X and Y coordinates of the Centroid of a Polygon (float used since it may have decimal values)
4. edge: To store the 'to' node and length of the edge in the AdjacencyList representation.

1.1.2 Classes

1. Polygon:
   (a) Important Members:
      i. bool convexity : true if shape was convex, false if not (intended to implement faster algorithms for convex polygons but spent time on other aspects of the program)
      ii. Vector of Side struct
         A. sides : Sides in the same order as inputed
         B. sidesOrderedByX: Sides ordered in ascending order of the leftmost coordinate
      iii. int xmax,xmin,ymax,ymin: To store the values needed to construct the bounding rectangle of the Polygon.

   (b) Important Member Functions:
      i. bool CheckIfAntiClockwise():
         A. Checks if point given are in the anticlockwise order.
         B. Uses the fact that the integral of the area is negative if going in the anticlockwise direction.
      ii. bool CheckConvexity(): Check if polygon is convex using the fact that any 3 points in order should have clockwise orientation if the polygon is convex.
      iii. int orientation(Point p, Point q, Point r):

        A. Returns 0,1 or 2 depending on if p,q, and r are collinear, anticlockwise or clockwise (in our unusual coordinate system where -y axis is +y.)
        B. Used the slopes of the line from Point p to q and q to r to make the decision.
        C. Algo provided here.

    iv. bool onSegment(Point p, Point q, Point r): Returns true if Point q is on the line segment connecting p and r.

    v. bool doIntersect(Point p1, Point q1, Point p2, Point q2, bool DoingPointInPolyCheck=false):
        A. if DoingPointInPolyCheck=false then Check if line p1q1 and p2q2 are intersecting excluding the case that one point lies on the other line (or is collinear)
        B. else consider that case as well.
        C. This was needed to include the condition that touching edges or a vertice on an edge was not considered to be intersecting (which was a major reason for most hiccups in the code and a significant part of the problem statement.)

    vi. bool CheckIfIntersecting():
        A. To check if any of the edges of the polygon intersects with the other.
        B. This was made before the idea of using an Edge structure or a vector of such structures came to mind and iterates through all points constructing edges as we move along.

    vii. bool CheckPointInPolygon(Point p):
        A. Checks if a given pont P is inside the polygon or not (includes edges).
        B. First checks if Point is outside the bounding rectange of the polygon.
        C. If it is inside, counts number of intersections of a line y=(x coordinate of P). If this is even then the point is outside, else it is inside.
        D. If collinear then check if point lies on the collinear edge else don't count it.

    viii. bool CheckPointStrictlyInPolygon(Point p): Checks if a given pont P is strictly inside the polygon or not.

    ix. Constructor function taking vector<int> x,vector<int> y: Calculates bounding rectangle, centroid, convexity and populates sides and sidesOrderedByX.

2. SetOfPolygons:

    (a) Important Members:
        i. vector<Polygon> PolyList : a vector of the Polygons in the program
        ii. vector<float> distances: The distances to each polygon from the first one (if graph is connected).
        iii. vector< vector<int> > AdjacencyMatrix: Matrix used to check for intersections. -2 for not checked, -1 for infinity, 1 for Intersecting (see constants).
        iv. vector< vector<edge> > AdjacencyList : Adjacency List representation of th graph used in the shortest path algorithm.
        v. int DisconnectedPoly, bool Disconnected :
        In case the graph is disconnected, store the position of the disconnected polygon.

    (b) Important Member Functions:
        i. void PopulateAdjMatrix():
            A. Makes our version of an Adjacency Matrix going row by row.
            B. Checks if bounding rectangles are intersecting first, if not, mark as non intersecting else next run CheckPolyIntersect()
            C. If PolyIntersect returns true, add to adjacency list with edge length as proposed distance.

D. Else mark as non-intersecting

E. Uses fact that Adjacency Matrix is symmetric (repeated calculations are skipped)

ii. void SetDistances(): Run Dijkstra's algo on the AdjList.

2 Polygon Intersection Algorithm bool CheckPolyIntersect(Polygon poly1,Polygon poly2):

1. Iterates side by side from the using the preprocessed SidesOrderedByX vector and is based on the line sweep algorithm done at the vecrtices.
2. Chooses the side further to the left out of both polygons and adds to the active set.
3. Check if side added to the active set intersects with any of the sides in the active set.
4. Checks if the side has the right end point of any of the sides in the active set (which is true for all sides other than the first two) and removes them if so.
5. Break if inserted side's left point is beyond xmax of the other polygon.
6. This skips the case where the polygon is wholly contained in the other so need to check if at least one point is inside the other polygon to catch this. This is where we use CheckIfPointInPolygon().
7. If all the vertices of one lie on the edges of the other polygon then the sides are not considered intersecting and this is a problem.
8. Returns true if the polygons are completely identical (side vectors are equal).
9. Now need to check cases where the polygon sides or vertices are touching as well the polygons intersect with non zero area.

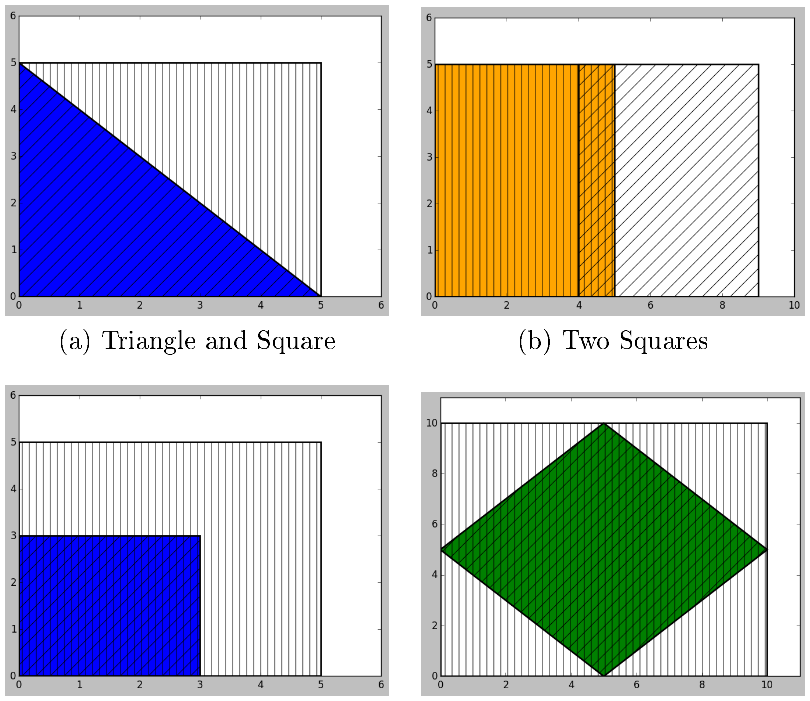
3 Problematic Testcases

Using the iteration along sides was yielding problems when the sides or vertices are touching the other polygons sides, without the sides intersecting (in our problem, touching was not considered intersecting) and having nonzero intersection area.

Examples are:

One possible way is to check if some randomised point along the line lies in the other polygon or if maybe the centroid lies in the other polygon given that all the points lie on the edge. The submitted code tried to check if the midpoint of any of the sides lie in the polygon but was largely imperfect since the midpoint was not considered to have decimal coordinates (since most of the functions would have to be overwritten to work with floats, or templates could have been used to enable this functionality) . This midpoint test would fail if the midpoints of the edges lie on the boundary of the polygons. A better way of strictly lying inside and touching the edge should have been implemented.

One method would have been to run the linesweep algorithm over all integers in the space (pass a line x=i for all i and check for order of sides intersecting) but this would have failed in cases such as the second one with the intersecting area being 1 unit wide. Need to make special conditions for edge touching cases here too.


(a) Triangle and Square

(b) Two Squares

(c) Two Squares

(d) Two Squares

Figure 1: Four Problematic Testcases

4 Analysis of Time Complexity

For each check of two polygons intersecting the worst case is when you go through all sides. - O(M+N) For a particular side, worst case of checking for intersection is when it intersects all the other polygons sides O(M+N).

So this is O(P(N)^2).
Where p is number of polygons and n is number of vertices.
This can be significantly improved using a O(nlogn) algo to check for polygon intersection or the O(M+N) algo for the convex polygons and the normal algo for the concave polygons and intersection between convex and concave polygons.