Rectilinear Partitioning of Rectilinear Polygon using Dynamic Programming

Dibyendu Saha, M.Tech, University Of Kalyani

Abstract

In this paper we propose an algorithm for partitioning a rectilinear polygon into a minimum number of rectangles using a minimum number of cut lines using dynamic programming. We take the input points, possible cut lines, and possible points that cut the existing edges on new points and we will also take the intersection points of each cut line and we take each possible combination from these set of points into consideration to create the rectangles and to find the optimal result. We tried to make the algorithm to find the optimal cuts we can have on the input polygon that will divide the polygon in n sub rectangles. This algorithm will take the orthogonal hull points of the polygon as an input and it will give output as a set of optimal rectangles it finds.

1 Introduction

Partitioning complex Polygons is the fundamental problem in digital geometry and it helps to identify the similar features between multiple building plans, load distribution, chip manufacturing, pattern recognition and Geo-informatics. The first time this problem arose because of the growing VLSI layout design image processing industries due to their real world applications. In this paper we will propose an algorithm to partition an rectilinear polygon using multiple rectilinear rectangles with the main objective to minimise the number of rectangles and cuts we need to have to create those polygons.

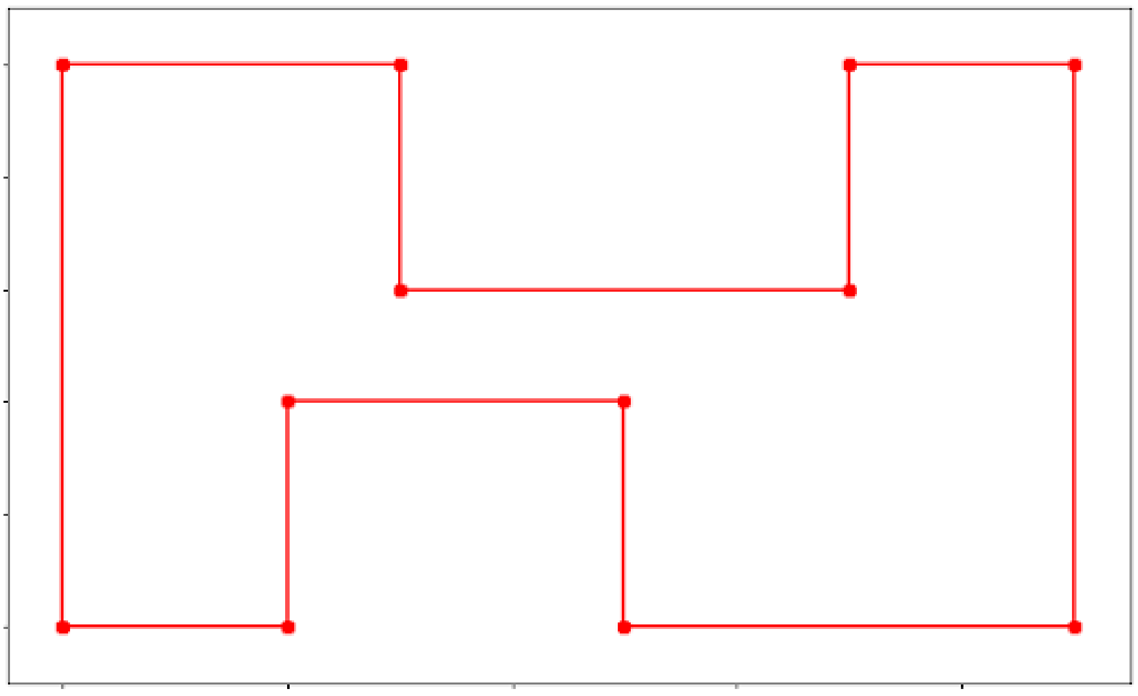

Figure 1: Input Rectilinear polygon

If we try to solve this problem using our human eye, the first thing we do is to find a rectangle that will consume the maximum area, and we repeat the process with leftover area until the whole area is covered. If any area is left and requires more rectangles we adjust the first rectangle and repeat the process once more time. This is the exact thing we implement in our algorithm. We will try to find the minimum rectangle by trying to maximise the area of each rectangle we find so that each rectangle will consume the maximum area of any given polygon figure [1] and the number of rectangles will be minimum. By finding the minimum number of rectangles we can also find the cut lines that will be needed to make the number of rectangles minimum by comparing and merging the edges of the new sub rectangles with the given hull edges. The new lines that will not be in the input hull will be our optimal cuts. In this kind of rectilinear partitioning some rectangles will have one or more of the rectilinear polygon points taken from initial orthogonal hull points as per work of Hwi Kim[**?**].

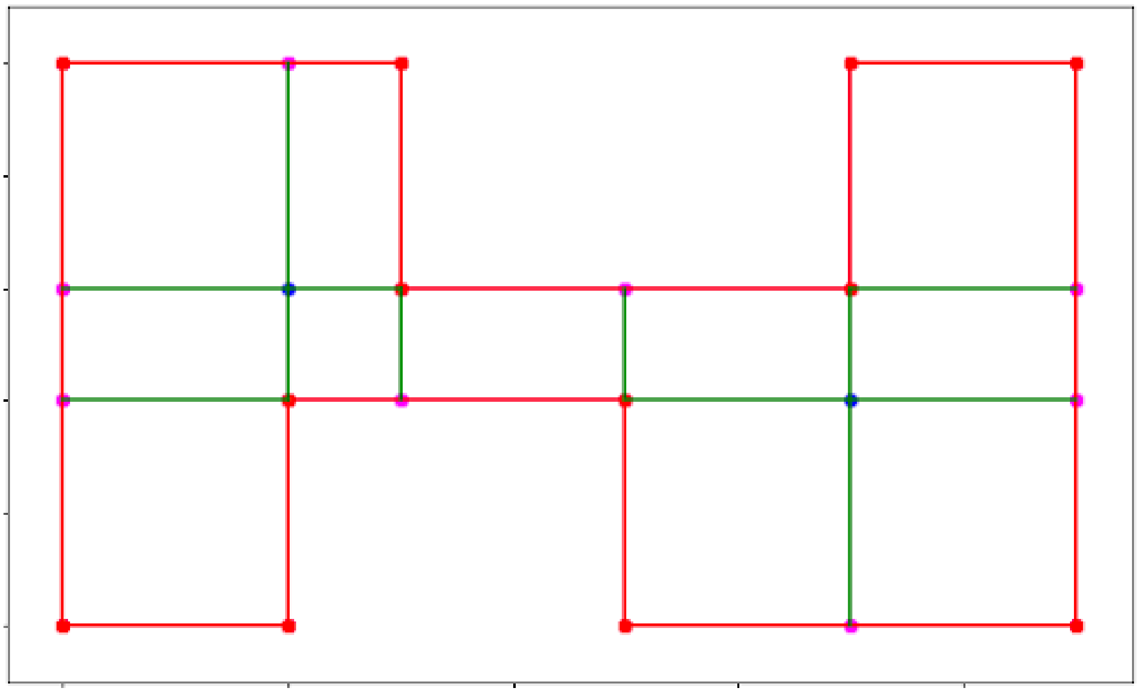

Figure 2: Points marking; 1.red vertices are the initial points, 2. Red lines are the initial rectilinear polygon lines, 3. Green lines are the cut lines by extending initial points, 4. Pink points are the projection points of the cut lines, 5. Blue points are the intersection points of the initial cut lines.

We will try to create rectangles from each corner point and so there will be horizontal and vertical lines intersecting the whole polygon and landing on other edges. These new edge points will be placed to the set of orthogonal hull points too and this will be our new set of hull points. We will also calculate the intersection points of two or more cut lines and now the set of the new hull points and intersection points will be passed as an input to the greedy algorithm. The points are shown in Figure[1]

Now the algorithm will compute the possible combination of 4 points among the new ponts set, and each possible rectangle will be processed to decide whether it is allowed to be a possible sub-rectangle or not. If it is eligible to be a rectangle we will compare it with the maximum area, if it's more than the previous maximum area, we will restart the process starting with this rectangle. After we found the eligible candidates we will then perform merging by taking each possible combination of rectangle set and merging two or more rectangle with fall to one, of there is an overlap and union of those two rectangle does not form a rectangle we will cut the portion of smaller rectangle to resolve it. After we got the final rectangle set, this needed to be compared with the input hull to find the cut lines to make the rectangles.

2 Paper Organisation

We will Start with the methodology we used to build our algorithm. After that we will discuss about the main algorithm and the various parts of it. Next We will summarize the algorithm and Later We will take few example rectilinear polygon and discuss the results of of our algorithm on those examples. Then we have a brief look on the future scope of this algorithm, and we will end our paper with conclusion and references.

3 Methodology

The proposed algorithm consists of the following steps; 1. At first we find the orthogonal hull of the given polygon object. Then we extend each hull point internally within the polygon to find the cut points that fall on other edges. We also compute the intersection points of line cut points. 2. Next we will take the union of a set of points of initial hull point, cut point on each edge and cut intersection points and we will try to compute each possible rectangle combination that falls within the input polygon and create a set of possible rectangles. 3. And at last we combine the rectangle in such a way that there is no overlap between any rectangle, no portion of any rectangle is not outside of the hull, total area of the given hull is covered and the number of rectangles is minimum.

4 Algorithm Description

Our algorithm has three parts: points marking, sub rectangle calculation, and then merging the rectangles.

4.1 Points Marking

In this algorithm we will first try to find potential candidates of the optimal rectangle solution. There are three types of candidate points, initial candidates those who are on the orthogonal hull, projection points that fall on any edge of the hull and the intersection points of the projection lines. Then these potential points are feed-ed into the rectangle creation part.

4.1.1 Initial Points

Initial points are the given input orthogonal hull points. We will start from these initial points and try to find other potential points for the optimal rectangles. This points are the corner points of each hull so we can assume there will be one or more react-angle for each hull corner point.

4.1.2 Projection Points

On any orthogonal hull if we want to divide it into multiple pieces we need to goto each edges one by one and need to check if there exist a perpendicular on that line passing through any hull points in a way there is no part of the cut line exist outside the given polygon this new point is the projection point, this projection point needs to be inline with the two vertices of the edge. As this projection point is on the edge we need to make sure this projection point is not any vertex of the initial point and the projection line or cut line is not an edge. If the projection point passed all the above criteria we will add the projection point into the initial hull points.

Algorithm 1 Pseudo code for generating projection points

```
 1: procedure $generateProjectionPoint(polygon)$        ▷ $INPUT : RectilinearPolygon$
 2:     $points \leftarrow getPointsFromPolygon(polygon)$
 3:     $edges \leftarrow getEdgesFromPolygon(polygon)$
 4:     $cutLines \leftarrow []$
 5:     for $each\ p \in points, \ldots$ do
 6:         for $each\ edge \in edges, \ldots$ do
 7:             $projectionPoint \leftarrow getProjectionPoint(edge,\ point)$
 8:             if $(edge[0],\ projectionPoint\ and\ edge[1]\ are\ in\ same\ line)$ & $(projectionPoint\ is\ not\ edge[0]\ or\ edge[1])$ then
 9:                 $cut \leftarrow [point, projectionPoint]$
10:                 if $(point\ is\ not\ projectionPoint)$ & $(cut\ not\ in\ edges)$ & $(cut\ not\ fall\ outside\ the\ polygon)$ & $(projectionPoint\ not\ in\ points)$ then
11:                     $insertprojectionPointintopointsafterindexofedge[0]$
12:                     $cutLines.append(cut)$
13:                 end if
14:             end if
15:         end for
16:     end for
17: end procedure
```

these are the sub algorithms we use to in order to get the Algorithm [1]. These all are the helper function of the Algorithm [1].

Algorithm [4.1.2] is to generate a list of points from a polygon object.

Algorithm 2 Pseudo code for finding polygon points

```
1: procedure $getPointsFromPolygon(polygon)$
2:     $points \leftarrow makeIterableArrayOfPoints(polygon)$
3:     return $points$
4: end procedure
```

Algorithm [**??**] is to generate a list of edges from a polygon object.

Algorithm 3 Pseudo code for finding polygon edges

```
1: procedure $getEdgesFromPolygon(polygon)$
2:     $index \leftarrow 0$
3:     $points \leftarrow getPointsFromPolygon(polygon)$
4:     $pointsCount \leftarrow points.length$
5:     $edges \leftarrow []$
6:     for $index < pointsCount \ldots$ do
7:         $line = [points[index], points[mod(index + 1, pointsCount - 1)]$
8:         $edges.push(line)$
9:         $index \leftarrow index + 2$
10:    end for
11:    return $edges$
12: end procedure
```

Algorithm [**??**] is to single projection point of a point to a edge.

Algorithm 4 Pseudo code for finding single projection point

```
1: procedure $getProjectionPoint(edge,\ p)$
2:     $x \leftarrow edge[0]$
3:     $y \leftarrow edge[1]$
4:     $x1, y1 \leftarrow x[0], x[1]$
5:     $x2, y2 \leftarrow y[0], y[1]$
6:     $xp, yp \leftarrow p[0], p[1]$
7:     $x12 \leftarrow x2 - x1$
8:     $y12 \leftarrow y2 - y1$
9:     $dotp \leftarrow x12 * (xp - x1) + y12 * (yp - y1)$
10:    $dot12 \leftarrow x12 * x12 + y12 * y12$
11:    if $dot12 == True$ then
12:        $coeff \leftarrow dotp/dot12$
13:        $lx \leftarrow x1 + x12 * coeff$
14:        $ly \leftarrow y1 + y12 * coeff$
15:        return$(int(lx), int(ly))$
16:    else
17:        return $False$
18:    end if
19: end procedure
```

After we find the projection points those points need to be inserted into proper places in the points array so that we can loop though the new points one by one.

4.1.3 Intersection Points

The main target is to collect maximum number of possible potential points that will become part of optimal rectangle. After we find the intersection lines we need to compute the intersection point of the cut lines as this points will be also potential points to form the optimal rectangles. We will also consider multiple points from the same cut line if there exist more overlap. this will be last step to generate potential points. To find the intersection points the pseudo code is described in Algorithm [**??**]

Algorithm 5 Pseudo code for finding Intersection points

```
1: procedure $generateIntersectionPoint(cutLines)$                ▷ $Input : cutLines$
2:     $intersections \leftarrow []$
3:     for $each\ line1 \in cutLines \ldots$ do
4:         for $each\ line2 \in cutLines \ldots$ do
5:             if $(line1 == line2)$ & $(line1[0] == line2[0])||(line1[0] == line2[1])||(line1[1] == line2[0])||(line1[1] == line2[1])$ then
6:                 continue
7:             end if
8:             $intersectionPoint \leftarrow lineIntersection(line1, line2)$
9:             if $(intersectionPointisnotNull)$ & $(intersectionPointnotinintersections)$ & $(intersectionPointnotinnew_points)$ then
10:                $intersections.append(intersectionPoint)$
11:            end if
12:        end for
13:    end for
14:    return $intersections$
15: end procedure
```

After finding all of the above points we will create the set by taking union of it. All types of points are shown in Figure [1]. Pink points are the projection points, blue points are the intersection points and green lines are the projection line from initial hull point to projection point.

4.2 Sub Rectangle Calculation

On this step we will perform dynamic programming to find all types of of rectangular polygon that can be formed by the points set got from section **??**. We will take any four points form the point set and we will check if its a rectangle and eligible to be a potential rectangle or not. Also

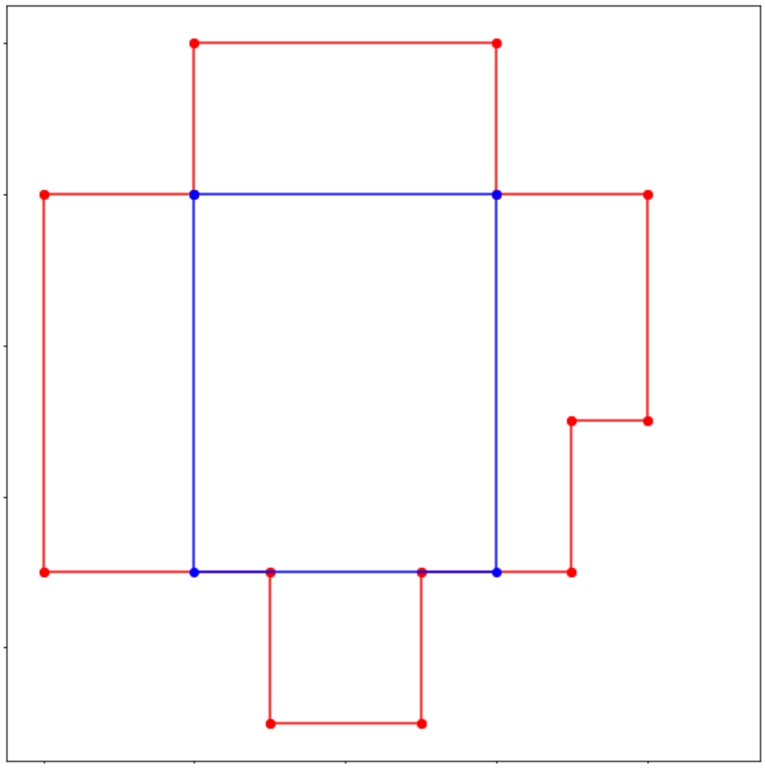

Figure 3: Capturing the rectangle having the the max_area

by this algorithm we will try to fit the maximum possible rectangle on to this polygon. To find this we will start with a rectangle that has the maximum area among other rectangle until now. If we found any point set(rectangle) that has the area more than the *max_area* we will start over the process with this rectangle as the starting point.

Rectangle finding function will take three arguments as input set of all points, *max_area* found till now, vertices of the *max_rectangle* and the primary polygon. Now we need to take the permutation combinations of all points to take 4 points $nP4$. We need to check if the taken 4 points are the valid rectangle or not. We can easily do so by drawing a bounding box rectangle of these 4 points by calculating $min\_x, min\_y; max\_x, min\_y,\ max\_x, max\_y\ and\ min\_x,\ max\_y$. And then check if the area of the bounding box is the same as the area of the taken points polygon and there are no two lines of these 4 points in the same line. We also need to test if the intersection area of this polygon and primary polygon is zero. the area of this rectangle is less than the max_area we found as the argument. If we found any case where the new rectangle area is more, the function will return this rectangle back, and we need to start over the permutation again so that we will start from the beginning. The main objective of doing this is to find the rectangle having the maximum area and then making the smaller rectangle so that there will be less number of rectangles to be processed in the merging function. After these test cases if passed we again need to check whether any form of these four rectangles is already interested into the new rectangle set if not insert it to the final_set. After all the permutation ends, return the final set. The final set is shown in Figure[**??**]

Algorithm 6 Pseudo code for finding sub rectangles

```
1: procedure $findRetangles(allPoints,\ rectangles,\ max\_area,\ primaryPoly)$
2:     for $each\ poly \in permutation(all\_points,\ 4)$ do
3:         if $(poly.area\ \neq 0)\ \&\ (poly.area == enclosed_rectangle(poly))\ \&\ (poly \notin rectangles)$ then
4:             $allowd\_to\_add \leftarrow False$
5:             if $rectangles.length == 0$ then
6:                 $allowd\_to\_add \leftarrow True$
7:             else
8:                 $has\_overlap \leftarrow False$
9:                 for $each\ \ \in rectangles$ do
10:                    if $hasOverlap(poly, rect)$ then
11:                        $has\_overlap = True$
12:                        break
13:                    end if
14:                end for
15:                if $has\_overlap == False$ then
16:                    $allowd\_to\_add \leftarrow True$
17:                end if
18:                if $allowd\_to\_add\ \&\ primary\_poly.intersection(poly).is\_closed\_area$ then
19:                    if $poly.area > max_area$ then
20:                        return [poly], True, poly.area
21:                    end if
22:                    $rectangles.append(poly)$
23:                end if
24:            end if
25:        end if
26:    end for
27:    return $rectangles, False, max_area$
28: end procedure
```

,,