Students:
Fatima Yousif Rustamani [fyousif30@gmail.com]
Syma Afsha [symaafsha.eece@gmail.com]

Lab2 : Visibility Graph

Introduction

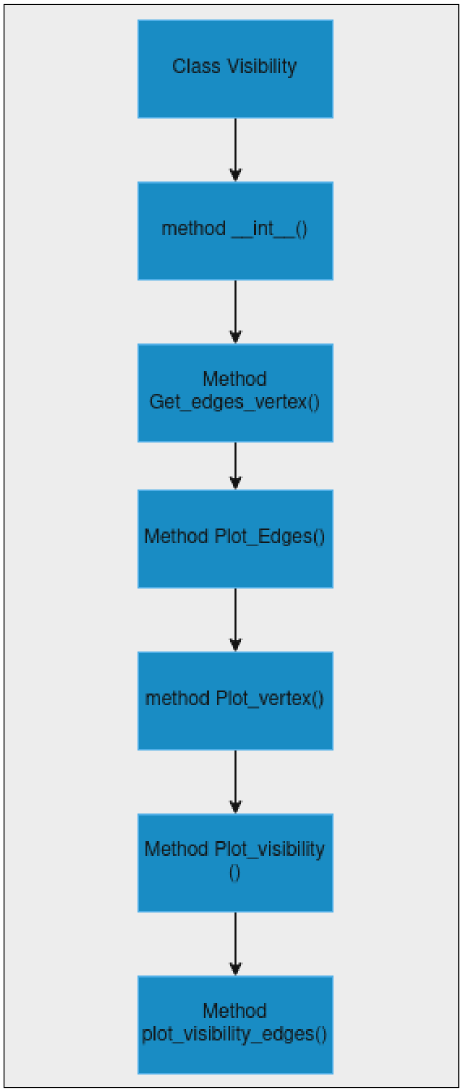
The goal of this laboratory exercise is to develop a visibility graph in a two-dimensional space with obstacles made by polygons that connect a starting point to a destination. To do this, we used the efficient rotational plane sweep (RPS) technique, which we coded in Python. For developing the code, we have used multiple classes with several methods. The rotating plane sweep is a path-planning algorithm that is based on topological maps, making it one of the most powerful techniques in the field of intelligent robot navigation. In the next parts, we will look at the algorithm's underlying concepts, complexities, and consequences. The lab report is divided into four different sections including methodology, result, problem and conclusion.

Methodology

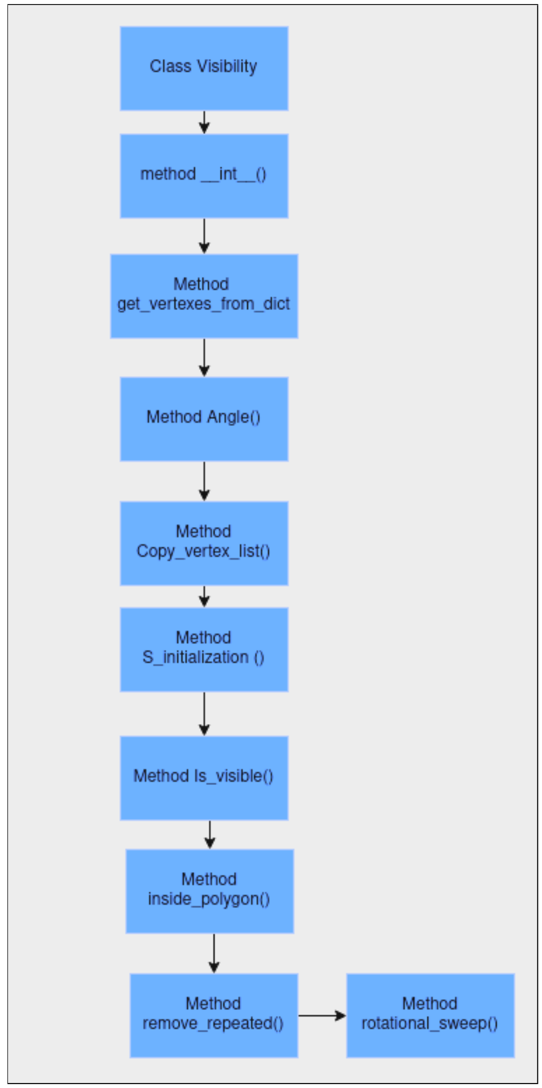
We make use of the four sections of the code as mentioned below:

1. Lab data resources used for Point and Segment classes
2. Vertex and edges manipulation and visualization classes
3. Implementation class for RPS algorithm
4. Main script file (.py)

The flowcharts of all the above classes are shown below:



(a) "Visibility" class and its methods



(b) "Implementation" class and its methods

Figure 1: different environment's output

0.1 Lab data class

We used the visgraph_skeleton.py file provided in the lab data to do the basic functionalities using Point and Segment classes and the provided functions needed for further implementation in the code explained below.

0.2 Plotting and printing class

In this class the first method we get all the required variables as vertexes and edges for our graph's main points to be plotted and printed for reference. The .csv input is first converted to a list then to an array (for manipulations) and to the final dictionary for quick access to keys and values of polygon IDs and their corresponding vertex points. Likewise, the edges are made by using given vertex points to segments using lab data class. Additionally, the second method likewise plots all the existing and potential edges along with the vertices. Lastly, the print function helps store the visited vertex in the counter variable and starts adding the two points of the visibility edge in the visibility list.

```python
# This function reads a csv file to retrieve the vertexes in the environment (start, goal, and obstacles' vertexes)
# It creates a dictionary of lists where each list represents the vertexes in a specific obstacle

    def get_edges_vertexes(self):

        # vertexes from csv
        v = []
        with open(self.map, 'r') as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            next(csv_reader)
            for row in csv_reader:
                v.append(row)

        # list -> arr
        v = np.array(v)
        v = v.astype(float)

        # arr -> dict
        vertexes_dict = {}

        # obstacle identifier, int(i[0])
        # value =[]
        # adds curr vertex (Point) for the corresponding obstacle (using obstacle indentifier) in the vertexes_dict

        for i in v:
            if not (bool(vertexes_dict.get(int(i[0])))):
                vertexes_dict[int(i[0])] = []
            vertexes_dict[int(i[0])].append(Point(i[1], i[2]))

        # This function also computes the Edges of each obstacle considering the vertexes that define it

        E = []

        # loop = obstacles except first(start) and last (goal)
        for i in range(1, len(vertexes_dict) - 1):

          # loop = each obstacle's vertex
            for j in range(0, len(vertexes_dict[i]) - 1):

              # connect= curr vertex -> next vertex
                E.append(Segment(vertexes_dict[i][j], vertexes_dict[i][j + 1]))

                # connect last vertex to the first vertex of the obstacle
            E.append(Segment(vertexes_dict[i][len(vertexes_dict[i]) - 1], vertexes_dict[i][0]))

        return E, vertexes_dict
```

Figure 2: get_edges_vertex() implementation

```python
# plot obstacle edges
    def plot_edges(self, edges):
        for j in edges:
            plt.plot([j.p1.x, j.p2.x], [j.p1.y, j.p2.y], color='black', linewidth=3)

# plot vertexes
    def plot_vertexes(self, vertexes):

      # for how many vertexes plotted?
        counter = 0

        # i = index
        # j = vertex

        for i in range(0, len(vertexes)):
            for j in vertexes[i]:

                plt.scatter(j.x, j.y, color='green', zorder=3)
                plt.annotate(counter, (j.x + 0.1, j.y))
                counter += 1

# visibility edges emanating from each vertex = potential edges
    def plot_visibility(self, visibility_edges):
        for j in visibility_edges:
            plt.plot([j.p1.x, j.p2.x], [j.p1.y, j.p2.y], color='blue', linestyle='dashed')
```

Figure 3: Functions for plotting vertexes and edges

```
# prints a list of edges.
    def print_visibility_edges(self, visibility_edges, vertexes):

        # edge = initial vertex and a final/second vertex.
        a = None  # First vertex
        b = None  # Second vertex
        v_list = []
        for edge in visibility_edges:

          # for which vertex?
            counter = 0
            for i in range(0, len(vertexes)):

              # for ->  a=b=?

              # vertex(x,y)= edge's points (x,y)

                for j in vertexes[i]:
                    if j.x == edge.p1.x and j.y == edge.p1.y:
                        a = counter
                    counter += 1

            counter = 0

            for i in range(0, len(vertexes)):
                for j in vertexes[i]:
                    if j.x == edge.p2.x and j.y == edge.p2.y:
                        b = counter
                    counter += 1

            v_list.append((a, b))
        print( len(v_list), "visibility edges are below:")
        print(v_list)
        return v_list
```

Figure 4: print_visibility_edges() implementation

0.3 RPS Implementation class

In this class, we make the functions necessary for the RPS implementation. Starting from converting vertices into a dictionary with PolyIDs to be used as keys to access the vertices and edges formed. The angle function makes sure the angle is positive between $[0, 2\pi]$. A copy vertex list function is used to make a copy of vertices before their sorting step. The S list is initialized using the S_initialization function to start with the RPS table-making procedure explained in theory lectures by taking the obstacle edges and checking if the intersection between the current vertex and half-line and edge intersection is nearly 0 (meaning) the vertex intersection point is not the same as the current vertex. We then implement the is_Visible() function by following the pseudo code and every required part of it is commented in the code snippets provided below. The inside_polygon() function checks whether the current point v and new point vi lie inside the polygon or not by first checking their polygons and then using midpoint to ensure achieving the results of visibility of v and vi vertices. Moreover, the remove_repeated() function makes sure to remove the duplicate edges from the final visibility graph as shown in the code snippet below in Figure 11. Lastly, the rotational_sweep() function then implements the above functions after it is initialized with vertices and their sorted angles along with initializing the half line of 100 units in x. Resulting, in starting to check the visibility of every vi w.r.t v to add or remove the edges which satisfy the distance between them and v and vi respectively. The sweep line is then rotated

by an angle offset of 0.001 and moving the coordinates of points with x 100 units. This continues for calculating the distance of the sweep line to every edge in S (obstacle edges) to finally sorting the S list. Finally, we plot the potential edges and check the v and vi visibility with the edge in S to conclude if it's visibility edge points to be printed and plotted.
The implementation is labeled below:

```python
# separate vertexes
    def get_vertexes_from_dict(self, v_dict):
        vertexes = []

         # keys = v_dict
         # values =v_dict[i]

        for i in v_dict:
            for i in v_dict[i]:
                vertexes.append(i)
        return vertexes
```

Figure 5: get_vertexes_from_dict() function

```python
#  For each vertex vi calculate αi (the angle from the horizontal axis to the line segment wi).
    def angle(self,y,x):
        angle = np.arctan2(y,x)

         # <0 = -ve angle -> + 2π = to make it +ve

        if angle < 0:
            angle = (angle + 2*np.pi)
        return angle
```

Figure 6: angle() function

```python
# before sorting = save the prev list
   def copy_vertex_list(self,list):
     new_lst = []
     for vertex in list:
       new_lst.append(Point(vertex.x, vertex.y))  # Use the custom Point class
     return new_lst
```

Figure 7: copy_vertex_list() function

```python
# S edges list
    def S_initialization(self,half_line, current_vertex):

        S = []

        for edge in self.obstacles_edges:

          # intersecting pt
            is_intersect = half_line.intersect(edge)
            temp_point= half_line.intersection_point(edge)

              # is_intersect[0] = half line and edge intersection pt
              # and
              # dist.curr/start=dist.temp_point !=0
              # vertex intersection point is not the same as the current vertex.

            if (is_intersect[0] and round(current_vertex.dist(temp_point),0) != 0):
                edge.distance = current_vertex.dist(temp_point)
                S.append(edge)

        # line 14
        S = sorted(S, key=lambda x: x.distance)

        return S
```

Figure 8: S_initialization() function

```python
# isVisible()

    def is_visible(self,v,vi,s, sweep_line):

        # line 8 -> 10
        if len(s) == 0:
            return True


        # If both v and vi lay on the same edge in S, vi is visible from v= line 1->4
        for i in s:

          #  v.dist=vi.dist == 0 (lying on the same INTERSECTING edge)
            if round(v.dist_segment(i),3) == 0. and round(vi.dist_segment(i),3) == 0.:
                return True


        # If vi and v are on the same obstacle and if the midpoint between them is inside the obstacle
        # vi is not visible from v = line 2 -> 4
        if self.inside_polygon(v,vi,s):
            return False


        # If the first edge in S intersect the sweepline going from v to vi, vi is not visible from v = line 11 -> 12
        for edge in s:
            is_interset = sweep_line.intersect(edge)


        # intersecting pt and v(curr/start) is NOT on the same INTERSECTING edge = OBSTACLE
            if is_interset[0] and not(round(v.dist_segment(edge),3) == 0.):
                return False
            else:
                return True
```

Figure 9: is_visible() function

```python
#  inside a polygon

    def inside_polygon(self, v, vi, s):

        #  both vertexes belong to same obstacle

        # id1/2 = obstacle in which v and vi belong = comes from .csv
        id1 = None   #first vertex
        id2 = None   # 2nd vertex

         # vertexes = # obstacles
         # vertexes[i] = vertexes in each obstacle

        for i in range(0,len(self.vertexes)):
            for j in self.vertexes[i]:

                # v, vi = belongs to same obstacle = store in id1, id2
                if (v.x,v.y) == (j.x,j.y):
                    id1 = i
                if (vi.x,vi.y) == (j.x,j.y):
                    id2 = i


         # if both vertexes belong to the same obstacle, the MP bw them is inside an obstacle, vi is not visible from v
        if id1 == id2:

          # create polygon
            poly_path = mplPath.Path(np.array([[vertex.x,vertex.y] for vertex in self.vertexes[id1]]))
            midpoint = ((v.x+vi.x)/2, (v.y+vi.y)/2)
            return poly_path.contains_point(midpoint)
        else:
            return False
```

Figure 10: inside_polygon() function

```python
# rmv repeated edges = from final visibility graph

def remove_repeated(self, visible):
    i = 0
    j = 1
    while i<len(visible) - 1:
        while j<len(visible):
            if (visible[i].p1.x == visible[j].p2.x and visible[i].p1.y == visible[j].p2.y and visible[i].p2.x == visible[j].p1.x and visible[i].p2.y == visible[j].p1.y) :
                visible.remove(visible[j])
                break
            j+=1
        i+=1
        j = i+1

    return [ x for x in visible if not(x.p1.x == x.p2.x and x.p1.y == x.p2.y)]
```

Figure 11: remove_repeated() function

```python
    # RPS
    def rotational_sweep(self):

        vertexes = self.get_vertexes_from_dict(self.vertexes)
        sorted_vertexes = self.copy_vertex_list(vertexes)
        visibility = []

        for k in range(0,len(vertexes)):
            v = vertexes[k] # Vertex = reference = start/curr

           # ε = sort vertex acc to angle
            for point in sorted_vertexes:
                point.alpha(self.angle(point.y-v.y,point.x-v.x))

            sorted_vertexes = sorted(sorted_vertexes, key=lambda x: x.alph)

            # create half line = 100 units
            half_line = Segment(v,Point(v.x+100,v.y))

            # begin S initialization
            S = self.S_initialization(half_line, vertexes[k])
```

Figure 12: rotational_sweep() function - initialization

```python
# start visibility checking of vi wrt v (start/curr)

        for vi in sorted_vertexes:

          # obstacle edges
            for edge in self.obstacles_edges:

                # dist(edge)= dist(vi) =0
                if round(vi.dist_segment(edge),2) == 0. and edge not in S:
                    S.append(edge)

                  # dist(edge)= dist(vi)= dist(v) =0
                elif (round(vi.dist_segment(edge),2) == 0.  and edge in S) or (round(v.dist_segment(edge),2) == 0. and edge in S):
                    S.remove(edge)
```

Figure 13: rotational_sweep() function - S edges add or removal

```python
# MOVE LINE in anticlockwise direction =
# sweep line from vertex v to vi with an angle offset of 0.001 and a magnitude of 100

vi_SL = Point(v.x+(100)*np.cos(vi.alph + 0.001),v.y+(100)*np.sin(vi.alph + 0.001))  # Point (x,y)
sweep_line = Segment(v,vi_SL) # point -> segment

# Calculate the distance of the sweepline to every edge in S (obstacle edges)
for s_edge in S:
    temp_point= sweep_line.intersection_point(s_edge)
    s_edge.distance = v.dist(temp_point)

# Sort the S list with respect which obstacle edge is closer to v
S = sorted(S, key=lambda x: x.distance)

# potential edge
sweep_line1 = Segment(v,vi)

# Check for visibility
if self.is_visible(v,vi,S, sweep_line1):
    visibility.append(Segment(v,vi))
```

Figure 14: rotational_sweep() function - complete

0.4 Main Script

Finally, in this main script, we make the system arguments to be accepted for the environment.csv file. We then begin initializing the graph for the provided environment to get the edges and vertices used to implement the Rotational Plane Sweep (RPS) algorithm which gives the visibility edges and lets us plot all the variables as required for our figure representation. The results of the main script are shown in Figure 5 in the results section.

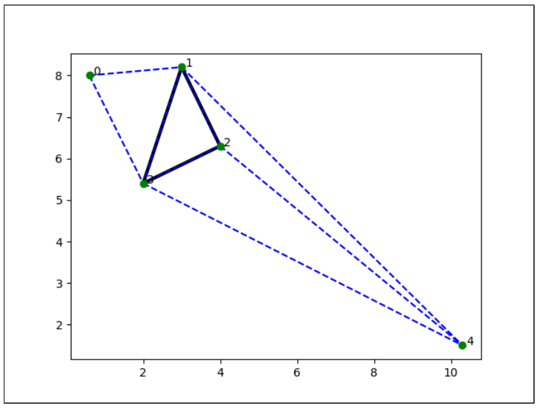
```python
if __name__ == "__main__":

    # argument parser to accept the CSV file path
    parser = argparse.ArgumentParser(description="Visibility Graph Generator")
    parser.add_argument("csv_file", help="Path to the CSV environment file")
    args = parser.parse_args()

    # CSV path from the command-line argument
    csv_path = args.csv_file

    # defining visibility graph
    graph = Visibility(csv_path)

    # edges and vertexes from the visibility graph
    E, vertexes = graph.get_edges_vertexes()

    # Plotting environment's edges
    graph.plot_edges(E)

    # RPS algo
    rps_algorithm = Implementation(vertexes, E)
    visibility_edges = rps_algorithm.rotational_sweep()

    # Plotting the visibility graph edges
    graph.plot_visibility(visibility_edges)

    # Plotting vertexes (in green)
    graph.plot_vertexes(vertexes)

    # Printing the visibility edges as required
    visibility_edges_list = graph.print_visibility_edges(visibility_edges, vertexes)

    plt.savefig('visibility_graph.png')

    plt.show()
```

Figure 15: Main Script

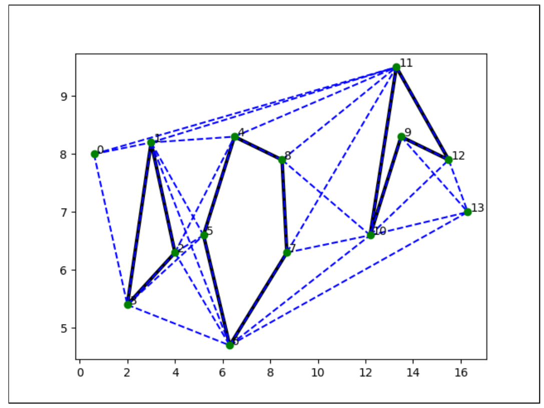
Results

The CSV input files provide the starting coordinates, the destination point, and the vertices that make up the polygonal environment. We opened the file in a data frame and retrieved the important information, which we stored in three distinct lists. We also made separate lists for visibility graphs, points, and edges. Following that, we used the Rotational Plane Sweep (RPS) algorithm from all vertices to produce a complete visibility graph.
The generated visibility graph included duplicate edges. To address this, we sorted each edge's points, making all edges unidirectional, and then removed duplicates. The final visibility graph was superimposed onto the existing environment.
The figures16a, 16b show the given context from the six CSV files, as well as the resulting visibility graph.

Different Environment Outputs



(a) Visibility graph for env-0.csv file



(b) Visibility graph for env-1.csv file

Figure 16: different environment's output

Video submission

Click to watch on YouTube

Problem Faced

The most difficult thing at first was acquiring a good understanding of the underlying concepts. As a result, we faced problems in aligning the is-visible function with the provided pseudo-code. To overcome these obstacles and ensure a faultless solution, we made the necessary changes to both the pseudo-code and the implementation, extensively evaluating it using a variety of test cases.

Discussion & Conclusions

The objective of the lab has been successfully achieved as we have been able to implement the Rotation Sweep Plane algorithm in the 2D place and observed the desired output. During the implementation of the class in function some problems were faced which was successfully recovered by discussing the group members. Learning outcomes from this lab will help us in the future to implement this algorithm where necessary