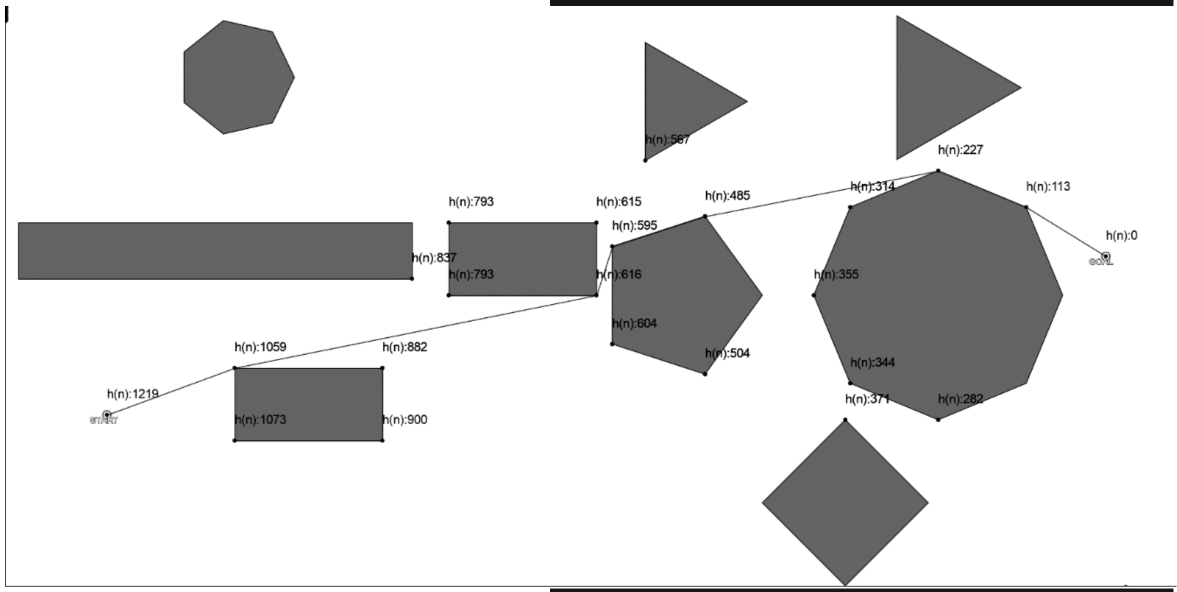
2017

Robot Path Planning



ANURAG SHAHI
ID : 801024552
Intelligent Systems
12/4/2017

Abstract

Robot Path Planning is a research project in which the robot finds an optimal path from source to destination using efficient algorithms. The algorithm used in this project is A Star Algorithm. The robot is blocked with obstacles present in the environment and the robot has to reach the destination avoiding these obstacles.

The main problem is defining an efficient data structure on which A Star Algorithms runs. The most conventional approach is to use a grid based system in which the whole environment is divided into a discrete set of obstacles. But in this project, visibility graph concept has been drawn into picture. The idea is to traverse the robot on a set of edges and vertices based on whether the obstacle vertices is visible by the current vertex. Based on collision detection algorithm the whole graph is constructed and the robot run on the constructed graph.

One of the challenging problem is to devise a collision detection algorithm that detects obstacles and avoid them to find an optimal path from start to goal. The algorithm used in the project is based on line intersection concept that will be detailed in the later section.

TABLE OF CONTENTS

Part 1: Introduction………………………....................................... iv

Part 2: Project Overview…………………………………………… x

Part 3: System Architecture…………………………………………… xi

Part 4: *Programming Language selection and Implementation………* xvi

Part 5: User Interface………………………………………………. xviii

Part 6: Future Perspectives ………………………………………… xxiiiiv

Part 7: Conclusions……………………………………………….. xxvv

Part 8: References…………………………………………………. xxvvi

Appendix A: User Manual………………………………………….. xxviii

Appendix B: Source Code…………………………………………. xxviii

Part 1: Introduction

Path-planning is an important primitive for autonomous mobile robots that lets robots find the shortest – or otherwise optimal – path between two points. Otherwise optimal paths could be paths that minimize the amount of turning, the amount of braking or whatever a specific application requires. Algorithms to find a shortest path are important not only in robotics, but also in network routing, video games and gene sequencing.

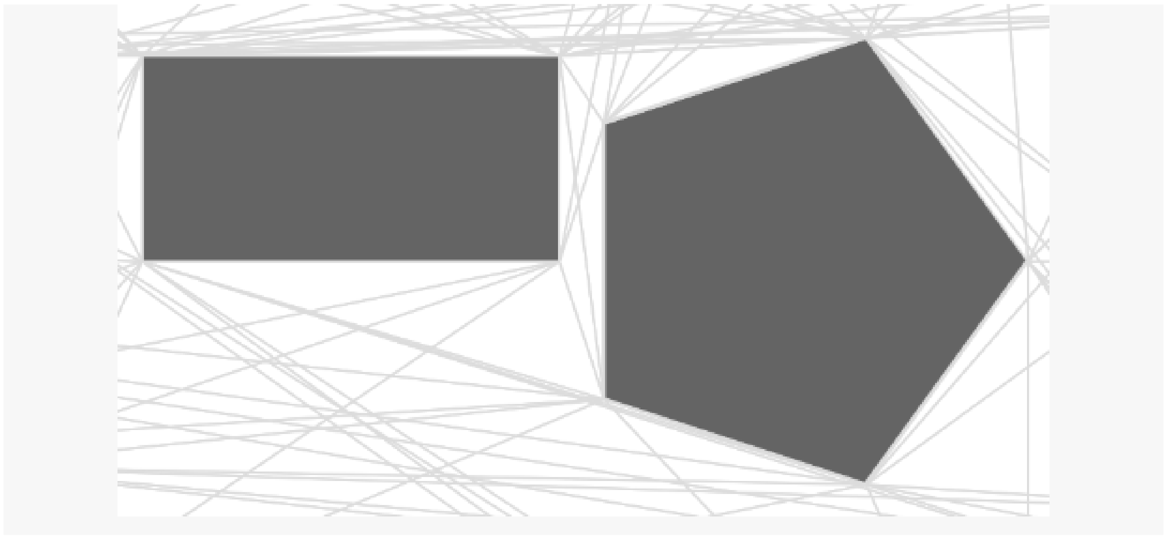
Path-planning requires a *representation* of the environment and the robot to be aware of its *location* with respect to the drawn representations. We will assume for now that the robot is able to localize itself, is equipped with a environment, and capable of avoiding temporary obstacles on its way. The problems addressed in this project are how to create an environment based on the concepts of Polygon, how to localize a robot, and how to deal with uncertain position information. The goals of this project are

- introduce suitable graph representations
- Implement any one of the path-planning algorithms such as A* algorithm.
- Make the environments more dynamic and user friendly.

Graph representations

In order to plan a path, we somehow need to represent the environment in the computer. We differentiate between two complementary
approaches: *discrete* and *continuous* approximations. In a discrete approximation, a map is sub-divided into chunks of equal (e.g., a grid or hexagonal map) or differing sizes (e.g., rooms in a building). The latter maps are also known as *topological* maps. Discrete maps lend themselves well to a *graph* representation. Here, every chunk of the map corresponds to a vertex (also known as “node”), which are connected by edges, if a robot can navigate from one vertex to the other. For example a road-map is a topological map, with intersections as vertices and roads as edges. Computationally, a graph might be stored as an adjacency or incidence list/matrix. A continuous approximation requires the definition of inner (obstacles) and outer boundaries, typically in the form of a polygon, whereas paths can be encoded as sequences of real numbers. Discrete maps are the dominant representation in robotics.

The best approach is to construct a visibility graph which is used in this project. In a conventional approach a grid based system is used, where the environment is discretized into squares of arbitrary resolution, e.g. 1cm x 1cm, on which obstacles are marked. In a probabilistic occupancy grid, grid cells can also be marked with the probability that they contain an obstacle. This is particularly important when the position of the robot that senses an obstacle is uncertain. Disadvantages of grid maps are their large memory requirements as well as computational time to traverse data structures with large numbers of vertices. A solution to the latter problem are topological maps that encode entire rooms as vertices and use edges to indicate navigable connections between them. There is no silver bullet, and each application might require a different solution that could be a combination of different map types.



Visibility Graph

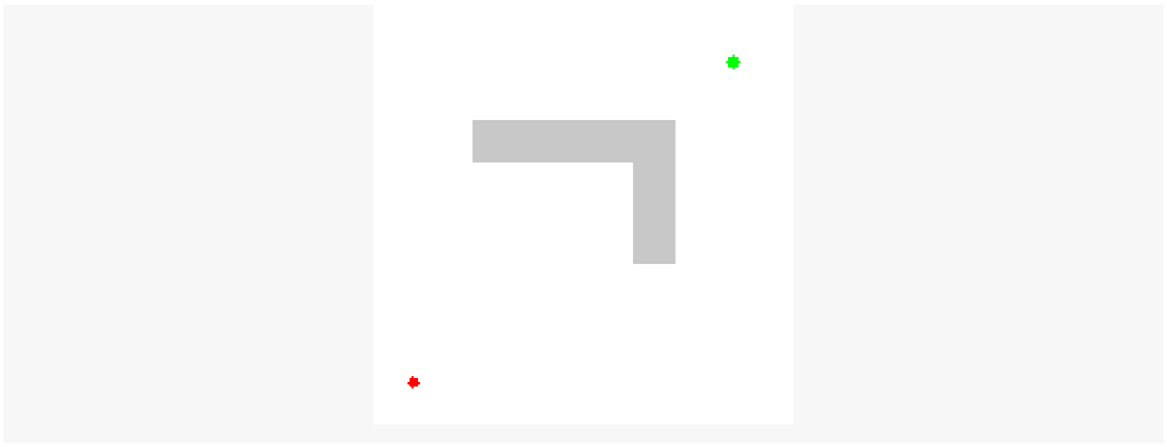
As given in the above diagram each point in the graph vertices traverses across all the obstacles to check if the line segment joining the two vertex is intersected by any polygon.

This gives the idea of implementing a visibility graph. Visibility graphs can be used to find Euclidean shortest paths among a set of polygonal obstacles in the plane: the shortest path between two obstacles follows straight line segments except at the vertices of the obstacles, where it may turn, so the Euclidean shortest path is the shortest path in a visibility graph that has as its nodes the start and destination points and the vertices of the obstacles. Therefore, the Euclidean shortest path problem may be decomposed into two simpler sub problems: constructing the visibility graph, and applying a suitable algorithm such as A* algorithm to the graph. For planning the motion of a robot that has non-negligible size compared to the obstacles, a similar approach may be used after expanding the obstacles to compensate for the size of the robot

There exist also every possible combination of discrete and continuous representation. For example, roadmaps for GPS systems are stored as topological maps that store the GPS coordinates of every vertex.

Path-Planning Algorithms

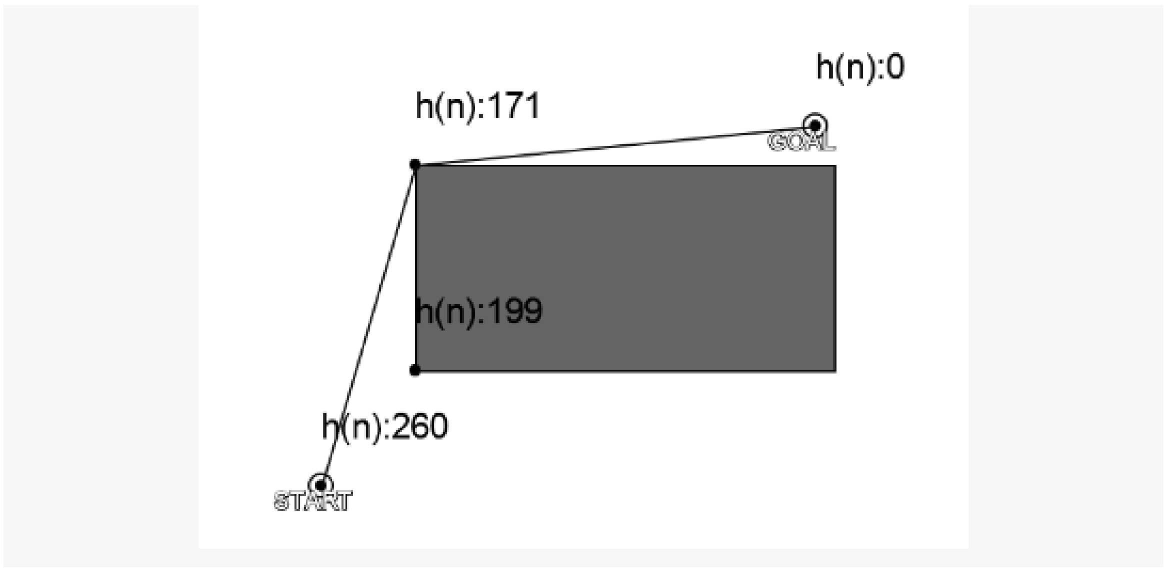
The problem to find a “shortest” path from one vertex to another through a connected graph is of interest in multiple domains, most prominently in the internet, where it is used to find an optimal route for a data packet. The term “shortest” refers here to the minimum cumulative edge cost, which could be physical distance (in a robotic application), delay (in a networking application) or any other metric that is important for a specific application.



Dijkstra’s algorithm for robotic path planning

One of the earliest and simplest algorithms is Dijkstra’s algorithm. Starting from the initial vertex where the path should start, the algorithm marks all direct neighbors of the initial vertex with the cost to get there. It then proceeds from the vertex with

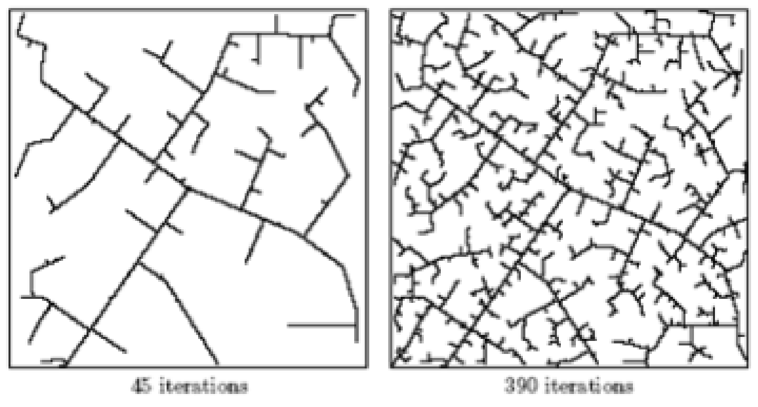
the *lowest* cost to all of its adjacent vertices and marks them with the cost to get to them via itself *if* this cost is lower. Once all neighbors of a vertex have been checked, the algorithm proceeds to the vertex with the next lowest cost. A more detailed description of this algorithm is provided here. Once the algorithm reaches the goal vertex, it terminates and the robot can follow the edges pointing towards the lowest edge cost.
Looking at the animation to the right shows that the algorithm performs a lot of computations that are "obviously" not going in the right direction. They are obvious because we have a birds-eye view of the environment, seeing the location of the obstacle, and we know the approximate direction of the goal. The latter extra knowledge that we have can be encoded using a heuristic function, a fancier word for a "rule of thumb". For example, we could give priority to nodes that have a lower estimated distance to the goal than others. For this, we would mark every node not only with the actual distance that it took us to get there (as in Dijkstra's algorithm), but also with the estimated cost "as the crows flies", for example by calculating the Euclidean distance or the Manhattan distance between the vertex we are looking at and the goal vertex. This algorithm is known as A*. Depending on the environment, A* might accomplish search much faster than Dijkstra's algorithm. The algorithm is provided in pseudo-code here.



A* planning

An extension of A* that addresses the problem of expensive re-planning when obstacles appear in the path of the robot, is known as D*. Unlike A*, D* starts from the goal vertex and has the ability to change the costs of parts of the path that include an obstacle. This allows D* to re-plan around an obstacle while maintaining most of the already calculated path.

A* and D* become computationally expensive when either the search space is large, e.g., due to a fine-grain resolution required for the task, or the dimensions of the search problem are high, e.g. when planning for an arm with multiple degrees of freedom. A more recent development known as Rapidly-Exploring Random Trees (RRT) addresses this problem by using a randomized approach that aims at quickly exploring a large area of the search space with iterative refinement. For this, the algorithm selects a random point in the environment and connects it to the initial vertex. Subsequent random points are then connected to the closest vertex in the emerging graph. The graph is then connected to the goal node, whenever a point in the tree comes close enough given some threshold. Although generally a coverage algorithm (see also below), RRT can be used for path-planning by maintaining the cost-to-start on each added point, and biasing the selection of points to occasionally falling close to the goal. RRT can also be used to take into account non-holonomic contraints of a specific platform when generating the next random way-point. One way to implement this is to calculate the next random point by applying random values to the robot’s actuators and use the forward kinematics to calculate the next point. Most recently, variations of RRT have been proposed that will eventually find an optimal solution. Nevertheless, although RRT quickly finds *some* solution, smooth paths usually require additional search algorithms that start from an initial estimate provided by RRT.



RRT search

Robot embodiment

In order to deal with the physical embodiment of the robot, which complicates the path-planning process, the robot is reduced to a point-mass and all the obstacles in the environment are grown by half of the longest extension of the robot from its center. This representation is known as *configuration space* as it reduces the representation of the robot to its x and y coordinates in the plane. Most planning algorithms also do not consider the orientation of the robot. If this is desired, an additional dimension needs to be introduced into the configuration space.

Other path-planning applications

Once the environment has been discretized into a graph (note, a graph is only *one* possible representation), we can employ other algorithms from graph theory to plan desirable robot trajectories. For example, floor coverage can be achieved by performing a depth-first search (DFS) or a breadth-first-search (BFS) on a graph where each vertex has the size of the coverage tool of the robot. “Coverage” is not only interesting for cleaning a floor: the same algorithms can be used to perform an exhaustive search of a configuration space, such as in the example seen in Lecture #3, where we plotted the error of a manipulator arm in reaching a desired position over its configuration space. Finding a minimum in this plot using an exhaustive search solves the inverse kinematics problem. Similarly, the *same* algorithm can be used to systematically follow all links on a website till a desired depth (or actually retrieving the entire world-wide web).

Part 2: Project Overview

The mission of the project is to provide an environment where user can set the start and goal position and an optimal path between them is drawn. The environment is discrete and vertices of the obstacles are captures and stored for path traversal.   As the project is opened the start and goal position is defaulted to end corners of the canvas. The users can switch between environment and can set start and goal positions. The users also has the flexibility to see the path traversal through g(n), f(n) and h(n) value. Log files are generated and kept att the console level for easier understanding of A Star traversal.

Part 3: System Architecture

Environment Setup

Polygon formulation: Obstacles represent polygons. To draw polygon either all the vertices are specified and edges are drawn or vertices are rotated based on size and radius on the basis of below formula.

```
polygon(x, y, radius, npoints) {
            angle = TWO_PI / npoints;
            for ( a = 0; a < TWO_PI; a += angle) {
             sx = x + cos(a) * radius;
             sy = y + sin(a) * radius;
            AddVertex(sx, sy);
}
```

Collision Detection Algorithm

Collision detection Algorithm is based on simple concepts of line intersection Algorithm.

Step 1: Check if line containing current node and goal node intersect with any obstacles.
Step 2: Capture the line segment with start(x, y coordinates) and stop(x, y coordinates).

Step 3: Traverse the line segment to all the obstacles (polygon in the space).

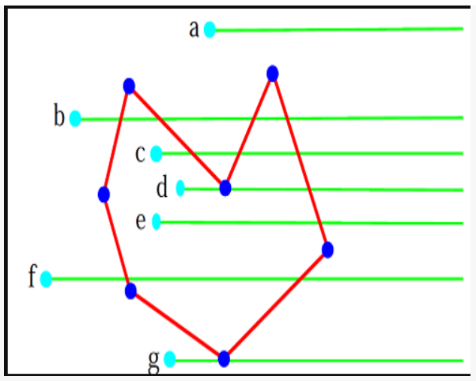
Step 4: Divide the line segment into parts where it intersects with the polygon.

Step 5: For each line segment that is inside the polygon check if the mid-point is inside the polygon as mentioned in step 6.

Step 6: To check if a point is inside a polygon, draw horizontal lines to the right of the point and extends it to the infinity.

Step 7: Count the number of times it intersects with the polygon.

Step 8: A point is inside the polygon of the number of intersections is odd or point lies at the edge of the polygon. If none of the conditions is true, then points lie outside. Example is described below.



Collision Detection

A Star Algorithm.

The most widely known form of best-first search is called **A∗** A **search**
. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost
to get from the node to the goal:

$$f(n) = g(n) + h(n) .$$

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have
f(n) = estimated cost of the cheapest solution through n .

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

Below is the A Star Algorithm as provided in the Wikipedia Page.

```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := an empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)
        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue                    // Ignore the neighbor which is already evaluated.

            if neighbor not in openSet      // Discover a new node
                openSet.Add(neighbor)

            // The distance from start to a neighbor
            //the "dist_between" function may vary as per the solution requirements.
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue                    // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

States: A state description specifies the location of each vertex in the graph and edges associated with the vertex. All obstacles are identified in the environment and the process is repeated for all vertex.

• **Initial state**: For clarity the start and end vertex occupies the corner points in the environment and usually acquires the greatest distance between the environments.

• **Actions**: The simplest formulation defines the actions as event of mouse click. The mouse click can be defined for start or goal position based on user defined testing.

• **Transition model**: Given a state and action, this returns the resulting state; for example, if we change the start or end position based in the environment, A Star is called recursively.

• **Goal test**: This checks whether the path provided is the optimal path

• **Path cost**: Each step costs as the distance between the two vertex, so the path cost is the number of steps in the path.

Heuristic Function

To calculate the heuristic function which is a very important concept in A Star, Euclidean distance is taken into account.

The **Euclidean distance** between points **p** and **q** is the length of the line segment connecting them (p, q).

In Cartesian coordinates, if **p** = (p_1, p_2,..., p_n) and **q** = (q_1, q_2,..., q_n) are two points in Euclidean *n*-space, then the distance (d) from **p** to **q**, or from **q** to **p** is given by the Pythagorean formula:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

<u>Part 4: Programming Language selection and Implementation</u>

4.1 Language Selected: JavaScript

JavaScript , often abbreviated as **JS**, is a high-level, dynamic, weakly typed, prototype-based, multi-paradigm, and interpreted programming language.
Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production. It is used to make webpages interactive and provide online programs, including video games. The majority of websites employ it, and all modern web browsers support it without the need for plug-ins by means of a built-in JavaScript engine. Each of the many JavaScript engines represent a different implementation of JavaScript, all based on the ECMAScript specification, with some engines not supporting the spec fully, and with many engines supporting additional features beyond ECMA.

For ease of doing things P5.js library is used in the project.

Processing (P5.js) Framework

Processing is an open source computer programming language and integrated development environment (IDE) built for the electronic arts, new media art, and visual design communities with the purpose of teaching the fundamentals of computer programming in a visual context, and to serve as the foundation for electronic sketchbooks.

4.2 Google App Engine

The project is deployed on Google App Engine Server. The project url is https://robotpathplan.appspot.com/

4.3 Execution Steps

1) Unzip RobotPathPlanning.zip
2) Go inside robot folder.
3) Double click index.html

Part 5: User Interface

On running the application, below screen appears as shown in Figure 1.

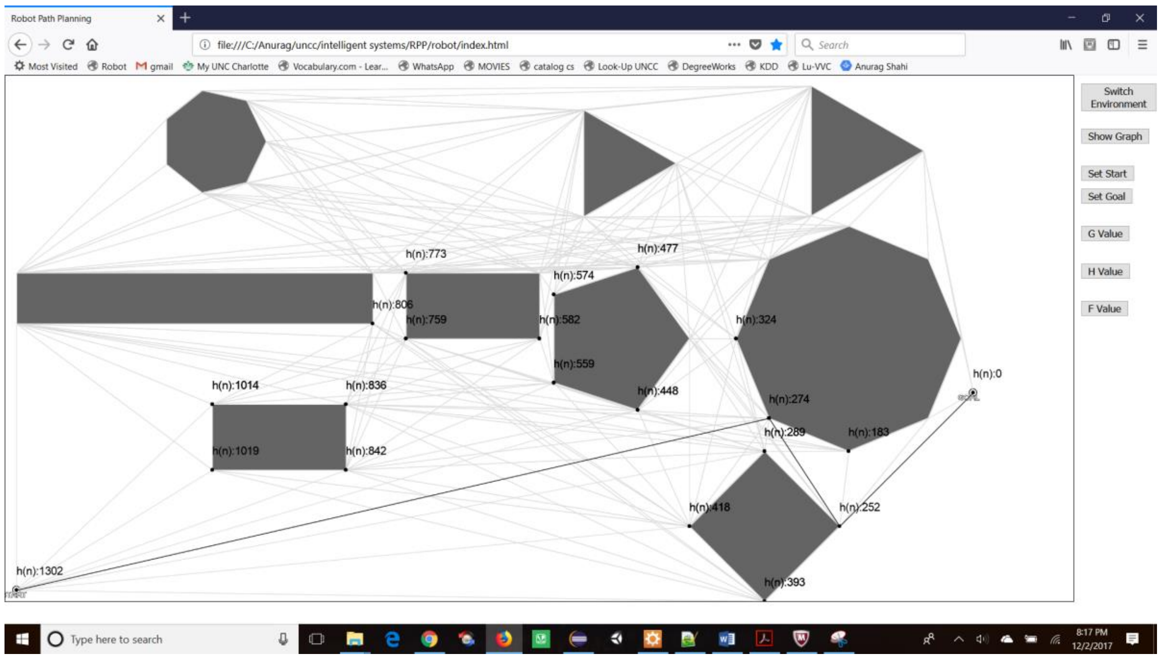


Figure 1:Index.html

The first page shows the default environment. The users can see the robot start and goal positions defaulted to the corners of the canvas.

.

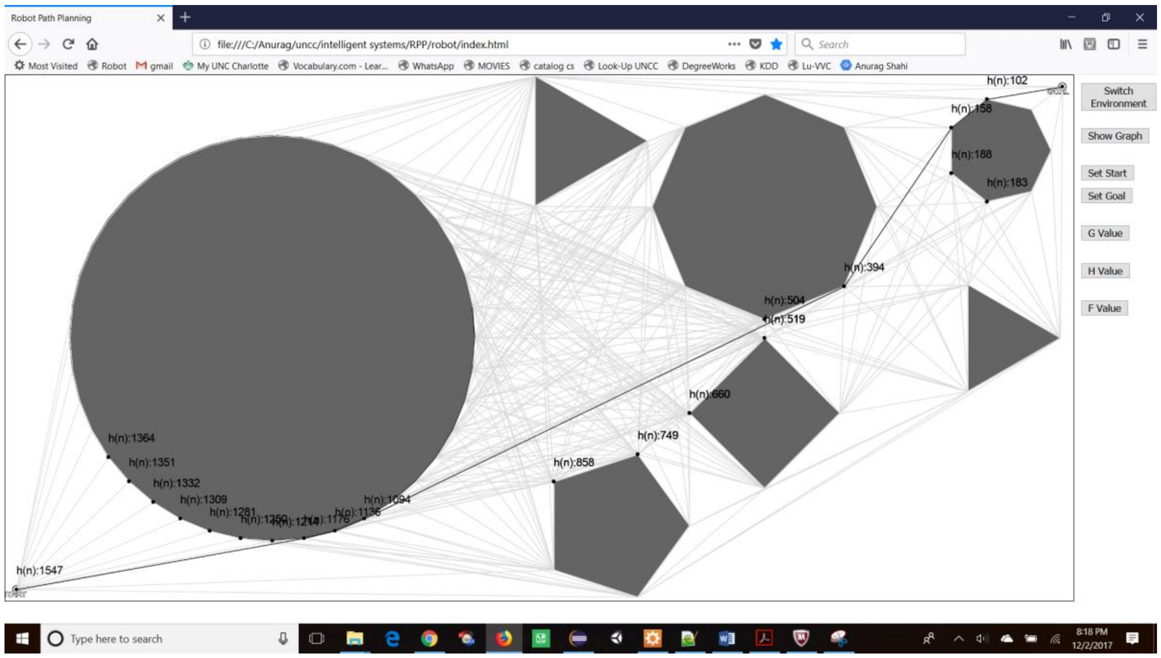


Figure 2: Switch Environment

When the 'Switch Environment' is clicked, the screen appears as shown in Figure 2 where the user can switch to a more complex environment.

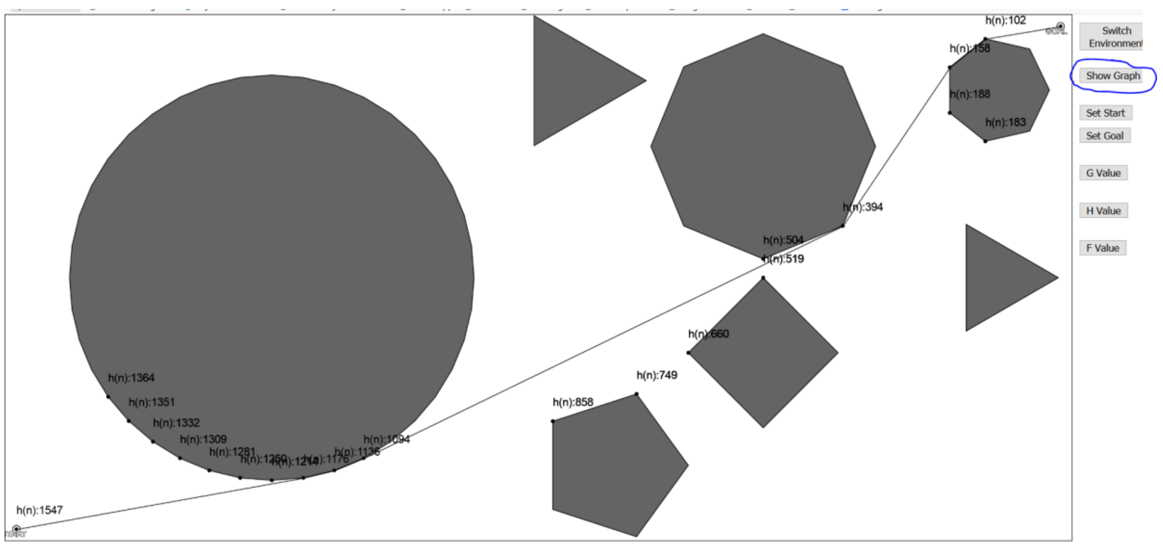


Figure 3: Turn on/off Graph Visibility

Figure 3 shows the canvas without the graph which can be turned on/off by one click of Show Graph Button.

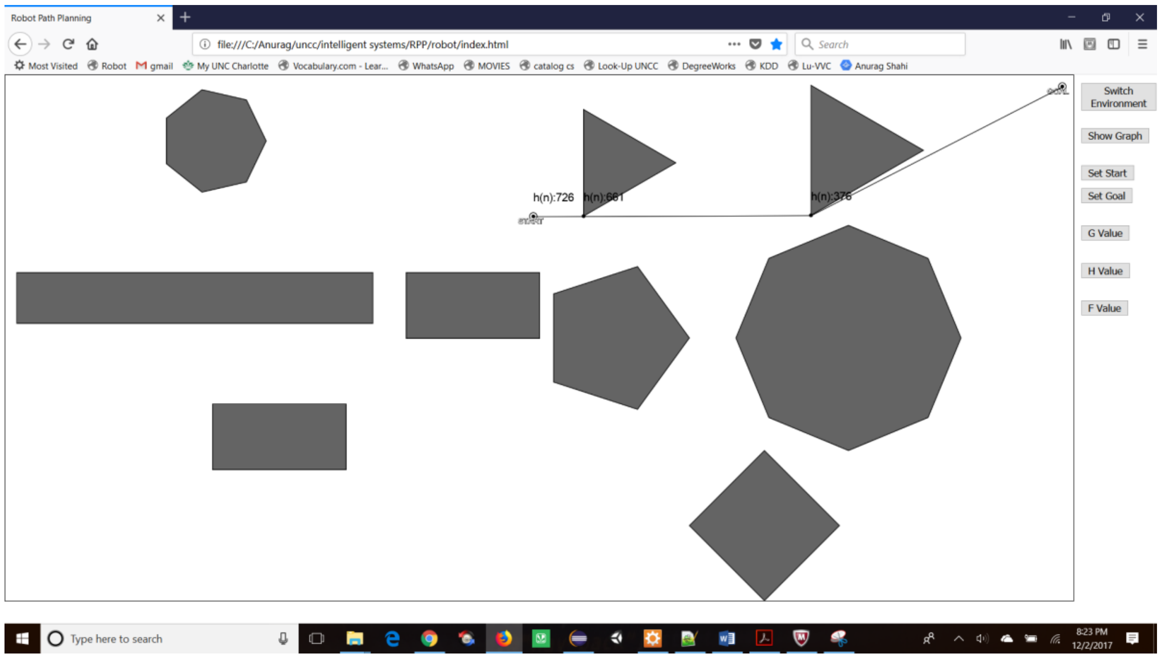


Figure 4: Set Start Position

Figure 4 shows the canvas where user can click and fix the start position of the robot.

Figure 5 below shows the canvas where user can click and fix the goal position.

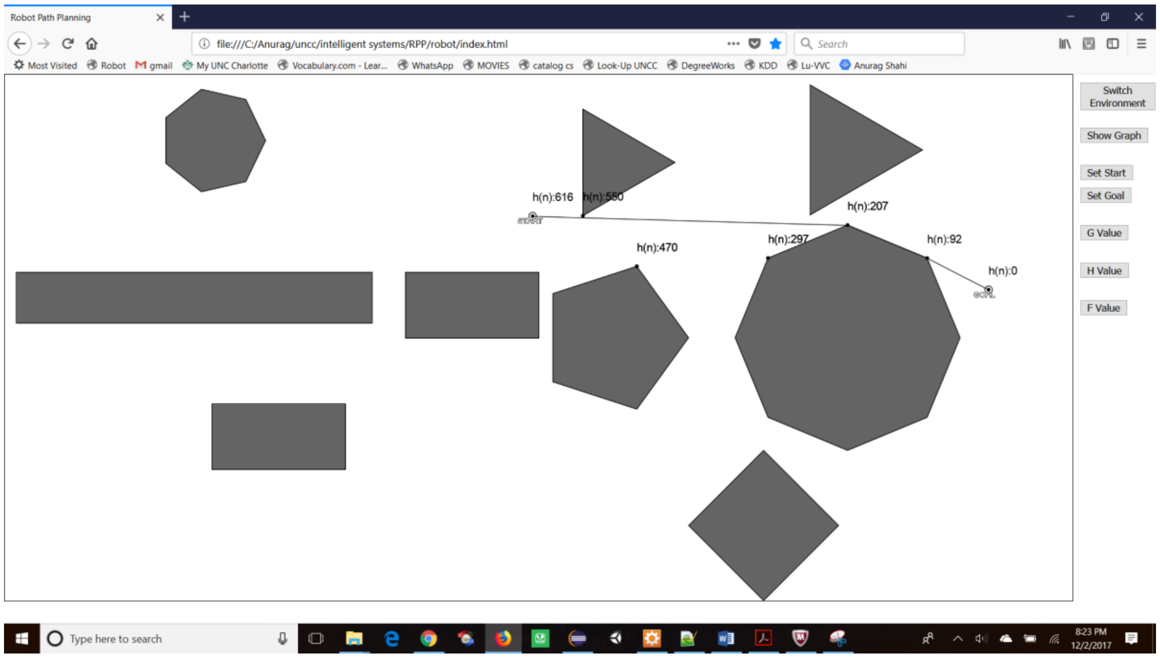


Figure 5: Set Goal Position.

Figure 6 shows the canvas which displays the G value of the nodes explored and path traversed.

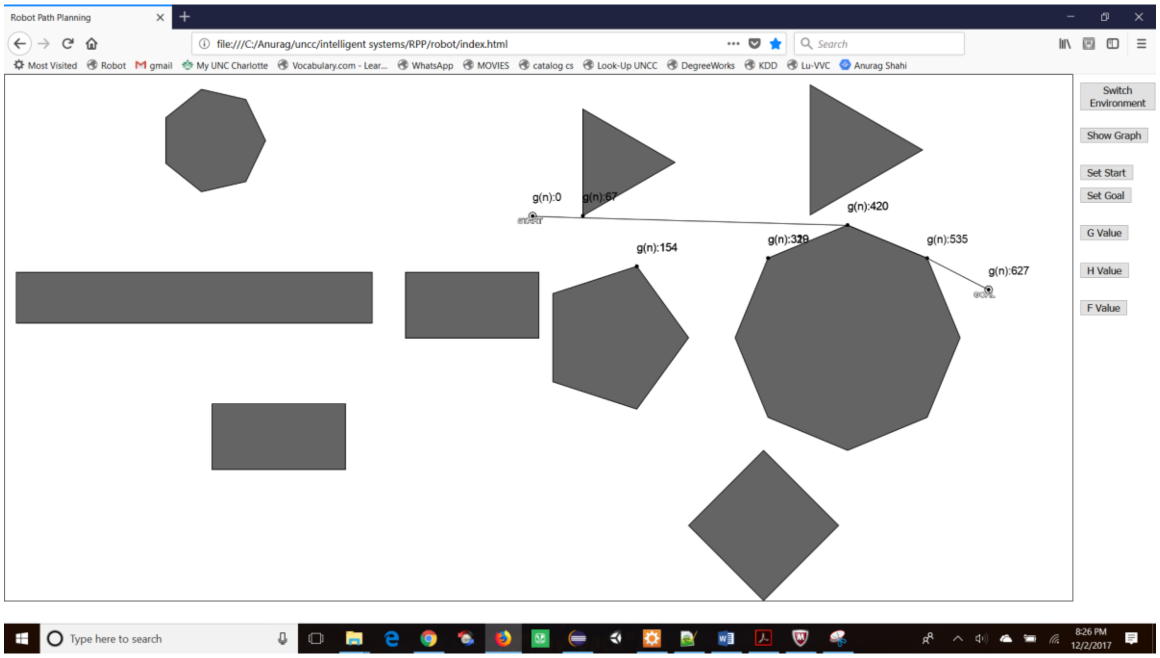


Figure 6: Set G Value

Figure 7 shows the canvas which displays the H value of the nodes explored and path traversed

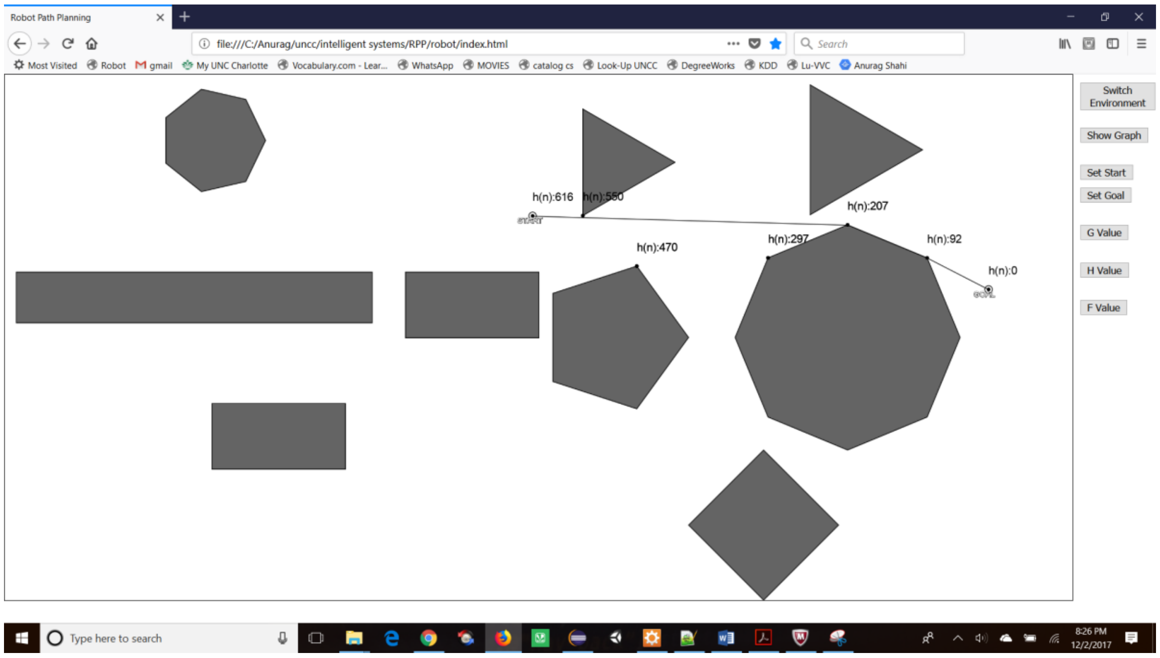


Figure 7: Set H Value

Figure 8 shows the canvas which displays the F value of the nodes explored / path traversed

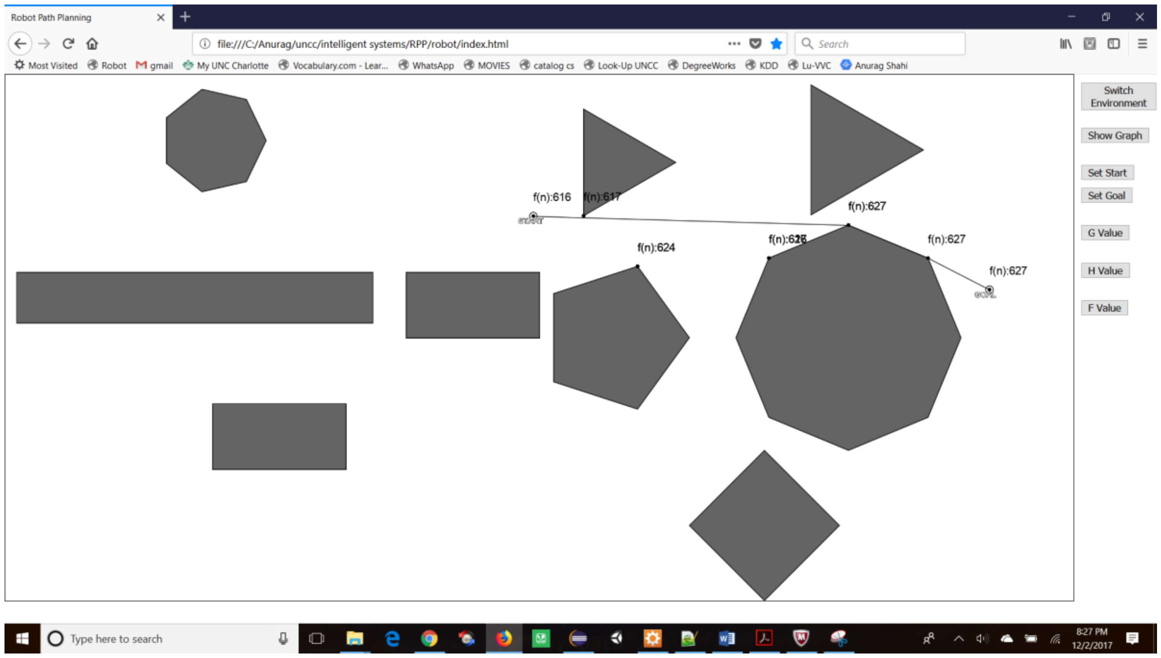
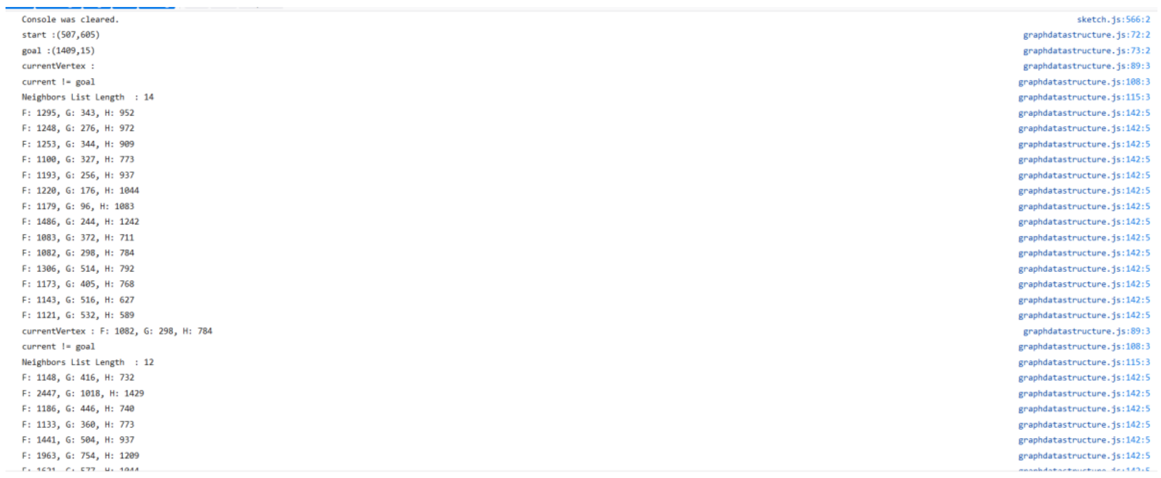


Figure 8: Set F Value



```
Console was cleared.                                 sketch.js:566:2
start :(507,605)                          graphdatastructure.js:72:2
goal :(1409,15)                           graphdatastructure.js:73:2
currentVertex :                           graphdatastructure.js:89:3
current != goal                          graphdatastructure.js:108:3
Neighbors List Length  : 14              graphdatastructure.js:115:3
F: 1295, G: 343, H: 952                  graphdatastructure.js:142:5
F: 1248, G: 276, H: 972                  graphdatastructure.js:142:5
F: 1253, G: 344, H: 909                  graphdatastructure.js:142:5
F: 1100, G: 327, H: 773                  graphdatastructure.js:142:5
F: 1193, G: 256, H: 937                  graphdatastructure.js:142:5
F: 1220, G: 176, H: 1044                 graphdatastructure.js:142:5
F: 1179, G: 96, H: 1083                  graphdatastructure.js:142:5
F: 1486, G: 244, H: 1242                 graphdatastructure.js:142:5
F: 1083, G: 372, H: 711                  graphdatastructure.js:142:5
F: 1082, G: 298, H: 784                  graphdatastructure.js:142:5
F: 1306, G: 514, H: 792                  graphdatastructure.js:142:5
F: 1173, G: 405, H: 768                  graphdatastructure.js:142:5
F: 1143, G: 516, H: 627                  graphdatastructure.js:142:5
F: 1121, G: 532, H: 589                  graphdatastructure.js:142:5
currentVertex : F: 1082, G: 298, H: 784   graphdatastructure.js:89:3
current != goal                          graphdatastructure.js:108:3
Neighbors List Length  : 12              graphdatastructure.js:115:3
F: 1148, G: 416, H: 732                  graphdatastructure.js:142:5
F: 2447, G: 1018, H: 1429                graphdatastructure.js:142:5
F: 1186, G: 446, H: 740                  graphdatastructure.js:142:5
F: 1133, G: 360, H: 773                  graphdatastructure.js:142:5
F: 1441, G: 504, H: 937                  graphdatastructure.js:142:5
F: 1963, G: 754, H: 1209                 graphdatastructure.js:142:5
```

Log File

This contains the log file for the a star algorithm that can be found by Going to console view in Web Browser.

Part 6: Future Perspectives

1. A Star Algorithm and visibility graph approach can be truncated to different algorithm that can be extended to discrete and continuous environment.
2. Instead of A star Algorithm, there are series of algorithms such as Bug2 Algorithm, Dijkstra Algorithm, and RRT and D* Algorithm.
3. The user interface can be changed to allow more control to the user. The user can draw polygons and drag and drop the obstacles.
4. The robot should be able to navigate on the path drawn.

Part 7: Conclusions

The project implementation gives clear picture of the A Star implementation in different environment setup. Astar algorithm is complete and optimal.

The complexity of A∗ often makes it impractical to insist on finding an optimal solution. One can use variants of A∗ that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search.

Part 8: References

1. “Robot Motion Panning” by Jean Claude Latombe
2. Artificial Intelligence A Modern Approach (3rd Edition). By Stuart J. Russell and Peter Norvig
3. Principles of Robot Motion By Kevin M Lynch.
4. Processing Javascript Language. https://p5js.org/examples/.
5. Javascript Tutorials https://www.w3schools.com/
6. http://blog.benoitvallon.com/data-structures-in-javascript/the-graph-data-structure/ Data Structures and Algorithms – Graph
7. https://en.wikipedia.org/wiki/A*_search_algorithm : A Star Algorithm Wikipedia
8. http://www.geeksforgeeks.org/how-to-check-if-a-given-point-lies-inside-a-polygon/ Collision Detection Concept.
9. https://www.youtube.com/user/shiffman/playlists Coding Train Video Tutorials.

Appendix A: User Manual

The list of the files included in the source code

1. Index.html contains references for all the javascript code and libraries.
2. Sketch.js contains the main code where the program executes . This also contains design for user interface using processing.js library framework.
3. Graph.js : for pushing and removing elements from the array.
4. Graphdatastructures.js : For constructing a visibility graph.
5. Visibilitydetection.js : for Collision Detection Algorithm

Appendix B: Source Code

Sketch.js

```
var img;
var graph;
var listObs;
var v1;
var v2;
var env = 1;
var canvas;
var canvasWidth = 1424;
var canvasHeight = 700;
var startX = 15;
var startY = canvasHeight - 15;
var showEdge = true;
var attr = 3;
var goalX = canvasWidth - 15;
var goalY = 15;

var changeStartGoal = 2;
var dataPoints = [];

var firstTime = true;

var EnvPoints = [];

function environment1() {

        EnvPoints.remove(0, EnvPoints.length);

        var box1x = canvasWidth / 8 + 100;
        var box1y = canvasHeight / 8;

        EnvPoints.push([box1x, box1y]);

        var box2x = canvasWidth / 2 + 100;
        var box2y = canvasHeight / 6;

        EnvPoints.push([box2x, box2y]);

        var box3x = canvasWidth / 2 + 100;
        var box3y = canvasHeight / 2;

        EnvPoints.push([box3x, box3y]);

        var box4x = canvasWidth / 2 + 300;
        var box4y = canvasHeight - 100;

        EnvPoints.push([box4x, box4y]);

        var box5x = canvasWidth - 300;
        var box5y = canvasHeight / 2;

        EnvPoints.push([box5x, box5y]);
        var box6x = canvasWidth - 300;
        var box6y = 100;

        EnvPoints.push([box6x, box6y]);

}

function environment2() {

        EnvPoints.remove(0, EnvPoints.length);

        var box4x = canvasWidth / 4 + 400;
        var box4y = canvasHeight / 8;

        var box3x = canvasWidth / 2 + 300;
        var box3y = canvasHeight / 4;

        var box2x = canvasWidth / 2 + 300;
        var box2y = canvasHeight / 2 + 100;

        var box1x = canvasWidth / 2 + 100;
        var box1y = canvasHeight - 100;

        var box5x = canvasWidth - 100;
        var box5y = canvasHeight / 2;

        var box6x = canvasWidth - 100;
        var box6y = 100;

        EnvPoints.push([box6x, box6y]);
        EnvPoints.push([box5x, box5y]);
        EnvPoints.push([box1x, box1y]);
        EnvPoints.push([box2x, box2y]);
        EnvPoints.push([box3x, box3y]);
        EnvPoints.push([box4x, box4y]);

}




function colors(point) {
        var c = color(0, 0, 0);
        // Define color 'c'
        fill(c);
        // Use color variable 'c' as fill color
        noStroke();
        // Don't draw a stroke around shapes
        ellipse(point.x, point.y, 5, 5);
        textSize(15);
        //text("g(n):" + point.g + ",  h(n):" + point.h, point.x, point.y - 20);
        if (attr == 1)
                text("g(n):" + point.g, point.x, point.y - 20);
        if (attr == 2)
                text("h(n):" + point.h, point.x, point.y - 20);
        if (attr == 3)
                text("f(n):" + point.f, point.x, point.y - 20);
}



function polygon(x, y, radius, npoints) {
        dataPoints.remove(0, dataPoints.length);
        var angle = TWO_PI / npoints;
        fill(100);
        beginShape();
        for (var a = 0; a < TWO_PI; a += angle) {
                var sx = x + cos(a) * radius;
                var sy = y + sin(a) * radius;
                vertex(sx, sy);

                var pushData = [sx, sy];
                dataPoints.push(pushData);
        }
        endShape(CLOSE);
        //console.log("data points >> ");
        //console.log(dataPoints);
}



function drawPoly(pol, x1, y1, x2, y2, x3, y3, x4, y4) {

        var p1 = new Vertex();
        p1.x = x1;
        p1.y = y1;
//      p1.type = "OPEN";
        //p1.parent = v1;
        graph.addVertex(p1);

        var p2 = new Vertex();
        p2.x = x2;
        p2.y = y2;
        //p2.type = "OPEN";
        graph.addVertex(p2);

        graph.addEdge(p1, p2);
        graph.addEdge(p2, p1);

        var p3 = new Vertex();
        p3.x = x3;
        p3.y = y3;
        //p3.type = "WALL";
        graph.addVertex(p3);
        graph.addEdge(p1, p3);
        graph.addEdge(p3, p1);

        var p4 = new Vertex();
        p4.x = x4;
        p4.y = y4;
        //p4.type = "OPEN";
        graph.addVertex(p4);
        graph.addEdge(p2, p4);
        graph.addEdge(p4, p2);

        graph.addEdge(p3, p4);
        graph.addEdge(p4, p3);

        pol.addVertex(p1.x, p1.y);
        pol.addVertex(p2.x, p2.y);
        pol.addVertex(p4.x, p4.y);
        pol.addVertex(p3.x, p3.y);
        pol.addEdge();
        fill(100);
        rect(x1, y1, x2 - x1, y3 - y1);

        return pol;
}

function PoltoGraph(pol, data) {

        var prv = null;
        var first = null;
        for (var i = 0; i < data.length; i++) {
                var temp = new Vertex();
                temp.x = data[i][0];
                temp.y = data[i][1];
                //temp.type = "OPEN";
                graph.addVertex(temp);
                pol.addVertex(temp.x, temp.y);
                if (prv != null) {
                        graph.addEdge(prv, temp);
                } else
                        first = temp;

                prv = temp;
        }
        graph.addEdge(temp, prv);
        pol.addEdge();

        return pol;
}

function traverse(path, previous) {
        var previous = previous;

        for (var i = 0; i < path.length; i++) {
                console.log((i + 1) + "->" + path[i]);

                //textSize(15);
                //text("g(n):" + previous.g + ",  h(n):" + previous.h, previous.x, previous.y - 20);
                //fill(0, 102, 153, 51);
                stroke(10);
                line(previous.x, previous.y, path[i].x, path[i].y);
                //var data = [[1, 2], [2, 3]];
                var data = [[previous.x, previous.y], [path[i].x, path[i].y]];
                move(data);
                previous = path[i];
        }

        //textSize(15);
        //text("g(n):" + previous.g + ",  h(n):" + previous.h, previous.x, previous.y - 20);
        //fill(0, 102, 153, 51);

}

function move(data) {
        var x_min = Math.min(data[0][0], data[1][0]);
        var x_max = Math.max(data[0][0], data[1][0]);
        var y_min = Math.min(data[0][1], data[1][1]);
        var y_max = Math.max(data[0][1], data[1][1]);
        ;

        //      var initial = data[0];
        var iteration = 1;

        data.length = 0;

        while (x_min < x_max || y_min < (y_max)) {

                //data.push([initial[0]+i, initial[1]+i]);
                //      console.log(x_min+':'+y_min);

                if (x_max - x_min > 0)
                        x_min = x_min + iteration;
                if (y_max - y_min > 0)
                        y_min = y_min + iteration;
                //              console.log(x_min + "," + y_min + " ------->" + x_max + "," + y_max);

                //      image(img, x_min, y_min, img.width / 8, img.height / 8);
        }

}

function loadDefaultObstacles() {

        //console.log("Env points : "+EnvPoints);

        if (env == 1) {

                var pol1 = new Polygon();

                var x1 = 15;
                var y1 = height / 4 + height / 8;
                var x2 = 15 + width / 3;
                var y2 = height / 4 + height / 8;
                var x3 = 15;
                var y3 = height / 4 + height / 8 + height / 8 - 20;
                var x4 = 15 + width / 3;
                var y4 = height / 4 + height / 8 + height / 8 - 20;

                var pol1 = drawPoly(pol1, x1, y1, x2, y2, x3, y3, x4, y4);
                listObs.addObstacle(pol1);

                var pol2 = new Polygon();

                x1 = width / 4 + width / 8;
                y1 = height / 4 + height / 8;
                x2 = width / 4 + width / 8 + width / 8;
                y2 = height / 4 + height / 8;
                x3 = width / 4 + width / 8;
                y3 = height / 4 + height / 8 + height / 8;
                x4 = width / 4 + width / 8 + width / 8;
                y4 = height / 4 + height / 8 + height / 8;
                pol2 = drawPoly(pol2, x1, y1, x2, y2, x3, y3, x4, y4);
                listObs.addObstacle(pol2);

                var pol4 = new Polygon();

                x1 = width / 2 - width / 4 - 80;
                y1 = height / 2 + height / 8;
                x2 = width / 2 - width / 4 - 80 + width / 8;
                y2 = height / 2 + height / 8;
                x3 = width / 2 - width / 4 - 80;
                y3 = height / 2 + height / 8 + height / 8;
                x4 = width / 2 - width / 4 - 80 + width / 8;
                y4 = height / 2 + height / 8 + height / 8;
                pol4 = drawPoly(pol4, x1, y1, x2, y2, x3, y3, x4, y4);
                listObs.addObstacle(pol4);

        }

        if (env == 2) {
                polygon(width / 4, height / 2, 270, 40);
                var pol10 = new Polygon();
                pol10 = PoltoGraph(pol10, dataPoints);
                listObs.addObstacle(pol10);

        }

        //polygon(width / 8 + 300, height / 8, 70, 7);
        polygon(EnvPoints[0][0], EnvPoints[0][1], 70, 7);
        var pol3 = new Polygon();
        pol3 = PoltoGraph(pol3, dataPoints);
        listObs.addObstacle(pol3);

        //polygon(width / 2 + 300, height / 6, 82, 3);
        polygon(EnvPoints[1][0], EnvPoints[1][1], 82, 3);

        var pol5 = new Polygon();
        pol5 = PoltoGraph(pol5, dataPoints);
        listObs.addObstacle(pol5);

        polygon(EnvPoints[2][0], EnvPoints[2][1], 100, 5);

        var pol6 = new Polygon();
        pol6 = PoltoGraph(pol6, dataPoints);
        listObs.addObstacle(pol6);
```

```
        polygon(EnvPoints[3][0], EnvPoints[3][1], 100, 4);

        var pol7 = new Polygon();
        pol7 = PoltoGraph(pol7, dataPoints);
        listObs.addObstacle(pol7);

        polygon(EnvPoints[4][0], EnvPoints[4][1], 150, 8);

        var pol8 = new Polygon();
        pol8 = PoltoGraph(pol8, dataPoints);
        listObs.addObstacle(pol8);

        polygon(EnvPoints[5][0], EnvPoints[5][1], 100, 3);

        var pol9 = new Polygon();
        pol9 = PoltoGraph(pol9, dataPoints);
        listObs.addObstacle(pol9);

}

function recursiveAStar() {

        var counter = 0;

        //for each (var obc in obj)
        for (var obc = 0; obc < listObs.obstacles.length; obc++) {
                //                      console.log(listObs.obstacles[obc].vertices + "");
                var tempPol = listObs.obstacles[obc];

                if (!listObs.collisionDetection(v1.x, v1.y, v2.x, v2.y, tempPol)) {
                        counter = counter + 1;
                }

                for (var i = 0; i < tempPol.vertices.length; i++) {

                        var pgx = tempPol.vertices[i].x;
                        var pgy = tempPol.vertices[i].y;
                        var pg = new Vertex();
                        pg.x = pgx;
                        pg.y = pgy;

                        /*
                         * CODE CHANGED TO FIX OBSTACLES PAASSTHROUGH
                         */
                        var countSrc = 0;
                        for (var t = 0; t < listObs.obstacles.length; t++) {
                                //          console.log(listObs.obstacles[obc].vertices+"");
                                var tempPolAll = listObs.obstacles[t];
                                if (listObs.collisionDetection(v1.x, v1.y, pgx, pgy, tempPolAll))
                                        countSrc += 1;
                        }
                        if (countSrc == listObs.obstacles.length)
                                graph.addEdge(v1, pg);

                        var countDes = 0;
```

```
                    for (var r = 0; r < listObs.obstacles.length; r++) {
                        //      console.log(listObs.obstacles[obc].vertices+"");
                        var tempPolAll = listObs.obstacles[r];
                        if (listObs.collisionDetection(pgx, pgy, v2.x, v2.y, tempPolAll))
                            countDes += 1;
                    }
                    if (countDes == listObs.obstacles.length)
                        graph.addEdge(pg, v2);

                    // Edge detection

                    for (var e = 0; e < listObs.obstacles.length; e++) {
                        //      console.log(listObs.obstacles[obc].vertices+"");
                        var tempPolyG = listObs.obstacles[e];
                        for (var k = 0; k < tempPolyG.vertices.length; k++) {

                            var countEdge = 0;
                            var ex = tempPolyG.vertices[k].x;
                            var ey = tempPolyG.vertices[k].y;
                            var eg = new Vertex();
                            eg.x = ex;
                            eg.y = ey;

                            var countEdge = 0;
                            for (var s = 0; s < listObs.obstacles.length; s++) {
                                //
        console.log(listObs.obstacles[obc].vertices+"");
                                var tempPolAllE = listObs.obstacles[s];
                                if (listObs.collisionDetection(pgx, pgy, eg.x, eg.y,
tempPolAllE))
                                    countEdge += 1;
                            }
                            if (countEdge == listObs.obstacles.length)
                                graph.addEdge(pg, eg);

                        }

                    }

                    /*
                    * CODE CHANGED TO FIX OBSTACLES PAASSTHROUGH
                    */

                    //      if (listObs.collisionDetection(v1.x, v1.y, pgx, pgy, tempPol))
                    //      graph.addEdge(v1, pg);

                    //if (listObs.collisionDetection(pgx, pgy, v2.x, v2.y, tempPol))
                    //graph.addEdge(pg, v2);

                }
        }

        if (counter == 0)
            graph.addEdge(v1, v2);

        //console.log(graph.vertices[0].toString());
        for (var i = 0; i < graph.vertices.length; i++) {

                var edgeList = graph.edges[graph.vertices[i]];

                for (var j = 0; j < edgeList.length; j++) {

                        //      console.log("G : " + edgeList[j]);

                }

        }

        var path = graph.astar(graph.vertices, v1, v2, "euclidean");

        var previous = v1;
        traverse(path, previous);

}

function preload() {
//      img = loadImage("rob.gif");
        song = loadSound('mario_game.mp3');

}

function visible() {

        if (showEdge == true) {
                showEdge = false;
                setup();
                return;
        }

        if (showEdge == false) {
                showEdge = true;
                setup();
                return;
        }

}

function visibleG() {

        attr = 1;
        setup();
}

function visibleH() {
        attr = 2;
        setup();
}

function visibleF() {

        attr = 3;
```

```
	setup();
}

function changeEnv() {

	firstTime = false;

	startX = 15;
	startY = canvasHeight - 15;

	goalX = canvasWidth - 15;
	goalY = 15;

	for (var i = 0; i < graph.vertices.length; i++) {

		graph.removeVertex(graph.vertices[i]);

	}
	if (env == 1) {
		env = 2;

		environment2();
		//console.log("env2 : " + EnvPoints);
		setup();
		return;

	}

	if (env == 2) {
		env = 1;
		environment1();
		//console.log("env1 : " + EnvPoints);
		setup();
		return;
	}

	alert(env);

}

function setStart() {

	changeStartGoal = 1;
	//console.log("changeStartGoal : " + changeStartGoal);
}

function setGoal() {

	changeStartGoal = 2;
	//console.log("changeStartGoal : " + changeStartGoal);
}

function setup() {

	console.clear();
	canvas = createCanvas(canvasWidth, canvasHeight);
```

```
canvas.style("border", "1px solid");

button1 = createButton('Switch Environment');
button1.position(canvas.width + 10, 10);
button1.mousePressed(changeEnv);

button2 = createButton('Set Start');
button2.position(canvas.width + 10, 120);
button2.mousePressed(setStart);

button3 = createButton('Set Goal');
button3.position(canvas.width + 10, 150);
button3.mousePressed(setGoal);

button4 = createButton('Show Graph');
button4.position(canvas.width + 10, 70);
button4.mousePressed(visible);

button5 = createButton('G Value');
button5.position(canvas.width + 10, 200);
button5.mousePressed(visibleG);

button6 = createButton('H Value');
button6.position(canvas.width + 10, 250);
button6.mousePressed(visibleH);

button7 = createButton('F Value');
button7.position(canvas.width + 10, 300);
button7.mousePressed(visibleF);
//var des_x = width / 12 + 40;
//var des_y = height / 8 -70;

//var des_x = goalX;
//var des_y = goalY;

//

graph = new Graph();

var robotObject = ellipse(startX, startY, 10, 10);
textSize(10);
//fill(0, 102, 153, 51);
text("START", startX - 20, startY + 10);

v1 = new Vertex();
v1.x = startX;
v1.y = startY;
//v1.type = "WALL";
graph.addVertex(v1);

v2 = new Vertex();
v2.x = goalX;
v2.y = goalY;
//v2.type = "OPEN";
graph.addVertex(v2);
var goalObject = ellipse(goalX, goalY, 10, 10);
```

```
		textSize(10);
		//fill(0, 255, 255);
		text("GOAL", goalX - 20, goalY + 10);

		/*var e = new edge();
		e.v1 = v1;
		e.v2 = v2;
		*/
		// console.log(e);

		//console.log("firstTime : "+firstTime);

		if (firstTime)
			environment1();
		listObs = new Collision();
		//code removed

		loadDefaultObstacles();

		recursiveAStar();

		/*	for (var i = 0; i < path.length; i++) {
		 console.log((i + 1) + "->" + path[i]);
		 stroke(10);
		 line(previous.x, previous.y, path[i].x, path[i].y);
		 previous = path[i];
		 }
		 */

}

function mouseDragged() {
		ellipse(mouseX, mouseY, 5, 5);
		// prevent default
		return false;
}

function mouseClicked() {

		if (mouseX < canvasWidth && mouseY < canvasHeight) {
			//	song.stop();
			//song.play();

			if (changeStartGoal == 2) {

				goalX = mouseX;
				goalY = mouseY;

				graph.removeVertex(v2);
			}

			if (changeStartGoal == 1) {

				startX = mouseX;
				startY = mouseY;
```

```
					graph.removeVertex(v1);
				}

				setup();
			}
			/*
			graph.removeVertex(v2);
			v2.x = goalX;
			v2.y = goalY;
			v2.type = "OPEN";
			graph.addVertex(v2);
			var goalObject = ellipse(goalX, goalY, 20, 20);
			recursiveAStar();
			console.log(goalX, goalY);

			*/
			//console.log(obk.test());

			//setup();
			// prevent default
			return false;
}

function draw() {

}
```

Graphdatastructures.js

```
function Graph() {
	this.vertices = [];
	this.edges = [];
	this.numEdges = 0;
}

Graph.prototype.addVertex = function(vertex) {
	this.vertices.push(vertex);
	this.edges[vertex] = [];
};

Graph.prototype.addEdge = function(vertex1, vertex2) {

	var e = new edge();
	e.src = vertex1;
	e.des = vertex2;
	e.cost = Math.round(graph.euclidean(vertex1, vertex2));
	this.edges[vertex1].push(e);
	this.numEdges++;
	if (showEdge) {
		stroke(220);
		line(vertex1.x, vertex1.y, vertex2.x, vertex2.y);
	}

};

Graph.prototype.removeVertex = function(vertex) {

	var index = this.vertices.indexOf(vertex);
	if (~index) {
		this.vertices.splice(index, 1);
	}
	while (this.edges[vertex].length) {
		var adjacentVertex = this.edges[vertex].pop();
		this.removeEdge(adjacentVertex, vertex);
	}
};
Graph.prototype.removeEdge = function(vertex1, vertex2) {
	var index1 = this.edges[vertex1] ? this.edges[vertex1].indexOf(vertex2) : -1;
	var index2 = this.edges[vertex2] ? this.edges[vertex2].indexOf(vertex1) : -1;
	if (~index1) {
		this.edges[vertex1].splice(index1, 1);
		this.numEdges--;
	}
	if (~index2) {
		this.edges[vertex2].splice(index2, 1);
	}
};
Graph.prototype.size = function() {
	return this.vertices.length;
};

/*
Graph.prototype.print = function() {
	console.log(this.vertices.map(function(vertex) {
```

```
                return (vertex + ' -> ' + this.edges[vertex].join(', ')).trim();
        }, this));
};
*/


Graph.prototype.astar = function(vertices, start, goal) {


        var openSet = [];

        start.h = graph.euclidean(start, goal);
        start.f = start.g + start.h;

        openSet.push(start);
        //colors(openSet[0]);
        console.log("start :" + start.toString());
        console.log("goal :" + goal.toString());

        while (openSet.length > 0) {

                // Grab the lowest f(x) to process next
                var lowestCostF = 0;
                for (var i = 0; i < openSet.length; i++) {
                        if (openSet[i].f < openSet[lowestCostF].f) {
                                lowestCostF = i;
                        }
                }
                var currentVertex = openSet[lowestCostF];
                colors(currentVertex);
                console.log("currentVertex : " + currentVertex.debug);

                //console.log("currentVertex :"+currentVertex);
                //console.log("Current Node Debug : " + currentVertex.debug);
                // End case -- result has been found, return the traced path

                if (currentVertex.toString() == goal.toString()) {
                        console.log("start == goal");
                        var curr = currentVertex;
                        var ret = [];
                        while (curr.parent) {
                                ret.push(curr);
                                curr = curr.parent;
                        }
                        return ret.reverse();

                }
                console.log("current != goal");
                openSet.remove(lowestCostF);
                //console.log("open list length  : "+ openSet.length);
                currentVertex.closed = true;

                var neighbors = graph.neighborsList(vertices, currentVertex);

                console.log("Neighbors List Length  : " + neighbors.length);
```

```
				for (var i = 0; i < neighbors.length; i++) {
					var neighbor = neighbors[i];
					if (neighbor.closed) {
						continue;
					}

					var gScore = currentVertex.g + graph.euclidean(currentVertex, neighbor);
					var bestGValue = false;

					if (!neighbor.visited) {

						bestGValue = true;
						neighbor.h = graph.euclidean(neighbor, goal);
						neighbor.visited = true;

						openSet.push(neighbor);
					} else if (gScore < neighbor.g) {
						bestGValue = true;
					}

					if (bestGValue) {
						neighbor.parent = currentVertex;
						neighbor.g = gScore;
						neighbor.f = neighbor.g + neighbor.h;
						neighbor.debug = "F: " + neighbor.f + ", G: " + neighbor.g + ", H: " +
neighbor.h;
						console.log(neighbor.debug);

					}
				}
		}

		// No result was found -- empty array signifies failure to find path
		return [];
};

Graph.prototype.distance = function(pos0, pos1) {
		// See list of heuristics: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

		var d1 = Math.abs(pos1.x - pos0.x);
		var d2 = Math.abs(pos1.y - pos0.y);
		return d1 + d2;
};

Graph.prototype.euclidean = function(pos0, pos1) {
		var x1 = pos0.x;
		var y1 = pos0.y;
		var x2 = pos1.x;
		var y2 = pos1.y;

		var dist = Math.sqrt(Math.pow((x1 - x2), 2) + Math.pow((y1 - y2), 2));
		return Math.round(dist);
};

Graph.prototype.neighborsList = function(vertices, vx) {
		var ret = [];
```

```
	//console.log("To find neighbours");

	for (var i = 0; i < graph.vertices.length; i++) {

		var edgeList = graph.edges[graph.vertices[i]];
		for (var j = 0; j < edgeList.length; j++) {

			if (edgeList[j].src.toString() == vx.toString()) {
				var tempVertex = new Vertex();
				tempVertex = edgeList[j].des;
				ret.push(tempVertex);
			}

		}

	}

	//console.log("neighbors of " + vx + ": " + ret);

	return ret;
};


function Vertex(x, y, type) {
	this.x = x;
	this.y = y;

	this.f = 0;
	this.g = 0;
	this.h = 0;
	this.visited = false;
	this.closed = false;
	this.debug = "";
	this.parent = null;
	this.print = function() {
		return '(' + this.x + "," + this.y + ')';
	};

}

function edge(src, des, cost) {
	this.src = src;
	this.des = des;
	this.next = null;
	this.cost = 0;

}



edge.prototype.toString = function edgeToString() {
	var ret = '[' + this.src.toString() + "->" + this.des.toString() + ']cost(' + this.cost + ')';
	return ret;
};
```

```
Vertex.prototype.compare = function vertexToString() {
            var ret = '(' + this.x + "," + this.y + ')';
            return ret;
};

Vertex.prototype.toString = function vertexToString() {
            var ret = '(' + this.x + "," + this.y + ')';
            return ret;
};
```

Graph.js

```
if (!Array.prototype.indexOf) {
    Array.prototype.indexOf = function(elt /*, from*/) {
        var length = this.length;
        var from = Number(arguments[1]) || 0;
        from = (from < 0) ? Math.ceil(from) : Math.floor(from);
        if (from < 0) {
            from += length;
        }
        for (; from < length; ++from) {
            if (from in this && this[from] === elt) {
                return from;
            }
        }
        return -1;
    };
}

if (!Array.prototype.remove) {
    Array.prototype.remove = function(from, to) {
        var part = this.slice((to || from) + 1 || this.length);
        this.length = from < 0 ? this.length + from : from;
        return this.push.apply(this, part);
    };
}
```

VisibilityDetection.js

```
var COUNTER = 0;
function Point(x, y) {
	this.x = x;
	this.y = y;

	this.Equals = function(p) {
		if (Math.abs(this.x - p.x) > 0.00001)
			return false;
		if (Math.abs(this.y - p.y) > 0.00001)
			return false;
		return true;
	};
	//calculate distance to a point
	this.distance = function(p) {
		var dx = x - p.x;
		var dy = y - p.y;
		return Math.sqrt(dx * dx + dy * dy);
	};
}

Point.prototype.toString = function pointToString() {
	var ret = this.x + ',' + this.y;
	return ret;
};

function Segment(p1, p2) {

	this.p1 = p1;
	this.p2 = p2;

	this.Segment = function(A, B) {
		if (A.x < B.x) {
			p1 = A;
			p2 = B;
		} else if (A.x > B.x) {
			p1 = B;
			p2 = A;
		} else// same x-coordinate
		if (A.y < B.y) {
			p1 = A;
			p2 = B;
		} else {
			p1 = B;
			p2 = A;
		}
	};

	this.midPoint = function() {
		//console.log("IsPointInPol MidPoint");
		var pt = new Point();
		pt.x = (this.p1.x + this.p2.x) / 2;
		pt.y = (this.p1.y + this.p2.y) / 2;

		return pt;
};



this.checkPointinPol = function(x, y) {
        if ((x == p1.x && y == p1.y) || ( x == p2.x && y == p2.y)) {
                return false;
        }
        var minx = Math.min(p1.x, p2.x);
        var miny = Math.min(p1.y, p2.y);

        var maxX = Math.max(p1.x, p2.x);
        var maxY = Math.max(p1.y, p2.y);

        if ((x > minx && x < maxX) || (y > miny && y < maxY)) {

                return true;
        }

        return false;
};

this.Equals = function(s) {
        if (!this.p1.Equals(s.p1))
                return false;
        if (!this.p2.Equals(s.p2))
                return false;
        return true;
};


this.InterSection = function(s) {
        if (this.p1.Equals(s.p1))
                return this.p1;
        if (this.p2.Equals(s.p2))
                return this.p2;

        if (this.p2.Equals(s.p1))
                return this.p2;

        if (this.p1.Equals(s.p2))
                return this.p1;

        var vx1, vy1, vx2, vy2;
        vx1 = p2.x - p1.x;
        vy1 = p2.y - p1.y;
        vx2 = s.p2.x - s.p1.x;
        vy2 = s.p2.y - s.p1.y;
        var t = (vy1 * (s.p1.x - p1.x) - vx1 * (s.p1.y - p1.y)) / (vx1 * vy2 - vx2 * vy1);
        var x = (int)(vx2 * t + s.p1.x);
        var y = (int)(vy2 * t + s.p1.y);
        if (checkPointinPol(x, y) && s.checkPointinPol(x, y))
                return new Point(x, y);
        else
                return null;
};
```

```
        this.checkIntersectionSegment = function(s) {
                if (this.p1.Equals(s.p1))
                        return null;
                if (this.p2.Equals(s.p2))
                        return null;
                if (this.p2.Equals(s.p1))
                        return null;
                if (this.p1.Equals(s.p2))
                        return null;
                                var vx1, vy1, vx2, vy2;

                //console.log("p2 : "+p2);
                //console.log("p1 : "+p1);
                vx1 = p2.x - p1.x;
                vy1 = p2.y - p1.y;
                vx2 = s.p2.x - s.p1.x;
                vy2 = s.p2.y - s.p1.y;
                var t = (1.0) * (vy1 * (s.p1.x - p1.x) - vx1 * (s.p1.y - p1.y)) / (1.0 * (vx1 * vy2 - vx2 *
vy1));
                var x = vx2 * t + s.p1.x;
                var y = vy2 * t + s.p1.y;

                //      console.log("x : "+x);
                //console.log("y : "+y);
                //      console.log("intersection point");
                //console.log("p2 : "+p2 +", p1 : "+p1);
                //console.log(s.checkPointinPol(x,y));
                //console.log(this.checkPointinPol(x,y));
                // (not include end points)
                if (this.isEndPoint(x, y))
                        return null;
                // check if the intersection inside this segment (not include end points)
                if (this.checkPointinPol(x, y) && (s.checkPointinPol(x, y) || s.isEndPoint(x, y)))
                        return new Point(x, y);
                else
                        return null;
        };


        this.isLeft = function(p) {
                var i_isLeft = ((p2.x - p1.x) * (p.y - p1.y) - (p2.y - p1.y) * (p.x - p1.x));
                if (i_isLeft > 0)// p is on the left
                        return 1;
                else if (i_isLeft < 0)
                        return -1;
                return 0;
        };

                this.isEndPointObj = function(p) {
                if (p1.Equals(p) || p2.Equals(p))
                        return true;
                return false;
        };
```

```
                this.isEndPoint = function(x, y) {
                var tmp = new Point(x, y);
                return this.isEndPointObj(tmp);
        };
}

Segment.prototype.toString = function toString() {

        return this.p1 + "->" + this.p2;
};


function Polygon() {

        this.vertices = [];

        this.Edges = [];


        this.addVertex = function(x, y) {

                var p = new Point(x, y);

                this.vertices.push(p);

        };

        this.addEdge = function() {

                var i = 0;
                for ( i = 1; i < this.vertices.length; i++) {

                        this.Edges.push(new Segment(this.vertices[i - 1], this.vertices[i]));

                }

                this.Edges.push(new Segment(this.vertices[i - 1], this.vertices[0]));

                for (var i = 0; i < this.Edges.length; i++) {

                        //      console.log(this.Edges[i].toString());
                }

        };


        this.size = function() {
                return this.vertices.length;
        };


        this.IndexOf = function(p) {
                for (var i = 0; i < this.size(); i++)
                        if (this.vertices[i] == p)
                                return i;
                return -1;
```

```
    };


    this.IsPointInPol = function(point) {
        var j = this.vertices.length - 1;
        var errorVertex = false;

        for (var i = 0; i < this.vertices.length; i++) {
            if (this.vertices[i].y < point.y && this.vertices[j].y >= point.y || this.vertices[j].y
< point.y && this.vertices[i].y >= point.y) {
                if (this.vertices[i].x + (point.y - this.vertices[i].y) / (this.vertices[j].y -
this.vertices[i].y) * (this.vertices[j].x - this.vertices[i].x) < point.x) {
                    errorVertex = !errorVertex;
                }
            }
            j = i;
        }
        // console.log('errorVertex :'+ errorVertex);
        return errorVertex;
    };

        this.Intersect = function(s) {

        breakPoint = null;
        for (var i = 0; i < this.size(); i++) {
            var p1 = vertices[i];
            var d = (i + 1) % this.size();
            var p2 = vertices[d];
            var edge = new Segment(p1, p2);
            breakPoint = s.checkIntersectionSegment(edge);
            if (breakPoint != null)
                break;
        }
            if (breakPoint != null)// then check each part
        {
            var first_part = Intersect(new Segment(s.p1, breakPoint));
            if (first_part == true)// a part intersects means whole segment intersects
                return first_part;
            // if first part doesn't intersect
            // it depends on second one
            var second_part = Intersect(new Segment(breakPoint, s.p2));
            return second_part;
        }
        // cannot split this segment
        else {
            var result = segPartInPol(s);
            return result;
        }
    };

    this.segPartInPol = function(s) {

        // if segment is a edge of this polygon
        var p1_pos = this.IndexOf(s.p1);
        var p2_pos = this.IndexOf(s.p2);
        if (p1_pos != -1 && p2_pos != -1) {
```

```
                        var pos_distance = Math.abs(p1_pos - p2_pos);
                        if (pos_distance == 1 || pos_distance == this.size() - 1)// adjcent vertices
                                return false;
                }
                                var mid = s.midPoint();
                //console.log('mid :'+ mid);
                if (this.IsPointInPol(mid))
                        return true;
                else
                        return false;

        };

        // check if a point is one of this polygon vertices
        this.isVertex = function(p) {
                for (var i = 0; i < vertices.length; i++)
                        if (vertices[i] == p)
                                return true;
                return false;
        };

}

Polygon.prototype.checkSegInPol = function(s, p) {


        // console.log(s+" inside checkSegInPol for time : "+ COUNTER++ +" segment :"+ s);

        var breakPoint = null;
        for (var i = 0; i < p.Edges.length; i++) {

                var edge = p.Edges[i];

                        breakPoint = s.checkIntersectionSegment(edge);
                //    console.log(i+ "  edge : "+edge +", breakPoint :"+ breakPoint);
                if (breakPoint != null)
                        break;
        }


        if (breakPoint != null)// then check each part
        {
                //console.log(" breakPoint : "+ s.p1);
                var first_part = this.checkSegInPol(new Segment(s.p1, breakPoint), p);
                if (first_part == true)
                        return first_part;
                var second_part = this.checkSegInPol(new Segment(breakPoint, s.p2), p);
                return second_part;
        }
                else {
                //console.log(" else segment : "+ s);
                var result = this.segPartInPol(s);
                //console.log(" result : "+ result);
                return result;

        }
```

```
};

function Collision() {

        this.obstacles = [];

        this.addObstacle = function(pol) {

                this.obstacles.push(pol);

        };

        this.collisionDetection = function(x1, y1, x2, y2, polyObject) {

                var p1 = new Point(x1, y1);
                var p2 = new Point(x2, y2);
                var segment = new Segment(p1, p2);

                var isVisible = true;
                for (var i = 0; i < this.obstacles.length; i++) {

                        if (polyObject.checkSegInPol(segment, this.obstacles[i])) {
                                isVisible = false;
                                break;
                        }

                }

                return isVisible;
        };

}
```