A Report on

Path Planning in Single and Multi-Robot Systems

ROBOTICS AND IT'S APPLICATIONS

Submitted to:
PROF NARAYANAMOORTHI M.

Prepared by:
JACOB JOHN (16BCE2205)

Acknowledgement

We would like to start by thanking our mentor and our university for giving us an opportunity to work on this project, under excellent guidance and support.

We would also like to express heartfelt gratitude towards our VIT staff members for constantly helping us out throughout our project.

Last but not the least, we would like to thank our guide PROF. NARAYANAMOORTHI M for his valuable insights and constant support.

ABSTRACT

In the world of Artificial Intelligence, exploration and path finding is one of the major problems in which extensive research is being carried out. Our lab is also actively participating towards development in this field and has devised an exploration and dynamic task allocation algorithm, also as CAC (Cluster, Allocate and Cover). In the smaller maps, CAC's performance is well accepted. However, when the map to be explored becomes very large, the algorithm tends to slow down. One of the major reasons for this is usage of elementary path finding algorithms.

Current work aims to improve the overall efficiency by using a better and more efficient path finding algorithm. In order to achieve the goal, a number of state-of-the art algorithms have been tested in simulation, eventually will be implemented in a controlled real-world environment through multiple-robot systems. So far, one of the more recently developed algorithms, Jump Point Search, has been simulated and shown to be the most effective path finding algorithm. Additionally, this has resulted in reduced time for exploration in an environment for which A* was previously used.

Path Planning in Single and Multi-Robot Systems

1 INTRODUCTION

Path finding is one of the most important problems in the world of artificial intelligence. It is used in game programming, navigation and many militaries as well as domestic applications. Path finding mainly involves finding the optimal path in a known environment. Exploration, a parent to path finding, is covering and mapping of unknown environments. An additional complexity in the problem stems from obstacles that may be present in a given environment (which may be dynamic or static in nature). The goal is to take an object from start to goal in a path avoiding obstacles, in minimum time. Another aspect this research would be dealing with is exploration and coverage. From the execution time and the space requirements computed via simulation, it would be providing the necessary analytical and experimental comparison of the various algorithms which were studied.

This work comprises of implementing some state-of-the-art path planning algorithms on the proprietary in-house simulator, as well as on the commercially available mobile robots.

2 LITERATURE REVIEW

Path Planning is the formal procedure that is followed to find the optimal path when the start and goal point of a robot is given. Pathfinding algorithms are deployed in areas where the map is known before- hand whereas coverage and exploration algorithms are deployed in areas where the map is unknown. Many coverage strategies have been devised in recent years. All of them principally work on frontier coverage. In frontier exploration, one recognizes a list of potential connected grid cells where the robot can move based on certain predefined constraints. The algorithms vary in the way the selection of these cells is made depending on the respective constraints.

Like other coverage algorithms, our lab has also devised one algorithm known as the CAC (Cluster, Allocate and Cover) which is currently using the A-Star algorithm as the pathfinding algorithm. However, it has been observed that as the map to be explored increases in special complexity, a lot of time is being wasted in path planning. This leads to heavy penalty on the time required to executing a given task. Therefore, to reduce the time for executing as given task, computationally faster and time-efficient algorithms are being studied, both in simulations and experimentally.

In recent years, a lot of techniques to find the optimal path have been developed, most of which are variants of the Breadth First Search. The aim of this research is to compare, the performance of various path planning algorithms. The algorithms studied and implemented in this research are Breadth-First Search, Dijkstra, A-Star, D-Star (Dynamic A-Star) and Jump Point Search. The basis of most of them is the flood fill technique. In this technique a circular wave front of the thickness of one cell is expanded. The way it expands is governed by the algorithms. Reduction of the number of cells processed is what most of these algorithms try to do. A comprehensive explanation of how these algorithms work and benchmarking and comparing their performance is what the further sections would be addressing.

3 PATH PLANNING

MAP REPRESENTATIONS

Pathfinding algorithms generally work on graphs or mazes. Map representations makes a huge difference in the algorithm performance. Path Planning algorithms generally have a time complexity worse than linear (that is, if you double the distance needed to travel, it takes more than twice the time to find the path). The fewer the number of nodes to be traversed, the faster the algorithm. Some of the commonly used map representations are as follows:

Graphs

Graphs are mathematical tools used to model relations between objects. They comprise of vertices (or nodes) and edges (connecting two vertices). The representation of a graph is shown as in fig .1. The circles represent the vertices and the lines connecting them represent the edges. This is the most basic representation of maps.

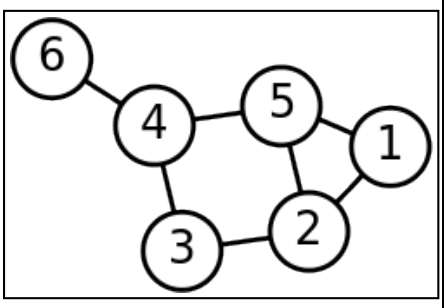


Fig.1 Graph Representation

Grids

Any two dimensional space can be subdivided into regular smaller shapes of a given size. These smaller segments of the whole space are called *tiles* and the whole space is said to represent a *grid map.* In general square, circular or triangular shapes are used to subdivide maps. A cost is usually allocated to each of the tiles and a movement cost is also associated when moving from one tile to another. If the movement costs do not vary across large areas, then using grid representations might be wasteful. Movement across the grid can be made in three basic ways. They are:

Tile Movement

When an object moves from one part to another just by traversing the center of the tiles, the type of motion is called *tile movement*. Tile movement is usually the default choice in a grid. Start and goal points are marked on the tiles and cell cost is assigned to every tile. An object can move from one cell to its four orthogonally adjacent cells by following a certain movement cost and heuristic. Diagonal movement can also be allowed, with the same or higher movement cost. An image representing tile movement is shown in fig.2.

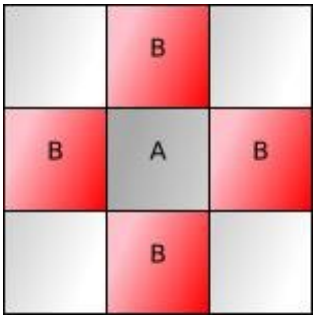


Fig.2 Tile Movement from A ->B

Edge Movement

When an object moves across a map just following the edges of the tiles, the type of motion is called *edge movement.* If the tile size for the given grid representation is very large, edge movement should be preferred. One drawback of this approach is that unlike tile movement, the bot cannot traverse diagonally and hence the path obtained in most cases may be sub-optimal. Representation of edge movement is shown in fig.3.

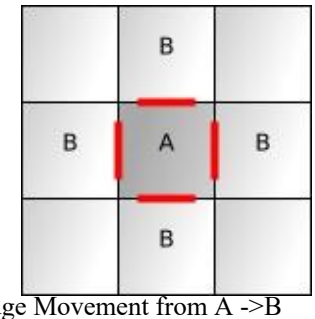


Fig.3 Edge Movement from A ->B

Vertex movement

When an object moves across a map following only the vertices of the tiles, the type of motion is called *vertex movement.* Since we move from corner to corner, vertex movement is the one with least wastage. However it is not applicable to all cases. Vertex Movement is represented in fig.4.

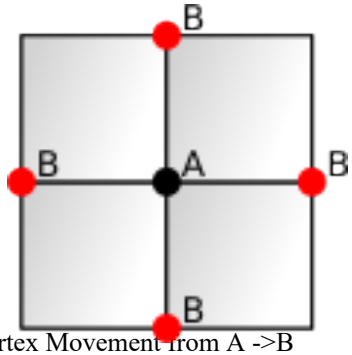


Fig.4 Vertex Movement from A ->B

Polygonal Maps

An alternative to the grid representation is polygonal maps. In particularly large areas, if movement costs are the same throughout and the robots are capable of moving in a straight line rather than following a grid, a non-grid representation may yield a more optimal path than the case of the grid representation. In the given representations (fig.5 and fig.6), the shortest path will be between obstacles' corners. Hence corners (red circles) are chosen as the key "navigation points" points as nodes for the pathfinding algorithms.

This type of representation yields a more optimal path than that of a grid representation but can also get more complex in other cases, where grid representations should be used. This happens in the case when lots of open areas or long corridors are present. In this case connecting every obstacle vertex to every other can result in N2 edges (where N is the number of vertices). This primarily affects space complexity of the algorithm. However, algorithms exist to remove redundant edges of visibility graphs. Hence, depending on the map size and open space ratio, we can prefer polygonal map representations over grid representations.

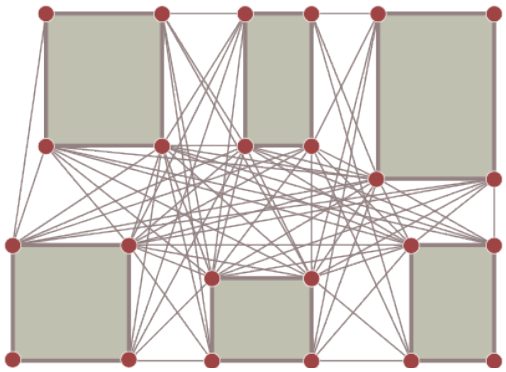


Fig.5 Complex Polygonal maps

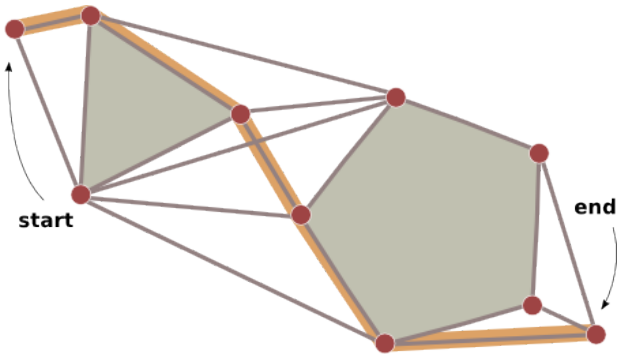


Fig.6 Path from start to end in map

Navigation Meshes

We can represent the walkable areas in a map as a polygons. This type of representation is called a navigation mesh. One major advantage of this representation is that locations of obstacles do not have to be stored. Meshes are somewhat analogous to grids in the method of traversal. A choice can be made whether to move on vertices, edges or centers of the polygons. Hybrid movement is also allowed in this case in which we can move in any of the three ways stated previously. The various representations are shown in fig.7 (a - d).

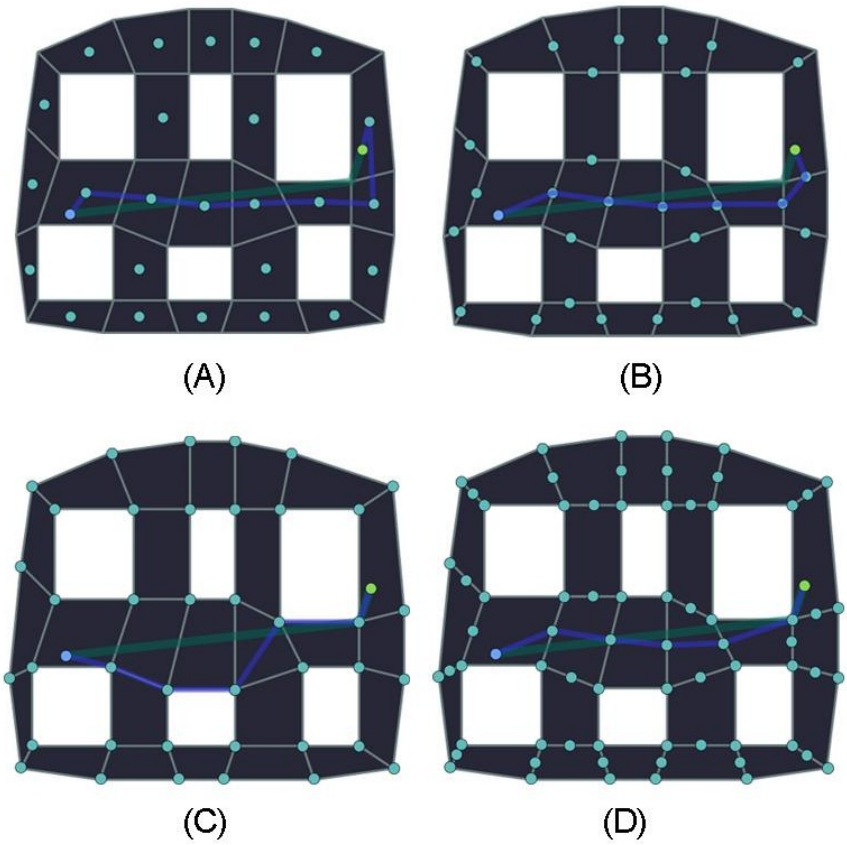


Fig.7 (A) Polygon center movement | (B) Edge Movement | (C) Vertex Movement | (D) Hybrid Movement

PATH FINDING ALGORITHMS

Pathfinding is deployed in areas where the map is known a-priori. Various algorithms have been developed in this area. Most of the path finding algorithms are graph-based algorithms (ie. They work on the graph data structure). This research deals with five of these pathfinding algorithms: Breadth First Search (BFS), Dijkstra's algorithm, A-Star Algorithm, Dynamic A-Star algorithm, and Jump Point Search . The following sections will give a brief of the pathfinding algorithms studied.

Breadth First Search

Breadth First search (BFS) is one of the most basic graph algorithms. It works on the queue data type. A queue is basically a First-In-First-Out (FIFO) data type. In a FIFO data type, the first element added to the queue will be the first one to be removed. In the Breadth First Search (BFS), we keep track of an expanding ring which is known as the *frontier*. The frontier expands as a wave front and this process of covering the surrounding is known as the *'flood fill'* approach. It starts at an arbitrarily specified vertex (usually root) of a graph, and explores the immediate neighbor vertices first, before moving on to the next level of neighbors.

Time and Space Complexity

The Big-O notation is used to determine the time complexity of this algorithm. Assuming the input graph is finite in size, the time complexity of the Breadth First Search can be represented as $O(|V| + |E|)$, where $|V|$ and $|E|$ are the number of vertices and edges of the graph respectively. This can be easily proven as in the worst case; every vertex and edge of the graph will be explored.

The Big- Θ notation is used to determine the space complexity of the algorithm. Assuming finite size of the input graph, the space complexity can be represented as $\Theta(|V| + |E|)$, where $|V|$ and $|E|$ are the vertices and edges respectively. This is self-explanatory as every graph and vertex needs to be stored in the worst case.

Dijkstra's Algorithm

The Dijkstra's Algorithm is similar in implementation to the Breadth First Search. The only difference is that the movement costs are now taken into consideration for different types of movement. In some cases, costs for different types of movements are different. For example, diagonal movement on a grid can cost more than orthogonal movement. Similarly moving on water can be costlier than moving on a road or grass. The Dijkstra's algorithm takes these costs into account and returns a more optimal path than BFS in such cases. Typical implementations of this algorithm use the Priority Queue Abstract Data Type.

Time and Space Complexity

The time complexity of the Dijkstra's algorithm can be represented as $O(|V| \log |V| + |E|)$, where V and E are the number of vertices and edges. The space complexity is represented as $\Theta((|V|+ |E|) \log |V|)$. As was the case with the Breadth First Search, the time and space complexity of Dijkstra's algorithm is also justifiable with a similar logic.

A-Star Algorithm

The A-Star is one of the most popular pathfinding algorithms in use. It is an informed search algorithm which means that it searches in all possible directions to reach the solution in a path that incurs the smallest cost and leads most quickly to the solution. It is formulated in terms of weighted graphs, in which each of the nodes are given weights in according to their priority (which is decided by heuristics). Starting from a specific node of a graph, it constructs a tree of paths and expanding them one at a time, until one encounters the goal node. Hence, it deploys the benefits of both Greedy Best First Search and Dijkstra's Algorithm. Dijkstra's Algorithm guarantees finding the shortest path but wastes a lot of time exploring in non-promising directions. This drawback is covered by the A* algorithm. Like Dijkstra, it also uses the Priority Queue as its data type. The most general Heuristics used in A* are as follows:

Heuristics

The heuristic function feeds the A* algorithm with a cost estimate from a node n to the goal node. A good heuristic function gives a path of better quality.

1. If the heuristic is 0, the A* algorithm turns into the Dijkstra's algorithm which guarantees the shortest path.
2. A* guarantees the shortest path if the heuristic is always lower than the movement cost from n to goal. The lower the heuristic, the more nodes A* expands and the slower it is.
3. A* expands the perfect path and nothing else if the heuristic equals the movement cost from node n to the goal. It results in the fastest operation of A* but this perfect behavior is difficult to achieve.
4. If the heuristic is occasionally larger than the movement cost from the current node (n) to goal, then the A* algorithm does not guarantee the shortest path. However, its performance increases.

Heuristics between two points (x1, y1) and (x2, y2) is given as follows.

- Manhattan Heuristic - $|x1 - x2| + |y1 - y2|$
- Euclidean Heuristic - $\sqrt{((x1 - x2)2 + (y1 - y2)2)}$
- Chebyshev Heuristic - $|x1 - x2| + |y1 - y2| - min(|x1 - x2|, |y1 - y2|)$
- Octile Heuristic - $|x1 - x2| + |y1 - y2| - 0.5857 * min(|x1 - x2|, |y1 - y2|)$

Algorithm

The algorithm maintains two sets: O and C, open and closed set respectively and takes a finite graph as input. The open set (O) uses the priority queue data type whereas the closed set (C) uses the dictionary data type and contains all the processed nodes. A snippet of the algorithm is shown in the snippet below. *Other definitions* include:

- Star(n) - *set of nodes which are adjacent to n*
- $c(n_1,n_2)$ - *length of the edge connecting* n_1 *and* n_2
- g(n) - *total length of a back pointer from n to the start node*
- h(n) - *estimate cost of the shortest path from n to goal*
- f(n) = g(n) + h(n) - *the total estimated cost of the shortest path from start to goal via n*

```
1 function A-Star (Graph, source, goal):
2 repeat
3    Pick n* from Open set (O) such that f (n*)≤ f (n), ∀ n∈ O
4              Remove n* from O and add to Closed Set (C)
5    If n* == goal
6       Exit
7    Expand n*: for all x ∈ Star (n*) that are not in C
8    if x ∈ O then
9                Add x to O
10    else if g (n*) + c (n*, x) < g (x) then
11               update x's back pointer to point to n*
12    end if
13    Until O is empty
```

Snippet of the A* algorithm

D-Star or Dynamic A* Algorithm

The algorithms before this considered only static environments for motion. However in real world applications, the environment keeps on changing in the sense that free tiles convert into obstacles and vice versa. To tackle this problem, we can use two approaches:

1. Use Dijkstra's algorithm or A* and calculate the initial path from the start to the goal and then follow that path until a dynamic obstacle makes an unexpected change (for example, a free tile converted to obstacle). When this happens, the bot can simply re-invoke A* and compute a path from its current position to the goal. However, this can be a computationally intensive and inefficient process if the number of changes is a lot in number.
2. Use Dijkstra (or A*) to plan the initial path and change the heuristics of the neighboring tiles once an anomaly is encountered in the original path. This method is known as the D* algorithm.

The D* algorithm uses a complex approach in which it redefines the heuristic on encountering an obstacle. This paradigm is explained in and interested readers may refer to it.

Jump Point Search

Most techniques used to speed up A* focus on lowering the number of nodes the algorithm has to account for. In a square grid with uniform costs looking at all the individual tiles can be detrimental to the performance of the algorithm. Building a graph of key coordinates (jump points) and using that when finding a path can be one approach. Jump Point Search moves ahead on the grid skipping tiles using two basic pruning algorithms. These pruning algorithms are described in the section below. When the algorithm considers neighbors of the current node for including them in the priority queue (O), the algorithm moves ahead to nodes visible to the robot, further away from the current node. There are lesser number of more expensive steps, hence reducing the count of nodes in O (the priority queue).

Pruning Algorithms

The pruning algorithms are the base of Jump Point Search. They basically reject nodes that need not be visited in order to find the best path. The two algorithms are as follows:

Neighbor Pruning

Currently, node x is being expanded. Arrows indicate the direction of travel (i.e. from its parent): straight or diagonal. In both cases, we can neglect the grey neighbors as these can optimally be reached from x's parent without passing through node x.

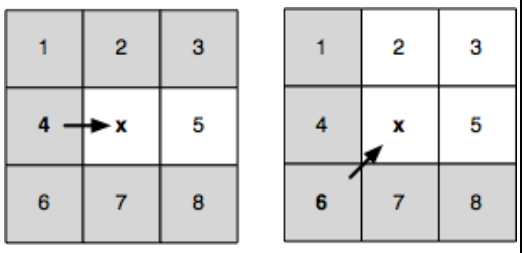


Fig.9 Neighbor Pruning

Forced Neighbors

Currently, node x is being expanded. Arrows indicate the direction of travel (i.e. from its parent): straight or diagonal. When x is adjacent to an obstacle the encircled neighbors cannot be pruned; any alternate path which is optimal, from the parent of x to each of these nodes, is blocked.

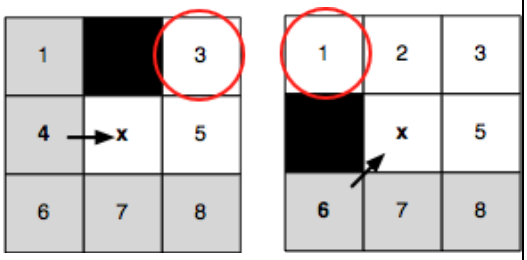


Fig.10 Forced Neighbor

These pruning rules are applied during search as follows: we recursively prune the set of neighbors around each node instead of generating natural and forced neighbors. The objective here is to remove symmetries by recursively "jumping over" all nodes which can be optimally reached via a path not visiting the current node. The recursion is stopped when an obstacle is hit, or a jump point successor is found. Jump points are significant as they have neighbors which cannot be reached by an alternate path. The optimal path has to go through the current node.

Performance

Jump Point Search is better than other algorithms for a number of reasons, some of them being:

I. It is optimal and the path quality is near ideal
II. It involves no preprocessing
III. No memory overheads are involved
IV. It speeds up A* nearly 10 times giving a reasonably better performance than approximate techniques such as the IDA*.

VARIATIONS OF THE A* ALGORITHM

Many variants of the A* algorithm exist which tend to decrease the number of nodes the algorithm is processing in order to make it a bit faster. Some of the recent practices include:

Iterative Deepening (IDA*)

In Iterative Deepening the algorithm starts with an approximate value for the answer and makes it more and more accurate by subsequent executions. The algorithm tries to make the search graph deeper by predicting further moves. The iteration stops when the answers plateau at some point and show little or no change. The "depth" is a threshold for the total cost value (f) in this kind of approach. The node isn't processed when this value is considerably large. After every pass the number of processed nodes increases. If the path is found to improve, continue to increase the cutoff; otherwise, stop.

One major drawback of this approach is that it tends to increase execution time while reducing space requirements.

Dynamic Weighting

In this approach we tend to reach anywhere quickly in the beginning and then when the goal is near, we give getting to the goal a priority. This can be represented as:

$$f(p) = g(p) + w(p) * h(p)$$

The weighting factor (w) generally greater than 1 is used to adjust the A* algorithm. The weight is decreased as one gets closer to the goal, decreasing the weightage of the heuristic, and increasing that of the actual cost of the path. Hence, we tend to find the goal faster.

Bidirectional Search

Unlike A* in which we search from the start to the finish, in this approach we start two parallel searches from start to goal, and from goal to start. Both these searches meet at a point and the path obtained is of better or approximately same quality as the original path. The idea behind is based on the divide and conquer approach. Hence, it's preferable to use two smaller search trees rather than a big tree for searching an element.

Instead of choosing the best search node in the forward direction "$g(start, x) + h(x, goal)$)" or in the opposite direction "$g(y, goal) + h(start, y)$", we choose a pair having best - "$g(start, x) + h(x, y) + g(y, goal)$".

Simultaneous forward and backward searches are cancelled by retargeting. For a small span of time, the algorithm performs a search in the forward direction. Once it chooses the best node in this direction, it performs a search in the backward direction looking for the node obtained by the forward search. Once over, the algorithm performs the same step again, but this time performs the backward search first, chooses a node and then performs a forward search to the chosen node. This cycle continues until the searches meet at a common point.

4 PERFORMANCE ANALYSIS

TIME COMPLEXITY

The number of instructions a program executes during its execution is called its *time complexity*. It depends on the size of the input and the algorithm used primarily. It quantifies the time taken to complete execution and output the result. It is estimated by counting the number of elementary operations such as addition, subtraction, etc. performed by the algorithm multiplied by the time to perform each of these operations. The time taken and the number of elementary operations differ by at most a constant factor.

Running time complexities are expressed in the *Big-O notation*. O is called the *Landau's symbol*. O(n) actually stands for a set of functions for which the function 'n' multiplied by a scaling factor is an upper bound. It can be represented as: $f(n) \leq c \cdot n2$.

Since an algorithm's execution time varies with different inputs of the same size, we generally use the worst-case time complexity, which is defined as the maximum time taken to execute the program with any input of size n. For graph algorithms as stated above, the time complexity is usually a function of the number of vertices (V) and the number of edges (E) of the graph. Similarly, a program that computes: $(x) = a_0x^3 +$
$a_1x^2 + a_2x^1 + a_3$, has a time complexity of O(x^3) because in the worst case, as x increases, x^3 is the most dominating factor in f(x). Hence, the complexity. Similarly, for graph algorithms like BFS and Dijkstra's
algorithm, the time complexity is given in their accompanying descriptions.

SPACE COMPLEXITY

The *space complexity* is the number of memory units an algorithm allocates while execution. An efficient algorithm uses the least space possible. There is often a compromise involved regarding execution time and space taken. A problem cannot always be solved in a small time and having minimum memory requirement. Hence a compromise has to be made and depending on the availability of space / time the algorithm is modified.

The space complexity is estimated by counting the space taken by each of the elementary operations and multiplying it by the number of such operations. An algorithm's space complexity varies with different input sizes; however, one uses the worst-case space complexity of an algorithm which is defined as the maximum size taken by the algorithm. It is generally represented by the *Big-Θ notation.* As was the case with time complexity, the space complexity of graph algorithms also depends on the number of edges and vertices of the graph in use.

PROGRAM PROFILING

Profiling of a program is breaking the program line by line and finding the parts which decrease the efficiency of the program . Once found, ways are devised to optimize these bottlenecks so that program efficiency is higher.

This can be very helpful as it gives us the data as to where the program needs to be optimized. Local optimizations are necessary as well. These consist of replacing in-built functions and data structures with ones which may be optimized for the particular kind of usage. Trying to optimize a program without measuring where it is spending time during execution is purposeless and leads to wastage of time.

In this research, focus has been laid on CPU utilization profiling, meaning the time spent by each function executing instructions. We can also do memory profiling which would give us a measure of the memory used by every line but that is unnecessary at this point of time.

Many tools are readily available for code profiling in almost all the languages.

Python Call Graph

The programming language used to program the simulator as well as the bots was Python. The code profiler used gives a visual representation of the number of calls to each of the functions, and the execution time per cycle per call for each.

The tool is a freely available open source library known as the *Python Call Graph* which creates graph visualizations for python applications.

To initialize the tool, we first need to install it and then execute it as follows:

I. Open Terminal and enter folder where the code is present
II. Execute the command: py callgraph graphviz -- ./ <program name>.py
III. The output would be stored as a '*.png' image in the same folder

5 SIMULATION RESULTS

The above algorithms were implemented in python and tested on a simulator proprietary to the Lab. Apart from testing the code on the simulator, they were also optimized and profiled using the tool as stated above. The simulator is programmed in Python. The dark blue cells represent the obstacles in the environment. The light purple cells represent tiles which can be traversed. The sky-blue cells represent the cells which have been visited in search for the goal.

In the following results, the figures on the left-hand side shows the path which is obtained whereas the ones on the right show the cells that were explored (or processed) in doing so. The number of visited cells and those in the path are obtained by profiling and are shown in table 1

The results are as follows:

BREADTH FIRST SEARCH

The Breadth First Search was run on the simulator and the result is as follows:

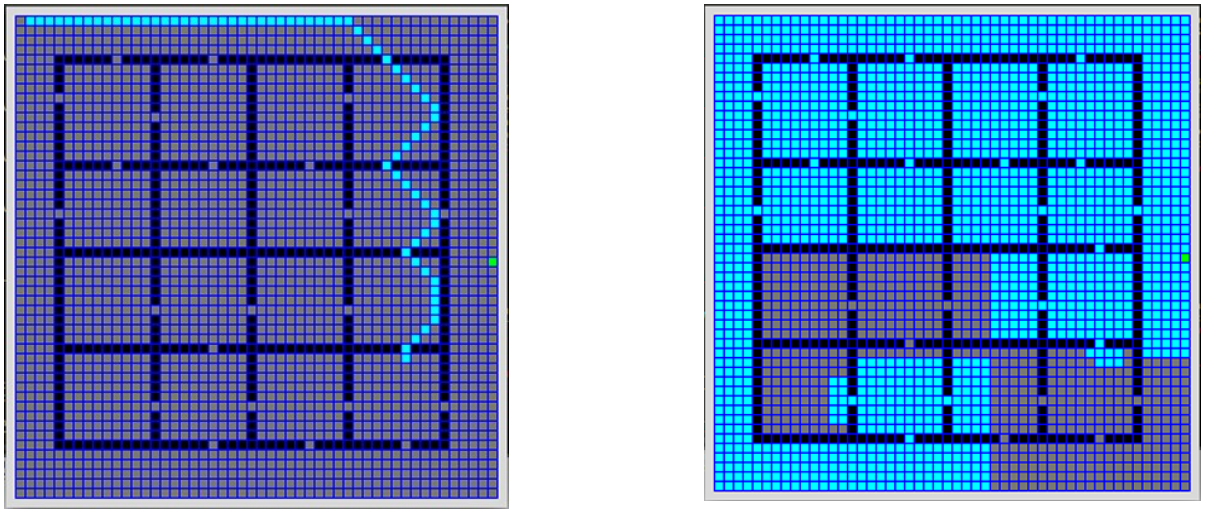


Fig.11 Breadth First Search: (A) The Path Found (B) the cells explored

DIJKSTRA'S ALGORITHM

The Open Set is the Priority Queue as described in the algorithm. The Total length is the list of processed nodes (closed set) + the open set. The execution time for the simulator is expressed in seconds and shown in table 1 . The cells processed and the path found is also given in the figure below. The results for the Dijkstra's Algorithm are as follows:

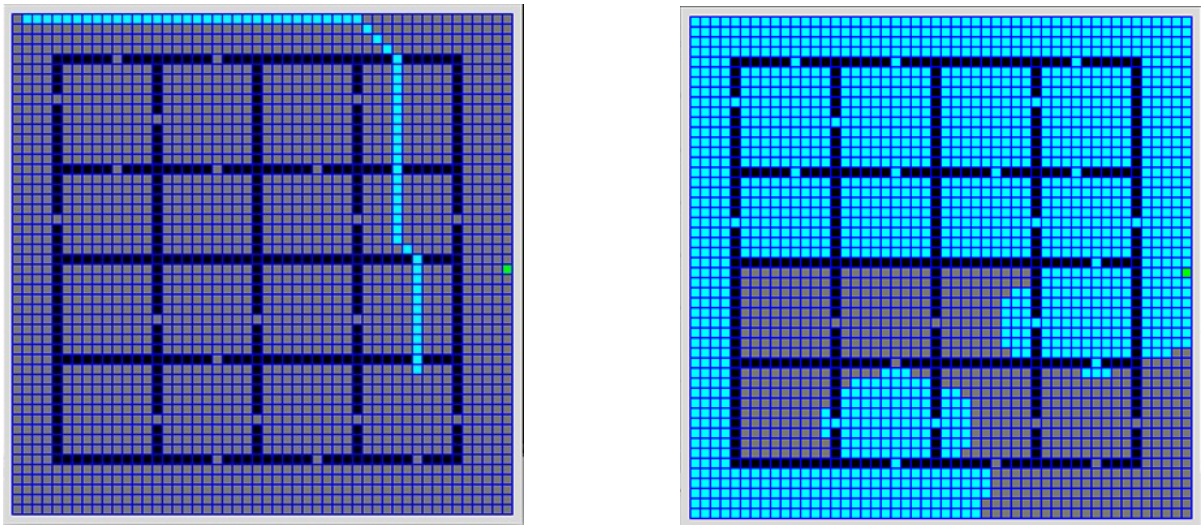


Fig.12 Dijkstra's Algorithm: (A) The Path Found (B) the cells explored

A* ALGORITHM

Open set and Total length are the same as in Dijkstra's. The results for A* are as follows:

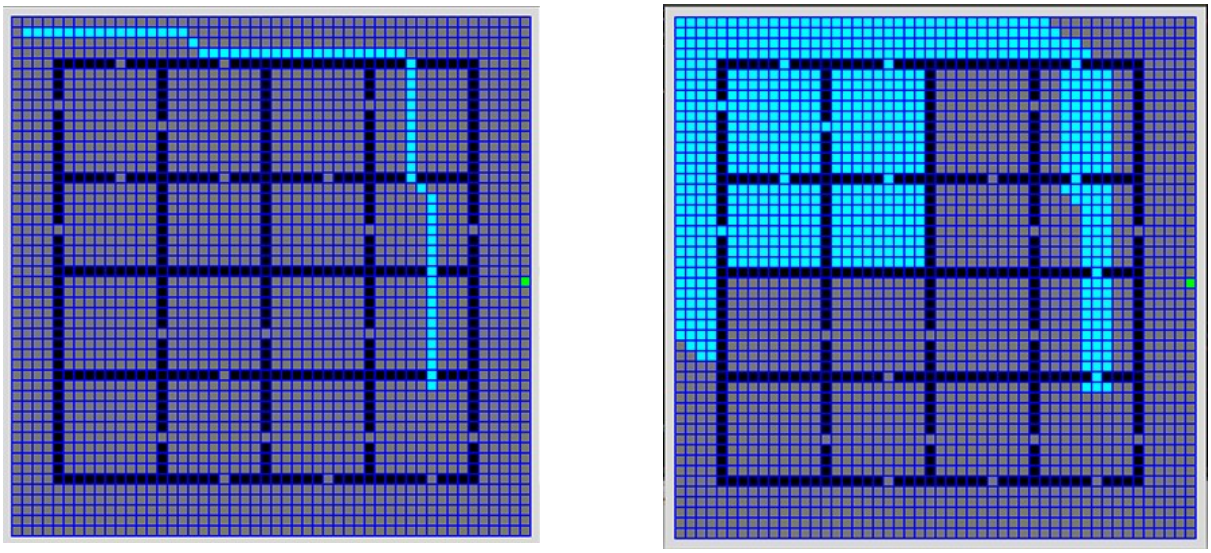


Fig.13 A*Algorithm: (A) The Path Found (B) the cells explored

D* ALGORITHM

Essentially D* and A* are one and the same until and unless the environment changes. The functionality of changing environments is yet to be incorporated into our simulator, so the results are same.

JUMP POINT SEARCH

The cells processed in total and the path found are shown in the figure below. JPS is claimed to be around 10 times faster than A* in almost all cases. The results for JPS are as follows:

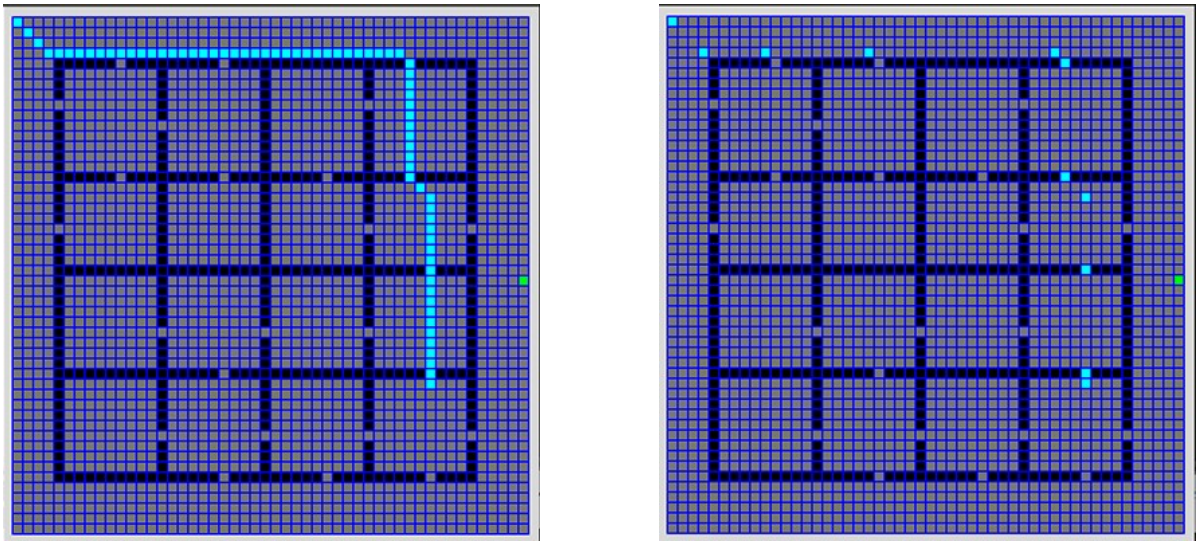


Fig.14 Jump Point Search: (A) The Path Found (B) the cells explored

PROFILING RESULTS

The variables are as follows:

- Function Calls – number of times the main loop executed before finishing
- Execution Time – Time taken by the function to complete execution and output result
- Visited – Number of tiles visited before finding the goal node
- Path – Number of tiles comprising the path
- Cost – Total cost of movement
    - For BFS – *cost = cost + 6* in all directions
    - For rest – *cost = cost + 5* if orthogonal direction, else *cost = cost + 7*

The results obtained are as follows:

SN	Algorithm	Function Calls	Execution Time	Visited	Path	Cost
1	Breadth First Search	2097	0.669954 sec	1242	70	1614
2	Dijkstra's Algorithm	2123	0.622114 sec	1228	69	1419
3	A* (and D*) Algorithm	2074	0.564133 sec	984	69	1417
4	Jump Point Search	205	0.021524 sec	10	69	1417

Table 1

As claimed, Jump Point Search is more than 10 times faster in this case than A*. We can also see that BFS and Dijkstra's Algorithm give comparable execution times in the specified case. The map being simulated is that of an office environment. The *start* position is *(0, 0)* and the *goal* position is *(35, 40)*.

6 EXPERIMENTATION

Index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Path Planning Simulator</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<link rel="stylesheet" type="text/css" href="stylesheets/canvas.css">
<link rel="stylesheet" type="text/css" href="stylesheets/main.css">
</head>
<body >
<div id = "container">
<h2 title="Back to Main website"><a href="#">Path Planner Simulation</a></h2>
<div id="SearchAlgo">
<input type="button" title="Click to Start" value="Depth First Search"
onclick="setAlgo(0)">
<input type="button" title="Click to Start" value="Breadth First Search"
onclick="setAlgo(1)">
<input type="button" title="Click to Start" value="Greddy Best First"
onclick="setAlgo(2)">
<input type="button" title="Click to Start" value="Dijkstra"
onclick="setAlgo(3)">
<input type="button" title="Click to Start" value="A-Star" onclick="setAlgo(4)">
<input type="button" title="Click to Start" value="RRT" onclick="setAlgo(5)">
<input type="button" title="Click to Start" value="RRT-Connect"
onclick="setAlgo(6)">
<input type="button" title="Click to Start" value="RRT-Star"
onclick="setAlgo(7)">
<input type="button" title="Click to Start" value="Stop" onclick="setAlgo(8)">
</div>
<canvas id="myCanvas" width = 1000 height = 800> </canvas>
<div id="stats" title="Drag to Move">
<div id="textbar">
<p style="padding: 42px; margin: auto">Press one of the buttons above to
start!</p>
</div>
<button id="toggle" title="Toggle On/Off "
onclick="disappear()"><strong>X</strong></button>
</div>
</div>
<script type="text/javascript" src="scripts/planners.js"></script>
<script type="text/javascript" src="scripts/interaction.js"></script>
<script>dragElement(document.getElementById(("stats")));</script>
</body>
</html>
```

Planners.js

```
function init(algo) {
      search_alg="";

      // get user input as to which algorith to use
      switch(algo){
            case 0:
                  search_alg = "depth-first";
                  break;
            case 1:
                  search_alg = "breadth-first";
                  break;
            case 2:
                  search_alg = "greedy-best-first";
                  break;
            case 3:
                  search_alg = "dijkstra";
                  break;
            case 4:
                  search_alg = "A-star";
                  break;
            case 5:
                  search_alg = "RRT";
                  break;
            case 6:
                  search_alg = "RRT-connect";
                  break;
            case 7:
                  search_alg = "RRT-star";
                 break;
          case 8:
              search_alg = "";
              break;
          case 9:
              location.reload();
      }

    initSearch(algo);
    if(search_alg != ""){
        animate();
    }
}


////////////////////////////////////////////////
/////     SEARCH INITIALIZATION LOOP
////////////////////////////////////////////////
```

```
function initSearch() {
      // World defined by self
    planning_scene = "multi_part";

    // eps defines the density of the grid cells
    eps = 0.1;
    path = [];

    // create event handlers for the mouse
    canvas = document.getElementById("myCanvas");
    mouse_x = 0;
    mouse_y = 0;

    // when the mouse moves, update the mouse's location
    canvas.onmousemove = function handleMouseMove(event) {
        mouse_x = event.clientX;
        mouse_y = event.clientY;
    };

    // when the mouse button is pressed, update mouseDown
    canvas.onmousedown = function() {
        mouseDown = 1;
    };

    // when the mouse button is released, update mouseDown
    canvas.onmouseup = function() {
        mouseDown = 0;
    };

    // Set the Start and Goal Points on the Canvas
    q_init = [-1.2,-1.2];
    q_goal = [5.3, 3.5];

        // Fixed code for parsing full url to get input parameters as given
    var url_parsed = window.location.href.split("?");
    for (i=1;i<url_parsed.length;i++) {
        var param_parsed = url_parsed[i].split("=");
        // console.log(param_parsed[0],param_parsed[1]);
        // eval(param_parsed[0]+"=\'"+param_parsed[1]+"\'");
        var param = param_parsed[0] ; var arg = param_parsed[1];
        switch(param) {
            case "q_init":
                arg = arg.slice(1,arg.length-1);
                var args = arg.split(",");
                q_init[0] = parseFloat(args[0]) ; q_init[1] =
parseFloat(args[1])
                break;
            case "q_goal":
                arg = arg.slice(1,arg.length-1);
                var args = arg.split(",");
```

```
                q_goal[0] = parseFloat(args[0]) ; q_goal[1] =
parseFloat(args[1])
                break;
            default:
                console.warn("Using default parameters");
                q_init = [-1.2,-1.2];
                q_goal = [5.3, 3.5];
                search_alg = "A-star";
        }
    }

    // Convert the Canvas Coordinates to Grid Coordinates
    start = getCoord(q_init);
    goal = getCoord(q_goal);

    // set the world for the planner
    setPlanningScene();

    // initialize search tree from start configurations (RRT-based algorithms)
    T_a = initRRT(q_init, "T_a");

    // also initialize search tree from goal configuration (RRT-Connect)
    T_b = initRRT(q_goal, "T_b");

    // variables for RRT
    nbhd = 2;
    stepSize = 1*eps;
    tree1 = T_a;
    tree2 = T_b;
    q_new = [];
    target = [];
    tolerance = 2*stepSize;

    saved_iter = 0;

      // Initialize everything to zero
    initSearchGraph();
    search_iterate = true;
    search_iter_count = 0;
    search_result = "starting";
    search_max_iterations = 100000;
    search_visited = 0;
    path_length = 0;
    path_found = false;
    cur_time = Date.now();
    min_msec_between_iterations = 20;

    // Add TextBar Stats Element
    textbar = document.getElementById("textbar");
}
```

```
////////////////////////////////////////////////
/////     ANIMATION AND INTERACTION LOOP
////////////////////////////////////////////////

function animate() {
    drawRobotWorld();

    // make sure the rrt iterations are not running faster than animation update
    if (search_iterate && (Date.now()-cur_time > min_msec_between_iterations)) {
        cur_time = Date.now();
        search_iter_count++;

        switch (search_alg) {
            case "depth-first":
                search_result = DFS();
                break;
            case "breadth-first":
                search_result = BFS();
                break;
            case "dijkstra":
                search_result = Dijkstra();
                break;
            case "greedy-best-first":
                search_result = Greedy();
                break;
            case "A-star":
                search_result = iterateGraphSearch();
                break;
            case "RRT":
                search_result = iterateRRT();
                break;
            case "RRT-connect":
                search_result = iterateRRTConnect();
                break;
            case "RRT-star":
                search_result = iterateRRTStar();
                break;
            default:

                break;
        }
    }

    var queue_size;
    if(search_alg=="depth-first" || search_alg=="breadth-first" ||
search_alg=="RRT" || search_alg=="RRT-connect" || search_alg == "RRT-star")
        queue_size = visit_queue.length;
```

```
    else if(search_alg=="")
        queue_size = 0;
    else
        queue_size = visit_queue.size();

    if(search_alg=="A-star" || search_alg=="depth-first" ||
search_alg=="breadth-first" || search_alg=="greedy-best-first" ||
search_alg=="dijkstra") {
        textbar.innerHTML =
        "<h3>Algorithm Statistics:</h3>"
        + search_alg
        + " progress: " + search_result
        + "<br>"
        + "<strong>Start</strong>: " + q_init
        + "  |  "
        + "<strong>Target</strong>: " + q_goal
        + "<br>"
        + "<strong>Iteration:</strong> " + search_iter_count
        + "  |  "
        + "<strong>Path Length:</strong> " + path_length.toFixed(2)
        + "<br>"
        + "<strong>Visited:</strong> " + search_visited
        + "   |   "
        + "<strong>Queued:</strong> " + queue_size
        + "<br>" ;

        //textbar.innerHTML += "<br> mouse ("+ mouse_x+","+mouse_y+")";
    }

    else if(search_alg=="RRT" || search_alg=="RRT-connect"){
        textbar.innerHTML =
        "<h3>Algorithm Statistics:</h3>"
        + search_alg
        + " progress: " + search_result
        + " <br> "
        + "<strong>Start</strong>: " + q_init
        + "  |  "
        + "<strong>Target</strong>: " + q_goal
        + "<br>"
        + "<strong>Iteration:</strong> " + search_iter_count
        + "  |  "
        + "<strong>Path Length:</strong> " + path_length.toFixed(2)
        + "<br>";
    }
    else if(search_alg == ""){
        textbar.innerHTML = "<p style = "
        + "'padding: 42px; margin: auto'>"
        + "Press one of the buttons above to start! </p>";
    }
    else{
```

```
        textbar.innerHTML =
        "<h3>Algorithm Statistics:</h3>"
        + search_alg
        + " progress: " + search_result
        + " <br> "
        + "<strong>Start</strong>: " + q_init
        + "  |  "
        + "<strong>Target</strong>: " + q_goal
        + "<br>"
        + "<strong>Iteration:</strong> " + search_iter_count
        + "  |  "
        + "<strong>Path Length:</strong> " + path_length.toFixed(2)
        + "<br>"
        + "<strong>Optimization</strong> | 2000 Iterations: " + path_found
        + "<br>";
    }

    textbar.innerHTML += "<br> Mouse Position: ("+
xformViewWorldX(mouse_x)+","+xformViewWorldY(mouse_y)+")";

    // callback request for the animate function be called again
    //   more details online:  http://learningwebgl.com/blog/?p=3189
    if(search_result=="succeeded" || search_iter_count>search_max_iterations)
        search_iterate =false;

    requestAnimationFrame(animate);
}

function BFS() {
    if(visit_queue.length == 0 || G[goal[0]][goal[1]].obstacle==true ||
G[start[0]][start[1]].obstacle==true) {
        console.log("Search Failed!")
        search_iterate=false;
        return "failed";
    }

    curr = visit_queue.shift();
    var xc = curr[0] ; var yc = curr[1];
    G[xc][yc].queued = false;
    ctx.fillStyle = "#ffe8a5";
    ctx.fillRect(xformWorldViewX(G[xc][yc].x)-3,xformWorldViewY(G[xc][yc].y)-
3,6,6);

    if(xc == goal[0] && yc==goal[1] || G[goal[0]][goal[1]].visited==true) {
        console.log("Goal found!");
        search_iterate=false;
        draw_path();
        return "succeeded";
    }
```

```
    var neighbors = get4nbhd(curr);
    for(var i=0; i<neighbors.length; i++) {
        var xi = neighbors[i][0]; var yi = neighbors[i][1];
        if(G[xi][yi].visited==false) {
            visit_queue.push([xi,yi]);
            G[xi][yi].visited = true;
            G[xi][yi].queued = true;
            G[xi][yi].parent = curr;
            G[xi][yi].distance = G[xc][yc].distance + gcost(curr,[xi,yi]);
            search_visited+=1;
            ctx.fillStyle = "#52483F";
            ctx.fillRect(xformWorldViewX(G[xi][yi].x)-
3,xformWorldViewY(G[xi][yi].y)-3,6,6);
            // console.log(curr,next,G[xi][yi].distance,G[xi][yi].visited,
G[xi][yi].priority,G[xi][yi].queued,G[xi][yi].parent,search_visited)
        }
    }
    return "iterating";
}

function DFS() {
    if(visit_queue.length == 0 || G[goal[0]][goal[1]].obstacle==true ||
G[start[0]][start[1]].obstacle==true) {
        console.log("Search Failed!")
        search_iterate=false;
        return "failed";
    }

    curr = visit_queue.pop();
    var xc = curr[0] ; var yc = curr[1];
    G[xc][yc].queued = false;
    ctx.fillStyle = "#ffe8a5";
    ctx.fillRect(xformWorldViewX(G[xc][yc].x)-3,xformWorldViewY(G[xc][yc].y)-
3,6,6);

    if((curr[0] == goal[0] && curr[1]==goal[1]) ||
G[goal[0]][goal[1]].visited==true) {
        console.log("Goal found!");
        search_iterate=false;
        draw_path();
        return "succeeded";
    }

    var neighbors = get4nbhd(curr);
    for(var i=0; i<neighbors.length; i++) {
        var xi = neighbors[i][0]; var yi = neighbors[i][1];
        if(G[xi][yi].visited==false) {
            visit_queue.push([xi,yi]);
            G[xi][yi].visited = true;
            G[xi][yi].queued = true;
```

```
            G[xi][yi].parent = curr;
            G[xi][yi].distance = G[xc][yc].distance + gcost(curr,[xi,yi]);
            search_visited+=1;
            ctx.fillStyle = "#52483F";
            ctx.fillRect(xformWorldViewX(G[xi][yi].x)-
3,xformWorldViewY(G[xi][yi].y)-3,6,6);
        }
    }
    return "iterating";
}

function Dijkstra() {
    if(visit_queue.size()==0 || G[goal[0]][goal[1]].obstacle==true ||
G[start[0]][start[1]].obstacle==true) {
        console.log("Search failed");
        search_iterate=false;
        return "failed";
    }

    curr = visit_queue.get();
    var xc=curr[0] ; var yc=curr[1];
    G[xc][yc].queued = false;
    ctx.fillStyle = "#ffe8a5";
    ctx.fillRect(xformWorldViewX(G[xc][yc].x)-3,xformWorldViewY(G[xc][yc].y)-
3,6,6);

    if(xc == goal[0] && yc==goal[1] || G[goal[0]][goal[1]].visited==true) {
        console.log("Goal found!");
        search_iterate=false;
        draw_path();
        return "succeeded";
    }

    var nbrs = get4nbhd(curr);
    for(var i=0 ; i<nbrs.length;i++) {
        var next = nbrs[i];
        var xi=next[0]; var yi=next[1];
        var new_cost = G[xc][yc].distance + gcost(curr,next);


        if(G[xi][yi].visited==false || new_cost < G[xi][yi].distance) {
            G[xi][yi].distance = new_cost;
            var pri = new_cost;
            visit_queue.put(next,pri);
            G[xi][yi].priority = pri;
            G[xi][yi].visited = true;
            G[xi][yi].queued = true;
            G[xi][yi].parent = curr;
            search_visited+=1;
            ctx.fillStyle = "#52483F";
```

```
            ctx.fillRect(xformWorldViewX(G[xi][yi].x)-
3,xformWorldViewY(G[xi][yi].y)-3,6,6);

        }
    }
    return "iterating";
}

function Greedy() {
    if(visit_queue.size()==0 || G[goal[0]][goal[1]].obstacle==true ||
G[start[0]][start[1]].obstacle==true) {
        console.log("Search failed");
        search_iterate=false;
        return "failed";
    }

    curr = visit_queue.get();
    var xc=curr[0] ; var yc=curr[1];
    G[xc][yc].queued = false;
    ctx.fillStyle = "#ffe8a5";
    ctx.fillRect(xformWorldViewX(G[xc][yc].x)-3,xformWorldViewY(G[xc][yc].y)-
3,6,6);

    if(xc==goal[0] && yc==goal[1] || G[goal[0]][goal[1]].visited==true) {
        console.log("Goal found!");
        search_iterate=false;
        draw_path();
        return "succeeded";
    }

    var nbrs = get4nbhd(curr);
    for(var i=0 ; i<nbrs.length;i++) {
        var next = nbrs[i];
        var xi=next[0]; var yi=next[1];
        var new_cost = heuristic(next,goal);

        if(G[xi][yi].visited==false || new_cost < G[xi][yi].distance) {
            G[xi][yi].distance = new_cost;
            var pri = new_cost;
            visit_queue.put(next,pri);
            G[xi][yi].priority = pri;
            G[xi][yi].visited = true;
            G[xi][yi].queued = true;
            G[xi][yi].parent = curr;
            search_visited+=1;
            ctx.fillStyle = "#52483F";
            ctx.fillRect(xformWorldViewX(G[xi][yi].x)-
3,xformWorldViewY(G[xi][yi].y)-3,6,6);
        }
    }
```

```
    return "iterating";
}

function iterateGraphSearch() {
    if(visit_queue.size()==0 || G[goal[0]][goal[1]].obstacle==true ||
G[start[0]][start[1]].obstacle==true) {
        console.log("Search failed");
        search_iterate=false;
        return "failed";
    }

    curr = visit_queue.get();
    var xc=curr[0] ; var yc=curr[1];
    G[xc][yc].queued = false;
    ctx.fillStyle = "#ffe8a5";
    ctx.fillRect(xformWorldViewX(G[xc][yc].x)-3,xformWorldViewY(G[xc][yc].y)-
3,6,6);

    if(xc==goal[0] && yc==goal[1] || G[goal[0]][goal[1]].visited==true) {
        console.log("Goal found!");
        search_iterate=false;
        draw_path();
        return "succeeded";
    }

    var nbrs = get4nbhd(curr);
    for(var i=0 ; i<nbrs.length;i++) {
        var next = nbrs[i];
        var xi=next[0]; var yi=next[1];
        var new_cost = G[xc][yc].distance + gcost(curr,next);

        if(G[xi][yi].visited==false || new_cost < G[xi][yi].distance) {
            G[xi][yi].distance = new_cost;
            var pri = new_cost + heuristic(goal,next);
            visit_queue.put(next,pri);
            G[xi][yi].priority = pri;
            G[xi][yi].visited = true;
            G[xi][yi].queued = true;
            G[xi][yi].parent = curr;
            search_visited+=1;
            ctx.fillStyle = "#52483F";
            ctx.fillRect(xformWorldViewX(G[xi][yi].x)-
3,xformWorldViewY(G[xi][yi].y)-3,6,6);
        }
    }
    return "iterating";
}

function iterateRRT() {
    var rand = randomConfig();
```

```
    if(testCollision(q_goal)==true || testCollision(q_init)==true ||
search_iter_count>=search_max_iterations) {
        console.log("Search Failed!");
        search_iterate=false;
        return "failed";
    }

    var result = extendRRT(rand,q_goal,T_a);
    if (result=="reached"){
        console.log("Reached target!");
        dfsPath(q_goal, q_init, T_a);
        return "succeeded";
    }
    return "iterating";
}

function iterateRRTConnect() {
    if(testCollision(q_goal)==true || testCollision(q_init)==true ||
search_iter_count>=search_max_iterations) {
        console.log("Search Failed!");
        search_iterate=false;
        return "failed";
    }

    var rand = randomConfig();
    if(extendRRT(rand, target, tree1)!="trapped")
        if(connectRRT(q_new,tree2)=="reached"){
            console.log("reached");
            dfsPath(q_new, q_init, T_a);
            dfsPath(q_new,q_goal,T_b);
            search_iterate=false;
            return "succeeded";
        }

    if(tree1.name == "T_a"){
        tree1 = T_b;
        tree2 = T_a;
        target = q_goal;
    }
    else{
        tree1 = T_a;
        tree2 = T_b;
        target = q_init;
    }

    return "iterating";
}

function iterateRRTStar() {
```

```
    if(testCollision(q_goal)==true || testCollision(q_init)==true ||
search_iter_count>=search_max_iterations) {
        console.log("Search Failed!");
        search_iterate=false;
        return "failed";
    }

    var rand = randomConfig(),
        nearest = findNearestNeighbor(rand,T_a),
        vertex = newConfig(nearest, rand, T_a);

    if(distance(vertex, q_goal) <= 1.4*stepSize)
        vertex = q_goal;

    if(testCollision(vertex)==false && inTree(vertex, T_a)==false){
        var near = findNeighborhood(vertex,T_a),
            parent_idx = chooseParent(vertex, nearest, near, T_a),
            parent = T_a.vertices[parent_idx].vertex,
            cost = T_a.vertices[parent_idx].cost + distance(parent,vertex);

        insertTreeVertex(T_a, vertex, parent, cost);
        insertTreeEdge(T_a, T_a.newest ,parent_idx);
        reWire(vertex, nearest, near, T_a);

        q_new = vertex;

        if(isEqual(vertex, q_goal)==true){
            saved_iter = search_iter_count;
            path_found = true;
            console.log("Reached target!");
            return "reached";
        }
    }
    var goal_idx = getIndex(q_goal,T_a);

    if (saved_iter>0) {
        if(search_iter_count - saved_iter>2000){
            search_iterate = false;
            dfsPath(q_goal, q_init, T_a);
            return "reached";
        }
        return "reached target! Optimizing";
    }

    else
        return "iterating";
}


////////////////////////////////////////////////
```

```
/////     RRT IMPLEMENTATION FUNCTIONS
////////////////////////////////////////////////

function extendRRT(rand, target, tree) {
    var nnbr_idx = findNearestNeighbor(rand, tree),
        vertex = newConfig(nnbr_idx, rand, tree),
        parent = tree.vertices[nnbr_idx].vertex,
        cost = tree.vertices[nnbr_idx].cost + distance(vertex,parent);

    if (distance(vertex, target) <= 1.2*stepSize)
        vertex = target;

    if (testCollision(vertex)==false && inTree(vertex,tree)==false){
        insertTreeVertex(tree,vertex,parent, cost);
        insertTreeEdge(tree,tree.newest,nnbr_idx);

        q_new = vertex;

        if (isEqual(vertex, target)==true){
            search_iterate = false;
            return "reached";
        }

        return "advanced";
    }

    else
        return "trapped";
}

function connectRRT(target, tree) {
    var result = extendRRT(target, target, tree);
    if (result=="advanced")
        result = extendRRT(target, target, tree);
    return result;
}

// Returns a random config in the C-Space
function randomConfig() {
    var x = strip(-1.5 + Math.random()*7.5),
        y = strip(-1.5 + Math.random()*7.5);
    return [x,y];
}

// Returns the new config to be added to the tree whether or not valid
function newConfig(q_idx, rand, tree) {
    var dx = rand[0] - tree.vertices[q_idx].vertex[0],
        dy = rand[1] - tree.vertices[q_idx].vertex[1],
        n = Math.sqrt(dx*dx + dy*dy),
```

```
        xd = tree.vertices[q_idx].vertex[0] + stepSize*dx/n,
        yd = tree.vertices[q_idx].vertex[1] + stepSize*dy/n;

    xd = Math.round(xd*100)/100;
    yd = Math.round(yd*100)/100;

    return [xd,yd];
}

// Returns the distance between 2 vectors
function distance(a,b) {
    var dx = a[0]-b[0],
        dy = a[1]-b[1],
        dist = Math.sqrt(dx*dx + dy*dy);
    return dist;
}

// Returns the index of the nearest neighbour
function findNearestNeighbor(target, tree) {
    var min = 999999,
        pos = 0,
        dist = 0;

    for(var i=0; i<=tree.newest; i++) {
        var dist = distance(target,tree.vertices[i].vertex);
        if (min >= dist) {min = dist; pos = i;}
    }
    return pos;
}

// Returns all node indexes in neighborhood
function findNeighborhood(target, tree) {
    var nbrs = [];
    for(var i=0; i<=tree.newest; i++)
        if (distance(target,tree.vertices[i].vertex) <= tolerance)
            nbrs.push(i);
    return nbrs;
}

// Chooses parent with minimum cost
function chooseParent(target, min_idx, nbr_idxs, tree) {
    var cmin = tree.vertices[min_idx].cost + stepSize,
        pos = min_idx;

    if(nbr_idxs.length>0){
        for(var i=0; i<nbr_idxs.length;i++){
            var cost = tree.vertices[nbr_idxs[i]].cost +
distance(tree.vertices[nbr_idxs[i]].vertex, target);
            if (cost<cmin)
                pos = nbr_idxs[i];
```

```
        }
    }
    return pos;
}

// Rewires the structure
function reWire(target, min_idx, nbr_idxs, tree) {
    if(nbr_idxs.length>0)
        for(var i=0; i<nbr_idxs.length; i++){
            var cost = tree.vertices[nbr_idxs[i]].cost;
            var new_cost = tree.vertices[tree.newest].cost +
distance(tree.vertices[tree.newest].vertex, tree.vertices[nbr_idxs[i]].vertex);
            if (new_cost<cost){
                tree.vertices[nbr_idxs[i]].cost = new_cost;
                var parent_idx =
getIndex(tree.vertices[nbr_idxs[i]].parent,tree);
                tree.vertices[nbr_idxs[i]].parent =
tree.vertices[tree.newest].vertex;
                insertTreeEdge(tree,nbr_idxs[i],tree.newest);
                removeTreeEdge(tree, parent_idx, nbr_idxs[i]);
            }
        }
}

// Generates path between the current "node" to the "target" in a given tree
function dfsPath(node, target, tree) {
    var curr = node,
        path = [],
        node_idx = getIndex(node,tree);

    while(isEqual(curr,target)==false){
        path.push(curr);
        var curr_idx = getIndex(curr,tree),
            parent = tree.vertices[curr_idx].parent;
        draw_2D_path_configurations(curr,parent);
        path_length+=distance(curr,parent);
        curr = parent;
    }
}

// Prints the elements of an array
function print(arr) {
    var st = "Path := \n";

    for(var i=0;i<arr.length;i++)
        st += "["+arr[i] + "]\n"
    console.log(st);
}

// Checks if the vertex is already in the tree or not
```

```
function inTree(vertex, tree) {
    for(var i=0; i<=tree.newest; i++)
        if (isEqual(vertex, tree.vertices[i].vertex))
            return true;
    return false;
}

// Checks if arrays are equal
function isEqual(q1,q2) {
    for(var i=0; i<q1.length; i++)
        if (q1[i]!=q2[i])
            return false;
    return true;
}

// Returns index of vertex, if it's in the tree
function getIndex(vertex, tree) {
    if (inTree(vertex,tree))
        for(var i=0; i<=tree.newest; i++)
            if (isEqual(vertex, tree.vertices[i].vertex))
                return i;
    return null;
}

////////////////////////////////////////////////
/////     STENCIL SUPPORT FUNCTIONS
////////////////////////////////////////////////

// functions for transforming canvas coordinates into planning world coordinates
function xformWorldViewX(world_x) {
    return (world_x*100)+200;  // view_x
}
function xformWorldViewY(world_y) {
    return (world_y*100)+200;  // view_y
}
function xformViewWorldX(view_x) {
    return (view_x-200)/100;  // view_x
}
function xformViewWorldY(view_y) {
    return (view_y-200)/100;  // view_y
}

function drawRobotWorld() {
    // draw start and goal configurations
    c = document.getElementById("myCanvas");
    ctx = c.getContext("2d");
    ctx.fillStyle = "red";
    ctx.fillRect(xformWorldViewX(q_init[0])-5,xformWorldViewY(q_init[1])-
5,12,12);
    ctx.fillStyle = "green";
```

```
    ctx.fillRect(xformWorldViewX(q_goal[0])-5,xformWorldViewY(q_goal[1])-
5,12,12);

    // draw robot's world
    for (j=0;j<range.length;j++) {
        ctx.fillStyle = "#e4e4e4";

ctx.fillRect(xformWorldViewX(range[j][0][0]),xformWorldViewY(range[j][1][0]),xfo
rmWorldViewX(range[j][0][1])-
xformWorldViewX(range[j][0][0]),xformWorldViewY(range[j][1][1])-
xformWorldViewY(range[j][1][0]));
    }
}

function initSearchGraph() {

    // initialize search graph as 2D array over configuration space
    //   of 2D locations with specified spatial resolution
    G = [];
    for (iind=0,xpos=-2;xpos<7;iind++,xpos+=eps) {
        G[iind] = [];
        for (jind=0,ypos=-2;ypos<7;jind++,ypos+=eps) {
            G[iind][jind] = {
                i:iind,j:jind, // mapping to graph array
                x:strip(xpos),y:strip(ypos), // mapping to map coordinates
                parent:null, // pointer to parent in graph along motion path
                distance:10000, // distance to start via path through parent
                visited:false, // flag for whether the node has been visited
                priority:null, // visit priority based on fscore
                queued:false, // flag for whether the node has been queued for
visiting
                obstacle:false  // flag to determine whether node is obstacle
            };
        }
    }

    define_obstacles();

    // For algorithms using priority queues as data structures - A-Star/
Dijkstra/ Greedy
    if(search_alg=="dijkstra" || search_alg=="A-star" || search_alg=="greedy-
best-first") {
        visit_queue = new PriorityQueue();
        visit_queue.put(start,0);
        G[start[0]][start[1]].visited = true;
        G[start[0]][start[1]].distance = 0;
        G[start[0]][start[1]].queued = true;
        G[start[0]][start[1]].priority = 0;
    }
    // else if(search_alg == "RRT" || search_alg == "RRT-connect" || search_alg
```

```
== "RRT-star") {
    //     visit_queue = [];
    // }
    // For algorithms using stacks/queues as data structures - DFS/ BFS
    else {
        visit_queue = [];
        visit_queue.push(start);
        G[start[0]][start[1]].visited = true;
        G[start[0]][start[1]].queued = true;
    }

    rows = G.length; cols = G[0].length;
}

function setPlanningScene() {

    // obstacles specified as a range along [0] (x-dimension) and [1] y-
dimension
    range = []; // global variable

    // world boundary
    range[0] = [ [-1.9,7.9],[-1.9,-1.7] ];
    range[1] = [ [-1.9,7.9],[5.7,5.9] ];
    range[2] = [ [-1.9,-1.7], [-1.9,5.9] ];
    range[3] = [ [7.7,7.9],   [-1.9,5.9] ];

    if (typeof planning_scene === 'undefined')
        planning_scene = 'multi_part';

      if (planning_scene == 'multi_part') {
        range[4] = [ [0.2,0.4], [-1.9,-1.5]];
        range[5] = [ [0.2,0.4], [-1.2, 4.1]];
        range[6] = [ [0.2,0.4],[4.4,5.9]];

        range[7] = [ [1.8,2],[-1.9,0.8]];
        range[8] = [ [1.8,2],[1.1,5.4]];

        range[9] = [ [3.6,6.5],[0.6,0.8]];
        range[10] = [ [3.6,6.5],[4.9,5.1]];
        range[11] = [ [3.6,3.8],[1.1,4.6]];
        range[12] = [ [6.3,6.5],[1.1,4.6]];

        range[13] = [ [3.6,4.5],[2.6,2.8]];
        range[14] = [ [4.9,6.5],[2.6,2.8]];
    }
}

function testCollision(q) {
    var j;
    for (j=0;j<range.length;j++) {
```

```
        var in_collision = true;
        for (i=0;i<q.length;i++) {
            if ((q[i]<range[j][i][0])||(q[i]>range[j][i][1]))
                in_collision = false;
        }
        if (in_collision)
            return true;
    }
    return false;
}

function initRRT(q,name) {
    var tree = {};

    tree.vertices = [];
    tree.vertices[0] = {};
    tree.vertices[0].vertex = q;
    tree.vertices[0].edges = [];
    tree.vertices[0].parent = null;
    tree.vertices[0].cost = 0;
    tree.name = name;

    tree.newest = 0;

    return tree;
}

function insertTreeVertex(tree, q, parent, cost) {
    new_vertex = {};
    new_vertex.edges = [];
    new_vertex.vertex = q;
    new_vertex.parent = parent;
    new_vertex.cost = cost;

    tree.vertices.push(new_vertex);
    tree.newest = tree.vertices.length - 1;
    draw_2D_configuration(q);
}

function draw_2D_configuration(q) {
    c = document.getElementById("myCanvas");
    ctx = c.getContext("2d");
    ctx.fillStyle = "#80685E";
    ctx.fillRect(xformWorldViewX(q[0])-3,xformWorldViewY(q[1])-3,6,6);
}

function draw_2D_edge_configurations(q1,q2) {
    c = document.getElementById("myCanvas");
    ctx = c.getContext("2d");
    ctx.beginPath();
```

```
    ctx.moveTo(xformWorldViewX(q1[0]),xformWorldViewY(q1[1]));
    ctx.lineTo(xformWorldViewX(q2[0]),xformWorldViewY(q2[1]));
    ctx.strokeStyle = "#2a2e2f";
    ctx.stroke();
}

function remove_2D_edge_configurations(q1,q2) {
    c = document.getElementById("myCanvas");
    ctx = c.getContext("2d");
    ctx.beginPath();
    ctx.moveTo(xformWorldViewX(q1[0]),xformWorldViewY(q1[1]));
    ctx.lineTo(xformWorldViewX(q2[0]),xformWorldViewY(q2[1]));
    ctx.strokeStyle = "#DC143C";
    // ctx.lineWidth = 2;
    ctx.stroke();
}

function draw_2D_path_configurations(q1,q2) {
    c = document.getElementById("myCanvas");
    ctx = c.getContext("2d");
    ctx.beginPath();
    ctx.moveTo(xformWorldViewX(q1[0]),xformWorldViewY(q1[1]));
    ctx.lineTo(xformWorldViewX(q2[0]),xformWorldViewY(q2[1]));
    ctx.strokeStyle = "#283747";
    ctx.lineWidth = 4;
    ctx.stroke();
}

function insertTreeEdge(tree,q1_idx,q2_idx) {
    tree.vertices[q1_idx].edges.push(tree.vertices[q2_idx]);
    tree.vertices[q2_idx].edges.push(tree.vertices[q1_idx]);

draw_2D_edge_configurations(tree.vertices[q1_idx].vertex,tree.vertices[q2_idx].v
ertex);
}

function removeTreeEdge(tree, q1_idx, q2_idx) {
    var edge1 = tree.vertices[q1_idx].edges,
        edge2 = tree.vertices[q2_idx].edges,
        vertex1 = tree.vertices[q1_idx].vertex,
        vertex2 = tree.vertices[q2_idx].vertex,
        l1 = edge1.length,
        l2 = edge2.length,
        pos = 0;

    for(var i=0; i<l1; i++)
        if (isEqual(edge1[i],vertex2))
            pos = i;
    edge1.splice(pos,1);
```

```
    for(var i=0; i<l2; i++)
        if (isEqual(edge2[i],vertex1))
            pos = i;
    edge2.splice(pos,1);

    remove_2D_edge_configurations(vertex1,vertex2);
}

////////////////////////////////////////////////
/////     PRIORITY QUEUE IMPLEMENTATION
////////////////////////////////////////////////

// STENCIL: implement min heap functions for graph search priority queue.
//   These functions work use the 'priority' field for elements in graph.
function PriorityQueue() {this.list = [];}

PriorityQueue.prototype.put = function(element, priority) {
  for (var i = 0; i < this.list.length && this.list[i][1] < priority; i++);
  this.list.splice(i, 0, [element, priority])}

PriorityQueue.prototype.get = function() {return this.list.shift()[0]}

PriorityQueue.prototype.size = function() {return this.list.length}

PriorityQueue.prototype.qprint = function() {
    var st="Queue:";
    for(var i=0;i<this.list.length;i++)
        st+= " | "+this.list[i][0];
    console.log(st)}

////////////////////////////////////////////////
/////     HELPER FUNCTIONS
////////////////////////////////////////////////

// Function to define all obstacles in the canvas frame
function define_obstacles() {
    for(var i= 0; i<G.length; i++) {
        for(var j=0;j<G[0].length;j++) {
            var x = G[i][j].x; var y = G[i][j].y
            if(testCollision([x,y]) == true)
                G[i][j].obstacle = true;
        }
    }
}
// Function to get graph coordinates[(0,90),(0,90)] by giving world coordinates
[(-2,7),(-2,7)] as inputs
function getCoord(node) {
    var x = Math.round(strip((node[0]+2)/eps));
    var y = Math.round(strip((node[1]+2)/eps));
    return [x,y]
```

```
}
// Function to get world coordinates[(-2,7),(-2,7)] by giving graph coordinates
[(0,90),(0,90)] as inputs
function getWorld(node) {
    var x = strip(eps*node[0]) - 2;
    var y = strip(eps*node[1]) - 2;
    return [x,y]
}
// Function to get 4 adjacent neighbours
function get4nbhd(node) {
    var nbhd = [];
    var space = [[0,1],[1,0],[0,-1],[-1,0]];
    for(var i=0; i< space.length; i++) {
        var new_x = node[0] + space[i][0];
        var new_y = node[1] + space[i][1];
        if((new_x>=0 && new_x<rows) && (new_y>=0 && new_y<cols) &&
G[new_x][new_y].visited==false && G[new_x][new_y].obstacle==false){
            nbhd.push([new_x,new_y]);
        }
    }

    // To filter out errors rising from undefined
    if(nbhd.length>0)
        return nbhd;
    else
        return 1
}
// Function to get 8 adjacent neighbours
function get8nbhd(node) {
    var nbhd = [];
    var space = [[0,1],[1,1],[1,0],[1,-1],[0,-1],[-1,-1],[-1,0],[-1,1]];
    for(var i=0; i< space.length; i++) {
        var new_x = node[0] + space[i][0];
        var new_y = node[1] + space[i][1];
        if((new_x>=0 && new_x<rows) && (new_y>=0 && new_y<cols) &&
testCollision([G[new_x][new_y].x , G[new_x][new_y].y])==false){
            nbhd.push([new_x,new_y]);
        }
    }
    // To filter out errors rising from undefined
    if(nbhd.length>0)
        return nbhd;
    else
        return 1
}
// Function to get the heuristic
function heuristic(node1,node2,version) {
    var x1 = node1[0]; var y1 = node1[1];
    var x2 = node2[0]; var y2 = node2[1];
    var dx = Math.abs(x2-x1);
```

```
    var dy = Math.abs(y2-y1);
    var hue;
    switch(version) {
        case "octile":
        hue = dx+dy+(Math.sqrt(2) - 2)*Math.min(dx, dy);
        break;

        case "chebyshev":
        hue = dx+dy- Math.min(dx, dy);
        break;

        case "Manhattan":
        hue = dx+dy;
        break;

        default:
        console.warn('Using Euclidean Heuristic as default');
        hue = Math.sqrt(dx*dx + dy*dy);
    }
    return strip(hue);
}
// Function for calculating graph movement costs
function gcost(node1,node2){
    var x1 = node1[0]; var y1 = node1[1];
    var x2 = node2[0]; var y2 = node2[1];
    var dx = Math.abs(x2-x1);
    var dy = Math.abs(y2-y1);

    if(dx==1 && dy==1)
        return 1.4   // returning cost of 1.4 for diagonal movements
    else
        return 1   // returning cost of 1.0 for cardinal movements
}
// Function for calculating precision values
function strip(number) {
    if(number<0.000001 && number>-0.000001)
        return 0;
    else {
        return parseFloat(number.toPrecision(2));
    }
}
// Function to draw path on graph
function draw_path() {

    var curr = goal;

    path.push(curr);

    while(curr!= start) {
        curr = G[curr[0]][curr[1]].parent;
```

```
        path.push(curr);
    }

    path_length = 0;
    var st = "Path: ";
    for(var i=path.length-1;i>=0;i--)
        st += path[i]+ " ; ";
    console.log(st);
    for(var i=0; i<path.length-1;i++) {
        var p1 = [G[path[i][0]][path[i][1]].x,G[path[i][0]][path[i][1]].y];
        var p2 =
[G[path[i+1][0]][path[i+1][1]].x,G[path[i+1][0]][path[i+1][1]].y];
        draw_2D_edge_configurations(p1,p2);
        path_length += Math.sqrt(Math.pow((p1[0] - p2[0]),2) + Math.pow((p1[1] -
p2[1]),2));
    }
    console.log("Path Length: ",strip(path_length));
}
```

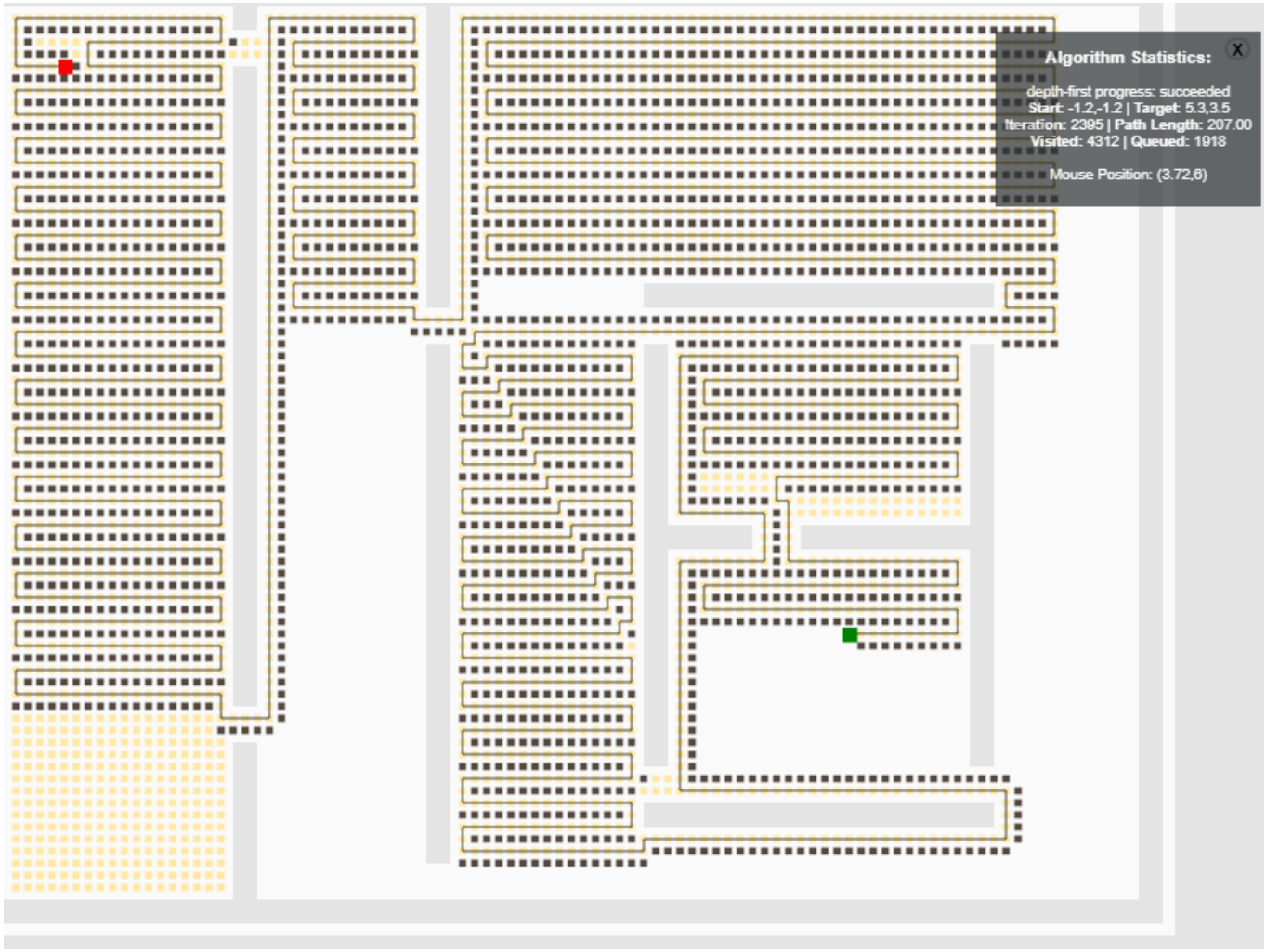
Interaction.js

```
var algo;

function setAlgo(algo) {
      init(algo);
      context = document.getElementById("myCanvas").getContext("2d");
      context.clearRect(0, 0, canvas.width, canvas.height);
}

function disappear() {
    var x = document.getElementById("textbar");
    if (x.style.display === "none") {
        x.style.display = "block";
    } else {
        x.style.display = "none";
    }
}

function dragElement(elmnt) {
  var pos1 = 0, pos2 = 0, pos3 = 0, pos4 = 0;
  if (document.getElementById(elmnt.id + "header")) {
    /* if present, the header is where you move the DIV from:*/
    document.getElementById(elmnt.id + "header").onmousedown = dragMouseDown;
  } else {
    /* otherwise, move the DIV from anywhere inside the DIV:*/
    elmnt.onmousedown = dragMouseDown;
  }

  function dragMouseDown(e) {
    e = e || window.event;
    // get the mouse cursor position at startup:
    pos3 = e.clientX;
    pos4 = e.clientY;
    document.onmouseup = closeDragElement;
    // call a function whenever the cursor moves:
    document.onmousemove = elementDrag;
  }

  function elementDrag(e) {
    e = e || window.event;
    // calculate the new cursor position:
    pos1 = pos3 - e.clientX;
    pos2 = pos4 - e.clientY;
    pos3 = e.clientX;
    pos4 = e.clientY;
    // set the element's new position:
    elmnt.style.top = (elmnt.offsetTop - pos2) + "px";
    elmnt.style.left = (elmnt.offsetLeft - pos1) + "px";
  }
```

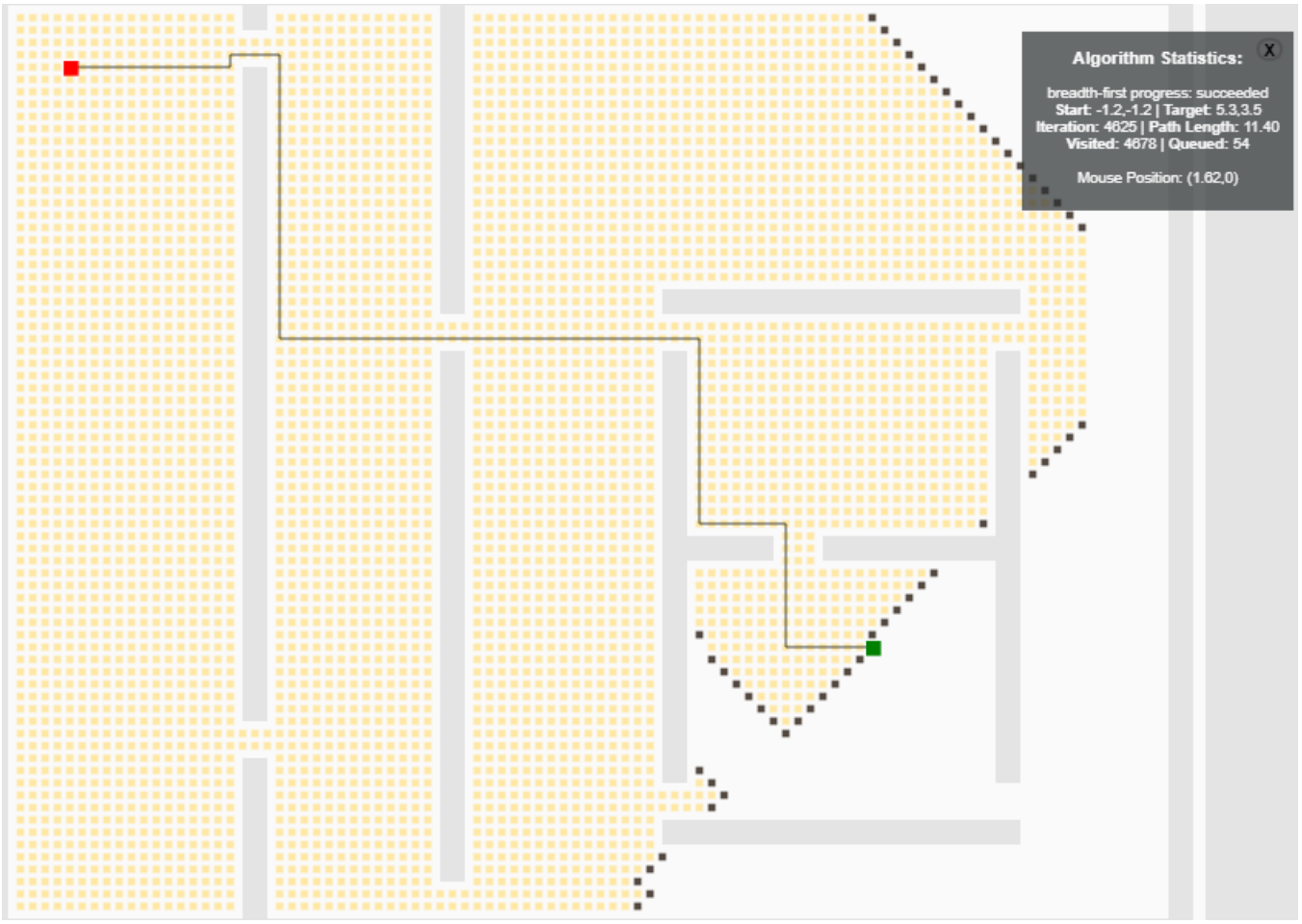
```
  function closeDragElement() {
    /* stop moving when mouse button is released:*/
    document.onmouseup = null;
    document.onmousemove = null;
  }
}
```

OUTPUT

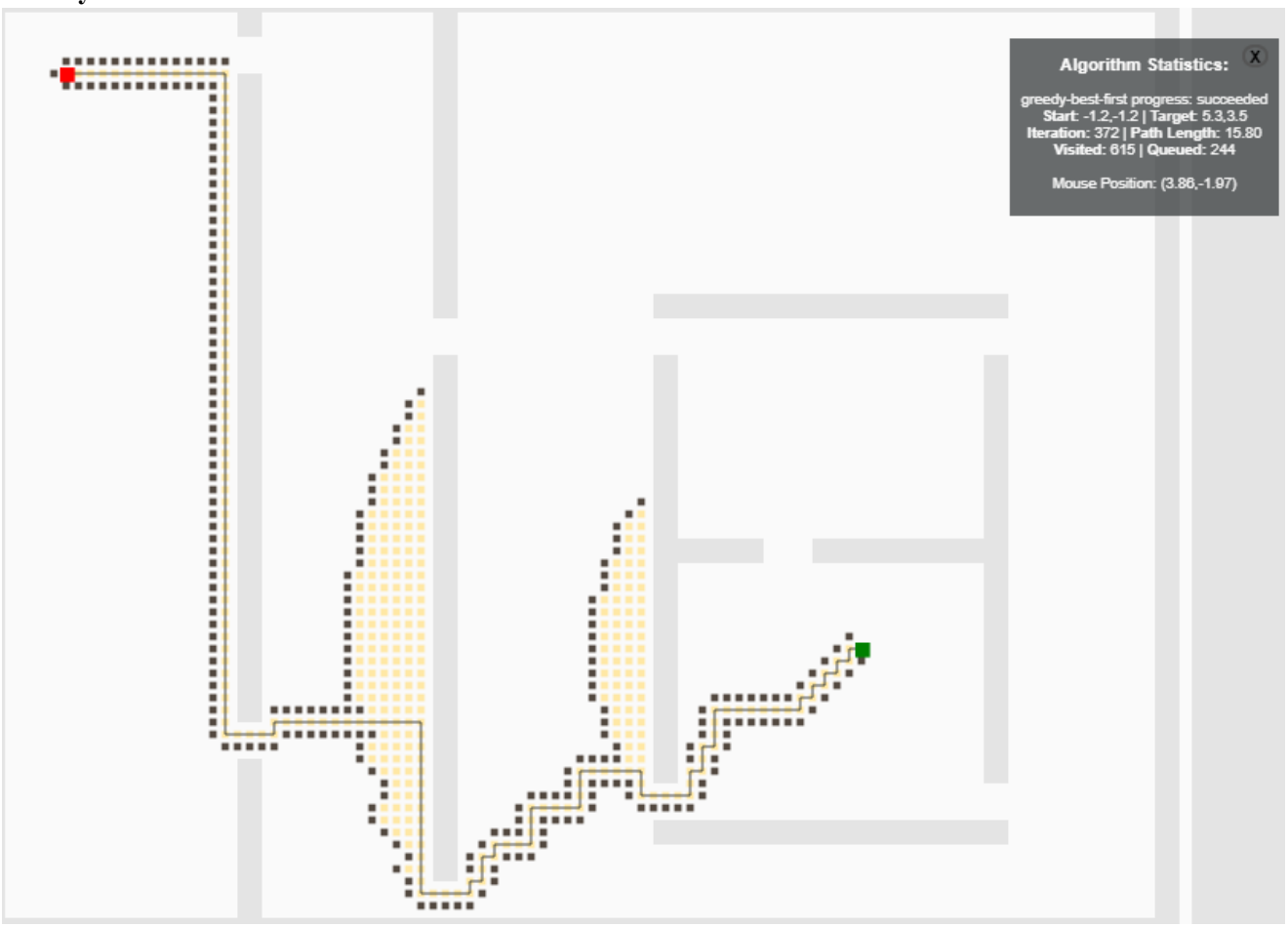
Depth First Search



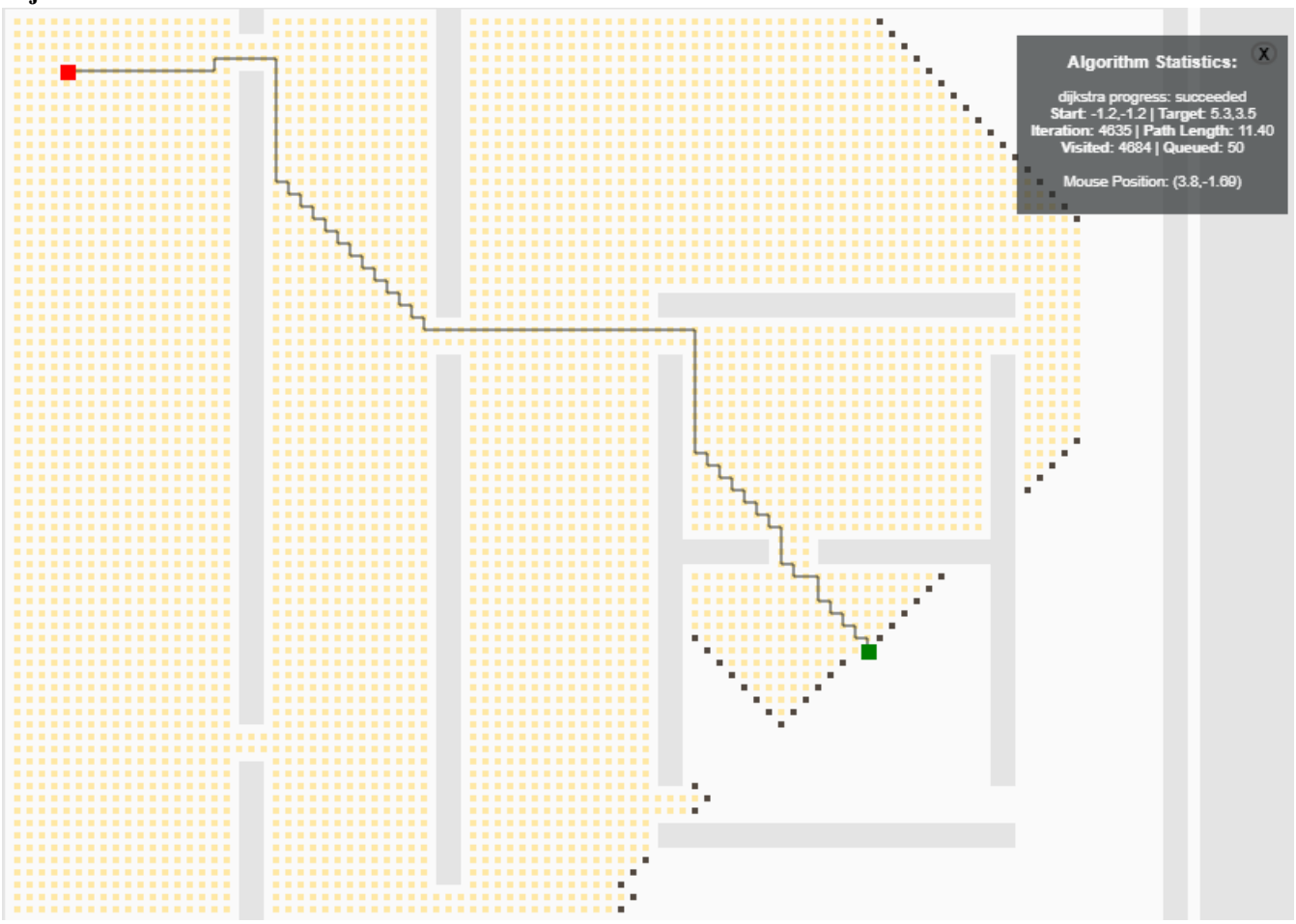
Breadth first search



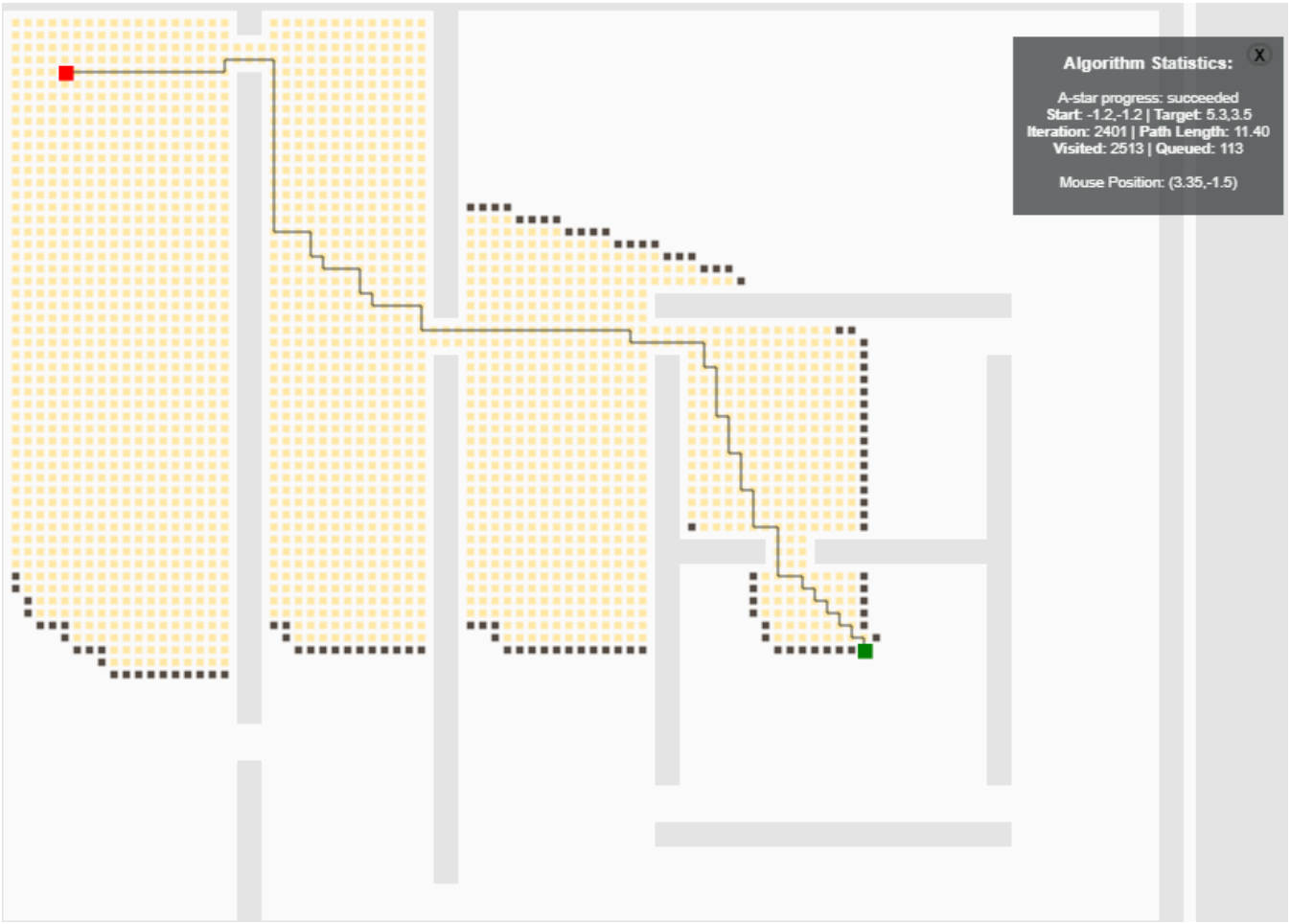
Greedy Best first search



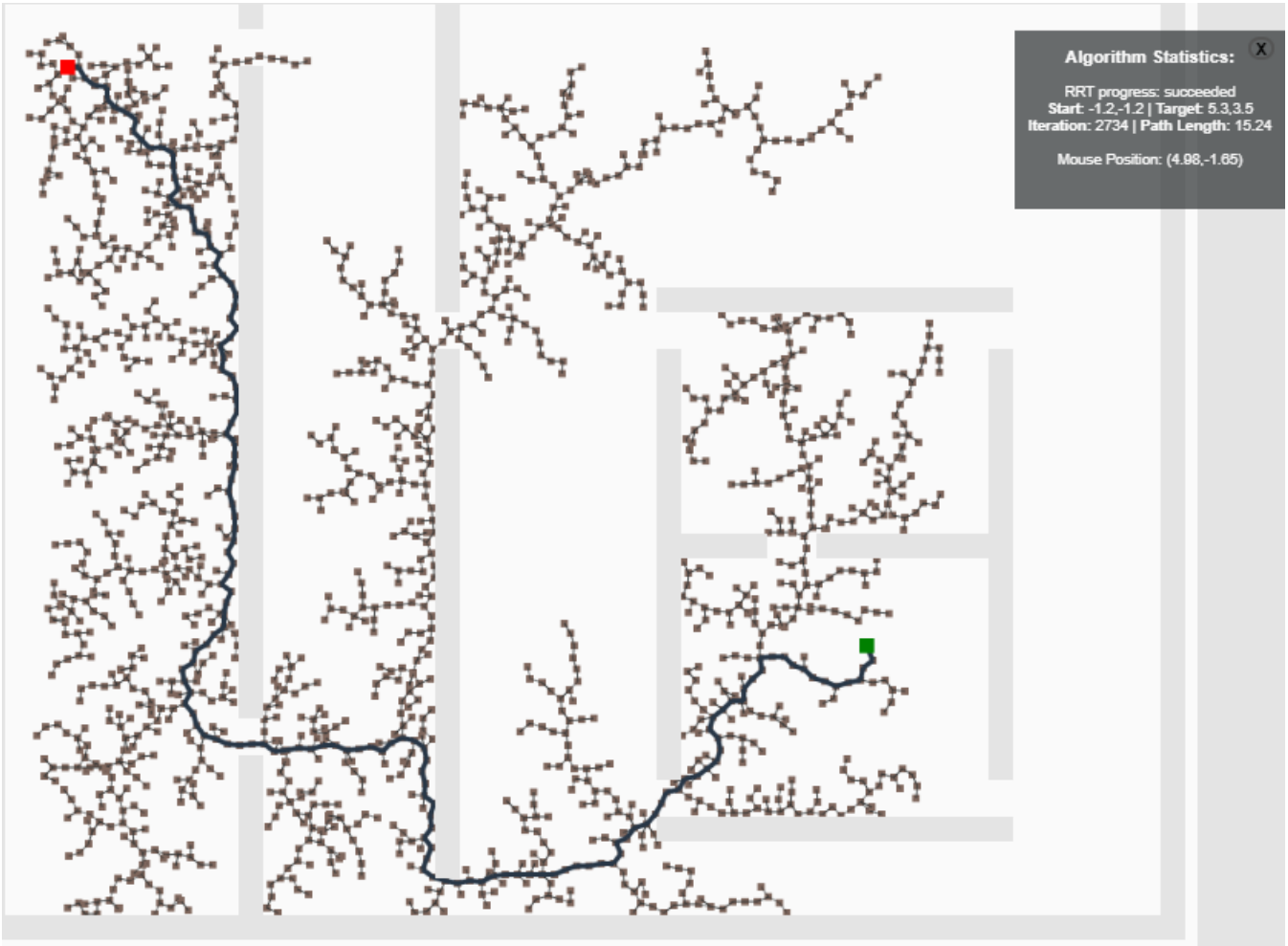
Dijkstra's


Algorithm Statistics:
dijkstra progress: succeeded
Start: -1.2,-1.2 | Target: 5.3,3.5
Iteration: 4635 | Path Length: 11.40
Visited: 4684 | Queued: 50
Mouse Position: (3.8,-1.69)

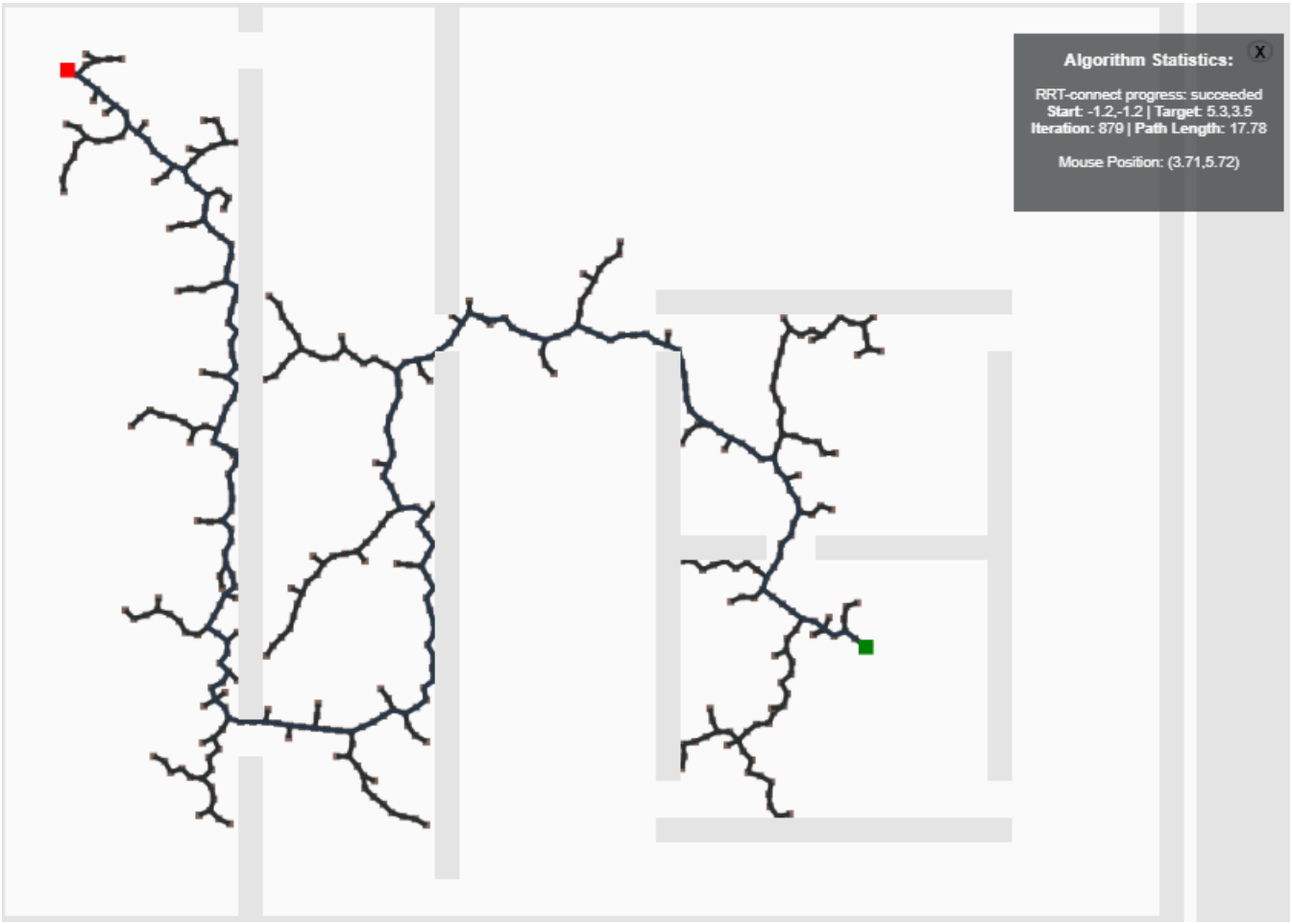
A*


Algorithm Statistics:
A-star progress: succeeded
Start: -1.2,-1.2 | Target: 5.3,3.5
Iteration: 2401 | Path Length: 11.40
Visited: 2513 | Queued: 113
Mouse Position: (3.35,-1.5)

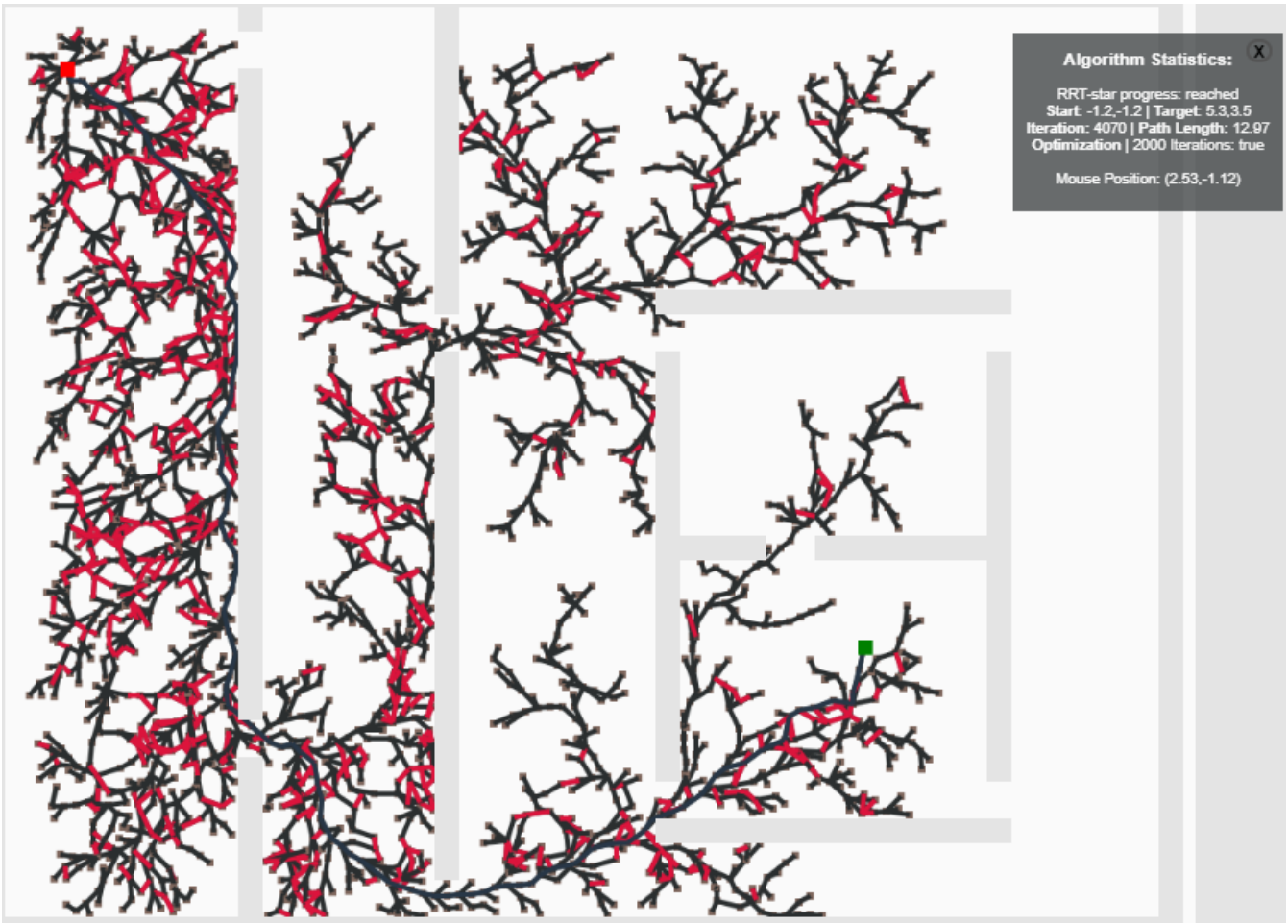
RRT


Algorithm Statistics:
RRT progress: succeeded
Start: -1.2,-1.2 | Target: 5.3,3.5
Iteration: 2734 | Path Length: 15.24
Mouse Position: (4.98,-1.65)

RRT Connect


Algorithm Statistics:
RRT-connect progress: succeeded
Start: -1.2,-1.2 | Target: 5.3,3.5
Iteration: 879 | Path Length: 17.78
Mouse Position: (3.71,5.72)

RRT-Star


Algorithm Statistics:
RRT-star progress: reached
Start: -1.2,-1.2 | Target: 5.3,3.5
Iteration: 4070 | Path Length: 12.97
Optimization | 2000 Iterations: true
Mouse Position: (2.53,-1.12)

7 CONCLUSION

As can be seen from the above results, Jump Point Search is the fastest algorithm. The basis of this is the pruning algorithms which cut down on the number of steps for path search and make the program faster by around ten times. Searching one full row, column or diagonal for the goal using the pruning rules defined is expensive, but the number of times this has to be done is very less. Hence, JPS takes lesser number of more expensive steps.

The A* algorithm (as well as the D* algorithm for the static case) performs ten times slower than the JPS in the average case. This is faster or slower than Dijkstra's algorithm based on the heuristic chosen. Dynamic weighting of the A* algorithm can also be done to improve its efficiency. It is faster because it gets an estimate as to what the cost from the current node to the goal could be and hence plans its path accordingly.

Dijkstra's algorithm comes after A*, which yields a path of somewhat better quality than BFS due to inclusion of cost. However, if the region to be covered is uniform, their performance is the same. Execution also depends on the implementation of the priority queue. If it performs slower than that of queues, Dijkstra's speed might suffer. Optimization of this code is hence necessary.

Breadth First Search offers the worst performance. It owes this to an unstructured blind approach of growing the size of the wave front until the goal node is reached. In large uniform spaces, it may perform equivalent to Dijkstra's algorithm, but in the rest of the cases it fails.

Apart from the comparisons based on execution time, we can see that the algorithms perform in the same order when compared in terms of the number of tiles explored by the algorithms. JPS explores the least number of nodes, followed by D*(in dynamic environments) then A*, Dijkstra and BFS.

Compiler optimizations can also be made to increase the speed of each of these algorithms. Methods to increase the speed of A* have been presented in as well. However, they have not been implemented and the claim that they outperform A* is based on literature reviews.

8 WEBPAGE

https://jacobjohn2016.github.io/robotics-path-planning-simulation/

9 CODE REPOSITORY

https://github.com/jacobjohn2016/robotics-path-planning-simulation