A Report on

Path Planning in Single and Multi-Robot Systems

Submitted to:

Dr. Sandeep Dhar
Assistant Professor (Mechanical Department)
BITS Pilani, Pilani Campus

Mr. Avinash Gautam
Lecturer (CSIS Department)
BITS Pilani, Pilani Campus

By:

Sahib Singh Dhanjal (2012A4PS289P)

A Report on

Path Planning in Single and Multi-robot Systems

Submitted in the partial fulfilment of requirements
BITS F421T Thesis

By:
Sahib Singh Dhanjal
(2012A4PS289P)

Under the supervision of
Dr. Sandeep Dhar & Mr. Avinash Gautam

May, 2016
Birla Institute of Technology and Science, Pilani Campus
Pilani (Rajasthan)

Acknowledgement

I would like to start by thanking my mentors, Dr. Sandeep Dhar (Assistant Professor, Mechanical Engineering Department) and Mr. Avinash Gautam (Lecturer, CSIS Department) for giving me an opportunity to work on this project, under their guidance and support.

I would also like to express heartfelt gratitude towards Arjun, Anant, Nitesh and Pranesh (friends) for constantly helping me out throughout my project.

I am also thankful to the people from the reddit and stackoverflow community for helping me out whenever necessary.

Last but not the least, I would like to thank Dr. Sudeept Mohan and Dr. B.K. Rout for their valuable insights and constant support.

Certificate

This is to certify that the Thesis entitled, Path Planning in Single and Multi-robot Systems and submitted by Sahib Singh Dhanjal, ID No. 2012A4PS289P in partial fulfillment of the requirement of BITS F421T/BITS F422T/BITS F423T/BITS F424T Thesis embodies the work done by him/her under my supervision.

Date: _____

_____

Signature of Supervisor

Dr. Sandeep Dhar
Assistant Professor
(Mechanical Engineering Dept.)

_____

Signature of Co-Supervisor

Mr. Avinash Gautam
Lecturer
(CSIS Dept.)

ABSTRACT

In the world of Artificial Intelligence, exploration and pathfinding is one of the major problems in which extensive research is being carried out. Our lab (Inspire Lab, BITS Pilani) 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 pathfinding algorithms.

Current work aims to improve the overall efficiency by using a better and more efficient pathfinding 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 pathfinding algorithm. Additionally, this has resulted in reduced time for exploration in an environment for which A* was previously used.

Table of Contents

Path Planning in Single and Multi-Robot Systems

1 INTRODUCTION

Pathfinding is one of the most important problems in the world of artificial intelligence. It is used in game programming, navigation and many military as well as domestic applications. Pathfinding mainly involves finding the optimal path in a known environment. Exploration, a parent to pathfinding, 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 ('*Turtlebots*' [1]). Based on the performance of each of the path-planning algorithms, in simulation and on Turtlebots, a comparative study is being presented. The basis of theoretical comparison is time and space complexity and experimental comparison would be carried out by program profiling. All these measures are introduced in chapter 4.

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 (for example, Yamauchi's method [2], and Burgard's method [3]). All of them principally work on frontier coverage. In frontier exploration, one recognizes a list of potential connected grid cells (Section 3.1.2) 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 [4] (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 (section 3.2.1). 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

3.1 MAP REPRESENTATIONS

Pathfinding algorithms generally work on graphs or mazes. Map representations [5] 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:

3.1.1 Graphs

Graphs [5] 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.

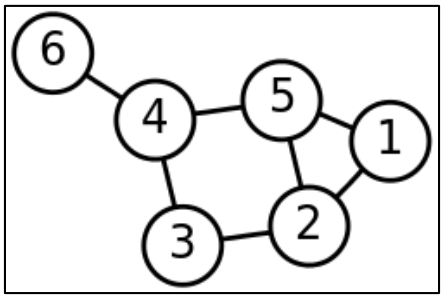


Fig.1 Graph Representation

3.1.2 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*** [5]. 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:

3.1.2.1 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*** [5]. 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.

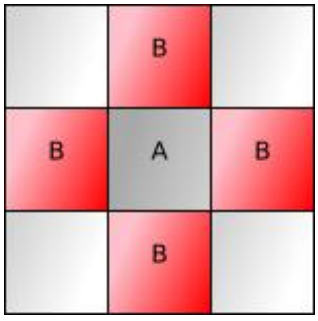


Fig.2 Tile Movement from A ->B

3.1.2.2 Edge Movement

When an object moves across a map just following the edges of the tiles, the type of motion is called ***edge movement*** [5]. 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.

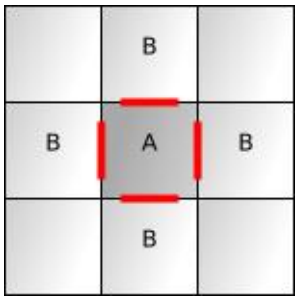


Fig.3 Edge Movement from A ->B

3.1.2.3 Vertex movement

When an object moves across a map following only the vertices of the tiles, the type of motion is called ***vertex movement*** [5]. 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.

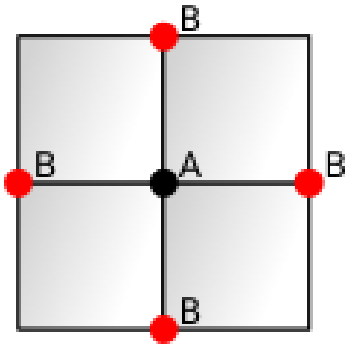


Fig.4 Vertex Movement from A ->B

3.1.3 Polygonal Maps

An alternative to the grid representation is polygonal maps [5]. 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.

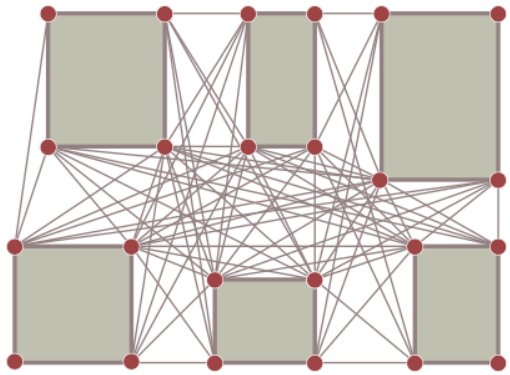


Fig.5 Complex Polygonal maps

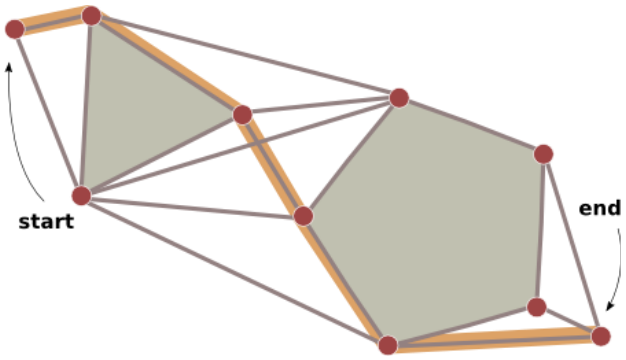


Fig.6 Path from start to end in map

3.1.4 Navigation Meshes

We can represent the walkable areas in a map as a polygons. This type of representation is called a navigation mesh [5]. 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).

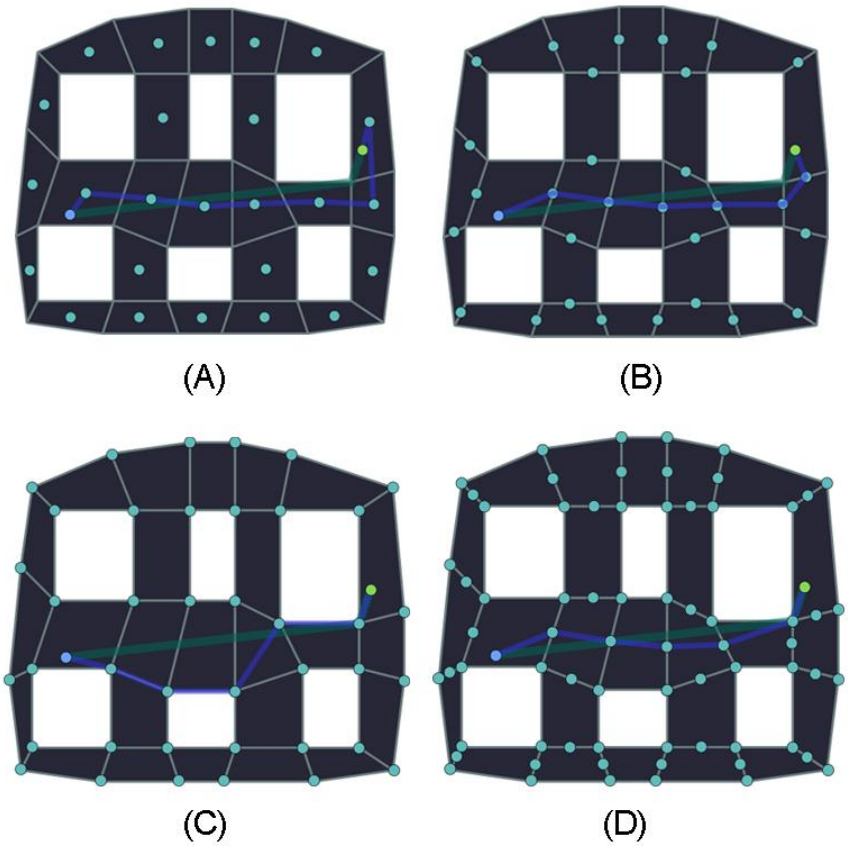


Fig.7 (A) Polygon center movement | **(B)** Edge Movement | **(C)** Vertex Movement | **(D)** Hybrid Movement

3.2 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) [6], Dijkstra's algorithm [7], A-Star Algorithm [8], Dynamic A-Star algorithm [9], and Jump Point Search [10]. The following sections will give a brief of the pathfinding algorithms studied.

3.2.1 Breadth First Search

Breadth First search (BFS) [6] is one of the most basic graph algorithm. 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 [6]. 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. The implementation has been given in the appendix.

3.2.1.1 Time and Space Complexity

The Big-O notation described in Section 4.1 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|) [6], 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 described in Section 4.2 is used to determine the space complexity of the algorithm. Assuming finite size of the input graph, the space complexity can be represented as Θ (|V| + |E|) [6], 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.

3.2.2 Dijkstra's Algorithm

The Dijkstra's Algorithm [7] 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. The implementation has been given in the appendix.

3.2.2.1 Time and Space Complexity

The time complexity of the Dijkstra's algorithm can be represented as O (|V| log |V| + |E|) [7], where V and E are the number of vertices and edges. The space complexity is represented as Θ ((|V|+ |E|) log |V|) [7]. 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.

3.2.3 A-Star Algorithm

The A-Star [8] 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 [5]. Like Dijkstra, it also uses the Priority Queue as its data type. The most general Heuristics used in A* are as follows:

3.2.3.1 Heuristics

The heuristic function [5] 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 - $|x1 - x2| + |y1 - y2| - 0.5857 * min(|x1 - x2|, |y1 - y2|)$

3.2.3.2 Algorithm

The algorithm maintains two sets: **O** and **C**, open and closed set respectively [8] 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. The implementation is given in the appendix. A snippet of the algorithm is shown in the snippet below [8]. ***Other definitions (cited from [8])*** 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

3.2.4 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 are 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 [9] [11].

The D* algorithm uses a complex approach in which it redefines the heuristic on encountering an obstacle. This paradigm is explained in [9] [11] and interested readers may refer to it. The implementation of the D*algorithm is given in the appendix.

3.2.5 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 [10] [12] 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). The implementation of the Jump Point Search can be found in the appendix.

3.2.5.1 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:

3.2.5.1.1 Neighbor Pruning

Currently, node x is being expanded. Arrows indicate the direction of travel (ie. 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.

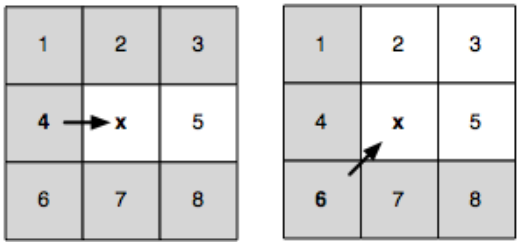


Fig.9 Neighbor Pruning

3.2.5.1.2 Forced Neighbors

Currently, node x is being expanded. Arrows indicate the direction of travel (ie. 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.

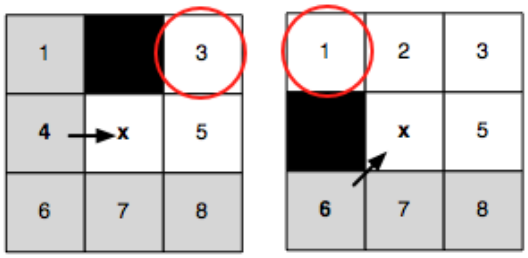


Fig.10 Forced Neighbor

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.

3.2.5.2 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*(Section 3.3.1)

3.3 VARIATIONS OF THE A* ALGORITHM

Many variants of the A* algorithm [5] 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:

3.3.1 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.

3.3.2 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:

$$f(p) = g(p) + w(p) * h(p) \quad - \quad \textit{cited form [5]}$$

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.

3.3.3 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

4.1 TIME COMPLEXITY

The number of instructions a program executes during its execution is called its ***time complexity*** [14]. 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.

4.2 SPACE COMPLEXITY

The ***space complexity*** [14] 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.

4.3 PROGRAM PROFILING

Profiling of a program is breaking the program line by line and finding the parts which decrease the efficiency of the program [13]. 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.

4.3.1 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*** [13] 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: pycallgraph 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 (Section 5.6)

The results are as follows:

5.1 BREADTH FIRST SEARCH

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

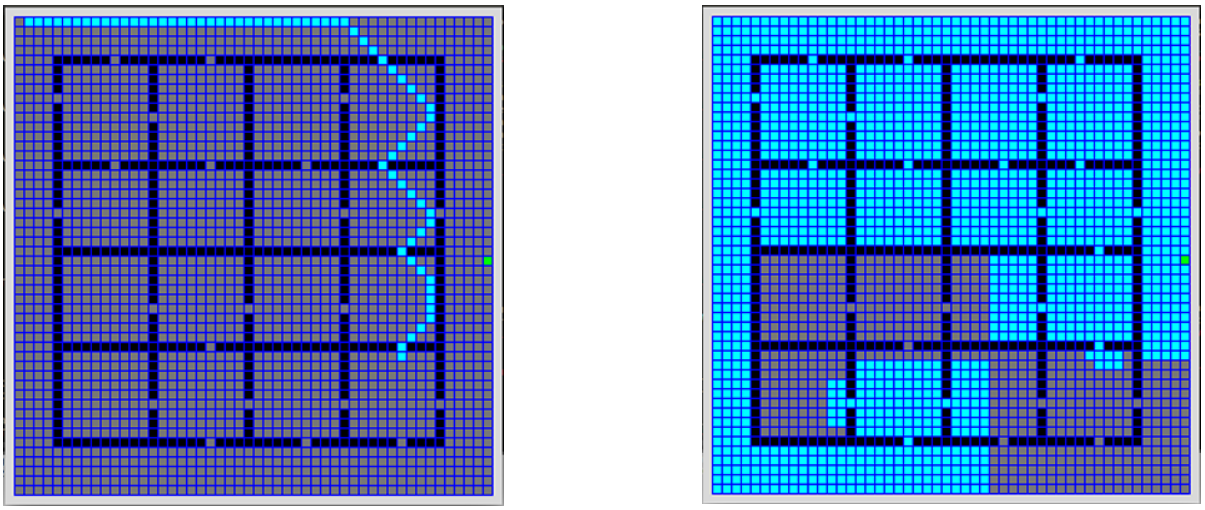


Fig.11 Breadth First Search: **(A)** The Path Found **(B)** the cells explored

5.2 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 (Section 5.6). The cells processed and the path found is also given in the figure below. The results for the Dijkstra's Algorithm are as follows:

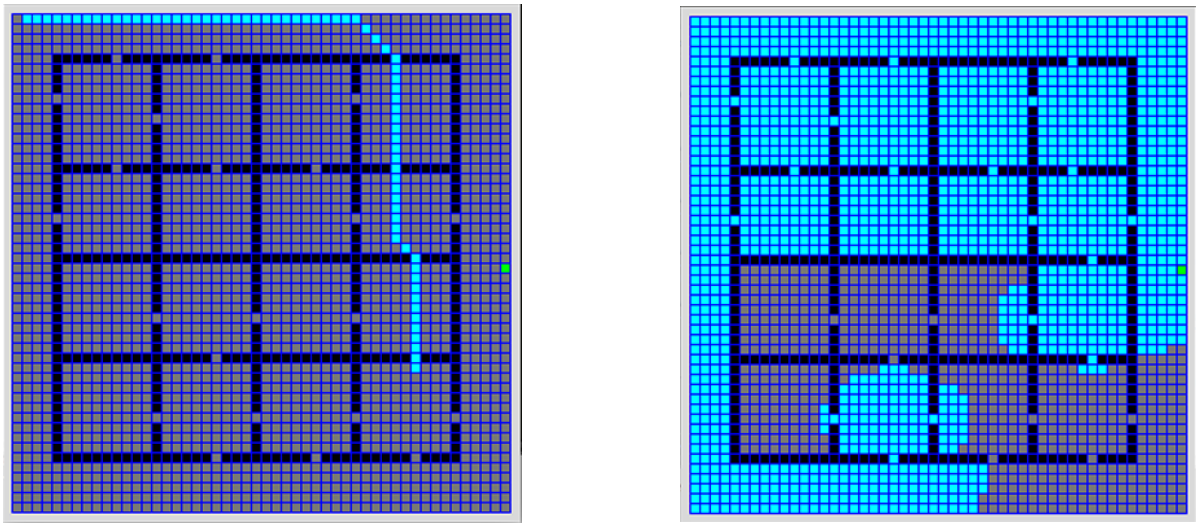


Fig.12 Dijkstra's Algorithm: **(A)** The Path Found **(B)** the cells explored

5.3 A* ALGORITHM

Open set and Total length are the same as in Dijkstra's. The cells processed and the path found is also given in table 1 (Section 5.6). The results for A* are as follows:

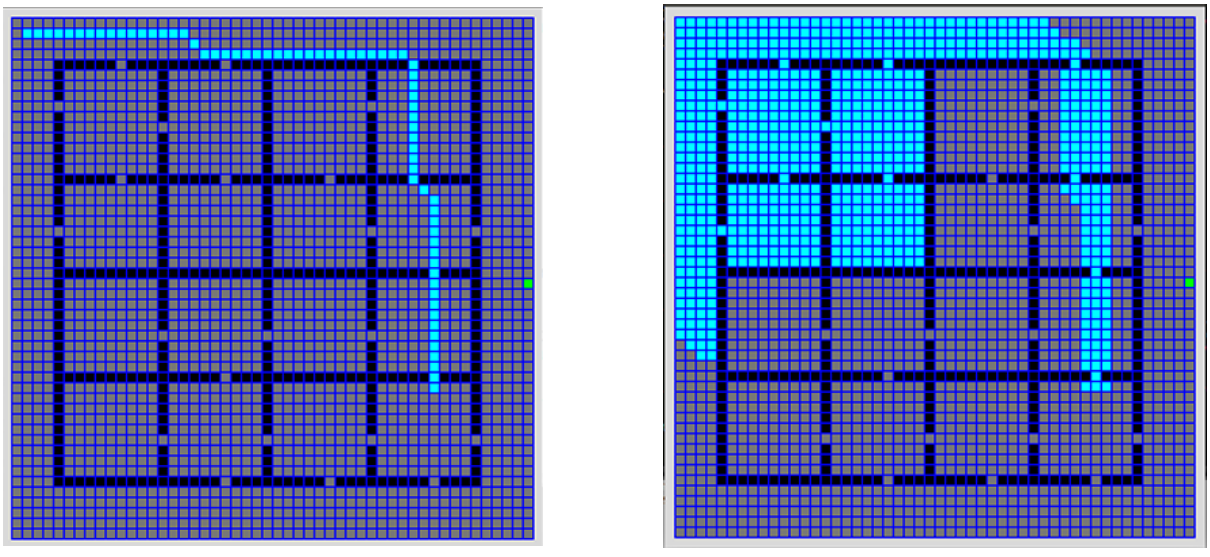


Fig.13 A*Algorithm: **(A)** The Path Found **(B)** the cells explored

5.4 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.

5.5 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:

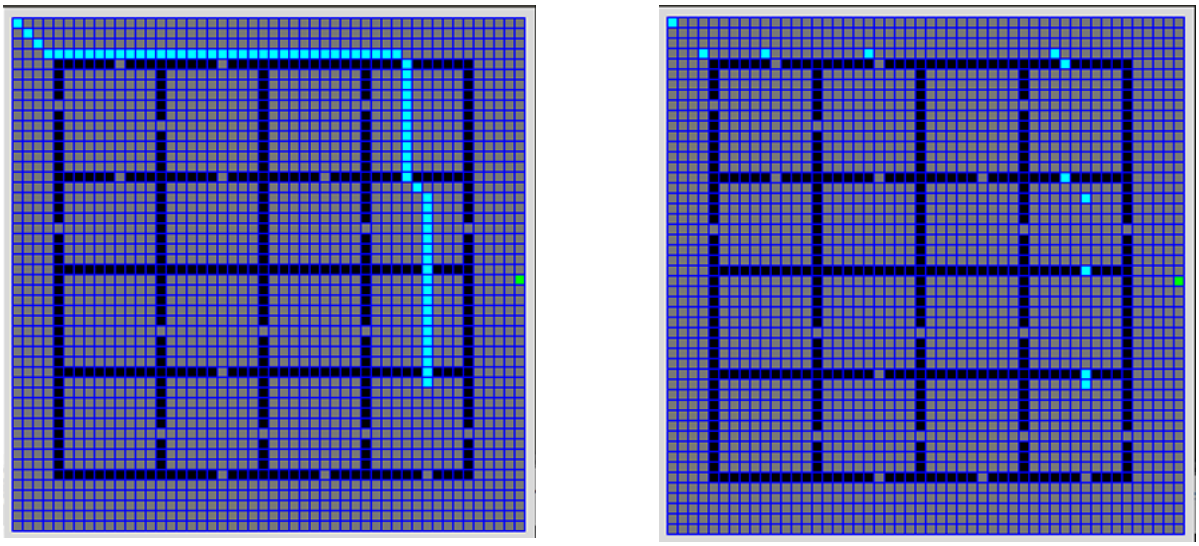


Fig.14 Jump Point Search: **(A)** The Path Found **(B)** the cells explored

5.6 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 graphical representations of the profiles obtained via Python Call Graph are shown in the appendix. The results obtained (from the graphs in the appendix) 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

6.1 EXPERIMENT SETUP

The experiment was performed on a commercially available robot known as the "Turtlebot" [1]. It operates on the open source ROS (Robot Operating System) [15] environment. A brief description of the ROS environment and how to set it up is given in the section below. Setting up communication on the Turtlebot is also necessary to operate it from a host computer. This will also be discussed in further sections.

6.1.1 Turtlebot

Turtlebot is an open source hardware platform and mobile base. When powered by ROS software, Turtlebot can handle vision, localization, communication and mobility. It can autonomously navigate avoiding obstacles along the way. It contains two basic components:

1. A mobile "Kobuki" base which is a two-wheeled differential drive robot with its own controller
2. A Microsoft Kinect 3D sensor which helps it sense it's environment

To communicate with the Turtlebot, it is required to attach a laptop to it. An ASUS Netbook is included when you buy a Turtlebot.

6.1.2 Robot Operating System (ROS)

One of the most predominantly used software framework in robotics is the ROS. It handles all the low level abstraction and lets developers focus on the higher part. Low level abstraction takes into account moving and maneuvering the robot by controlling its wheel speeds, getting sensor data and mapping them into useful data, network connections and other basic underlying frameworks necessary for the whole electro-mechanical robot to function perfectly. The higher level which developers work on focuses on how the robot should move, what to do with the sensor data, and other tasks that one would like to accomplish. A lot of research institutions have started to develop projects in ROS by adding hardware and sharing their code samples. Also, the companies have started to adapt their products to be used in ROS. The sensors and actuators used in robotics have also been adapted to be used with ROS. ROS provides standard operating system facilities such as hardware abstraction, low-level device control, implementation of commonly used functionalities, message passing between processes, and package management. It is based on graph architecture with a centralized topology where processing takes place in nodes that may receive or post, such as multiplex sensor, control, state, planning, actuator, and so on.

The library is geared towards a Unix-like system (Ubuntu Linux is listed as supported while other variants such as Fedora and Mac OS X are considered experimental). ROS also comes with its own simulator environment which we would be using to visualize the paths produced in this work. A brief on setting up ROS and the Turtlebot is given in the following section.

6.1.2.1 Setting up Ubuntu

Ubuntu is a (free) community-based Linux distribution with ample support for each of its versions. The version we would be using is the Ubuntu 14.04 LTS version compatible with the ROS Indigo environment. You can follow this thread [16] to have a detailed description on how to install Ubuntu.

6.1.2.2 Setting up ROS Indigo

Follow the given procedure step-by-step to install ROS.

1. open a terminal and run the following snippets
2. Get the latest updates by running

```
sudo apt-get update
```

3. Setup your sources.list by running

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

4. Set up your keys by running

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116
```

5. Install ROS by running

```
sudo apt-get install ros-indigo-desktop-full
```

6. Initialize rosdep by running

```
sudo rosdep init
rosdep update
```

7. Setup up environment

```
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

8. Install rosinstall for other packages

```
sudo apt-get install python-rosinstall
```

ROS is now successfully installed. To verify, restart the computer, open a new terminal and type `roscore`. ROS is correctly installed if you view a message `started core service [/rosout]`.

6.1.3 Setting up networking in the Turtlebot

The first step is to install ssh-server on both the Turtlebot-PC and the remote-PC. To do this open a terminal and type in: `sudo apt-get install openssh-server`

Once successfully installed, follow the article here [17] to configure the network.

6.1.4 Writing the Plugin

The plugin is specifically targeted to work with ROS Indigo. It may not work with other versions of ROS. The plugins are to be written in C++ only. All the steps followed in this section are to be done on the Turtlebot netbook. To make a plugin, please follow the steps in this section:

1. Write a class header file (eg. pathplanners.h). The header file used for this experiment is given in the appendix.
2. Implement the class (eg. Pathplanners.cpp). The class used for this experiment is given in the appendix.
3. Create a catkin workspace for both the Turtlebot and the remote-PC. To create a workspace, simply follow this tutorial [18].
4. Once a catkin workspace is created, open a terminal and type in the following commands:
   4.1. **cd catkin_ws/src** -> here catkin_ws is the name of your workspace
   4.2. **catkin_create_pkg pathplanners nav_core roscpp rospy std_msgs pluginlib** -> catkin_create_pkg is the standard function to make a new package. Pathplanners is the name of the package and it is dependent on the libraries roscpp, rospy, nav_core, std_msgs, and pluginlib. All of these are inherited from the ROS directories.
5. Now open the folder *~/catkin_ws/src/pathplanners/src* and move the class header (*pathplanners.h*) and class implementation (*pathplanners.cpp*) files in it.
6. To compile the files, open the **CMakeLists.txt** file in the folder *~/catkin_ws/src/pathplanners/* and search for *add_library* in the document. Once found add the following snippet below the list of comments -> **add_library(pathplanners src/pathplanners.cpp)**
   The default snippet to add a third party library is *add_library*. *pathplanners* here is the name of the library which would be created. *src/pathplanners.cpp* is the location of the .cpp file
7. Now return back to the terminal and type in **cd ..**
8. This would take us to the directory /catkin_ws
9. Now type in the command **catkin_make.** This is the default command to build the packages in the workspace.
10. If you receive a message saying *[100%] Built target pathplanners*, it means that your plugin is successfully been compiled and added.
11. To register your plugin to the ROS environment you need to add the following snippet to the pathplanners.cpp file

```
PLUGINLIB_EXPORT_CLASS (PathPlanners::PathPlanners, nav_core::BaseGlobalPlanner)
```

    PLUGINLIB_EXPORT_CLASS is the default function using which custom plugins are registered. In the arguments we pass <namespace> (PathPlanners) :: <class name> (PathPlanners) and the class to which the plugin needs to be registered which in our case is <namespace>nav_core :: <class name> BaseGlobalPlanner.
12. Once done. Run **catkin_make** once again to register the plugin.
13. Now open the folder *~/catkin_ws/devel/lib* and you can see a file named **libpathplanners.so** in there.
14. This means that our plugin has been created. We need to make a plugin description file now.
15. The plugin description file (*pathplanners_plugin.xml*) used in this experiment is given in the appendix.
16. Place the plugin file in the folder *~/catkin_ws/src/pathplanners/*

17. To register the plugin with the ROS package system, open the package.xml file in the same folder and find the term *<export>*. Once found between the <export> and </export> tags add the snippet

```
<nav_core plugin="${prefix}/pathplanners_plugin.xml" />
```

    Here pathplanners_plugin is the name of the plugin file that we made in step 15.
18. Now open a new terminal closing all the opened ones and type in

```
rospack plugins --attrib=plugin nav_core
```

    You would see a line in the output similar to pathplanners */home/bot/catkin_ws/src/pathplanners/pathplanner_plugin.xml*. This means that our plugin has been successfully registered.
19. Now since the plugin is created, we can run it. To do so open a new terminal and type in the following commands:

```
$ roscd turtlebot_navigation/
$ cd launch/includes/
```

20. This will navigate you directly to the directory */opt/ros/hydro/share/turtlebot_navigation/launch/includes*
21. Open the file move_base.launch.xml by typing the following command:
    sudo gedit move_base.launch.xml
22. Now search for the line "*node pkg =*"and add the following snippet

```
.....
<node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
<param name="base_global_planner" value="PathPlanners/PathPlanners "/>
....
```

23. Once registered, you need to make a launch file for this package. The launch file used for this purpose is given in the appendix and is well commented.
24. Once the launch file is made, you need to create a pseudo-map of the environment so that the bot has some initial physical origin and an idea of where the obstacles can be. To make a pseudo-map follow the next section closely.
25. Once you have an initial map, make 2 folders in the directory */home/bot/catkin_ws/src/pathplanners/*. One would be to store the maps (name the folder maps) and the other (name it launch) to store the launch file. The maps would be a *.pgm file + a *.yaml file.
26. Now once the map files and the launch files are placed in the respective folders, the plugin is ready to be launched. The procedure to launch the code is as follows:

    Turtlebot:

    1. Connect the netbook of the Turtlebot to the Kinect and the Turtlebot via the USB-ports.
    2. Launch a terminal and type in the code roslaunch <folder name> <launch-file name>

3. In our case, it was

```
roslaunch pathplanners pathplanner.launch
```

Remote – PC:

1. Launch a terminal on the remote PC and after launching the code on the Turtlebot, type the following snippet in the terminal and launch.

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

2. You would now see an RVIZ window opening up. Once the bot gets sensor data, you need to give the estimated initial pose to the robot and then a goal node. Once the bot receives both, it will localize itself and will launch the underlying libraries responsible for the motion.

27. If any part of the procedure is difficult to comprehend, a video of the whole procedure has been made and you can watch it here [19].

6.1.5 Mapping the environment

The mapping of the environment will be done via teleoperation. Make sure the Turtlebot and the workstation are able to communicate over Wi-Fi.

On the **Turtlebot**, open two terminal windows and run:

```
roslaunch turtlebot_bringup minimal.launch
roslaunch turtlebot_navigation gmapping_demo.launch
```

On the **workstation**, open two terminal windows and run:

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Now on the workstation, you can see the keys you need to press to move the Turtlebot. Move the bot in the desired area until a satisfactory map is made. To save the made map, type the following command on the workstation terminal (new terminal) `rosrun map_server map_saver -f /home/bot/catkin_ws/src/pathplanners/maps mymap.`

Rosrun is the default command to run an app from the ROS ecosystem. *Map_server* is the application which has a class *map_saver* which saves a map in the directory */home/bot/…/maps* with the name *mymap*.

6.2 Experimentation Results

The results obtained are derived on the basis of the written plugins and the path planners which were already there in the global_planner package in ROS. The results obtained are shown in Table 2

SN	Algorithm	Function Calls	Execution Time
1	Breadth First Search	1	0.003246 secs
2	Dijkstra's Algorithm	1	0.003071 secs
3	A* Algorithm	1	0.002793 secs
4	Jump Point Search	1	0.000352 secs

Table 2

Jump Point Search performs the fastest in this case too. The execution times are medians of the time taken to calculate the path length which varies from a path of 1m to over 10m.

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 Section 3.3 as well. However they have not been implemented and the claim that they outperform A* is based on literature reviews.

8 REFERENCES

[1] http://www.turtlebot.com/

[2] B. Yamauchi, "A frontier-based approach for autonomous exploration," in Proc. of IEEE Int. Symp. On Computational Intelligence in Robotics and Automation. IEEE Computer. Soc. Press, 1997, pp. 146-151

[3] W. Burgard, et al., "Coordinated multi-robot exploration," in IEEE Trans. on Robotics, June 2005, vol.21, no.3, pp.376-386

[4] http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7379179

[5] http://theory.stanford.edu/~amitp/GameProgramming/

[6] https://en.wikipedia.org/wiki/Breadth-first_search

[7] https://en.wikipedia.org/wiki/Dijkstra's algorithm

[8] Principles of Robot Motion – Howie Choset, Lynch, and Hutchinson

[9] http://robotics.usc.edu/~geoff/cs599/D_Lite.pdf

[10] https://harablog.wordpress.com/2011/09/07/jump-point-search/

[11] www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf

[12] https://users.cecs.anu.edu.au/~dharabor/pathfinding.html

[13] http://stackoverflow.com/questions/582336/how-can-you-profile-a-python-script

[14] https://julien.danjou.info/blog/2015/guide-to-python-profiling-cprofile-concrete-case-carbonara

[15] http://www.ros.org/

[16] http://askubuntu.com/questions/6328/how-do-i-install-ubuntu

[17] http://wiki.ros.org/turtlebot/Tutorials/indigo/Network%20Configuration#Robots.2BAC8-TurtleBot.2BAC8-Robot_Setup.2BAC8-ntp.Network_Time_Protocol

[18] http://wiki.ros.org/catkin/Tutorials/create_a_workspace

[19] https://www.youtube.com/watch?v=We1gGDXAO_o&list=PL93n88K6Qpb63pyaaPaOTudQB30Z2Qn4r

APPENDICES

1. Appendix A – Algorithm Implementations
2. Appendix B – Profiling Results
3. Appendix C – Turtlebot Code

APPENDIX A

1. Algorithm Implementations (in Python v2.7)
    1.1. Breadth First Search
    1.2. Dijkstra's Algorithm
    1.3. A* Algorithm
    1.4. D* Algorithm
    1.5. Jump Point Search

1. Breadth First Search

```
class BFS:
    def __init__(self):
        pass

    def BFSsearch(self,gridworld,start,goal):
        frontier=Queue()
        came_from={}
        frontier.put(start,[start]) ; came_from[start] = None
        path=[] ; cost={}
        cost[start] = 0

        while not frontier.isEmpty():
            current,path=frontier.get()

            if current==goal:
                break
            for next in gridworld.get8Neighbors(current):
                if next not in came_from:
                    frontier.put(next,path+[next])
                    came_from[next]=current
                    cost[next] = cost[current]+1

        return path,cost
```

2. Dijkstra's Algorithm

```
class Dijkstra:
    def __init__(self):
        pass

    def DijkstraSearch(self,gridworld,start,goal):
        frontier=PriorityQueue()
        came_from={} ; path=[] ; cost={}

        frontier.put((start,[start]),0) ; came_from[start]=None ; cost[start]=0

        while not frontier.isEmpty():
            current,path=frontier.get()

            if current==goal:
                break

            for next in gridworld.get8Neighbors(current):

                if next[0]==current[0] or next[1]==current[1]:
                    newcost = cost[current] + 5
                else:
                    newcost = cost[current] + 7

                if next not in came_from and newcost < cost[next]:
                    cost[next] = newcost
                    priority = newcost
                    frontier.put((next,path+[next]),priority)
                    came_from[next]=current

        return path, cost
```

3. A* Algorithm

```
class AStar:
    def __init__(self):
        pass

    def heuristics(self, a, b, flag=1):
        dx=abs(b[0]-a[0])
        dy=abs(b[1]-a[1])

        # Manhattan Heuristic
        if n==1:
            return dx+dy

        # Euclidean Heuristic
        elif n==2:
            return (dx**2 + dy**2)**0.5

        # Chebychev Heuristic
        elif n==3:
            return max(dx,dy)

        # Octile Heuristic
        else:
            return dx+dy+(2**0.5-2)*min (dx,dy)


    def AStarSearch(self, gridworld,start,goal,dynamic=False):
        frontier=PriorityQueue()
        came_from={} ; cost= {} ; path=[]
        frontier.put((start,[start]),0) ; came_from[start] = None ; cost[start]=0

        while not frontier.isEmpty():
            current,path=frontier.get()

            if current==goal:
                break
```

```
    for next in gridworld.get8Neighbors(current):
        if next[0]==current[0] or next[1]==current[1]:
            newcost = cost[current] + 5
        else:
            newcost = cost[current] + 7

        if next not in came_from and newcost<cost[next]:
            cost[next] = newcost

            # dynamic weighting added to a-star algorithm for achieving results faster
            if not dynamic:
                priority = newcost + self.heuristic(next,goal)
            else:
                if self.heuristic(next,goal)<5:
                    w=1
                else:
                    w=5
                priority = newcost + w * self.heuristic(next,goal)

            frontier.put((next,path + [next]),priority)
            came_from[next]=current

return path,cost
```

4. D* Algorithm

```
from heapq import *


maxint=9999

def readmap(filename):
   fp=open(filename,"rb")
   r=0;m=[]
   for line in fp.readlines():
      c=0;row=[]
      for i in line.strip():
         row.append(int(i))
         c+=1
      m.append(row);r+=1
   return m


def Heuristic(m,g): #input as map and goal, output as matrix with heuristic values
            nrow=len(m);ncol=len(m[0])
            (r,c)=g
            kmap=[[0]*ncol for i in range(nrow)]
            hmap=[[0]*ncol for i in range(nrow)]
            kmap[r][c]=0

            for i in range(nrow):
                        for j in range(ncol):
                                    (x,y)=(i,j)
                                    dr=abs(r-x);dc=abs(c-y);
                                    if dr<dc:
                                                kmap[i][j]+=1.4*dr+(dc-dr)
                                                if m[i][j]==1:
                                                            hmap[i][j]=kmap[i][j]+maxint
                                                else:
                                                            hmap[i][j]=kmap[i][j]
                                    else:
                                                kmap[i][j]+=1.4*dc+(dr-dc)
                                                if m[i][j]==1:
                                                            hmap[i][j]=kmap[i][j]+maxint
                                                else:
                                                            hmap[i][j]=kmap[i][j]
```

```
        result=[];result.append(kmap);result.append(hmap)
        return result


def neighbors(n):
        l=[(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]          # adjacency list
        (x,y)=n; alist=[]
        for i in l:
                (dx,dy)=i
                if 0<=x+dx<len(m) and 0<=y+dy<len(m[0]):
                        alist.append((x+dx,y+dy))
        return alist


def changestate(a):
        (x,y)=a
        if m[x][y]==0:
                m[x][y]=1
        else:
                m[x][y]=0
        return m[x][y]


def Dijkstra(m,start,goal):
        h=[];hmap=Heuristic(readmap("maze_2.txt"),(0,6))[0]
        kmap=hmap;(srcx,srcy)=start;(desx,desy)=goal

        prev={};close_set=set();open_set=set();new_set=set()

        gscore={goal:0}
        fscore={goal:Heuristic(m,start)[0]}
        heappush(h,(kmap[desx][desy],goal))

        while h:
                curr=heappop(h)[1]

                if curr== start:
                        path=[]

                        while curr in prev:
                                path.append(curr)
                                curr=prev[curr]
                        path.append(goal)

                        for i in h:
```

```
(c,p)=i
                                open_set.add(p)
                        return path,cost

                close_set.add(curr)
                a=neighbors(curr)
                for i in a:
                        (x,y)=curr
                        (nx,ny)=i
                        temp=kmap[x][y]+kmap[nx][ny]
                        if i in close_set and temp>=gscore.get(i,0):
                                continue
                        if temp<gscore.get(i,0) or i not in [j[1] for j in h]:
                                prev[i]=curr
                                gscore[i]=temp
                                fscore[i]=temp + kmap[nx][ny]
                                heappush(h,(fscore[i],i))
                (cost,pt)=h[0]
        return False

def Dstar(a,hmap,kmap,c=(3,3)):
        (path,cost)=a
        hnew=[];n=[];newpath=[];pnew=[]
        changestate(c)
        for i in range(len(path)):

                (x,y)=path[i]
                prev=path[i-2]

                if m[x][y]==0:
                        hnew.append(path[i])

                else:
                        curr=path[i-1]
                        alist=neighbors(curr)
                        plist=neighbors(prev)

                        for j in alist:
                                (x,y)=j
                                if m[x][y]==0:
                                        n.append(j)
                                        for k in plist:
                                                if k==j:
                                                        hmap[x][y]+=10000
                                                        n.remove(k)
```

```
n.remove(prev)

                        ls=maxint
                        for t in n:
                                (x,y)=t
                                if kmap[x][y]<ls:
                                        ls==kmap[x][y]
                                        temp=t
                        (pnew,cnew)=Dijkstra(m,temp,(0,6))
                        break
        return hnew+pnew


if __name__=='__main__':
        m=readmap("maze_2.txt")
        h,k=Heuristic(m,(0,6))

        for i in range(len(h)):
                for j in range(len(h[0])):
                        print h[i][j],
                print
        print '\n'
        (path,cost)=Dijkstra(m,(5,1),(0,6))

        print('Old Path= {0} \nNew Path= {1} \n'.format(path,Dstar(Dijkstra(m,(5,1),(0,6)),h,k)))
```

5. Jump Point Search

```
# global variables required for the operation of the JPS algorithm
grid = [] ; expanded = [] ; visited = [] ; h = 0 ; w = 0

class JPS:
    def __init__(self):
        pass

    def construct(self,sources,start,goal):
        jump_points=[]
        curr_x,curr_y=goal

        while curr_x!=start[0] or curr_y!=start[1]:
            jump_points.append((curr_x,curr_y))
            curr_x,curr_y=sources[curr_x][curr_y]
            jump_points.reverse()
        return [start]+jump_points


    def full_path(self,a):
        '''
        Take individual jump points as a list of tuples and return full path

        RETURN:
        Full pathof traversal or [] if no path is found
        '''
        a.sort()
        if a==[]:
            return []

        path=[] ; l=len(a) ; cost={} ; c=0

        for i in range(l-1):
            path.append(a[i])

            cost[a[i]]=c

            curr_x=a[i][0] ; curr_y=a[i][1] ; next_x=a[i+1][0] ; next_y=a[i+1][1]
            dx=next_x-curr_x ; dy=next_y-curr_y
```

```
if dy==0:
    if dx > 0:
        for j in range(1,dx):
            c+=5
            path.append((curr_x+j,curr_y))
            cost[(curr_x+j,curr_y)]=c
    if dx < 0:
        for j in range(dx+1,0):
            c+=5
            path.append((curr_x+j,curr_y))
            cost[(curr_x+j,curr_y)]=c

if dx==0:
    if dy >0:
        for j in range(1,dy):
            c+=5
            path.append((curr_x,curr_y+j))
            cost[(curr_x,curr_y+j)]=c
    if dy < 0:
        for j in range(dy+1,0):
            c+=5
            path.append((curr_x,curr_y+j))
            cost[(curr_x,curr_y+j)]=c

if abs(dx) == abs(dy) and abs(dx)>1:
    if dx>0 and dy>0:
        for j in range(1,dx):
            c+=7
            path.append((curr_x+j,curr_y+j))
            cost[(curr_x+j,curr_y+j)]=c
    elif dx<0 and dy<0:
        for j in range(dx+1,0):
            c+=7
            path.append((curr_x+j,curr_y+j))
            cost[(curr_x+j,curr_y+j)]=c
    elif dx>0 and dy<0:
        for j in range(1,dx):
            c+=7
            path.append((curr_x+j,curr_y-j))
            cost[(curr_x+j,curr_y-j)]=c
    else:
        for j in range(1,dx):
            c+=7
            path.append((curr_x-j,curr_y+j))
```

```
                cost[(curr_x-j,curr_y+j)]=c

    path.append(a[l-1])
    return(path,cost)


def jps(self,gridworld,start,goal):
    '''
    Run a Jump point search on a field with obstacles

    gridworld - 2d array representing obstacles as 1 and traversable paths as -1
    grid - 2d array representing the cost to get to that node
    start - starting point for the search
    goal - finishing point for the search

    Return:
    List of Jump Points as Tuples OR [] if no path is found
    '''
    class FoundPath(Exception):
        pass

    global grid , expanded , visited , h , w

    h=gridworld.height+1;w=gridworld.width+1

    grid = [[0]*w for i in range(h)]

    for i in range(h):
        grid[i][0]=1 ; grid[i][-1]=1
    for j in range(w):
        grid[0][j]=1 ; grid[-1][j]=1


    expanded = [[False]*w for i in range(h)]
    visited = [[False]*w for i in range(h)]
    sources = [[(None,None)]*w for i in range(h)]

    for obstacle in gridworld.obstacles:
        grid[obstacle[0]][obstacle[1]]=1

    start_x , start_y = start
    goal_x , goal_y = goal

    if start_x==0:
```

```
start_x+=1

    if start_y==0:
        start_y+=1

    if grid[start_x][start_y]==1:
        raise ValueError("No path exists: start point is an obstacle")
    if  grid[goal_x][goal_y]==1:
        raise ValueError("No path exists: goal point is an obstacle")

    def add_jump_point(node):
        if node is not None:
            p.put(node , grid[node[0]][node[1]] + max(abs(node[0]-goal_x),abs(node[1]-goal_y)))

    def diagonal(start,dirn):
        curr_x,curr_y=start[0],start[1]
        dir_x,dir_y=dirn

        curCost=grid[start[0]][start[1]]

        while (True):
            curr_x+=dir_x ; curr_y+=dir_y ; curCost +=1

            if grid[curr_x][curr_y] == 0:
                grid[curr_x][curr_y]=curCost
                sources[curr_x][curr_y]=start
                visited[curr_x][curr_y]=True

            # Destination found
            elif curr_x==goal_x and curr_y==goal_y:
                grid[curr_x][curr_y]=curCost
                sources[curr_x][curr_y]=start
                visited[curr_x][curr_y]=True
                raise FoundPath()

            # collided with obstacle
            else:
                return None

            # If jump point is found
            if grid[curr_x+dir_x][curr_y] == 1 and grid[curr_x+dir_x][curr_y+dir_y]!=1:
                return (curr_x,curr_y)
            # Extend horizontal search
            else:
```

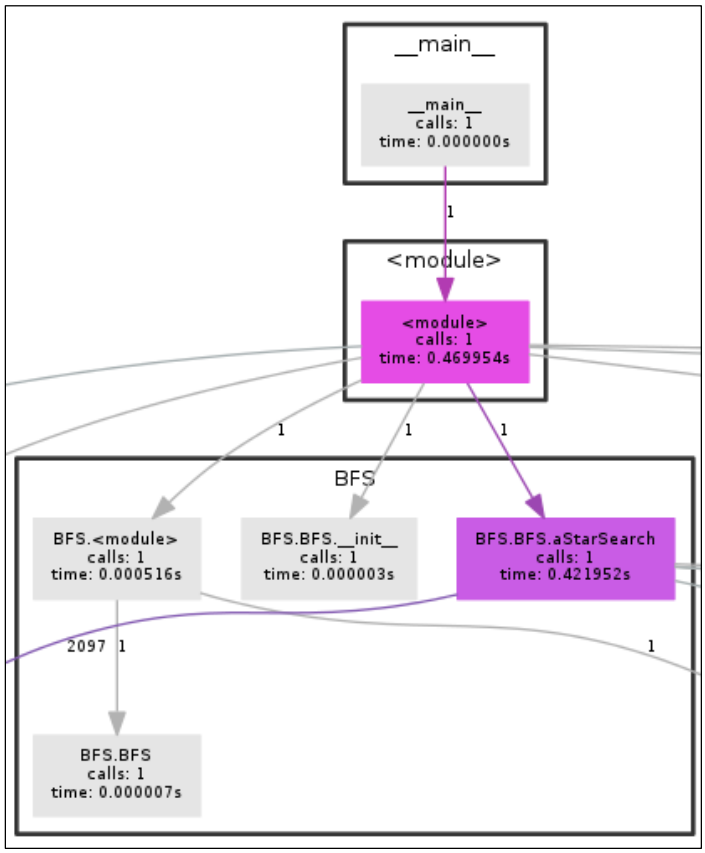
```
            if grid[curr_x][curr_y-1]==1 and grid[curr_x+dir_x][curr_y-1]!=1:
                return (curr_x,curr_y)

    grid[start[0]][start[1]]=0 ; grid[goal[0]][goal[1]]=-2

    p=FastPriorityQueue()
    add_jump_point(start)

    # Main loop: iterate through queue
    while not p.isEmpty():
        pX,pY=p.get()
        expanded[pX][pY]=True

        try:
            add_jump_point(cardinal((pX,pY),(1,0)))
            add_jump_point(cardinal((pX,pY),(-1,0)))
            add_jump_point(cardinal((pX,pY),(0,1)))
            add_jump_point(cardinal((pX,pY),(0,-1)))

            add_jump_point(diagonal((pX,pY),(1,1)))
            add_jump_point(diagonal((pX,pY),(1,-1)))
            add_jump_point(diagonal((pX,pY),(-1,1)))
            add_jump_point(diagonal((pX,pY),(-1,-1)))

        except FoundPath:
            return self.construct(sources,start,goal)

    raise ValueError('No path found')
```

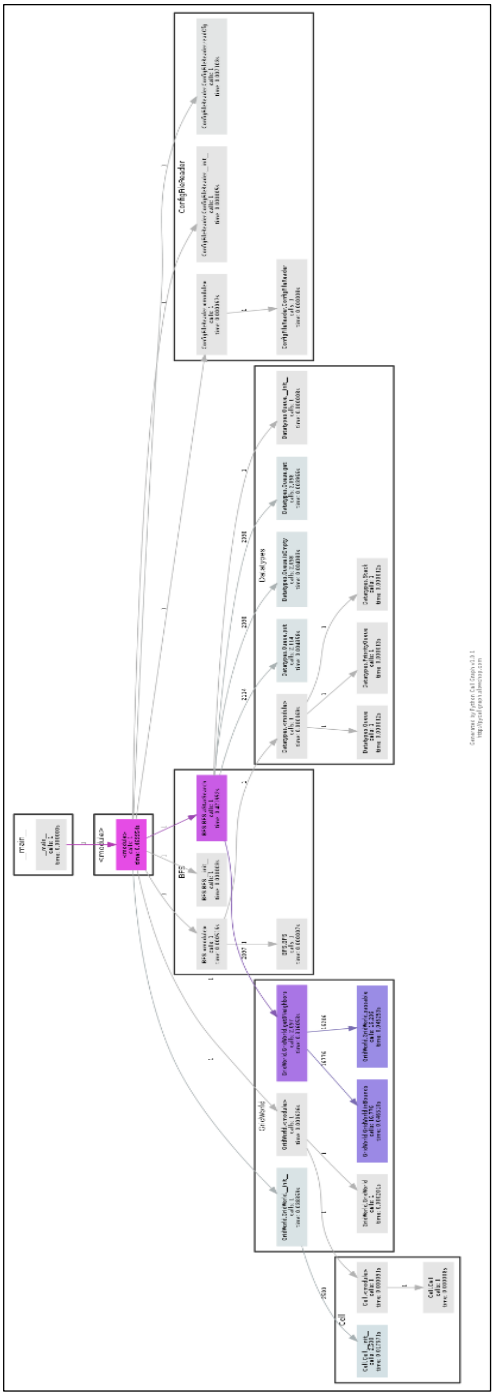
APPENDIX B

1. Profiling Results
    1.1. Breadth First Search
    1.2. Dijkstra's Algorithm
    1.3. A* Algorithm
    1.4. Jump Point Search

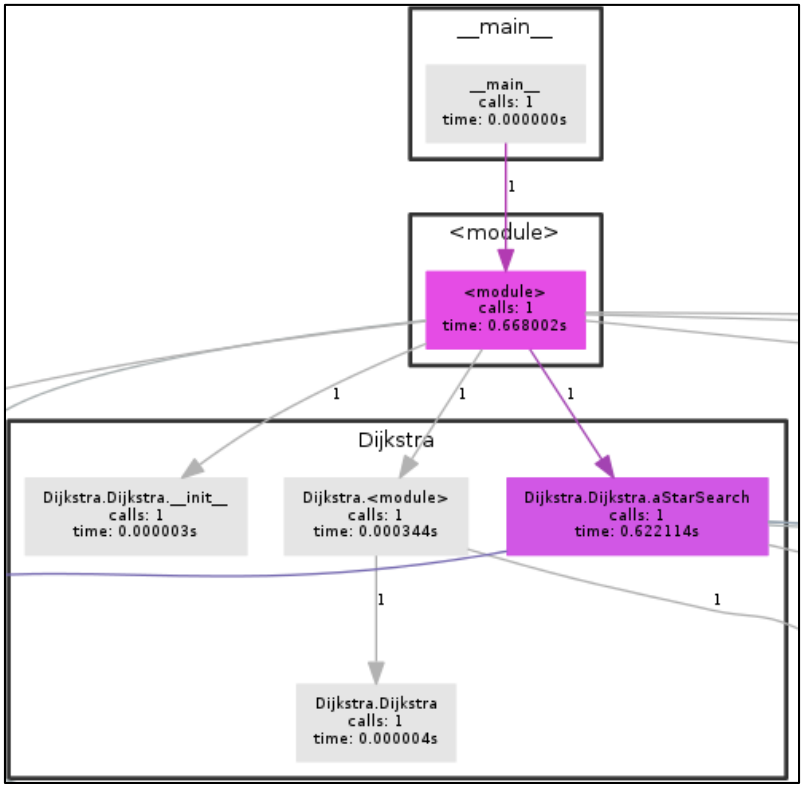
[NOTE: *In the following section, the first image for each algorithm shows an enlarged view of the second image laying focus on the execution time and the number of function calls. The second image is the one obtained by profiling using Python Call Graph (Section 4.3.1).*]

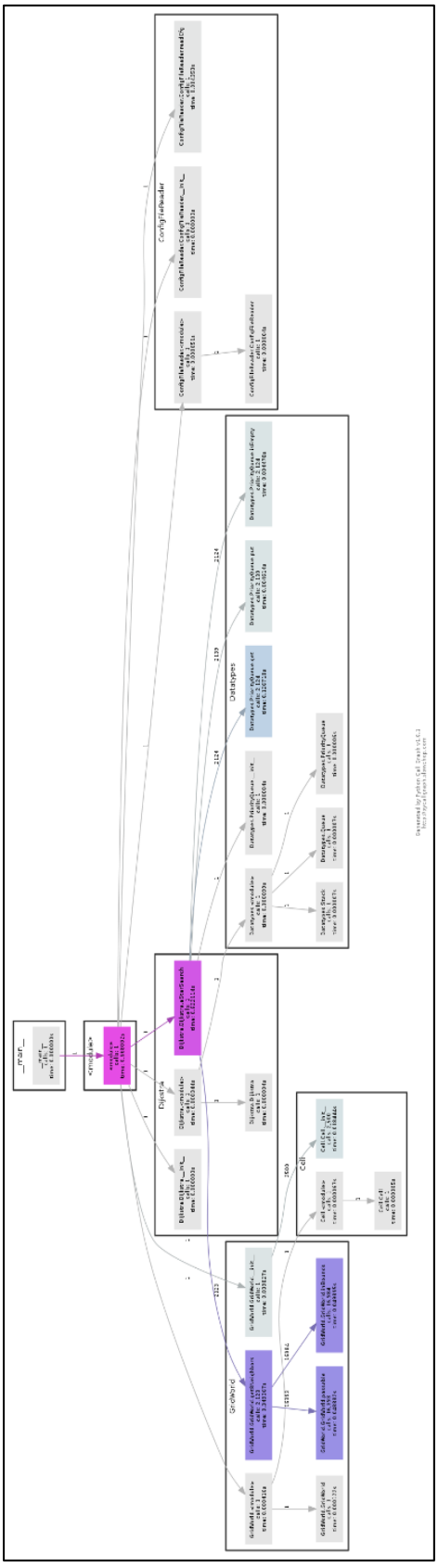
1. Breadth First Search



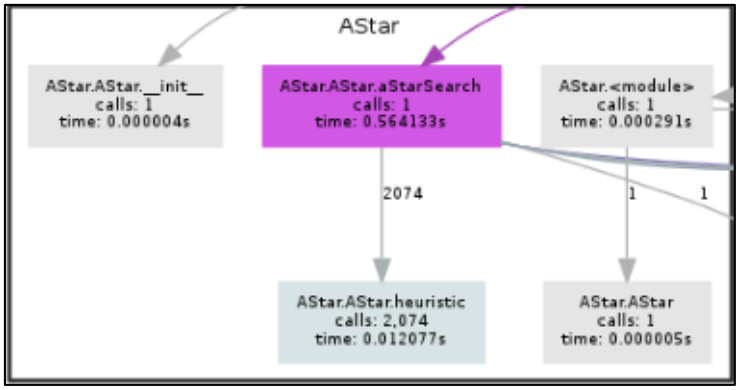


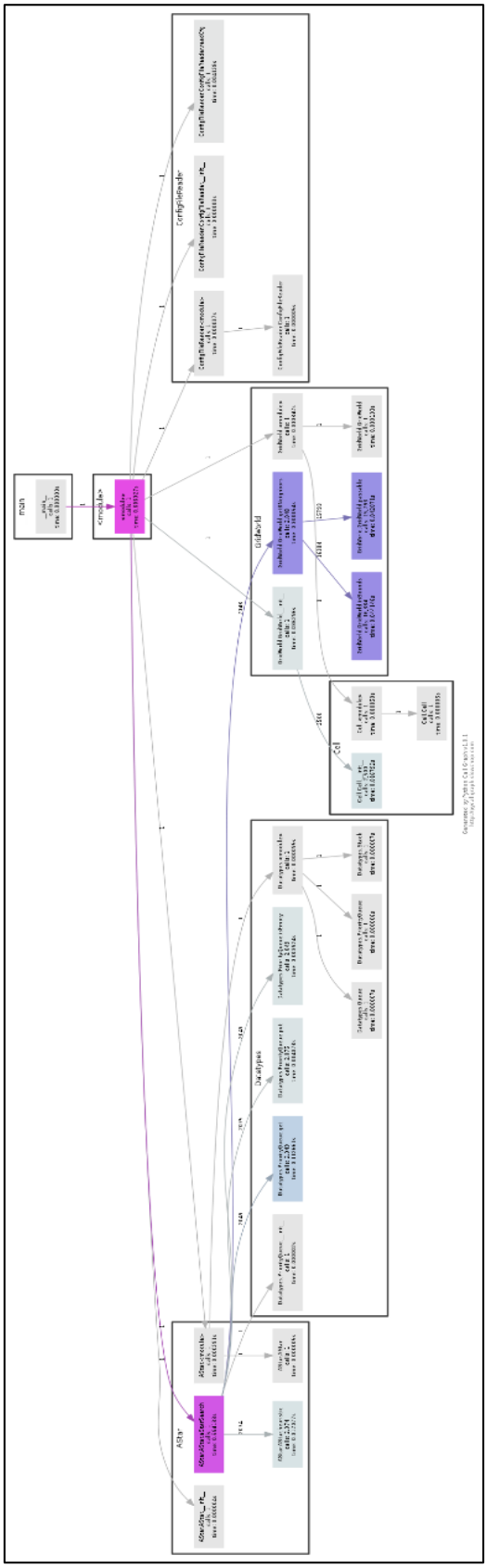
2. Dijkstra's Algorithm



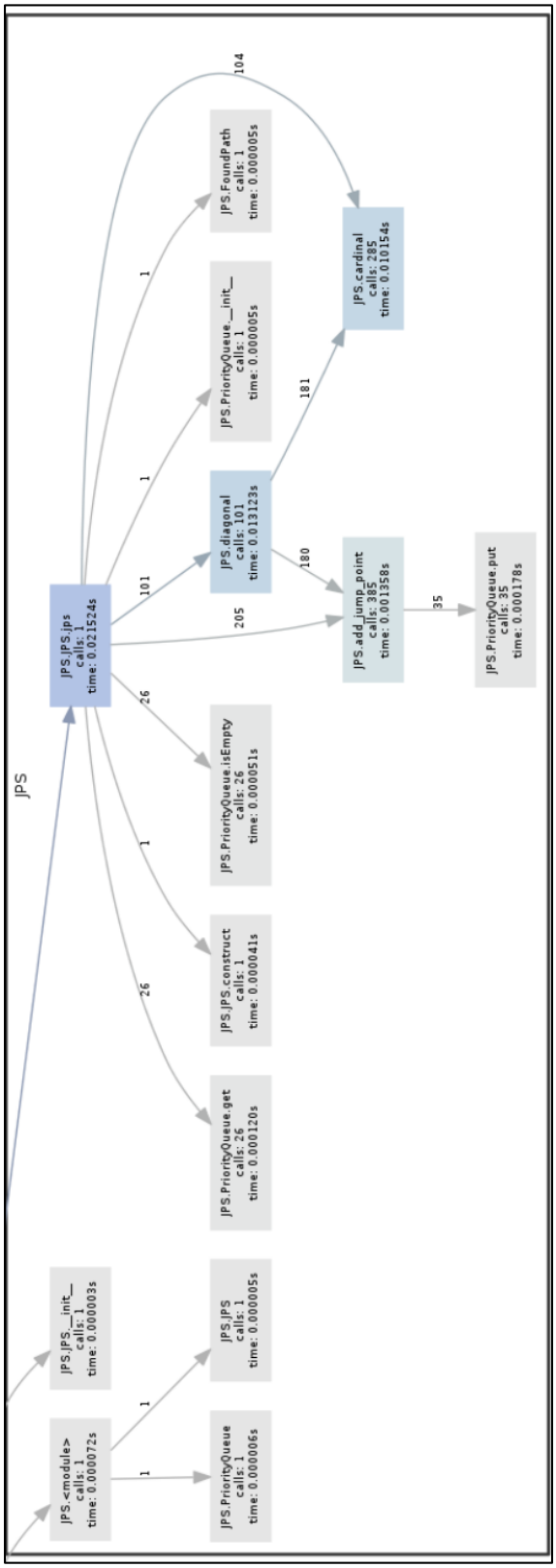


3. A* Algorithm





4. Jump Point Search



APPENDIX C

1. Path Planners plugin
   - 1.1. C++ file (pathplanners.cpp)
   - 1.2. Header file (pathplanners.h)
   - 1.3. Launch file (pathplanners.launch)
   - 1.4. Package file (package.xml)
   - 1.5. Plugin file (pathplanners_plugin.xml)

1. C++ File

```
#include "pathplanners.h"

PLUGINLIB_EXPORT_CLASS(PathPlanners_all::PathPlannersROS, nav_core::BaseGlobalPlanner);

int mapSize;
bool* OGM;
static const float INFINIT_COST = INT_MAX;
float infinity = std::numeric_limits< float >::infinity();

int clock_gettime(clockid_t clk_id, struct timespect *tp);

timespec diff(timespec start, timespec end){

		timespec temp;

		if ((end.tv_nsec-start.tv_nsec)<0){
			temp.tv_sec = end.tv_sec-start.tv_sec-1;
			temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
		}

		else{
			temp.tv_sec = end.tv_sec-start.tv_sec;
			temp.tv_nsec = end.tv_nsec-start.tv_nsec;
		}
		return temp;
}


inline vector <int> getNeighbour (int CellID);


namespace PathPlanners_all
{
PathPlannersROS::PathPlannersROS(){}
PathPlannersROS::PathPlannersROS(std::string name, costmap_2d::Costmap2DROS*
costmap_ros){initialize(name, costmap_ros);}

void PathPlannersROS::initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros){
		if (!initialized_){
			costmap_ros_ = costmap_ros;
			costmap_ = costmap_ros_->getCostmap();

			originX = costmap_->getOriginX();
			originY = costmap_->getOriginY();

			width = costmap_->getSizeInCellsX();
```

```
                height = costmap_->getSizeInCellsY();
                resolution = costmap_->getResolution();
                mapSize = width*height;

                OGM = new bool [mapSize];
                for (unsigned int iy = 0; iy < height; iy++){
                        for (unsigned int ix = 0; ix < width; ix++){
                                unsigned int cost = static_cast<int>(costmap_->getCost(ix,iy));

                                if (cost <= 255){
                                        OGM[iy*width+ix]=true;
                                        // cout <<"Traversable"<< ix<<","<<iy<<"   cost:"<<cost<<endl;
                                }

                                else{
                                        OGM[iy*width+ix]=false;
                                        // cout <<"Obstacle"<< ix<<","<<iy<<"   cost:"<<cost<<endl;
                                }
                        }
                }
                ROS_INFO("Jump Point Search initialized successfully");
                initialized_ = true;
        }
        else
        ROS_WARN("Planner already initialized");
}



bool PathPlannersROS::makePlan(const geometry_msgs::PoseStamped& start, const
geometry_msgs::PoseStamped& goal,std::vector<geometry_msgs::PoseStamped>& plan){
        if (!initialized_){
                ROS_ERROR("The planner has not been initialized, please call initialize() to use the
planner");
                return false;
        }

        ROS_DEBUG("Got a start: %.2f, %.2f, and a goal: %.2f, %.2f", start.pose.position.x,
start.pose.position.y,goal.pose.position.x, goal.pose.position.y);
        plan.clear();

        if (goal.header.frame_id != costmap_ros_->getGlobalFrameID()){
                ROS_ERROR("This planner as configured will only accept goals in the %s frame, but a goal
was sent in the %s frame.",
                costmap_ros_->getGlobalFrameID().c_str(), goal.header.frame_id.c_str());
                return false;
        }

        tf::Stamped < tf::Pose > goal_tf;
```

```
tf::Stamped < tf::Pose > start_tf;

poseStampedMsgToTF(goal, goal_tf);
poseStampedMsgToTF(start, start_tf);

float startX = start.pose.position.x;
float startY = start.pose.position.y;

float goalX = goal.pose.position.x;
float goalY = goal.pose.position.y;

getCoordinate(startX, startY);
getCoordinate(goalX, goalY);

int startCell;
int goalCell;

if (validate(startX, startY) && validate(goalX, goalY)){
	startCell = convertToCellIndex(startX, startY);
	goalCell = convertToCellIndex(goalX, goalY);
}

else{
	ROS_WARN("the start or goal is out of the map");
	return false;
}

if (isValid(startCell, goalCell)){
	vector<int> bestPath;
	bestPath.clear();
	bestPath = PathFinder(startCell, goalCell);
	if(bestPath.size()>0){
		for (int i = 0; i < bestPath.size(); i++){
			float x = 0.0;
			float y = 0.0;
			int index = bestPath[i];
			convertToCoordinate(index, x, y);
			geometry_msgs::PoseStamped pose = goal;

			pose.pose.position.x = x;
			pose.pose.position.y = y;
			pose.pose.position.z = 0.0;

			pose.pose.orientation.x = 0.0;
			pose.pose.orientation.y = 0.0;
			pose.pose.orientation.z = 0.0;
			pose.pose.orientation.w = 1.0;

			plan.push_back(pose);
```

```
                        }

                        float path_length = 0.0;
                        std::vector<geometry_msgs::PoseStamped>::iterator it = plan.begin();
                        geometry_msgs::PoseStamped last_pose;
                        last_pose = *it;
                        it++;

                        for (; it!=plan.end();++it){
                                path_length += hypot((*it).pose.position.x - last_pose.pose.position.x,
(*it).pose.position.y - last_pose.pose.position.y );
                                last_pose = *it;
                        }

                        cout <<"The global path length: "<< path_length<< " meters"<<endl;
                        return true;
                }
                else{
                        ROS_WARN("The planner failed to find a path, choose other goal position");
                        return false;
                }
        }

        else{
                ROS_WARN("Not valid start or goal");
                return false;
        }
}

void PathPlannersROS::getCoordinate(float& x, float& y){
        x = x - originX;
        y = y - originY;
}

int PathPlannersROS::convertToCellIndex(float x, float y){
        int cellIndex;
        float newX = x / resolution;
        float newY = y / resolution;
        cellIndex = getIndex(newY, newX);
        return cellIndex;
}

void PathPlannersROS::convertToCoordinate(int index, float& x, float& y){
        x = getCol(index) * resolution;
        y = getRow(index) * resolution;
        x = x + originX;
        y = y + originY;
}
```

```
bool PathPlannersROS::validate(float x, float y){
		bool valid = true;
		if (x > (width * resolution) || y > (height * resolution))
			valid = false;
		return valid;
}

void PathPlannersROS::mapToWorld(double mx, double my, double& wx, double& wy){
		costmap_2d::Costmap2D* costmap = costmap_ros_->getCostmap();
		wx = costmap->getOriginX() + mx * resolution;
		wy = costmap->getOriginY() + my * resolution;
}

vector<int> PathPlannersROS::PathFinder(int startCell, int goalCell){
		vector<int> bestPath;
		float g_score [mapSize];
		for (uint i=0; i<mapSize; i++)
			g_score[i]=infinity;

		timespec time1, time2;

		clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);

		bestPath=AStar(startCell, goalCell,  g_score);
		// bestPath=Dijkstra(startCell, goalCell,  g_score);
		// bestPath=BFS(startCell, goalCell,  g_score);
		// bestPath=JPS(startCell,goalCell,g_score);

		clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);

		cout<<" Time taken to generate path= " << (diff(time1,time2).tv_sec)*1e3 +
(diff(time1,time2).tv_nsec)*1e-6 << " microseconds" << endl;

		return bestPath;
}

vector<int> PathPlannersROS::AStar(int startCell, int goalCell, float g_score[]){
		vector<int> bestPath;
		vector<int> emptyPath;
		cells CP;

		multiset<cells> OPL;
		int currentCell;

		g_score[startCell]=0;
		CP.currentCell=startCell;
		CP.fCost=g_score[startCell]+heuristic(startCell,goalCell,1);
		OPL.insert(CP);
		currentCell=startCell;
```

```
		while (!OPL.empty()&& g_score[goalCell]==infinity){
			currentCell = OPL.begin()->currentCell;
			OPL.erase(OPL.begin());
			vector <int> neighborCells;
			neighborCells=getNeighbour(currentCell);
			for(uint i=0; i<neighborCells.size(); i++){
				if(g_score[neighborCells[i]]==infinity){

		g_score[neighborCells[i]]=g_score[currentCell]+getMoveCost(currentCell,neighborCells[i]);
					add_open(OPL, neighborCells[i], goalCell, g_score, 1);
				}
			}
		}

		if(g_score[goalCell]!=infinity){
			bestPath=constructPath(startCell, goalCell, g_score);
			return bestPath;
		}

		else{
			cout << "Path not found!" << endl;
			return emptyPath;
		}
}

vector<int> PathPlannersROS::Dijkstra(int startCell, int goalCell, float g_score[]){
		vector<int> bestPath;
		vector<int> emptyPath;
		cells CP;

		multiset<cells> OPL;
		int currentCell;

		g_score[startCell]=0;
		CP.currentCell=startCell;
		CP.fCost=g_score[startCell];
		OPL.insert(CP);
		currentCell=startCell;

		while (!OPL.empty()&& g_score[goalCell]==infinity){
			currentCell = OPL.begin()->currentCell;
			OPL.erase(OPL.begin());
			vector <int> neighborCells;
			neighborCells=getNeighbour(currentCell);
			for(uint i=0; i<neighborCells.size(); i++){
				if(g_score[neighborCells[i]]==infinity){

		g_score[neighborCells[i]]=g_score[currentCell]+getMoveCost(currentCell,neighborCells[i]);
```

```
                                        add_open(OPL, neighborCells[i], goalCell, g_score, 0);
                                }
                        }
                }

                if(g_score[goalCell]!=infinity){
                        bestPath=constructPath(startCell, goalCell, g_score);
                        return bestPath;
                }

                else{
                        cout << "Path not found!" << endl;
                        return emptyPath;
                }
}

vector<int> PathPlannersROS::BFS(int startCell, int goalCell, float g_score[]){
                vector<int> bestPath;
                vector<int> emptyPath;
                cells CP;

                multiset<cells> OPL;
                int currentCell;

                g_score[startCell]=0;
                CP.currentCell=startCell;
                CP.fCost=g_score[startCell];
                OPL.insert(CP);
                currentCell=startCell;

                while (!OPL.empty()&& g_score[goalCell]==infinity){
                        currentCell = OPL.begin()->currentCell;
                        OPL.erase(OPL.begin());
                        vector <int> neighborCells;
                        neighborCells=getNeighbour(currentCell);
                        for(uint i=0; i<neighborCells.size(); i++){
                                if(g_score[neighborCells[i]]==infinity){
                                        g_score[neighborCells[i]]=g_score[currentCell]+1;
                                        add_open(OPL, neighborCells[i], goalCell, g_score, 0);
                                }
                        }
                }

                if(g_score[goalCell]!=infinity){
                        bestPath=constructPath(startCell, goalCell, g_score);
                        return bestPath;
                }

                else{
```

```
                cout << "Path not found!" << endl;
                return emptyPath;
        }
}



vector<int> PathPlannersROS::constructPath(int startCell, int goalCell,float g_score[])
{
        vector<int> bestPath;
        vector<int> path;

        path.insert(path.begin()+bestPath.size(), goalCell);
        int currentCell=goalCell;

        while(currentCell!=startCell){
                vector <int> neighborCells;
                neighborCells=getNeighbour(currentCell);

                vector <float> gScoresNeighbors;
                for(uint i=0; i<neighborCells.size(); i++)
                        gScoresNeighbors.push_back(g_score[neighborCells[i]]);

                int posMinGScore=distance(gScoresNeighbors.begin(),
min_element(gScoresNeighbors.begin(), gScoresNeighbors.end()));
                currentCell=neighborCells[posMinGScore];

                path.insert(path.begin()+path.size(), currentCell);
        }

        for(uint i=0; i<path.size(); i++)
                bestPath.insert(bestPath.begin()+bestPath.size(), path[path.size()-(i+1)]);

        return bestPath;
}

void PathPlannersROS::add_open(multiset<cells> & OPL, int neighborCell, int goalCell, float g_score[] ,int
n){
        cells CP;
        CP.currentCell=neighborCell;
        if (n==1)
                CP.fCost=g_score[neighborCell]+heuristic(neighborCell,goalCell,1);
        else
                CP.fCost=g_score[neighborCell];
        OPL.insert(CP);
}

vector <int> PathPlannersROS::getNeighbour (int CellID){
        int rowID=getRow(CellID);
```

```
	int colID=getCol(CellID);
	int neighborIndex;
	vector <int>  freeNeighborCells;

	for (int i=-1;i<=1;i++)
		for (int j=-1; j<=1;j++){
			if ((rowID+i>=0)&&(rowID+i<height)&&(colID+j>=0)&&(colID+j<width)&& (!(i==0
&& j==0))){
				neighborIndex = getIndex(rowID+i,colID+j);
				if(isFree(neighborIndex) )
					freeNeighborCells.push_back(neighborIndex);
			}
		}

	return  freeNeighborCells;
}

bool PathPlannersROS::isValid(int startCell,int goalCell){
	bool isvalid=true;

	bool isFreeStartCell=isFree(startCell);
	bool isFreeGoalCell=isFree(goalCell);

	if (startCell==goalCell){
		cout << "The Start and the Goal cells are the same..." << endl;
		isvalid = false;
	}

	else{

		if(!isFreeStartCell && !isFreeGoalCell){
			cout << "The start and the goal cells are obstacle positions..." << endl;
			isvalid = false;
		}

		else{
			if(!isFreeStartCell){
				cout << "The start is an obstacle..." << endl;
				isvalid = false;
			}
			else{
				if(!isFreeGoalCell){
					cout << "The goal cell is an obstacle..." << endl;
					isvalid = false;
				}
				else{
					if (getNeighbour(goalCell).size()==0){
						cout << "The goal cell is encountred by obstacles... "<<
endl;
```

```
                                                isvalid = false;
                                        }
                                        else{
                                                if(getNeighbour(startCell).size()==0){
                                                        cout << "The start cell is encountred by
obstacles... "<< endl;
                                                        isvalid = false;
                                                }
                                        }
                                }
                        }
                }
        }

return isvalid;
}

float  PathPlannersROS::getMoveCost(int CellID1, int CellID2){
        int i1=0,i2=0,j1=0,j2=0;

        i1=getRow(CellID1);
        j1=getCol(CellID1);
        i2=getRow(CellID2);
        j2=getCol(CellID2);

        float moveCost=INFINIT_COST;
        if(i1!=i2 && j1!=j2)
              moveCost=1.4;
        else
              moveCost=1;
        return moveCost;
}

bool  PathPlannersROS::isFree(int CellID){
        return OGM[CellID];
}
};

bool operator<(cells const &c1, cells const &c2){return c1.fCost < c2.fCost;}
```

2. Header File

```
// Including general libraries
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <set>
#include <pluginlib/class_list_macros.h>

// Including ROS specific libraries
#include <ros/ros.h>

#include <actionlib/client/simple_action_client.h>
#include <move_base_msgs/MoveBaseAction.h>

// To accomodate for moving base
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <move_base_msgs/MoveBaseGoal.h>
#include <move_base_msgs/MoveBaseActionGoal.h>
#include <angles/angles.h>

// To accomodate sensory input
#include "sensor_msgs/LaserScan.h"
#include "sensor_msgs/PointCloud2.h"

// Navigation messages
#include <nav_msgs/Odometry.h>
#include <nav_msgs/OccupancyGrid.h>
#include <nav_msgs/Path.h>
#include <nav_msgs/GetPlan.h>

// Costmap transform
#include <tf/tf.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_listener.h>

// To get costmap
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>
#include <nav_core/base_global_planner.h>
#include <base_local_planner/world_model.h>
#include <base_local_planner/costmap_model.h>

// Defining the whole thing in a namespace
using namespace std;
```

```
using std::string;

#ifndef PathPlanners_ROS
#define PathPlanners_ROS

// Structure to store the cells in a data type
struct cells{
   int currentCell;
   float fCost;
};

namespace PathPlanners_all{

class PathPlannersROS : public nav_core::BaseGlobalPlanner {
public:
   PathPlannersROS();
   PathPlannersROS(std::string name, costmap_2d::Costmap2DROS* costmap_ros);

   // Overriden classes from interface nav_core::BaseGlobalPlanner
   void initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
   bool makePlan(const geometry_msgs::PoseStamped& start,const geometry_msgs::PoseStamped&
goal,std::vector<geometry_msgs::PoseStamped>& plan);

   // Base variables for Planners
   float originX;float originY;float resolution;
   double step_size_, min_dist_from_robot_;
   bool initialized_;
   int width;int height;

   costmap_2d::Costmap2DROS* costmap_ros_;
   costmap_2d::Costmap2D* costmap_;

   // Base functions for Planners
   void getCoordinate (float& x, float& y);
   void convertToCoordinate(int index, float& x, float& y);
   void mapToWorld(double mx, double my, double& wx, double& wy);
   void add_open(multiset<cells> & OPL, int neighborCell, int goalCell, float g_score[],int n);

   bool validate(float x, float y);
   bool isValid(int startCell,int goalCell);
   bool isFree(int CellID); //returns true if the cell is Free

   int convertToCellIndex (float x, float y);
   int getIndex(int i,int j){return (i*width)+j;}
   int getRow(int index){return index/width;}
   int getCol(int index){return index%width;}

   float getMoveCost(int CellID1, int CellID2);
```

```
    vector <int> getNeighbour (int CellID);
    vector<int> PathFinder(int startCell, int goalCell);
    vector<int> AStar(int startCell, int goalCell, float g_score[]);
    vector<int> Dijkstra(int startCell, int goalCell, float g_score[]);
    vector<int> BFS(int startCell, int goalCell, float g_score[]);
    vector<int> constructPath(int startCell, int goalCell, float g_score[]);

    float heuristic(int cellID, int goalCell, int n){
        int x1=getRow(goalCell);int y1=getCol(goalCell);int x2=getRow(cellID);int y2=getCol(cellID);
        int dx = abs(x2-x1) ; int dy = abs(y2-y1);
        // Manhattan Heuristic
        if(n==1)
            return dx + dy;
        // Euclidean Heuristic
        else if(n==2)
            return sqrt(dx*dx + dy*dy);

        // Chebyshev Heuristic
        else if(n==3)
            return max(dx,dy);

        // Octile Heuristic
        else
            return dx+dy+(sqrt(2)-2)*min (dx,dy);
    }
};
};
#endif
```

3. Launch File

```
<launch>

<include file="$(find turtlebot_bringup)/launch/minimal.launch"></include>

<include file="$(find turtlebot_bringup)/launch/3dsensor.launch">
                        <arg name="rgb_processing" value="true" />
                        <arg name="depth_registration" value="true" />
                        <arg name="depth_processing" value="true" />
                        <arg name="scan_topic" value="/scan" />
</include>

<arg name="map_file" default="/home/kipp/catkin_ws_turtlebot/maps/main.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

<arg name="initial_pose_x" default="0.0"/>
<arg name="initial_pose_y" default="0.0"/>
<arg name="initial_pose_a" default="0.0"/>

<include file="$(find turtlebot_navigation)/launch/includes/amcl.launch.xml">
                        <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
                        <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
                        <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
</include>

<include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml"/>
</launch>
```

4. Package File

```
<?xml version="1.0"?>
<package>
  <name>pathplanners</name>
  <version>1.0.0</version>
  <description>The Path Planners package written by Sahib Singh Dhanjal, BITS Pilani. </description>
  <maintainer email="sahibdhanjal@bits-apogee.org">Sahib Singh Dhanjal</maintainer>
  <license>MIT</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>nav_core</build_depend>
  <build_depend>pluginlib</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>nav_core</run_depend>
  <run_depend>pluginlib</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
    <nav_core plugin="${prefix}/pathplanners_plugin.xml" />
  </export>

</package>
```

5. Plugin File

```
<library path="lib/libpath_planners">
  <class name="PathPlanners_all/PathPlannersROS" type="PathPlanners_all::PathPlannersROS"
base_class_type="nav_core::BaseGlobalPlanner">
    <description>This is the global path planner plugin which includes JPS, A*, Breadth First Search and
Dijkstra's algorihtm.</description>
  </class>
</library>
```