INDIAN INSTITUTE OF INFORMATION TECHNOLOGY , DHARWAD.

MINI PROJECT REPORT

PATH PLANNING

GROUP MEMBERS:

A. SAI CHANDRA KOUSHIK	**19BCS006**
B. LOHITH REDDY	**19BCS014**
B. SAIRAJ PATEL	**19BCS016**
V. KRISHNA SAI ROHITH	**19BCS061**

Abstract

The objective of this paper is to study the different well-known and important path planning algorithms, namely A*, D*, Rapidly exploring Random Tree (RRT) and RRT*. A* and D* are graph-search algorithms, which are deterministic, i.e., the solution will only be found if it exists. RRT on the other hand is a probabilistic-sampling based method meaning there is no guarantee that a solution will be found out. This paper discusses the comparative study between the path-planning algorithms with respect to their computational complexity, speed, path-optimality, reliability etc. It also briefly discusses the scenarios/ situations where they can be applied and are analysed for advantages and disadvantages. The paper also tries to extend the concepts of graph search in 2D to 3D in a much uncomplicated way.

Introduction

In several applications, such as defence and security, autonomous robots have become an unavoidable requirement. Scientific curiosity Rovers have already been employed in planetary exploration and how do these robots determine the path from an initial point to the end point? This is where the path planning algorithms come in handy. This procedure must be done in such a way that the rover/robot determines an optimal path without being interfered by / colliding into any obstacle that might come in its way. A path planning algorithm as the name implies, finds a path from an initial configuration to the final configuration, steering clear of any obstacles that might be present along the way. A path planning algorithm finds a solution to the path planning problem. It may also be called a path-planner or a guidance algorithm. If any new hurdle pops up in the path planned by the planner, it is detected by the robot and a new path is planned, which is done either online or offline.

Offline re-planning requires the planner to have prior knowledge about the profile of the environment, including all the obstacles present, i.e., the global information about the environment. Online replanning can also be done by global planners with the constraint that the computational and time requirements are practical online.

Instead of online planning, a better approach can be used- "**collision-avoidance"**. It avoids hurdles the moment they are detected. An algorithm that employs this is known as a collision-avoidance algorithm or a local path planner. This may be a part of the control loop or a separate inner loop. Such an algorithm performs an evasive manoeuvre as an obstacle is detected, without considering other obstacles or the position and orientation of the final point. Once the obstacle has been circumvented, the rover may follow the earlier planned path or re-plan a new path to the goal. A path is determined in a practical sense by combining both path planning and collision avoidance. Other factors like the robot dynamics and kinematics, path-optimality, practicality of time and computational resources, etc also need to be taken into account while path-planning.

This paper aims at studying and analysing various path planning algorithms, implementing them in MATLAB and comparing them on the basis of time and computational complexity, speed, optimality,
reliability etc.

A global path-planner preserves global data about the obstacles and plans the path from the initial point to the end point while avoiding the obstacles.

The algorithms discussed in this paper are:
1) A-star algorithm.
2) D-star algorithm.

3) Rapidly exploring random tree (RRT) algorithm.
4) Rapidly exploring random tree (RRT*) algorithm.

All these algorithms may be used for online path-replanning by changing the starting position to the present position, and keeping the goal position the same.

A* algorithm

Peter Hart, Nils Nilsson, and Bertram Raphael initially described this algorithm in 1968. The A* algorithm is part of a category of algorithms known as "Graph Search Algorithms." Grids are used to partition the entire region. The search area has now been reduced to a two-dimensional array. Each item in the array is a rectangle that can be walked or cannot be walked. Finding the path entails determining which squares/rectangles must be passed through in order to reach the destination square/rectangle from the starting one. Once the rectangles have been identified, the rover must go from one rectangle's centre (called a "node") to the next until it reaches its destination.

Starting at the beginning, we check the adjacent squares and search outward until we find our target. Examine all the squares that are reachable or walkable from the starting point, excluding those with walls, water, or other illegal terrain (which are basically obstacles). The "PARENT SQUARE" pointers in adjacent squares will point to square A. Now we select one of the adjacent squares from the open list (basically a list of squares that need to be checked out) and repeat the process until we reach our destination. But which of the adjacent squares must be chosen to continue the algorithm? This is determined by the F-score, which must be as low as possible in order to obtain a low-cost path.

$$\mathbf{F=G+H}$$

Here is the cost of moving from the starting point of G = the movement cost to move from the starting point to a given square on the grid, and the path generated to get there follows (the G score for a particular square is Equal to the G score of that parent square plus past traversal. The cost from the parent to that particular square.) And H = the final destination from that particular square on the grid. Estimated move cost of moves to point B. This is often called heuristic. It can be a little confusing. The reason it is called is because it is a guess. Really doesn't know the actual distance until it finds a way. This is because anything of any kind (walls, water, etc.) can get in the way. Heuristic functions that are actually guesses or guesses can be formulated in various ways . But in this work, we use the “Manhattan Heuristic”, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way.

This process continues until either the destination square is added to the closed list (indicating the discovery of a path) or the open list becomes empty (no path exists). Now, working backwards from the target yields the path. Utilising the parent pointer to square to the starting square mapping back from the final to each of the squares
via the parent pointers to the first square.

D* algorithm

This is a network search method as well. It is named after the A* algorithm, except that it is dynamic, meaning that the cost functions might change over time or throughout the problem-solving process. In other terms, it can be described as path planning in an environment where complete topographic profile information isn't available, i.e., there are some impediments in the environment and complete transversal cost information from one grid to the next isn't known in advance. When the sensors on board the robot identify a difference from the previous data, the costs and obstacle information in the database must be updated.

"The problem space can be defined as a set of states signifying robot locations connected by directional arcs, each with an associated cost," Stentz writes. The robot starts in one state and goes over arcs to other states (at a cost of traversal) until it reaches the goal state, represented by G. Except for G, every state X has a backpointer to the following state Y, which is denoted by b(X) =Y. Backpointers are used by D* to represent paths to the objective. The arc cost function c(X, Y) returns a positive integer while traversing an arc from state Y to state X. c(X, Y) is undefined if Y does not have an arc to X. X two states.

Similar to A*, D* also maintains a list of OPEN states. This list is used to transmit information about changes to the supply cost function and to calculate move costs. to states in space. Each state X has an associated tag t(X), whose value is "NEW" if it is never in the list OPEN, t(X) = "OPEN" if X is already in the list book OPEN and t(X) = “CLOSED” if it is no longer in OPEN list. Unlike algorithm A* which starts at the initial point and tracks the path to the target using a heuristic function, D* starts from the target position G and returns to the original position , don't use heuristics like this used by A * algorithm. Quoting Stentz again, "For each state X, D* maintains an estimate of the total cost of arc from X to G given by the path cost function h(G, X). Under the conditions appropriate, this estimate equates to the (minimum) optimal cost of state X at G, given by the implicit function o(G, X) For each state X of the OPEN lists (i.e., t(X) = OPEN), the main function, k(G, X), is defined as equal to the minimum value of h(G, X) before modification and all values are assumed by h(G, X) since X was placed in OPEN list. Key function classifies X state on OPEN list as of two types: RAISE state if k(G, X) < h(G, X) and state LOWER if k(G, X) = h(G, X) D * uses RAISE state in OPEN list to transmit information about increment path cost (for example e.g., due to increased provisioning cost) and LOWER state for information transmission d reduces path cost (e.g., due to reduced overhead arc or new path to target). Propagation takes place by repeatedly removing states from the OPEN list . Every time a state is removed from the list, it is expanded to propagate cost changes to its neighbours. its. These neighbours are in turn added to the OPEN list to continue the process.

The states in the OPEN list are ordered by their main function value . The kmin parameter is set to min(k(X)) for any X such that t(X) = OPEN. The parameter k min represents an important threshold in D*: travel costs less than or equal to kmin are optimal, and those greater than km may not be optimal. The parameter kold is set in kmin before removing the most recent state from the OPEN list. If no states are removed, kold is undefined.

Rapidly exploring Random Tree (RRT)

Most mobile robots often find themselves in situations where they must find a trajectory to another position in the environment, subject to limitations posed by the hurdles and the capabilities of the robot. RRT is an advanced algorithm on which fast continuous domain path planners can be based. This search algorithm finds the path in high dimension cluttered environments. The RRT is a probabilistic-sampling based approach, and the generated random points should lead towards the goal. This method is used for motion planning in various applications.

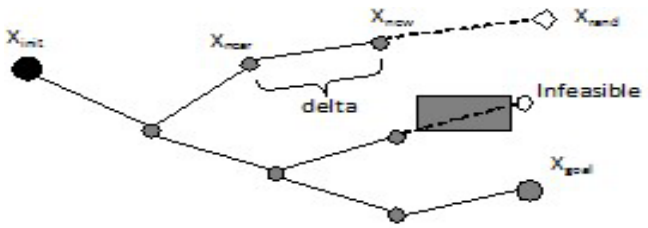
The algorithm to grow an RRT and find a feasible path is as follows:

a) The tree is initiated with an initial point or say start point Xinit

b) A random point Xrand is generated

c) A node in the tree that is nearest to Xrand (say Xnear) is taken using some criterion (e.g., distance)

Figure 1: Growing an RRT to reach the
goal point Xgoal.



d) A branch is grown in the direction of Xrand from Xnear. At a fixed distance of ‘delta’ from Xnear another point is generated say Xnew.

e) Now Xnew is checked for collisions and feasibility

f) If the path is collision-free and feasible, Xnew will become a node of the tree, i.e., it gets included in tree

g) Note that the probability of generating random points should lie between 20% to 25%.

h) Repeat steps 2-6 until the goal point is reached.

The algorithm allows the designer to choose ‘delta’ and Xrand. The choice of ‘delta’ must be determined very carefully because a very minute value may result in the algorithm running too many iterations to determine the path, and a very large value may result in the robot getting

stuck in cluttered spaces. Since this method is probability-based, the random samples are generated such that the 20% to 25% of the generated random samples should be the goal itself. This will lead the tree towards the goal. RRT may get stuck in much cluttered environments. Thus, an iteration limit is given so that it does not go in an infinite loop. Once the iteration limit is reached it comes out of the loop and the path is shown irrespective of whether the goal point is reached or not. Also, this method is not useful in case of
moving targets, because the position of the object is changing and it may intersect with the path.

Rapidly exploring random tree – variant (RRT*)

RRT* is a modification or a variant of the RRT algorithm which has been optimized for better performance or results (in this case, a “better” path) . The RRT* method will give the shortest path (closest to optimal path) to the goal as the number of nodes approaches infinity. While being realistically unfeasible, the statement suggests that the algorithm does work to develop a shortest path. RRT* has the same fundamental premise as RRT, but two critical changes to the algorithm provide dramatically different outcomes.

First, RRT* records the distance each vertex has travelled with respect to its parent vertex.

This is referred to as the cost() of the vertex. After the closest node is found in the graph, a neighbourhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper cost() than the proximal node is found, the cheaper node replaces the proximal node. The effect of this feature can be seen with the addition of fan shaped twigs in the tree structure. The cubic structure of RRT is eliminated.

The second difference RRT* adds is the rewiring of the tree.

After a vertex has been connected to the cheapest neighbour (i.e., the neighbour with the lowest cost) , the neighbours are again examined. The neighbours are checked if being rewired to the newly added vertex will make their cost lower. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex.

This feature makes the path smoother and creates incredibly straight paths, thus resulting in a shorter or a more optimal path. The graphs of RRT* are characteristically different from those of RRT. For finding an optimal path, especially in a dense obstacle-field, the structure of RRT* is very useful. The graph vines or turns around objects, finding shorter paths than when compared to RRT. If the destination was to change, the original graph can still be used because it represents the quickest/shortest path to most locations in the region.

One significant drawback of the RRT* algorithm is the reduction in performance which is due to thorough examination of the neighbouring nodes and rewiring of the graph. A major part of the computational power or capacity goes into avoidance of obstacles. Obstacle avoidance must be checked when a node is placed, when a node is connected to its neighbor, and for each node that is to be rewired. This is a considerable number of checks to make. Yet, one cannot deny the success of the generated paths.

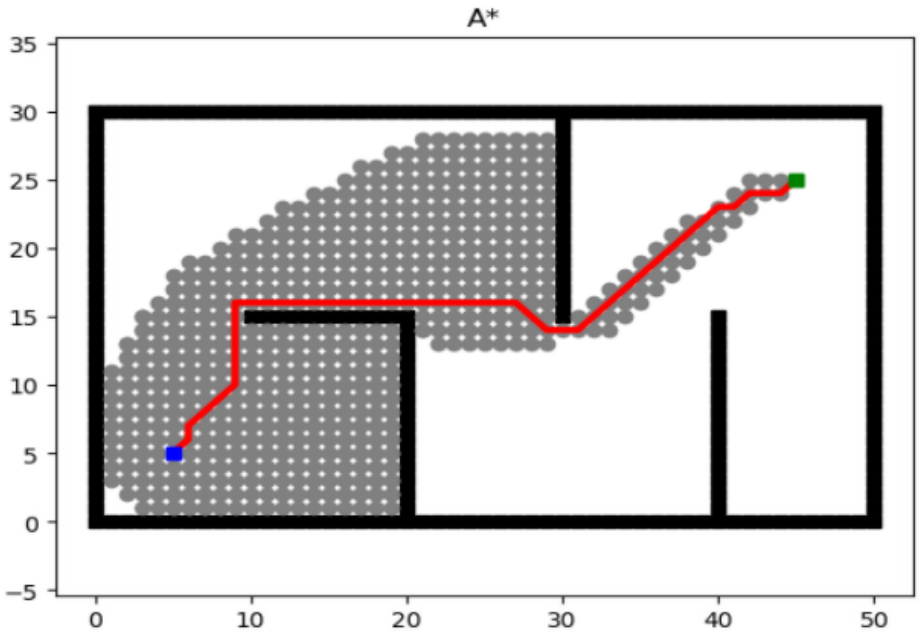
Simulation Results

In all the MATLAB programs which were coded, the whole world was divided into GRIDS and the obstacles were defined in terms of these grids. The grids were laid out in the below given manner, with each number representing the grid number.

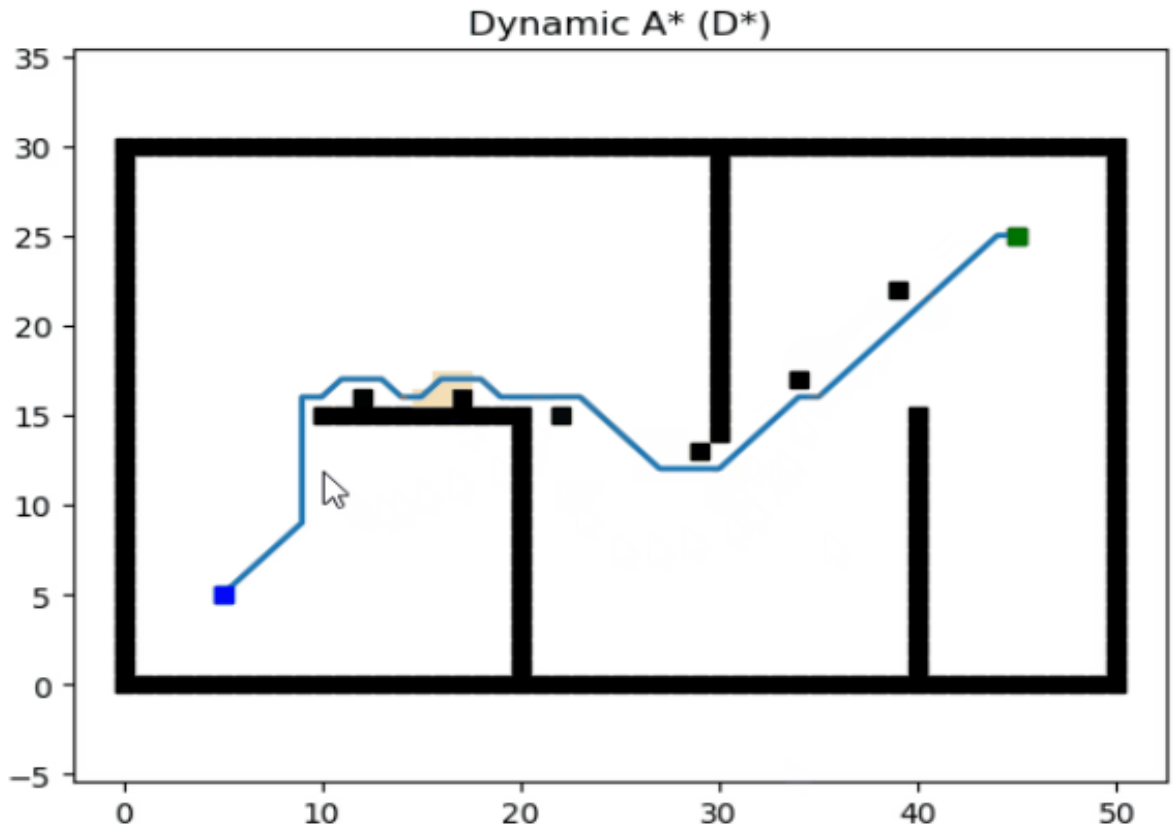
1 2 3 4 .... n
n+1 n+2 n+3 n+4 ... . 2n
.
n(m-1)+1.. .... mn

But at the same time, the coordinates along the axes have been defined as usual, i.e., the x-coordinate being zero at the origin and increasing along the axis towards
the right and the y-coordinate being zero at the origin (which is the lower-left-most point) and increases along the axis in the upward direction. In fact, the coordinates have been shown in every plot. Also, the grids have been taken to be of unit length and breadth in dimension.

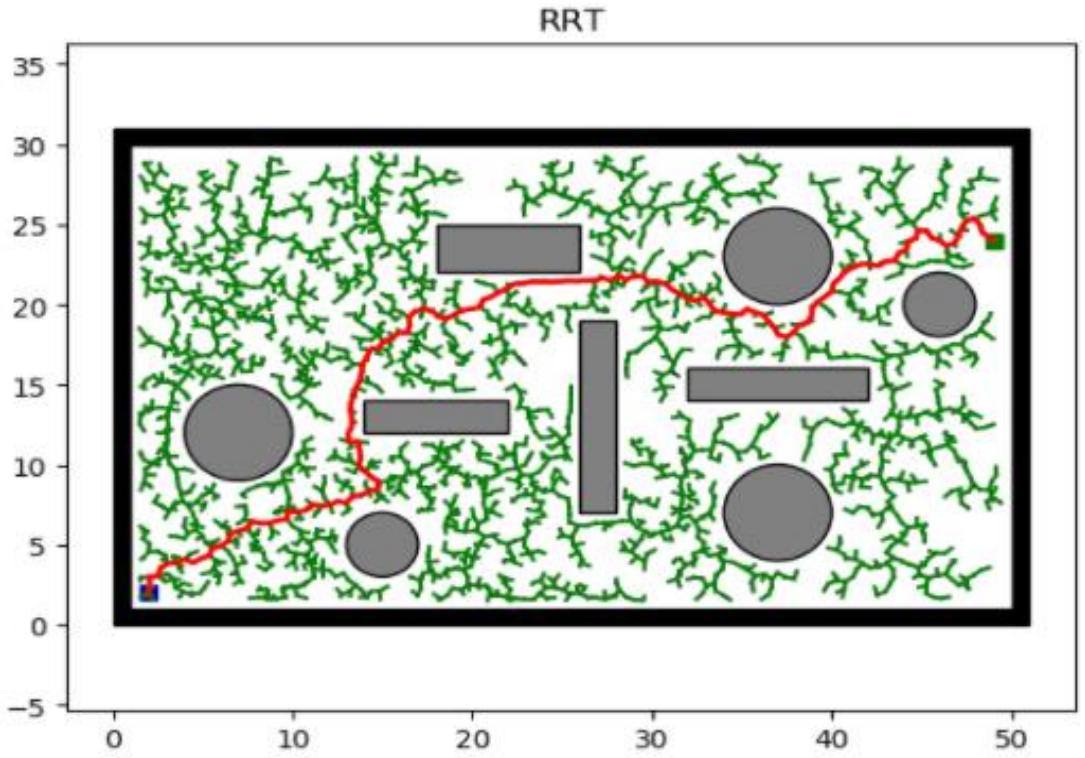
A*



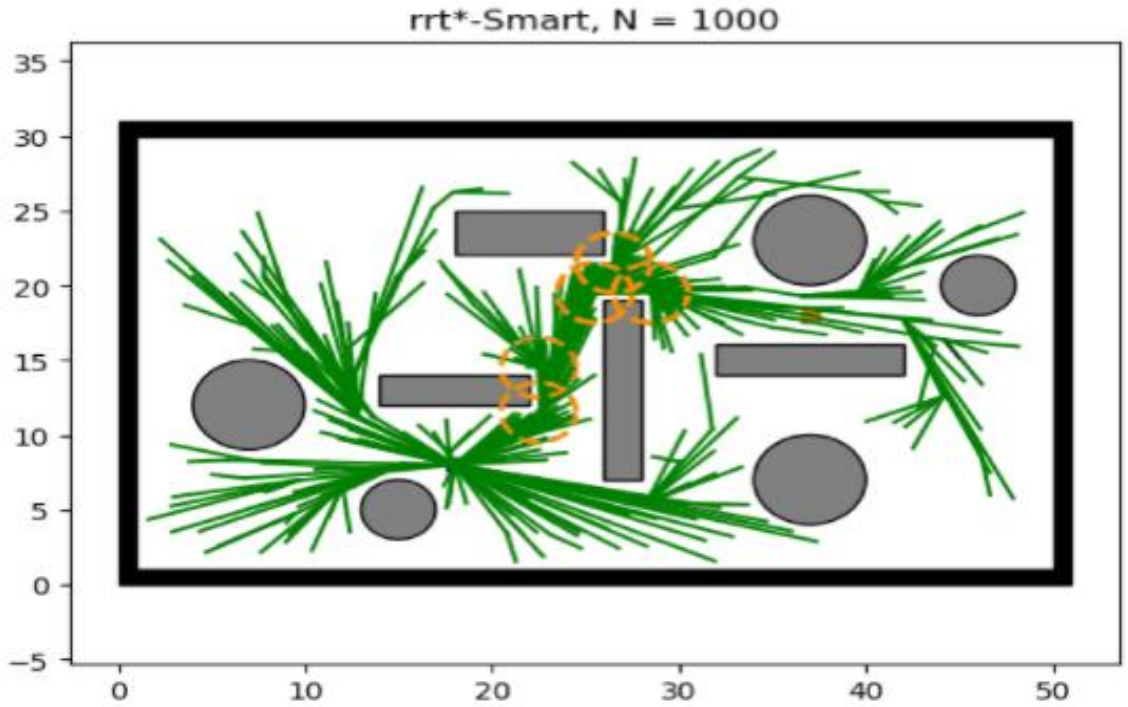
D*



RRT



RRT*



Comparison

All the algorithms have been tested with the same or similar problem statements for a comparative study based on several important factors like the computational delay, path-optimality and number of nodes explored as well as those present in the path , in the test environment.

ALGORITHM	TIME TAKEN (ms)	No.of nodes explored.	No.of nodes in the path.
A*	23.1	920	48
D*	251	2246	48
RRT (1000 iterations)	267	484*	151*
RRT* (1000 iterations)	1732	363*	58*

Conclusion

Four path planning algorithms have been implemented in MATLAB, namely “A*, D*, rapidly exploring random tree (RRT) and RRT*. These have been tested to plan the path from different start and end point configurations as well as obstacle configurations. The A* and D* algorithms, which are Deterministic graph search algorithms are both strong algorithms, which search through the graph (in terms of grids) to find the path from the starting point to the goal if such a path exists. In fact, the grid edge-length i.e. the unit length can be chosen to be greater than or equal to the lateral i.e., the side-to-side span of the robot, and the WORLD mapped with respect to that unit length, and then the algorithms applied.

In almost all the cases, A* can be used if the environment has been completely mapped already, i.e. The entire topography of the environment is already known, and will remain so during the course of the robot’s mission. Also, the requirements for computation are not huge and it can be implemented easily.

D* algorithm, on the other hand, may be used in dynamic environments whose topography isn’t known beforehand. Therefore, while determining its path in a partially-known environment, the robot must be equipped with a sensor or detector in order to sense any variation from the already mapped topography. But in the comparative study, the D* algorithm appears to have a larger computational delay in cluttered environments. This happens because the D* keeps a record of the cost from each of the grids to its neighbouring grids, unlike the A*. Thus, it is not advisable to use the D* algorithm when there is a strict memory limitation.

The RRT algorithm is a rather slow algorithm. Also, it gives a solution that is not desirable in most cases (i.e., non-optimal solutions). It also doesn’t guarantee that the solution will be obtained even if it exists. But it comes in handy in highly cluttered environments. In fact, the algorithm must be selected in such environments only if high memory requirements can be afforded by the path planner. If the resources are available, the RRT gives a fairly good response in cluttered environments.

References

[1] Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation by Y. Koren, Senior Member, IEEE and J. Borenstein, Member, IEEE

[2] Prospective Unmanned Aerial Vehicle Operations in the Future National Airspace System, Matthew DeGarmo and Gregory M. Nelson, American Institute of Aeronautics and Astronautics.

[3] PLANNING ALGORITHMS, Steven M. LaValle, University of Illinois.

[4] Path Planning for UAVs, Scott A. Bortoff, Proceedings of the American Control Conference, Chicago, Illinois June 2000.

[5] Anthony Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments," Proceedings IEEE International Conference on Robotics and Automation, May 1994.

[6] Anusha Mujumdar and Radhakanth Padhi, "Evolving philosophies on autonomous obstacle / collision avoidance of unmanned aerial vehicles," Journal of Aerospace Computing, Information, and Communication, Vol.8, February 2011.

[7] Real-Time randomized Path Planning for Robot Navigation, James Bruce, Manuela Veloso, Carnegie Mellon University.

[8] http://en.wikipedia.org/wiki/A*_search_algorithm

[9]http://www.policyalmanac.org/games/aStarTutorial.htm

[10]http://www.edenwaith.com/products/pige/tutorials/a-star.php

[11] Real-time obstacle avoidance for manipulators and mobile robots, 0. Khatib, Artificial Intelligence Laboratory Stanford University

Plagiarism Report(https://papersowl.com/free-plagiarism-checker)