New path planning

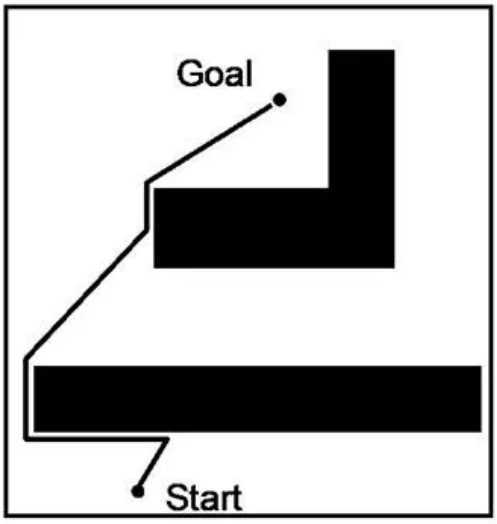
A* algorithm- it is used to find the shortest path planning path. In this first the distance is calculated between the rover and the obstacle.the place where the obstacle is noted is marked as a node. After that from that point to the end point the distance is calculated. That is the max the distance is calculated from the obstacle. Like that various distance is calculated from the rover to the obstacle and the goal. An f(x) function is formed which is the addition of the g(x) and h(x)

The function with the smallest value is chosen that becomes my most feasible travel path.Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.
We define '**g**' and '**h**' as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). successor.f = successor.g + successor.h.

Bug0 algorithm

Robot goes towards the goal. When it has faced with an obstacle follow the obstacle until the way to the goal is revealed. The algorithm is terminated when the goal is reached. The only draw-back of the algorithm is that it is an incomplete algorithm.



Bug 1

Robot goes towards the goal until it hits an obstacle. When an obstacle is encountered, the robot encircles the obstacle and then goes to the point on the boundary of the obstacle which is nearest to the goal. The algorithm is terminated when the goal is reached.Robot has a memory to remember the nearest point to the goal on the boundary of the obstacle. It is a complete algorithm.

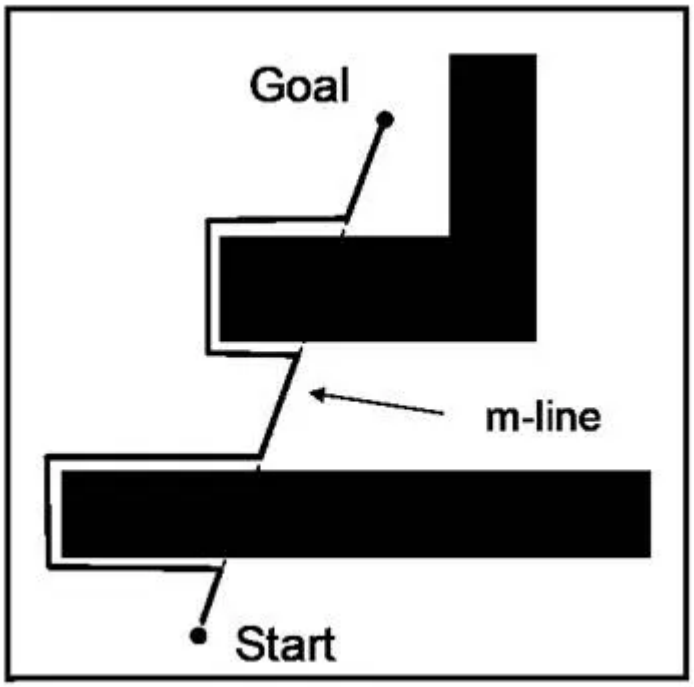
The figure below explains the exact sequence of motion of the robot. As you can see, the robot circles around the entire obstacle at least once which is unnecessary. This makes this method inefficient but it has an added advantage, it is a complete algorithm, which means if there is a way to reach the goal, bug 1 will find it.. It will encircle the entire obstacle and find a safe exit point. It encircle the entire obstacle an dfind the point with the minimum distance from the goal. After it finds that it will go to that exit point and escape from there.

Bug2

Robot goes towards the goal until it hits an obstacle. The line from the start point to the goal point is called m-line. When an obstacle is encountered, the robot encircles the obstacle until it hits m-line. If the encountered point on m-line is nearer to the goal than the point that the robot starts following the obstacle then it stops following and starts moving towards the goal. The algorithm is terminated when the goal is reached. It is a complete algorithm.

The figure below shows the hit points and the leave points that the robot would take. The line in blue is the m-line that it creates at the beginning of its journey. There are certain environments in

which Bug 2 would fail. In the figure below, following the m-line (in red) would return us back



to our start position.

Bug 2 draws a m line between the start and the stop it alings itself towards the goal and travels towards the m line. It moves and when it finds the m line again it takes that line to reach the target

Rrt

The premise of RRT is actually quite straight forward. Points are randomly generated and connected to the closest available node. Each time a vertex is created, a check must be made that the vertex lies outside of an obstacle. Furthermore, chaining the vertex to its closest neighbor must also avoid obstacles. The algorithm ends when a node is generated within the goal region, or a limit is hit.Additionally, the method of chaining the randomly generated vertex is customizable. One method involves calculating the vector that forms the shortest distance between the new vertex and the closest edge. At the point of intersection, a new node is added to the edge and connected to the randomly generated vertex. Alternatively, the vertex can be attached to the closest node by chaining a link of discretized nodes to it. This method requires less computation and is simpler to implement, but requires more points to be stored.

RRT produces very cubic graphs. This is expected as nodes are attached to their nearest neighbor. The structural nature of these graphs hinders the probability of finding an optimal path. Instead of taking the hypotenuse between two points, the two legs of a triangle are navigated across. This is evidently a longer distance. The cubic nature and irregular paths generated by RRT are addressed by RRT*.

rrt*

RRT* is an optimized version of RRT. When the number of nodes approaches infinity, the RRT* algorithm will deliver the shortest possible path to the goal. While realistically unfeasible, this statement suggests that the algorithm does work to develop a shortest path. The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results.First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the `cost()` of the vertex. After the closest node is found in the graph, a neighborhood 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 neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth.

Djkstar algorithm

set initial distances for all vertices: 0 for the source vertex, and infinity for all the other. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.In the end we are left with the shortest path from the source vertex to every other vertex in the graph. This algorithm is used in google maps.

Difference between rrt and a* algorithm

RRT (Rapidly-exploring Random Tree) Sampling-based: Grows a tree by randomly sampling points in the space and connecting them to the nearest existing node. Exploratory: Designed for fast motion planning in high-dimensional spaces.Graph-based: Systematically expands nodes from a start point to a goal based on a cost function. Uses a heuristic function (e.g., Euclidean or Haversine distance) to guide the search toward the goal efficiently.

Tangent bug

The basic idea of the Tangent Bug algorithm is that it uses both motion-to-goal and boundary following behaviors to reach a desired goal. The motion-to-goal behavior is commanded as long as there is no blocking obstacle or if the blocking obstacle does not increase the heuristic distance. The heuristic distance is calculated using the following equation.

$$h=d(x,Oi)+d(Oi,qgoal)$$

where

Oi

is an edge point of a continuous segment of an obstacle as obtained from LIDAR data. A visual representation of these endpoints can be seen below.

Boundary following mode is commanded when the heuristic distance increases. The bug them remains in Boundary following mode until

dreach<dfollowed

. The value

dfollowed

is the shortest distance between the sensed boundary and the goal. The value

dreach

is the shortest distance between the blocking obstacle and the goal.If there is no blocking obstacle,

dreach

is the distance between the robot and the goal.

Algorithmic breakdown

Motion to Goal (While the heuristic distance is decreasing or remaining the same)

1. Compute Lidar Data
2. Move toward the obstacle
3. Terminate if the goal is reached

Boundary Following (While dreach≥dfollowed)

1. Update the Lidar data, dreach and dfollowed
2. Terminate if a complete cycle is completed (goal is unreachable)
3. Terminate if the goal is reached