Some of the most common algorithms used for motion planning in robotics are broadly divided into two categories,

1. Search based algorithm – A\*
2. Sampling based algorithm – RRT & RRT\*

To reach from the start position to the end position, the robot follows a given path. The state of the robot at any given points along the path can be described using three variables

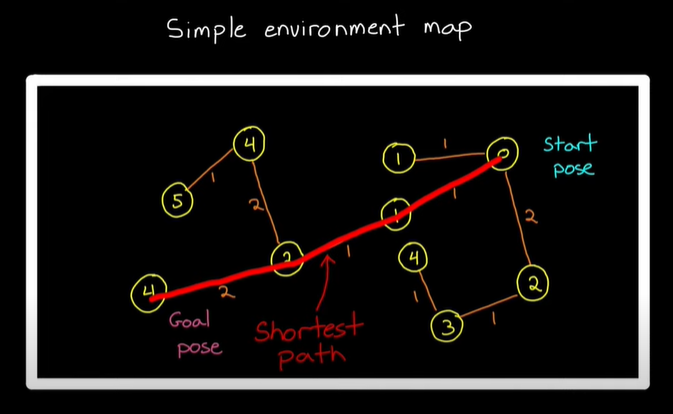
1. X
2. Y
3. Orientation

A path planning problem includes finding the path to reach from point A to point B and is a subset of the motion planning problem. In case of motion planning, a wider set of variables are estimated – velocity, acceleration, angular rotation etc.

Graph based methods to find the path between a start and goal pose

General idea:

* Create nodes in random directions on a graph such that the cost of getting to a node is the distance between the two nodes. The total cost of reaching the end goal would be a total of the lengths between nodes. Randomised points create paths although not the shortest
* In the next iteration create new nodes and keep checking the costs associated with reaching nodes. If a previous node is encountered, compare the total lengths between this new path and the old one and keep only the path with a shorter length.
* In this kind of a search algorithm, only trees (subset of interconnect paths wherein the nodes can connect to other nodes in whatever way possible) are maintained (each node has only one parent) by always keeping the smallest path
* Keep updating and the shortest path is obtained. This kind of search algorithm doesn’t render the shortest path precisely but keeps approaching it as the number of nodes keep approaching infinity
* Generally speaking, this building of trees from random wandering isn’t the best solution and that’s where the graph-based methods of tree building come into picture



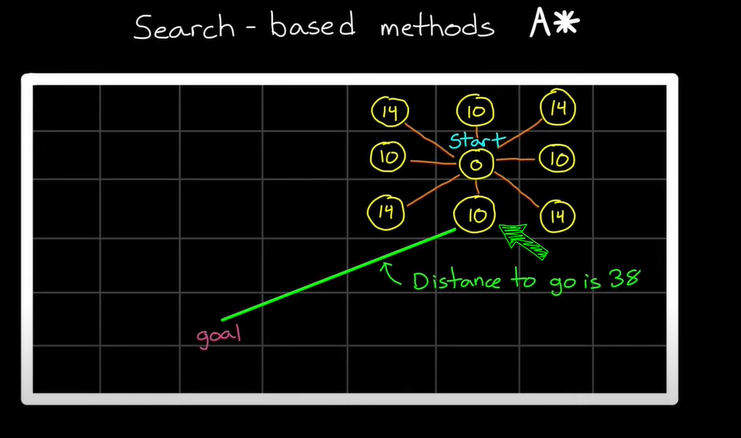
**Search based algorithms**

Build trees by adding nodes in an ordered pattern

* Start with a grid-based occupancy map and create nodes in each cell, the algorithm would then check for the cost that reaching each cell would take
* Keep populating each cell in the grid with a node and calculate and eliminate the paths with greater costs
* When the entire grid is covered, while maintaining the tree structure, the shortest path would be the one with the least cost associated to it
* Optimal path will be generated where the degree of optimal path would depend on the resolution of the grids
* Computationally expensive, since it is a brute force method of finding the more optimal paths

As a result, the A\* algorithm was found out.

* Here everything else remains the same, but instead of having to search through a brute force method, an additional cost is calculated along with the cost of getting to a node
* The total cost in case of the A\* algorithm is the cost to get to the node and the straight-line distance between the node and the end goal.
* As it keeps a track of the distance between the node and the end goal, it chooses a path that is more likely to produce a path with a shorter distance
* As a result, you don’t have to check at every node

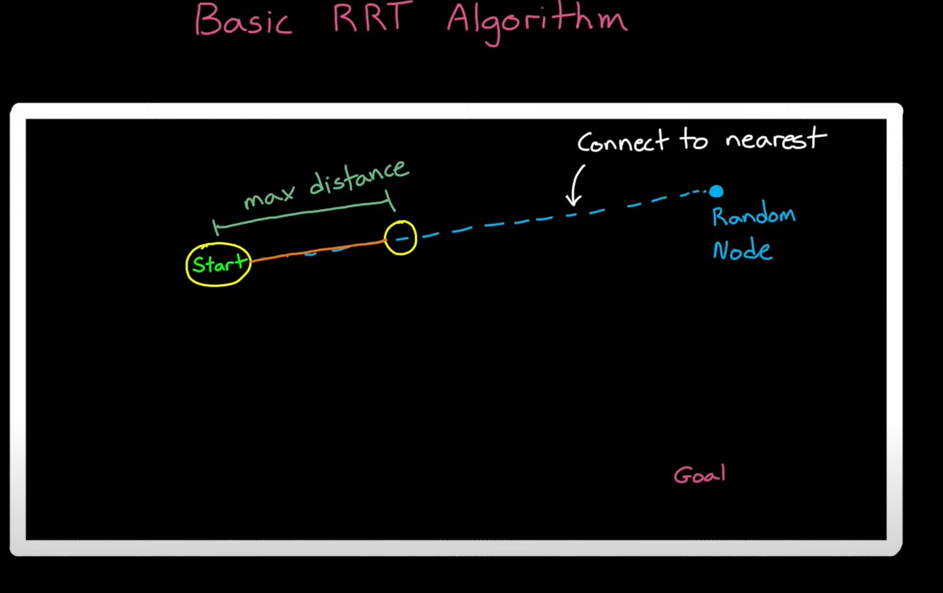


Issue – Computationally intensive if the size and dimensions of the state space increase, and hence the search-based algorithms are generally not preferred. **Hence, sampling-based algorithm**

Rapidly Exploring Random Trees

This algorithm has the following steps,

* Define the start and the end goal points
* Start by randomly creating nodes anywhere on the occupancy map. This node point can be created too far away as compared to the previous point which can cause the path to move away from the goal point, or wander off in a direction that is away from the goal point. Hence, in a RRT algorithm a max distance is defined. Thus, when a new node is created, there might be three situations that might arise,
  + If the new node created is within the max distance from the previous node, it is placed where it is produced
  + If the new node produced is more than the max distance from the previous node, then it is placed at the max distance from the previous node, along the same direction
  + If in any of the two cases, the path between the two nodes passes an obstacle then that node is completely ignored and a next node is created

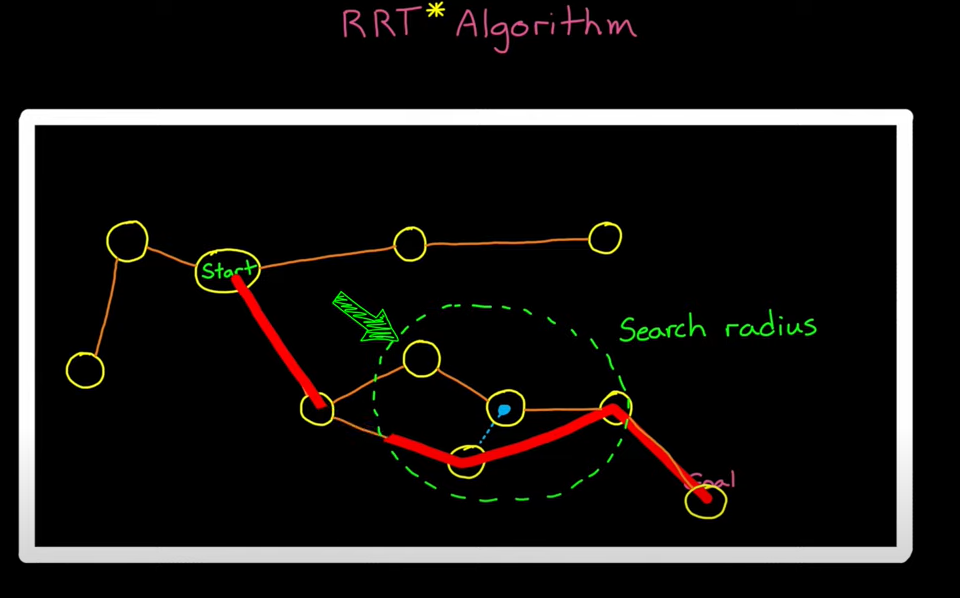


* The new node created is then joined to an existing node which is closest to the new node. Hence, in such a case there will be a path formed that is rapidly approaching the goal destination.
* Even if there are nodes created away from the goals, the paths they form will keep growing in the opposite direction, leaving the actual optimal paths untouched. Hence, there will always be some paths that will always keep moving towards the goal destination
* Once a single path reaches within a threshold of the goal position, the algorithm stops producing nodes/expanding the trees

RRT\*

In case of the RRT\*, the searching proceeds as follows,

* Random nodes are created on the map and placed based on the maximum distance criterion (if the new node is at a distance greater than the max distance, the new node is placed at the max distance; if not, then the new node is placed where it is created)
* After a particular node is created, the next step in case of the RRT was to join it to the nearest node. This is where RRT\* is different from the RRT algorithm
* Instead of directly attaching it to the nearest node, it looks for a node within a particular search distance that not only keeps the tree structure intact but also reduces the total path length. This means the new node can connect to an old node, even if the old node is not the closest to the new one.



Implementing RRT\* on MATLAB:

Implementation requires the definition of a state space and a state validator object. These are

1. State space – is the space in which the states of the vehicle can lie, and the variables used to describe the state of a vehicle are called as the state space variables. For e.g., the x and y coordinates and the orientation angle theta are the state space variables used to describe the motion of a robot. These state space variables can be interpolated to find out the motion of the robot. In case of the mobile robot example on MATLAB, there are three state space objects defined on MATLAB – SE(2), ReedsShepp state space and Dubins state space. In each of these state spaces, the vehicle is allowed to move in a certain way
2. To check whether the motion created in the state space, is valid or not, a state validator is used. On MATLAB, the binary occupancy map is an example of the state validator.

OBSERVATIONS WHILE USING THE RRT\* ALGORITHM

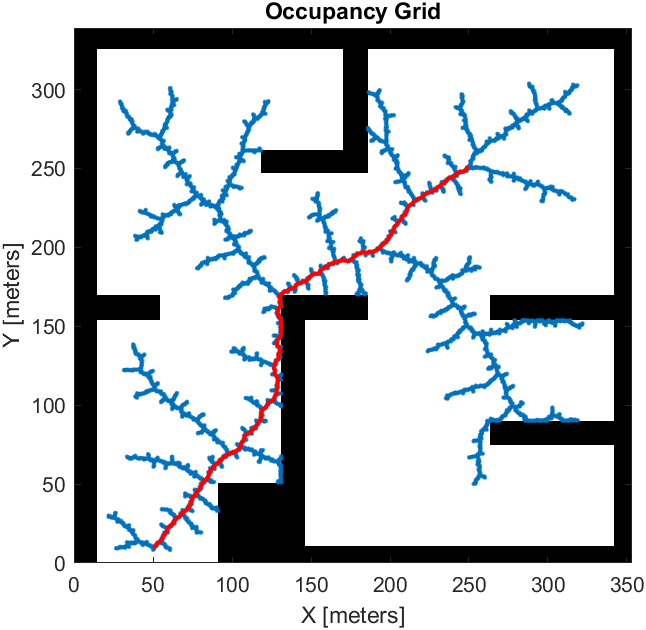
The planner parameters that need to be tuned while setting up the RRT\* is the max connection distance and the number of iterations.

**The max search distance** decides the maximum distance up to which the vehicle/robot is allowed to move. For e.g., if it is set at 0.1m, then at a time, the next node in a tree will lie at a distance of 0.1m from the previous node in a given direction. If the search distance is set to low for a given number of iterations, then the tree will not be able to spread out much and the goal destination will be not be reached. As compared to this, if the search distance is increased while keeping the number of iterations same, the tree spreads out more and reaches the destination.

**The max iterations,** this decides the number of random nodes that are created in the state space. If the number of iterations is not enough for a given max search distance, it is better to increase the search distance OR increase the number of iterations to let the nodes spread out.

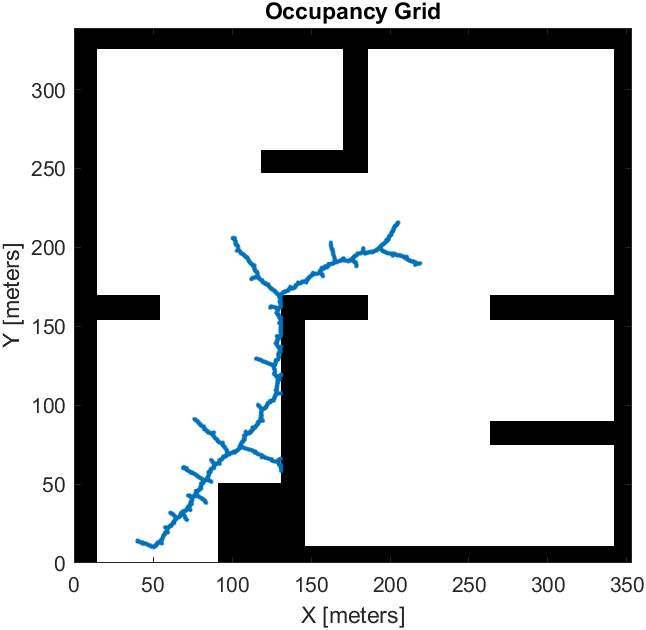
The **BallRadiusConstant** is the radius in which the shortest path node is found in case of the RRT\*. Like mentioned earlier, in the RRT\*, a new node is not directly attached to the nearest node. Instead, a search is carried out where the new node is attached to every node that lies within a given search radius, and the branch that results in the shortest path is the one that is taken forward even if it is not the nearest node. For e.g., if the ball constant radius is set at 10, then every node created will be checked for shortest path nodes within a 10-unit radius.

1. Max iterations – 10000 & max search distance – 0.4, somewhat the shortest path



1. Now if the number of iterations is decreased while keeping the same max search distance,

Max iterations – 2000 & max search distance – 0.4m, the iterations are not enough to reach the goal



1. If we check the sensitivity of the max search distance, max iterations – 6000 & max search distance – 4. As expected, we find out that the path obtained is not the shortest path, and the tree is denser than the one with a lower max search distance value. It is denser because the robot can now move greater distances and hence the nodes are able to explore more for the given number of iterations. The path is longer because the new nodes created travel farther away as compared to a smaller max search distance. A longer search distance would also mean that the resultant path isn’t necessarily a straight-line path but crooked one.

