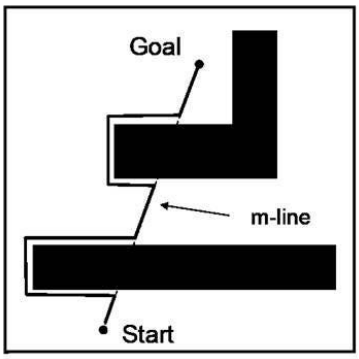
PATH PLANNING

Date : 8-02-2025 Full Name : Vighnesh Reddy

1. **Bug 0 Strategy** :
   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.
   Sensors needed : Proximity, Contact Sensors

2. **Bug 1 Strategy** :
   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.
   Sensors Needed : Proximity, Contact, Odometry, GPS Sensors

3. **Bug 2 Strategy :**
   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.
   Sensors Needed : Proximity, Contact, Odometry, GPS Sensors



4. **Tangent Bug Strategy** :
   The **Tangent Bug Strategy** is a local path-planning algorithm designed for navigating a robot toward a goal while avoiding obstacles. It combines obstacle boundary-following with goal-directed movement to efficiently find a

path, leveraging range sensors (like LiDAR) to detect and analyze nearby obstacles. The strategy calculates tangents to the detected obstacles and evaluates which direction minimizes the distance to the goal. If a direct path is clear, the robot moves toward the goal; otherwise, it follows the boundary of the closest obstacle, continuously checking for a shorter path. This approach balances simplicity and adaptability, making it suitable for cluttered environments where obstacles may block the direct path.
Has Mainly 2 modes : Boundary Following Mode and Motion to Goal Mode

Sensors Required : LIDAR/Depth Camera/Ultrasonic/Infrared – Obstacle Detection
Odometry/GPS/SLAM – Position Estimation
IMU – Goal Reacquisition

Pseudocode :

```
initialize_robot()
while not goal_reached():
    if path_to_goal_is_clear():
        move_directly_to_goal()
    else:
        obstacle_detected = True
        while obstacle_detected:
            follow_obstacle_boundary()
            if path_to_goal_is_clear():
                obstacle_detected = False
                move_directly_to_goal()
            if full_loop_around_obstacle():
                declare_goal_unreachable()
                break
stop_robot()
```

5. **Dijkstra's Algorithm** :

Dijkstra's algorithm is a graph-based pathfinding algorithm used to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights. It works by systematically exploring nodes in increasing order of their cumulative cost from the start, ensuring the shortest path is found for each node before moving on. The algorithm uses a priority queue to efficiently track the next node with the lowest cost. While it guarantees optimality and completeness, it can be computationally expensive as it explores all possible paths without prioritizing a specific goal, making it less efficient.
It explores all the available path to a goal, to find the most optimal path but it is very costly and time in-efficient.

Disadvantages :
No heuristic, leading to inefficient exploration.

Slow in large search spaces.

High memory usage.

Static nature; not suited for dynamic environments without re-computation.

6. **A* Algorithm :**

A* is a heuristic-based pathfinding algorithm designed to find the shortest path from a start node to a goal node in a graph. It combines the actual cost from the start to a node g(n) with an estimated cost to the goal (h(n) forming a total cost function f(n)=g(n)+h(n). This heuristic allows A* to focus on paths that are likely to lead to the goal, making it more efficient than Dijkstra's algorithm in many cases. A* guarantees an optimal solution if the heuristic is admissible (never overestimates the true cost) and consistent.

Heuristic Function: Commonly Euclidean distance, Manhattan distance, or diagonal distance

Disadvantages:

Dependence on heuristic (over/underestimation)

Computationally very expensive

Very High Memory usage

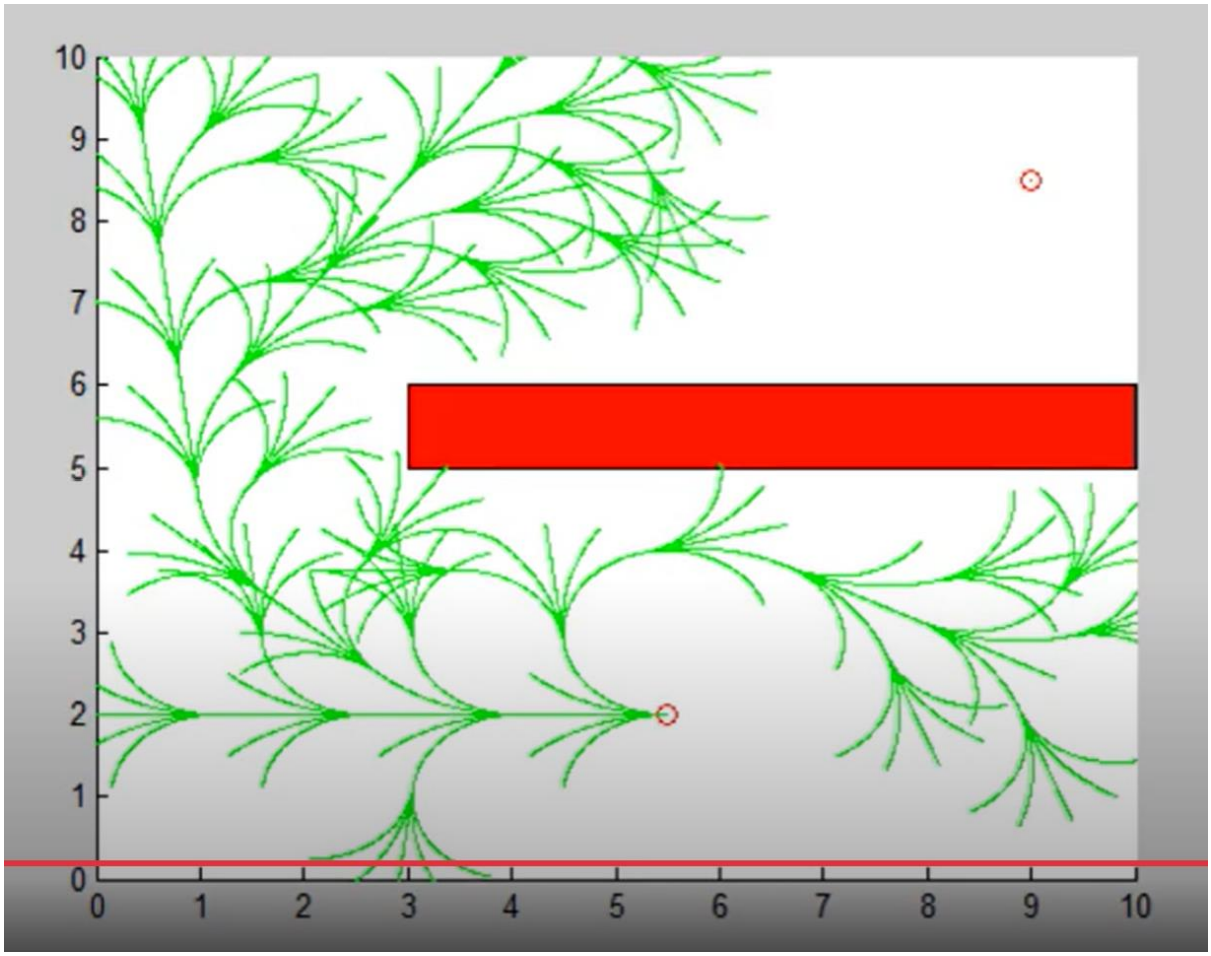
Struggles in dynamic environments

Inaccuracy in fixed grids and irregular obstacles
Found in nav2 stack , nav2_bt

7. **Hybrid A* Algorithm :**

Hybrid A* combines the discrete search efficiency of A* with continuous-space considerations.

It operates on a grid but expands nodes based on continuous motion primitives, representing feasible vehicle maneuvers (like turns and reversing). Each grid cell represents a state, including position and orientation. A heuristic function, often combining distance and non-holonomic estimates, guides the search. Collision checking is crucial, ensuring generated paths are obstacle-free. The algorithm finds near-optimal paths by balancing grid-based exploration with the constraints of continuous motion.



8. **Breadth First Search (BFS) and Depth First Search (DFS) :**
   BFS : Explores all nodes at the current depth before moving to nodes at the next depth.
   DFS : Explores as far as possible along one branch before backtracking.

   BFS can give us the the fastest path in unweighted graphs
   DFS uses lesser memory compared to BFS as it explores a region first

   Disadvantages:
   BFS needs a lot of memory for computing the most optimal path
   DFS can either get stuck in cycles or fail to find the most optimal path.

   BFS cares about hopping the least number of edges, whereas Dijkstra cares about the lowest cost of the function
   BFS goes layer by layer.

9. **Rapidly Exploring Random Trees (RRT)**

RRT builds a tree by randomly sampling points in the configuration space.
Each new sample connects to the nearest node in the tree if the path is collision-free.
Useful in high dimensional spaces and fast in open spaces.

RRT is a single query algorithm

Disadvantages:
May not find the most shortest path.
Needs a lot of post processing to make the path smooth.

10. Bi-Directional RRT (Bi-RRT):

Builds two trees (one from the start, one from the goal).

11. RRT*:

Improves path quality by optimizing connections.
It considers nearby neighbouring nodes when connecting a new sample. It reconnects nodes to potentially better parents, optimizing the tree.
Aims to find asymptotically optimal paths, meaning it converges to an optimal solution as the number of samples increases.
It prioritizes in finding the most optimal and efficient path

It keeps altering the main parent tree until the shortest path is found.But its computationally very expensive

Use RRT when you need a fast & feasible path, and RRT* when path optimality and efficiency are also needed

12. Probabilistic Roadmaps (PRM)

Randomly samples points in free space, connects these points into a graph, and finds paths using graph search algorithms.

Efficient in static environments but not dynamic environments.

PRM is a multi query algorithm, It requires a pre-processing phase
PRM mainly focuses on building a reusable roadmap
Is used in virtual environments where multiple bots might need to navigate

13. Potential Field Method:

The robot is attracted to the goal and repelled from obstacles using a mathematical potential field.
The goal creates a attractive field and obstacles create a repelling field, so the robot just slides down the potential field

Disadvantages:
Can get stuck in the local minima
Might not work well in complex environment