IIT MADRAS

ED5215: Introduction to Motion Planning

Path Planning for Multiple Mobile Robots

Eshant Kumar Jha(OE22S300)
Mrityunjay Upadhyay(OE22S002)

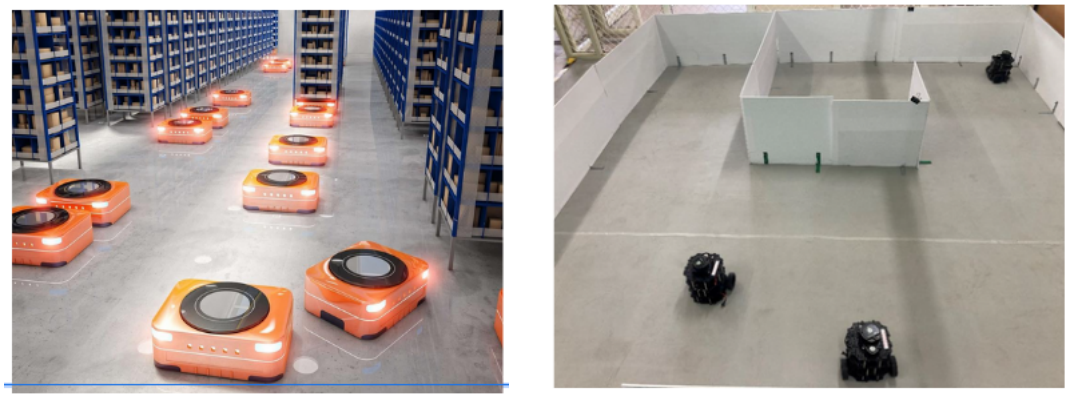

Figure 1: Path planning of multiple robots.

1 Abstract

In this study, we address the problem of path planning for multiple robots. Path planning is an essential task in robotics, and it becomes more complex when multiple robots are involved. In order to efficiently plan the paths of multiple robots, we propose the implementation of the A star search algorithm using the Euclidean heuristic. The A star search algorithm is a well-known algorithm for path finding, and the Euclidean heuristic provides an effective method for estimating the distance between two points in space. By combining these two approaches, we aim to develop a path planning algorithm that can handle multiple robots while minimizing the time and computational resources required to plan their paths. We evaluate the performance of our algorithm through simulations and compare it with existing path planning methods. Our results demonstrate the effectiveness of the proposed approach in terms of planning time and path optimality, making it a promising solution for path planning of multiple robots in various applications.We evaluate the performance of our algorithm through simulations. We also demonstrate that our algorithm can handle dynamic obstacles effectively.

2 Methodology

There are two separate architectures: the Motion Planner Architecture and the Controller Architecture. The main function establishes the connection between these two architectures. Setting up the Controller:

- Main function: Sets up the simulation
- Simulator interface function: Connect to simulator, get object handles, obtain wall positions, read robot and goal positions, set wheel velocities, start/shutdown simulation.
- Control function: Implements a simple waypoint control.
- Robot parameter: Stores the physical parameter values that are important for the simulation

Implementation of the Search Algorithm :

- Maze : To Plot the Maps and define the start state , goal state and successor states of the Robot
- Search : where we implement A * algorithm

For path planning of multiple robot we started implementing A* algorithm for each robot.
State space: (x, y, t), we introduced time here as the third dimension so as to avoid the collision
Action : (up, down, right, left, stop), action spaces consists of 5 variables
Cost function is 1 for each action
Search Algorithm: Implement A* with Euclidean distance as the heuristic.

Initially while setting up the controller we set the robot parameter and control function which consits of The gtg method implements the main control logic for the robot. It first calculates the angle between the current robot position and the goal position using the np.arctan2 function. It then computes the difference

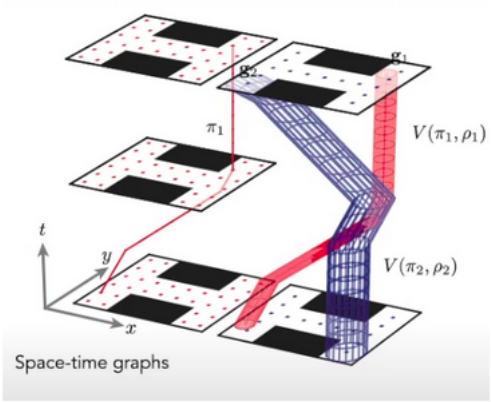

Figure 2: space time graph of multiple robots

between this angle and the current robot heading (orientation), which is the error that the controller tries to minimize using a PID algorithm.

In Figure 2 we can see how choosing time as the 3rd coordinate helping us to avoid collision that is clash in the robots can be avoided using space time concept

The PID algorithm used in this code has three gains: Kp, Ki, and Kd, which control the proportional, integral, and derivative terms of the algorithm, respectively. The control input for the angular velocity of the robot is calculated using the PID formula W = Kpe_new + Ki_total_heading_error + Kd*e_dot, where e_new is the current heading error, e_dot is the derivative of the error, and total_heading_error is the cumulative heading error.

The Euclidean distance heuristic is a commonly used admissible heuristic in path planning problems.The Euclidean distance heuristic is calculated by taking the absolute difference between the x-coordinates and the y-coordinates of the current and goal states of the robot and summing them up. This is done using the formula: $h = (abs(x_s - x_g) + abs(y_s - y_g))$

We define our state space to include the robot's current position and time to avoid collision. Our actions consist of moving in different directions or stopping. We then implement the A* search algorithm using the Euclidean distance heuristic to plan the paths of multiple robots. To test the performance of our algorithm, we consider dynamic obstacles that can change their position over time.

We Plan the paths of the robots in a sequential order such that each robot waits for the previous robot to finish its path planning before planning its own path. In this way, each robot can take into account the paths of previously planned robots to avoid collisions.

1. Robot 1 Plans its Path
2. Robot 1 sends its path's information to Robot 2 and according to that Robot 2 plans its path
3. Robot 1 and Robot 2 sends its path's information to Robot 3 and according to that Robot 3 plans its path

Getsuccessor function generates a list of all possible successor states for a given state, filtering out states that are obstructed by static or dynamic obstacles. The successors list is then used by other search algorithms, such as A* search, to explore the search space and find an optimal solution.

Now, let's go through the step-by-step explanation of the algorithm:

- Initialize two lists, fringe and explored, which will store the nodes that need to be explored and the nodes that have already been explored, respectively.
- Add the starting state of the problem to the fringe list along with a cost of 0. The starting state is obtained using the getStartState function of the problem instance.
- While the fringe list is not empty, perform the following steps:
  a. Sort the fringe list in ascending order based on the total cost of the nodes.
  b. Pop the node with the least total cost from the fringe list and add it to the explored list.
  c. Check if the popped node is the goal state. If yes, return the path to reach the goal state.
  d. Otherwise, expand the current node by generating all possible successor nodes. The successors

are obtained using the getSuccessors function of the problem instance.
e. For each successor node, compute the total cost and heuristic value.
f. Check if the successor node is already in the explored list. If yes, ignore the node and move on to the next successor.
g. Check if a duplicate of the successor node already exists in the fringe list. If yes, compare the costs of the two nodes and keep the one with the lower cost. If the existing node has a lower cost, ignore the successor node and move on to the next successor.
h. If the successor node is not already in the explored or fringe list, add it to the fringe list.

If the fringe list becomes empty and the goal state has not been found, return failure.

The major challenge we had faced was the collision avoidance part initially we thought of planning like 1st Robots goes then it will transfer its path to robot 2 than robot 2 moves and then robot 3 which is not taking place simultaneously we overcome this problem by considering other robot as dynamic obstacle with respect to first robot,

To Avoid static Obstacle i.e wall and dynamic Obstacle i.e other robots w.r.t to one robot

For Static Obstacle:

if isObstacle(new successor)

This is to check whether a newly generated state is an obstacle or not. If it is an obstacle, it should be skipped

For Dynamic Obstacle:

if (abs(new successor[0] - dynamic obstacle[0]) + abs(new successor[1] - dynamic obstacle[1])) <2

i.e., if the distance between the successor state and the dynamic obstacle is less than 2, then the method considers this a collision with the dynamic obstacle and skips to the next possible movement.

The code calculates the Euclidean distance between the successor state and each dynamic obstacle using the formula (abs(new successor[0] - dynamic obstacle[0]) + abs(new successor[1] - dynamic obstacle[1])), where new successor is the newly generated successor state and dynamic obstacle is the position of the dynamic obstacle. If the distance is less than 2, then the code considers this a collision with the dynamic obstacle and skips to the next possible movement. This is done using the continue keyword, which skips the current iteration of the loop and moves on to the next iteration. to avoid collisions between the robot and the dynamic obstacles during the path planning process. If a collision occurs, the robot will have to choose a different action to avoid the obstacle and continue on its path.

3 Results

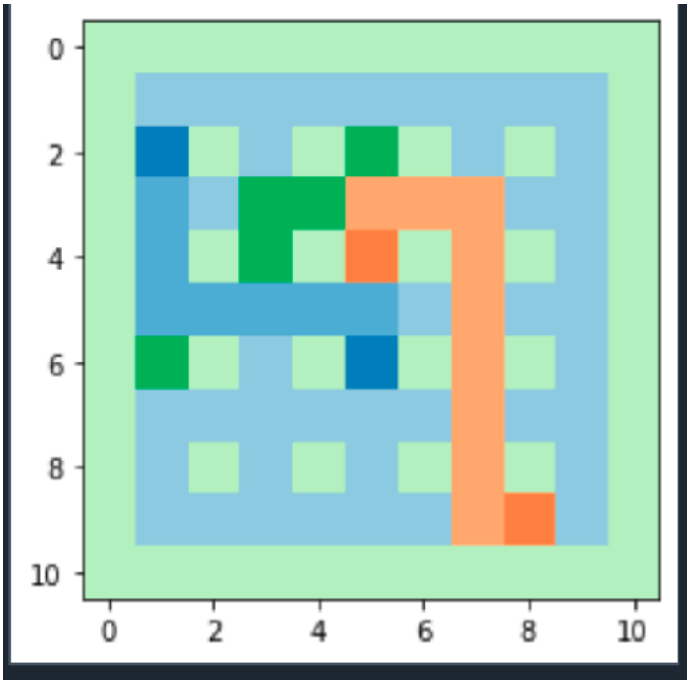

Figure 3: Path Planning of multiple robots in maze 1 which is coupled with coppeliaSim environment

Here in Figure 3 as we can see the blue path is the path traced by robot 1 and the green is the path traced by robot 2 and the orange is path traced by robot 3 and as through the simulation we have validated that they did not collide while running and ran simultaneously since we have used 5 actions that is up down left right and stop so accordingly the robots pick action depending on the other robots so they can reach their Goals simultaneously
Here A* gives an optimal and complete path

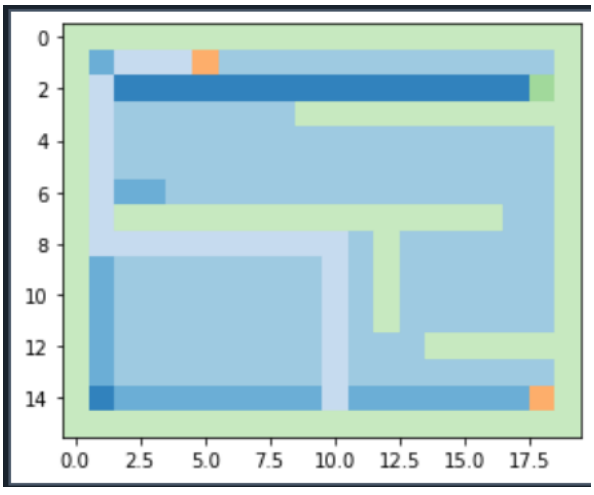

Figure 4: Path Planning of multiple robots in maze 2

Here in Figure 4 we have a different maze configuration with an alternative set of start state and goal state of all three robots and here as we can see that as the first and second robot are required to travel on the same path so the second robot gives way the path to robot 1 and stops there as the robot 1 crosses that particular state at say time $t = t_o$ meanwhile robot 2 just waits there or stopped and after the robot 1 crosses that path robot 2 continues on the same path

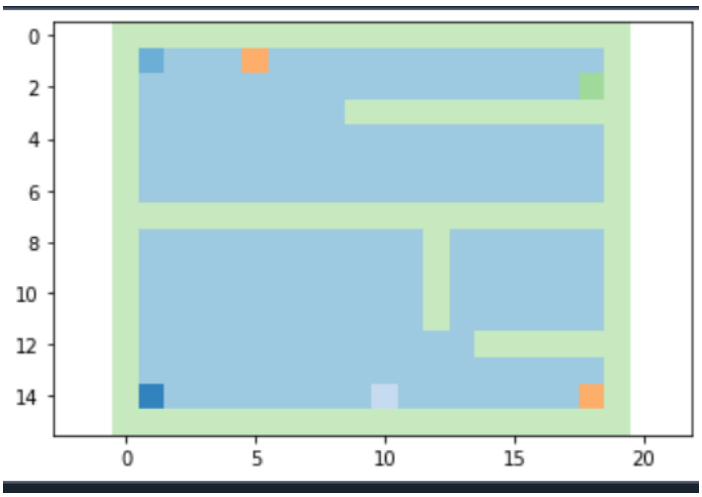

Figure 5: Obstruction in the path

In figure 5 as we can observe the Robots does not move since they would not be able to find the path to goal as their is obstacle in the path

Figure 6 is the coppeliaSim scene setup of 3 robots standing at their starting state it is the 3D representa-

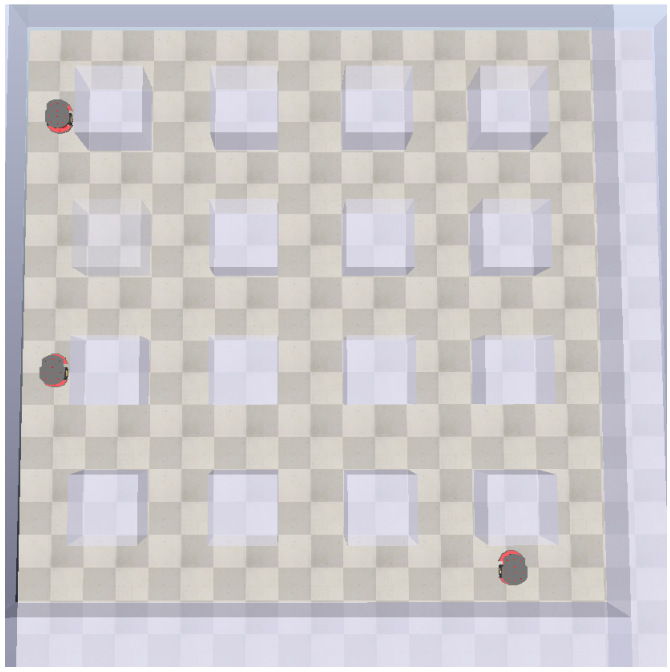

Figure 6: CoppeliaSim Environment before running the simulation for multiple Robots

tion of the maze 1 that is just plotted as the figure 1 In figure 7 we can see that after running the simulation the robots starts moving towards their respective goal positions in the CoppeliaSim environment

Figure 7: CoppeliaSim Environment after running the simulations for multiple Robots

4 Discussions

Strengths of the A* algorithm for path planning:
a. Completeness: A* is a complete algorithm, meaning it guarantees finding a solution if one exists, given that the search space is finite.
b. Optimality: In its original form, A* guarantees finding the optimal path, i.e., the shortest path from the start to the goal, under certain conditions.
c. Heuristic-based search: A* uses heuristics to guide the search process efficiently, which can lead to faster convergence compared to uninformed search algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS).
d. Versatility: A* can be adapted to different problem domains by modifying the heuristic function and the definition of the state space.

Limitations of the A* algorithm for path planning:
a. Memory and computation requirements: A* may require substantial memory and computation resources, especially for larger search spaces or complex environments, due to the need to maintain and update the open and closed sets.
b. Optimality trade-offs: While A* guarantees optimality under certain conditions, the presence of obstacles, dynamically changing environments, or conflicting goals for multiple robots can lead to sub-optimal or non-optimal solutions.
c. Heuristic selection: The effectiveness of A* heavily depends on the choice and accuracy of the heuristic function. Designing an admissible heuristic that provides accurate estimates of the remaining cost can be challenging in some scenarios.
d. Coordination of multiple robots: When applying A* for multiple robot path planning, ensuring collision avoidance, synchronized movements, and efficient coordination among the robots can be complex.

Initially, we implemented a *centralized* approach where a central controller was responsible for controlling all the robots. However, we realized that this approach had certain drawbacks. If the central controller were to fail, the entire system would become non-functional. To address this issue, we decided to explore the decentralized approach.

In the *decentralized* approach, each robot operates independently without direct control from a central authority. However, to ensure coordination and avoid conflicts, we established a priority system. Robot 1 is given the highest priority and moves first, followed by Robot 2, and so on. This sequential movement ensures that robots do not interfere with each other's paths.

To further enhance the coordination, we introduced a set of rules that govern the movement of the robots. These rules can be based on various principles, such as *COLREG's* (Collision Regulations at Sea) rules commonly used for ship maneuvering in the ocean. By defining specific rules, we can guide the robots' movements and mitigate the risk of collisions or conflicts.

It is important to note that the choice of rules and coordination mechanisms can vary depending on the specific requirements and characteristics of the system. By adopting a decentralized approach with defined priorities and rule-based movement, we aimed to enhance system robustness while ensuring effective coordination among the robots.

Here is the link of the video explaining the results we get through the simulation of the algorithm implemented. `https://drive.google.com/file/d/1ERVliEOFIaShTRi1TOwMANhVSt6sTfip/view?usp=sharing`