ECE 276B PR2: Motion Planning

Varun Vupparige
Mechanical and Aerospace Engineering
University of California San Diego
La Jolla, USA

I. Introduction

An autonomous system consists of various subsystems such as sensing, perception, path planning and control systems, and the study of path planning algorithms has been an important research topic. One of the major challenges on the way to obtain robot autonomy is path generation which is safe, reliable and dynamically feasible. In a path planning problem, an agent is trying to navigate from point A to point B without hitting any obstacles. The planned path of the agent can be evaluated based on metrics such as path length, movement time, energy consumption and risk levels.

Generally, a discrete feasible planning model is used to formulate the path planning problem. This path planning problem is similar to a deterministic shortest path problem. In graph theory, shortest path problem computes the shortest path between any two given nodes in the graph Broadly, motion planning algorithms can be divided into two types: Search-based and Sampling-based planning algorithms. Sampling based path planning algorithms are widely used today for higher dimensional and complex systems. Rapidly exploring random trees (RRT) is one of the popular sampling algorithms which involves incremental construction of search trees that rapidly and randomly explores the environment map to form a path towards the goal.

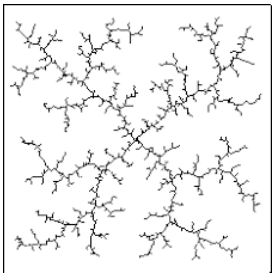

Fig. 1. RRT: Sampling-based algorithm

Search-based algorithm divides the environment map using grid method and expands outward from start point by choosing the grid with minimum evaluation function value. The evaluation function is dependent on actual path cost from start to present state and estimated path cost from present state to goal. There are many different versions of RRT and A* which address some bottlenecks posed by the original sampling algorithms such as ARA*, LRA etc.

In this project, we are implementing a search-based algorithm to plan a path from robot and its moving target. The movement of target is governed by its planner which is based on maximizing the minimal distance that can be achieved by robot using a mini-max decision rule. The planning problem considered here is deterministic and has a finite state space. Our planner is supposed to compute the next step of robot within 2 seconds.

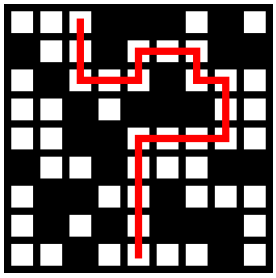

Fig. 2. A*: Search-based algorithm

II. Problem Formulation

In this section, the problem of path planning using search-based methods is formulated as Discrete Feasible Planning model by carefully explaining each of its parameter:

A. Discrete Feasible Planning

A typical discrete planning model is composed of following parameters:

- X: Discrete/continuous set of states
- U: Discrete/continuous set of controls
- x_0: Initial robot/agent state
- $x_{t+1} = f(x_t, u_t)$: Motion Model/State Transition Function
- x_g: Goal state

Lets formulate the parameters of the Discrete Feasible Planning model in context of our project:

1) State Space: The state X is a vector consisting the location of the agent in terms of the x and y coordinates which specify the location of the agent with respect to the origin $(0, 0)$.

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \text{ where, } x, y \in \mathbb{R} \tag{1}$$

In this project, environments considered are two-dimensional grids consisting of static obstacles, moving

goal and an agent. It is a occupancy grid map where the map of the environment is represented as evenly spaced binary variables. Therefore the state space is bounded by the size of the environment. The robot position is indicated by a blue square, while the target position is indicated by a red square and obstacles are denoted by black blocks.

$$\begin{aligned} 0 \leq x \leq envmap_{x,max} \\ 0 \leq y \leq envmap_{y,max} \\ envmap_{black} = obstacles \\ envmap_{blue} = robot \\ envmap_{red} = target \end{aligned} \tag{2}$$

2) Control Space: The control space $\mathcal{U}$ is the set of given actions taken by the agent to reach the final goal (τ). The control is u is taken to reach these neighboring states. The robot has eight control actions and the target has only four control actions since it has no diagonal movement. The cost of reaching any valid possible state is equal to unity. These action can be mathematically formulated as follows:

$$u_t = \begin{bmatrix} (1,0) & (1,1) & (1,-1) & (-1,1) \\ (-1,-1) & (0,1) & (0,-1) & (-1,0) \end{bmatrix} \tag{3}$$

Each element of the tuple denotes the translation in x axis and y axis. Using the definition of state space and control space, lets define the motion model.

3) Motion Model: Based on the definition of state space X and control space U, we can now define the motion model. Given a robot's state x_t, and control input u_t, the state x_{t+1} can be modelled as a probabilistic/deterministic function given by,

$$x_{t+1} = f(x_t, u_t) = p_f(.|x_t, u_t) \tag{4}$$

Since our project is deterministic in nature, we can discard the probability mass/density function. For a particular state, there can be eight possible next states based on the control actions. Few of the states which results in an obstacle state or out of bounds state are invalid.

$$x_{t+1} = x_t + u_t \tag{5}$$

4) Initial State and Goal State: The initial state x_0 represent the starting state of the robot/agent. The goal state x_g represent the target state of the robot/agent.
This formulated problem is real time motion planning problem where in we have to decide a move in every two seconds. Thus, the task of planning algorithm is to find a finite sequence of actions that when applied, transforms the initial state to some state in goal. Lets define the deterministic shortest path problem to complete our problem formulation:

B. Deterministic Shortest Path problem

Consider a graph with a finite vertex/node set V, edge set E, edge weights C, start node s and end node tau. Let a path be sequence of nodes belonging to to the vertex/node set V. Sum of edge costs along the path is defined as $J^{i:q} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$. We assume that there are no negative cycles in the graph. The objective of this problem is to find the shortest path from start node s to goal node tau such that we use least possible cost. This can be mathematically formulated as follows:

$$\text{dist}(s, \tau) = \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}} \tag{6}$$

$$i^*_{1:q} = \arg\min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}} \tag{7}$$

where, $\mathcal{P}_{s,\tau} := \{i_{1:q} \mid i_k \in \mathcal{V}, i_1 = s, i_q = \tau\}$ are all the paths from the start node s to goal node tau.
In correlation to our project, the environment, defined as a binary occupancy map is represented as a graph where each coordinate in the map is a node. Edge weight is defined to be unity for any valid control action. We are required to find the optimal path from start node to goal node, such that it results in least possible summation of edge costs.

III. Technical Approach

In this section, the implementation of a search-based path planning algorithm to compute shortest path between start node and goal node is explained in detail. This section also introduces the mathematical framework of weighted a* algorithm to solve a finite horizon optimal control problem.

A. Weighted A algorithm*

Now that we have formulated the deterministic shortest path problem using the discrete feasible planning model, lets discuss about one of popular search based planning algorithm to solve this problem. Search-based algorithm are an extension of the label correcting algorithm where we discover the nodes with lowest cost, and update the cost of their children nodes.

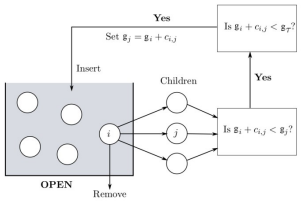

Fig. 3. Label Correcting Algorithm

These set of nodes which can potentially be part of the shortest path are stored in OPEN list. The order in which we remove a node from the OPEN list defines different type of

search algorithms such as Best-first search, Depth-first search and Dijkstra's algorithm.

The fig 3 depicts the flowchart explaining all the steps involved in the label correcting algorithm. Here, g_j is the present lowest cost of reach node j from the start node s, c_{ij} is the cost of travelling from node i to node j. Before diving into the A* algorithm, let's introduce a special type of function called as heuristic function. This function gives an estimate of the distance from the present node to the goal node. For A* algorithm to compute an optimal path, the heuristic function should be admissible, meaning that the function should output an underestimate of the actual distance between present node and goal node.

Introducing the heuristic function in the A* algorithm would result in a more stringent criterion such that it biases the search direction towards the goal node. Furthermore, the accuracy of the heuristic function is directly proportional to the efficiency of the algorithm. So, we change the condition for node expansion and g value update as follows:

$$g_i + c_{ij} < g_{tau} \tag{8}$$

$$g_i + c_{ij} + h_j < g_{tau} \tag{9}$$

Few commonly used heuristic functions are as follows:

- Euclidean Distance:$h_i = \|\mathbf{x}_\tau - \mathbf{x}_i\|_2$
- Manhattan Distance:$h_i = \|\mathbf{x}_\tau - \mathbf{x}_i\|_1 := \sum_k |x_{\tau,k} - x_{i,k}|$
- Diagonal Distance:$h_i = \|\mathbf{x}_\tau - \mathbf{x}_i\|_\infty := \max_k |x_{\tau,k} - x_{i,k}|$

The eqn 8 depicts the condition for a typical label correcting algorithm and eqn 9 depicts the condition for a* algorithm. Thus, the f value is defined as the sum of g value and the heuristic value as shown in eqn 10. In an $epsilon$ consistent heuristic, we scale the heuristic by a parameter $epsilon$ which trusts more on the heuristic value as shown in the eqn 11. Let's discuss few of the salient features of this algorithm:

$$g_i + h_i = f_i \tag{10}$$

$$g_i + \epsilon h_i = f_i \tag{11}$$

1) Optimality: Since this algorithm uses ϵ-consistent heuristic, it guarantees to return an ϵ-sub-optimal path with cost dist(s,τ)$\leq g_\tau \leq \epsilon$ dist(s,τ) for $\epsilon \geq 1$. In this literature the algorithm is iterated on various ϵ, and the value of 1 is chosen for most of the environments.

2) Completeness: Sine the environment is discretized into a finite set of points and all the edge costs are non-negative, the weighted A* algorithm is guaranteed to return a path if it exists and hence it is complete.

3) Memory: Let us assume that each coordinate is discretized to to n points, then the memory complexity of this algorithm is $\mathcal{O}(n^3)$.

4) Time efficiency: The implementation can be seen in two steps. In the code priority queue was implemented using pqdict(). If the total number of points in the space where $|V|$ then the time complexity of this part ids $\mathcal{O}(|V|^2)$.

B. Implementation

The algorithm 1 summarises the step by step implementation of A* path planning algorithm for our project:

To implement this algorithm, we need to store nodes in two lists namely OPEN and CLOSED. The OPEN list is a priority queue. A priority queue is an abstract data-type similar to a regular queue or stack data structure in which each element additionally has a "priority" associated with it. We use pqdict() method in python where the lowest value has the highest priority, which is exactly what we need for implementing A* algorithm. We initialize the start state in OPEN list and set it's value to be zero and g value of all other nodes is set to be infinity. The CLOSED list is initialized as an empty list.

Algorithm 1 Weighted A* Algorithm

```
OPEN ← {s}, CLOSED ← {}, ε ≥ 1
g_s = 0, g_i = ∞ ∀i ∈ 𝕆
while τ ∉ CLOSED do
    Remove i with smallest f_i = g_i + ε h_i
    Insert i into CLOSED
    for j ∈ Children(i) and j ∉ CLOSED do
        if g_i + c_ij < g_j and g_j > 0 then
            g_j ← g_i + c_ij
            Parent(j) ← i
            if j ∈ OPEN then
                Update priority of j
            else
                OPEN ← OPEN ⋃{j}
```

Under the while loop, we remove the node with smallest f value and insert it into the CLOSED list. Then we update the g value of its children nodes, and store the information of parent node. These promising nodes are added into the OPEN list and pqdict() method takes care of updating its priority. In order to recover the path with the lowest cost, we back track the parent node from goal until we reach the start. For this particular project, the robotplanner function computes only the immediate next step of the robot.

IV. Results

The A* path planning algorithm was implemented in 9 different environments. The logic behind the target's movement is unknown to us, and we successufly caught the target in 9 different environments. Fig 4-12 depicts the visualization of the path computed by the weighted A* algorithm with L1 Norm.

itr	Map	ϵ	Heuristic	No of Moves
1	Map 0	1	L2 Norm	4
2	Map 1	5	L2 Norm	1279
3	Map 2	1	L2 Norm	12
4	Map 3	1	L2 Norm	223
5	Map 3b	5	L2 Norm	430
6	Map 3c	5	L2 Norm	762
7	Map 4	1	L2 Norm	9
8	Map 5	1	L2 Norm	74
9	Map 6	1	L2 Norm	40
10	Map 0	1	L1 Norm	4
11	Map 1	5	L1 Norm	1279
12	Map 2	1	L1 Norm	13
13	Map 3	1	L1 Norm	300
14	Map 3b	5	L1 Norm	530
15	Map 3c	5	L1 Norm	907
16	Map 4	1	L1 Norm	9
17	Map 5	1	L1 Norm	113
18	Map 6	1	L1 Norm	40
19	Map 0	1	INF Norm	4
20	Map 1	5	INF Norm	1279
21	Map 2	1	INF Norm	12
22	Map 3	1	INF Norm	223
23	Map 3b	5	INF Norm	430
24	Map 3c	5	INF Norm	762
25	Map 4	1	INF Norm	9
26	Map 5	1	INF Norm	74
27	Map 6	1	INF Norm	40

The table summarises the different iterations implemented by altering the values of $epsilon$ and the choice of heuristic function. Three different types of heuristic function: Euclidean Distance (L2 Norm), Manhattan Distance (L1 Norm) and Diagonal Distance (inf Norm). From the table, we can observe that the inf Norm and L2 norm performs better than the L1 norm. The inf norms and L2 norm results in lesser number of moves, thus computing a more optimal path than other norms. Since, our robot has the ability to move in 8 directions, the L2 norm and inf norm gives a better estimate and L1 norm is more suited for control space in only 4 directions (no diagonal movement). The choice of norms did not make any difference for smaller maps since the size of state space is smaller compared to bigger maps

Thus, A*'s ability to vary its behavior based on the heuristic and cost functions can be very useful in a path planning. The trade-off between speed and accuracy can be exploited to make path generation faster whenever needed. For most games, you don't really need the best path between two points. The choice between speed and accuracy does not have to be global. You can choose some things dynamically based on the importance of having accuracy in some region of the map. For example, it may be more important to choose a good path near the current location, on the assumption that we might end up recalculating the path or changing direction at some point. For a robotic arm, the accuracy while grasping or interacting with other agents is more important. During such phases, we can alter the accuracy.

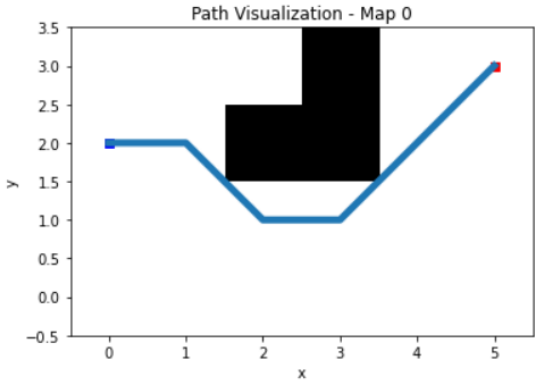

Fig. 4. Map 0: A* Visualization, ϵ = 1

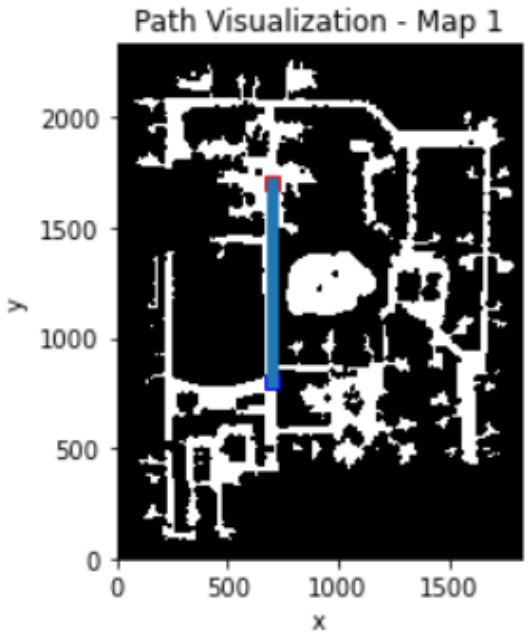

Fig. 5. Map 1: A* Visualization, ϵ = 5

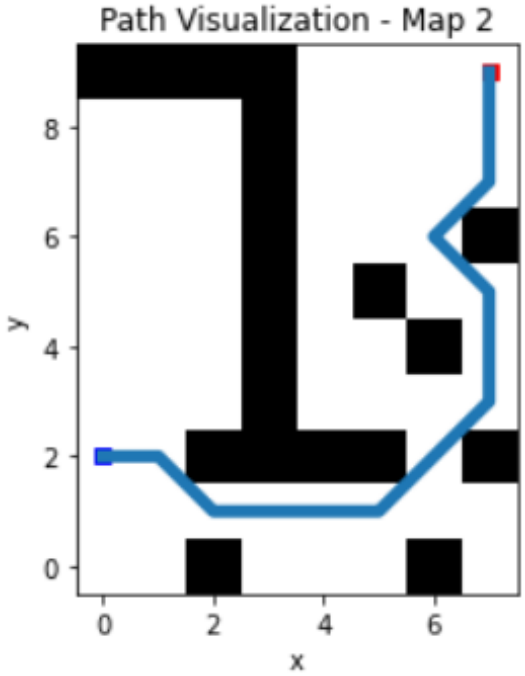

Fig. 6. Map 2: A* Visualization, $\epsilon = 1$

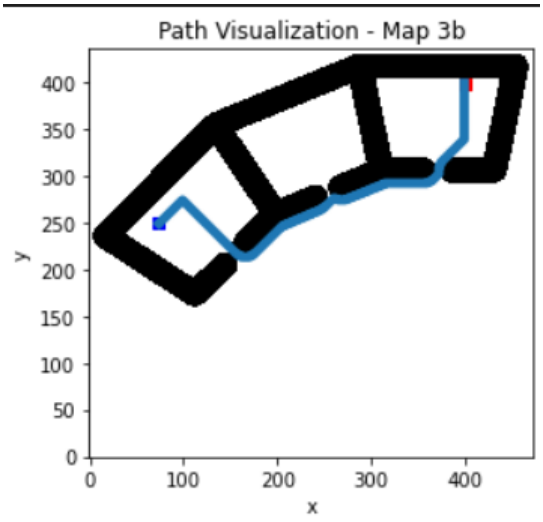

Fig. 8. Map 3b: A* Visualization, $\epsilon = 5$

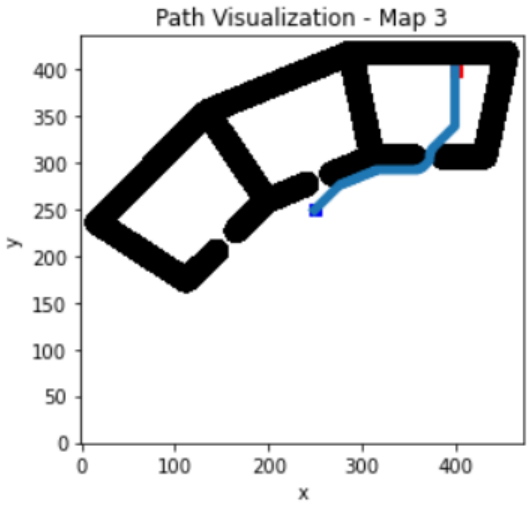

Fig. 7. Map 3: A* Visualization, $\epsilon = 1$

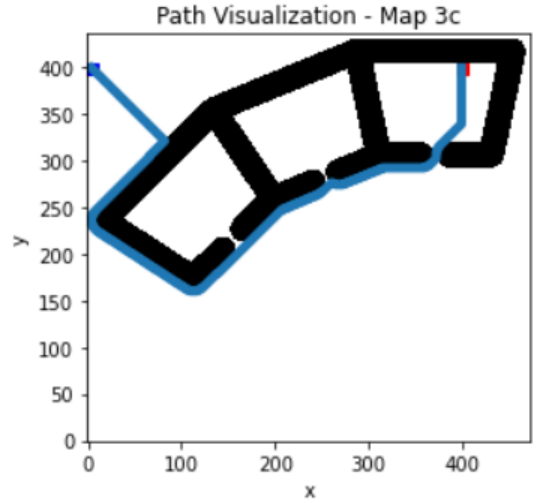

Fig. 9. Map 3c: A* Visualization, $\epsilon = 5$

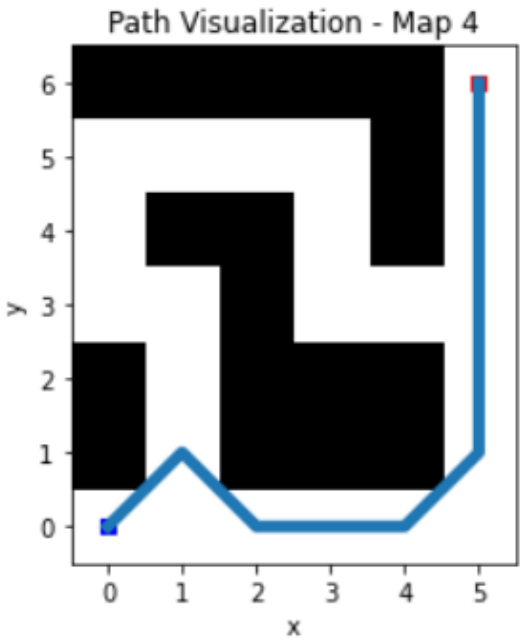

Fig. 10. Map 4: A* Visualization, $\epsilon = 1$

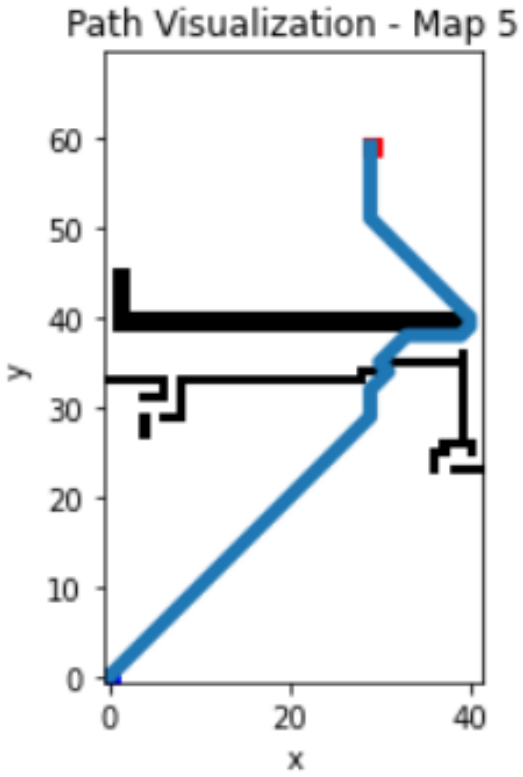

Fig. 11. Map 5: A* Visualization, $\epsilon = 1$

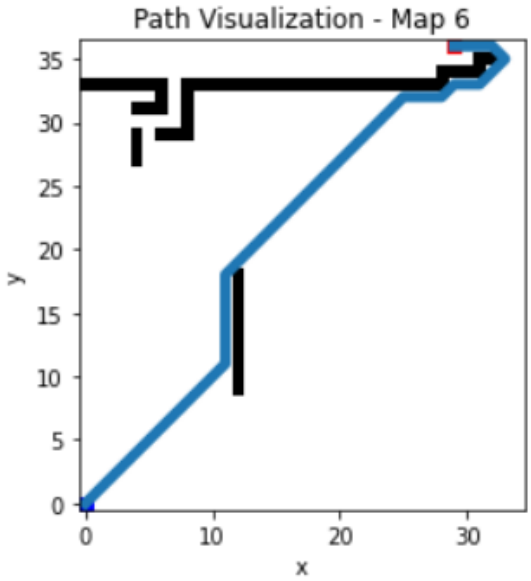

Fig. 12. Map 6: A* Visualization, $\epsilon = 1$