Yijun Mao NetID: ym134

Final Report

Table of Contents

1. Overview

In final A, it is required to design a motion planner for a differential drive robot. The overall approach I applied to solve this problem can be divided into the following steps:

- Represent the state space of the robot in a 3-D graph
- Define the nodes, edges, costs, and actions of the graph
- Implement appropriate graph search searching algorithms to find the path from starting state to the final state
- Evaluate and compare the results of different search algorithms with unit testing
- Analysis of problems in the applied method and algorithm and overlook of possible improvements

2. Dynamics and strategy

The general framework for motion planning can be summarized as the following steps: Continuous representation -> discretization -> graph searching. For this specific problem, a differential robot with two wheels travelling on a plane, it has 3 DOFs, two represent its coordinates on the plane and another one represents its orientation, these can be denoted as x, y and Θ. The state space of the robot could thus be defined as q = (x, y, Θ). Considering drive the robot from one pose to another, we must apply some controls or actions on it. Here the two wheel speeds are selected as controls, which can be denoted as $u(v_l, v_r)$. The dynamics of the robot are defined as follows:

$$\dot{q} = f(q,u) = \begin{bmatrix} \frac{1}{2}(v_l + v_r)\cos\theta \\ \frac{1}{2}(v_l + v_r)\sin\theta \\ (v_r - v_l)/b \end{bmatrix}$$

The continuous representation of this problem is the continuous state space transition of the robot travelling from the start state to the final state. Notice here the rotation is also included as one parameter in the state space since the orientation of the robot is naturally also part of its state and plays a role in determining its travelling path. Using this representation, the motion planning problem could be transferred into a graph searching problem in a 3-D graph with x, y, and Θ as its dimensions. To make a graph search practical, discretization should be performed to specify discrete vertices in the overall searching space. Here the searching space is all the possible states covered by the robot travelling from the start to the end state. After defining vertices, edges of the graph could be built associated with appropriate costs. Then an appropriate graph searching algorithm should be applied to find a reasonable path represented by corresponding vertices. To make the robot actually follow the path found by the algorithm, appropriate actions or controls should be computed and act upon the robot. Comparing the planned path with the executed path, results of different algorithms as well the correctness, reasonability of controls, costs are evaluated.

For discretization, the default setting is twenty vertices in each dimension. This parameter can naturally be modified by the user later to explore its effect on the searching performance. For graph searching, two algorithms will be implemented, Dijkstra and A star search. With this big picture, the remaining core problem is how to define the vertices and edges, as well as how to associate appropriate edges costs and actions.

To solve this problem, search is considered only to be performed along one of the three dimensions. This may lead to a non-optimal searching solution, but can simplify the procedure to define costs and controls. And when carefully selecting the granularity and appropriately define costs and actions, good results should still be achieved. When applying this policy, each vertex v in the graph (except those on the searching space border) has six neighboring vertices (I take all the six neighbors into consideration instead of throw any of them when building the graph). Three with a step-wise increment (here the resolution along the corresponding axis) in a single dimension when keeping the other two dimensions the same as vertex v. Three with a step-wise decrement while keeping the other two dimensions the same as vertex v. Assume vertex v has index (i, j, k) and coordinate (x, y, Θ). Its neighboring nodes have indices and coordinates as following:

Neighbor name	**index**	**coordinates**
Right neighbor	(i+1, j, k)	$(x+\Delta x, y, \theta)$
Left neighbor	(i-1, j, k)	$(x-\Delta x, y, \theta)$
Forward neighbor	(i, j+1, k)	$(x, y+\Delta y, \theta)$
Backward neighbor	(i, j-1, k)	$(x, y-\Delta y, \theta)$
Upward neighbor	(i, j, k+1)	$(x, y, \theta+\Delta\theta)$
Downward neighbor	(i, j, k-1)	$(x, y, \theta-\Delta\theta)$

This indicates for each vertex (except those on the searching border), six edges with its corresponding neighbors should be built. To associate appropriate costs and actions on these edges, the current state of the observed vertex v is divided into five possible cases to prepare

for the later discussion.

Case number	orientation	Reachable neighbors with constant control
1	Along positive x axis	Right, left, upwards, downwards
2	Along positive y axis	Forwards, backwards, upwards, downwards
3	Along negative x axis	Right, left, upwards, downwards
4	Along negative y axis	Forwards, backwards, upwards, downwards
5	Not along x nor along y axes	All six neighbors

The first column of this table lists the case number while the second describes the facing direction of the robot under current state. The third column lists all the neighbor states the robot can move to under the constant control. The controls we can act upon the robot are the left and right wheel speeds. Considering to move to the maximal six reachable neighbors, whatever the wheel speed is, only two possible actions should be generated regardless of their directions: move straight or rotation in place. To move straight, left and right wheel should have the same speed in the same direction. To rotate in place, two wheels should have speeds with the same value but act in opposite directions. Since the maximal allowed wheel speed is known as ulim, the controls as well as the required time to move to the neighbor can be computed. Just take the case number 1 as an example. In this case, the vertex face along the positive x axis, to move to its right neighbor, the control should be

$$u(v_l, v_r) = (ulim, ulim)$$

The time spent can be computed as

$$t = \frac{\Delta x}{ulim}$$

To move to its upwards neighbor, the control should be

$$u(v_l, v_r) = (ulim, ulim)$$

The time spent can be computes as

$$t = \frac{\Delta \theta}{ulim}$$

Here $\Delta\theta$ is the relative angle between the vertex and its neighbor upwards and always takes the positive value. Computations of controls and time spent for other cases could be done in a similar way and have been summarized in codes of the program. To choose an appropriate edge cost, I consider using the time spent to travel from the current state to its neighbor state as an appropriate candidate. This is reasonable since the travelling time physically represents the time cost to move from the current state to a neighboring state and could be taken as edge cost to search in a graph. Now the remaining question is what the cost is for an edge consisting of the current vertex and its unreachable neighbor. I assign infinity (in Python the maximum float) to this edge. This is a reasonable choice since an edge with infinity cost would not be searched and thus represents the neighbor is unreachable.

3. Theoretical behavior and implementation

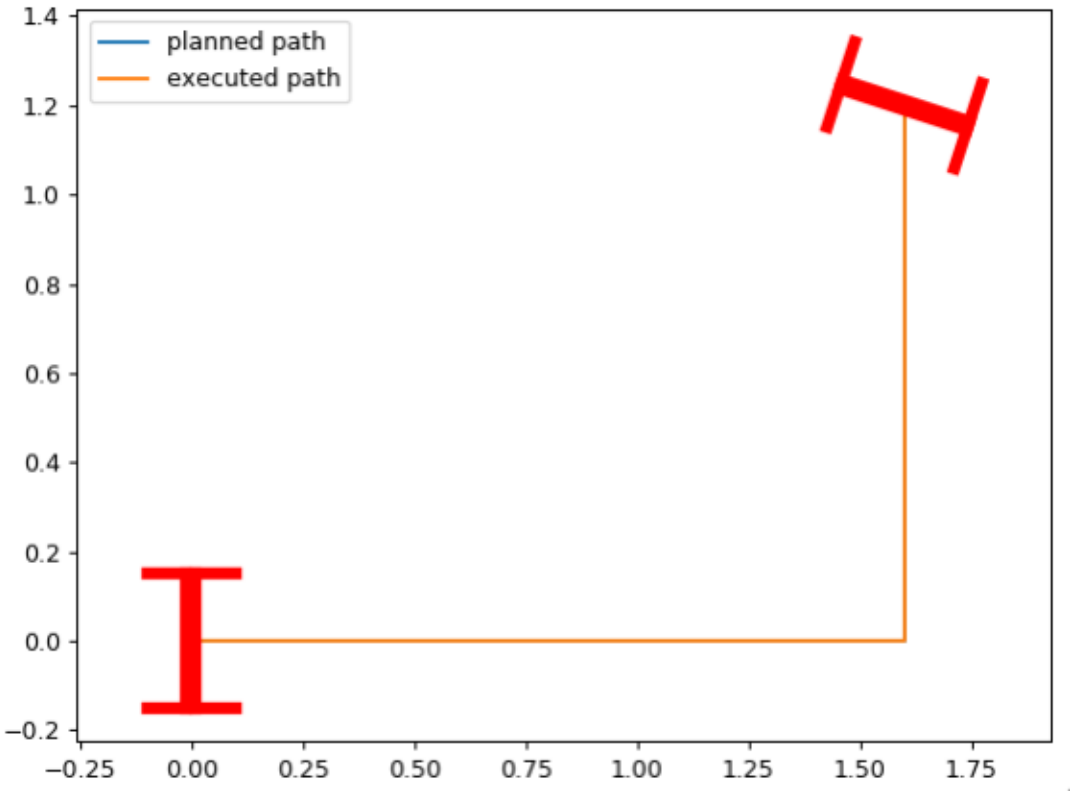
The inputs of my planner are states of vertices (x, y, Θ), edge costs, edge actions, and robot parameters (wheel base width, speed limit). Outputs of my planner are planned path (represented as a sequence of vertices along the path), executed path (the actual path travelled by the robot under specified actions), and the sum of all visited edge costs.

For the graph searching part, apart from the given Dijkstra algorithm, A star searching is also implemented. The main difference between Dijkstra and A star is that Dijkstra explores all the possible paths and give no priority to specific vertices while A star tries to look for a better path by using heuristic function. This function will give priority to certain vertices by adding higher weights to them. A star combines the advantages of best first search and Dijkstra to achieve a balance in performance, completeness, and optimality. I choose the Euclidean distance here as the heuristic function since it reflects the searching cost in a physically more reasonable way. And an overall better result is observed for A star search. More precisely in formulas,

$$f(N) = g(N) + h(N)$$

$$h(N) = \sqrt{(x_N - x_g)^2 + (y_N - y_g)^2 + (\theta_N - \theta_g)^2}$$

here $f(N)$ is the total cost, $h(N)$ is an estimate of the cost from a node to a goal, and $g(N)$ is the cost from the start to a node.



Dijkstra searching result with start state (0,0,0) and end state (1.8,1.2,$\pi/2$), grid size 20

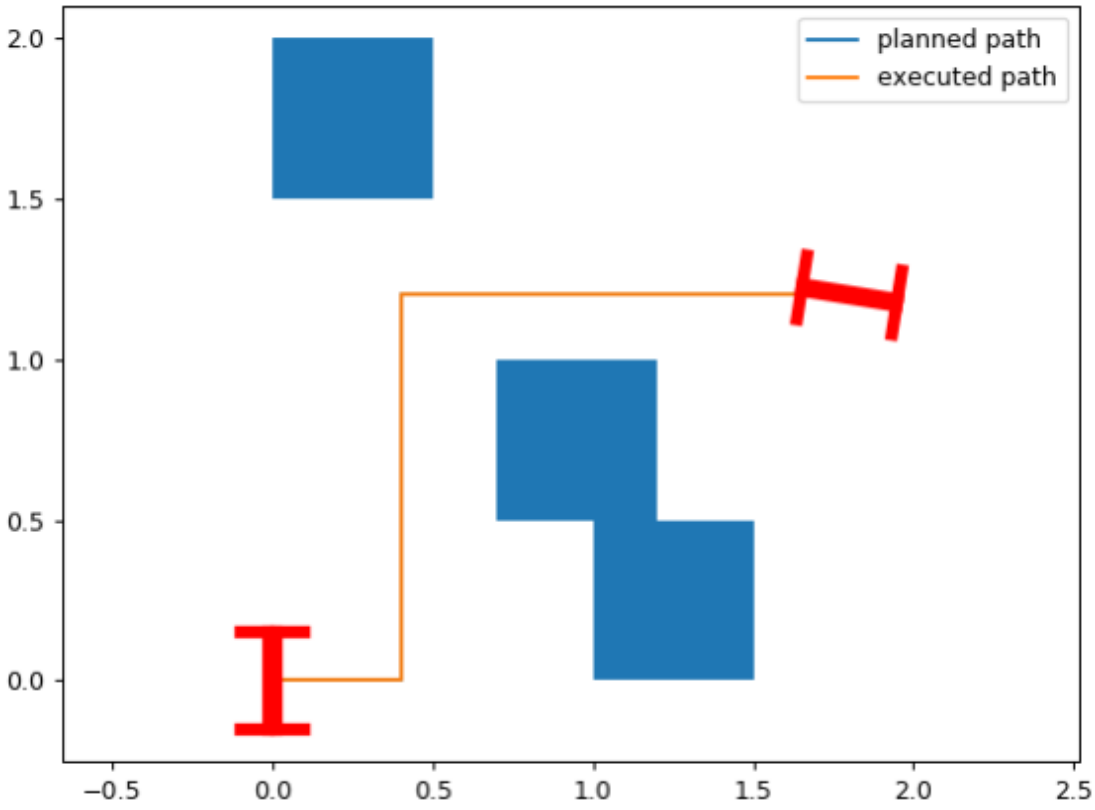
Yijun Mao NetID: ym134

Grid size	Number of iterations	cost
20	898	4.75
30	failed	
40	7131	4.73

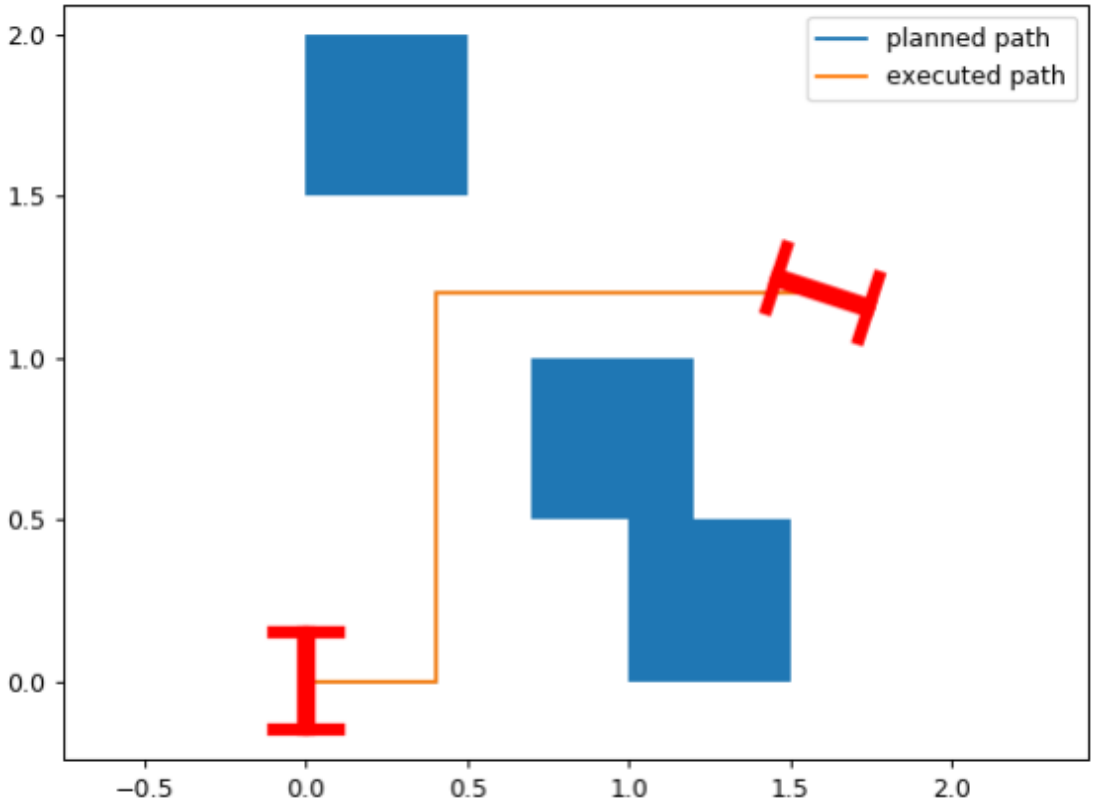
A star searching result with start state (0,0,0) and end state (2,2,$\pi/2$)

Grid size	Number of iterations	cost
20	508	3.46
30	failed	
40	4904	3.68

A star searching result with start state (0,0,0) and end state (1.8,1.2,$\pi/2$)



A star searching result with start state (0,0,0) and end state (1.8,1.2,$\pi/2$), grid size 40



A star searching result with start state (0,0,0) and end state (1.8,1.2,$\pi/2$), grid size 20

Whatever searching algorithm is applied (Dijkstra or A star), increasing the grid size may yield a little better performance but takes definitely more iterations. As an example, we can see from the figures and tables above that for A star searching algorithm, increasing grid size from 20 to 40 does decrease the error between the planned and executed final state. But takes almost as much as 9 times iterations. Decrease the distance between start and goal (keep the start state unvaried and change the end state from (2,2,$\pi/2$) to (1.8, 1.2,$\pi/2$)) would definitely decrease the required number of iterations. The effect of choice of actions on the performance is not studied thoroughly for this problem due to the time limit. But from a theoretical point of view, it should have influence on the performance since it will change the executed path. In addition, changing actions would also change edge costs, which would then affect the searching performance.

Algorithm	**complete**	**Optimal**	**probabilistic**
Dijkstra	yes	yes	no
A star	yes	no	no

4. Unit testing

I've tested the program with no blocks in the field, a single block in the field, two blocks in the field, and finally three blocks in the field. Switching between different test modes can be achieved by modifying the codes commented with "set obstacles" at the beginning of "cost_action" function as well those close to the end within the "build graph" module.

5. Enhancement and result discussion

I've implemented two things to enhance the performance, the block avoidance functionality and the A star algorithm for graph searching. By implementing the former one, I add three obstacle blocks into the field. A star algorithm implemented here uses an extra heuristic function to compute cost. Observing the obtained results show that with A start an overall better performance is achieved. Even for some cases Dijkstra fail, A star succeeds. However, despite the implemented functionalities and enhancements, there still exist many points to be improved, one of which is the way to build the graph. Currently all the nodes in the searching range are used in building the graph, to disable choosing some nodes, an infinite cost is applied here. A better solution may be to throw away some nodes in the process of building the graph so that assignment of costs is not necessary and a faster search due to fewer nodes could be expected.