AI 3603 Artificial Intelligence: Principles and Techniques

By: Ji Ai (521030910050)

HW#: 1 Search Algorithm

October 4, 2023

I. INTRODUCTION

A. Purpose

In this homework we imagine a service robot finding its way in a room, and simulates the original system without actually doing what the real system does. This homework consists of 3 tasks. All the tasks share the same 120m $\times$ 120m world map with obstacles, start position and goal position.

In task 1, the robot can move forward, backward, left, and right. we use a basic A* algorithm.

In task 2, the robot can move in 8 directions. Aside from the original 4 horizontal and vertical directions, it is allowed to go to a node at $\frac{\pi}{4}$, $\frac{3\pi}{4}$, $\frac{5\pi}{4}$, $\frac{7\pi}{4}$ in one step. Moreover, we try to reduce unnecessary turns and avoid being too close to the obstacles.

In task 3, we plan a route with and smoothness for the robot. This makes sense because blunt turns in reality have greater cost and difficulty. In my implementation, I just smoothed the route in Task 2 and didn't choose to build a car model.

B. Environment and Structure

My code works in both Python 3.9.7 and Python 3.11.5 in my machine.

Apart from several modules in the provided file, I used another module, **queue**, from the Python standard library.

It should be noted that the module **queue** is written as "Queue" in python2 and "queue" in python3.

```
.
├── Task_1.py
├── Task_2.py
├── Task_3.py
└── HW1_report.pdf
```

FIG. 1: Structure of my submitted folder

According to the code blocks provided in python file, to run the code, e.g. Task_X.py, we should ensure the following structure.

```
.
├── 3-map
│   └── map.npy
└── Task_X.py
```

FIG. 2: Structure for code running

II. SOLUTIONS

A. Ideas

This part will present my general ideas for solving problems. What the A_star function return i.e. the path we planned for the robot, is a N*2 array. What we need do is fill the array with nodes, i.e. 1*2 array.

- **Task 1**

First we take a look at two simpler cases, BFS and UCS.

Algorithm 1 Breadth First Search

```
Initialize frontier                                          ▷ frontier is a queue
add start_pos to frontier
Initialize explored                                          ▷ explored is a set
add start_pos to explored
Initialize came_from ▷ came_from is of the same size of the world map. Each node contains a vector of direction
while frontier not empty do
    frontier_node ← front.get()
    for temp_node in frontier_node's neighbors do
        if temp_node is traversable then
            if temp_node is goal_pos then
                construct the path depending on came_from
                return path
            else if temp_node not in explored then
                add temp_node to frontier
                add temp_node to explored
                record current vector in came_from
```

BFS can be used to find shortest path in graphs without weights. The pseudocode above is adapted to a series of finding-path problems including the problem in this homework.

Algorithm 2 Uniform Cost Search

```
Initialize frontier                                          ▷ frontier is a priority queue
add (0,start_pos) to frontier
Initialize explored                                          ▷ explored is a set
add start_pos to explored
Initialize came_from ▷ came_from is of the same size of the world map. Each node contains a vector of direction
while frontier not empty do
    frontier_node ← front.get()          ▷ Now front.get() is prior, i.e. has smallest cost in frontier
    for temp_node in frontier_node's neighbors do
        if temp_node is traversable then
            new_cost=frontier_node_cost+movement_cost
            if temp_node is goal_pos then
                construct the path depending on came_from
                return path
            else if temp_node not in explored or new_cost < temp_node_cost then
                temp_node_cost ← new_cost
                add (new_cost,temp_node) to frontier
                add temp_node to explored
                record current vector in came_from
```

Algorithm 3 Uniform Cost Search

```
Initialize frontier                                          ▷ frontier is a priority queue
add (0,start_pos) to frontier
Initialize explored                                          ▷ explored is a set
add start_pos to explored
Intialize came_from ▷ came_from is of the same size of the world map. Each node contains a vector of direction
while frontier not empty do
    frontier_node ← front.get()          ▷ Now front.get() is prior, i.e. has smallest cost in frontier
    for temp_node in frontier_node's neighbors do
        if temp_node is traversable then
            new_cost=frontier_node_cost+movement_cost
            if temp_node is goal_pos then
                construct the path depending on came_from
                return path
            else if temp_node not in explored or new_cost < temp_node_cost then
                temp_node_cost ← new_cost
                priority=new_cost+heuristic_estimate  ▷ Evaluation Cost= Cost function + Heuristic function
                add (priority,temp_node) to frontier
                add temp_node to explored
                record current vector in came_from
```

Dijkstra algorithm is similar to UCS. In UCS we focus on a goal vertex while Dijkstra algorithm finds the shortest distance between the start vertex and all vertices in the graph.

Comparing Algorithm 1 and Algorithm 2, in while loop UCS select the node with smallest costs. In task 1 we will not see **new_cost** < **temp_node_cost**, thus the priority we designed does not change the searching sequence. Therefore the searching process of BFS and UCS here will be the same.

But if we take a look at the formal definition, UCS is able to change the order of state searches, giving priority to searching for states that are most likely to be optimal. Then we add the heuristic function to the **cost**, the evaluation in **if** sentence consists of two parts: the known cost from the **start_pos** and the cost related to the **goal_pos**.

Now my strategy reach A* algorithm.

Their further analysis:

- **b**: branching factor, number of children at each node.
- **s**: depth which the shallowest solution correspond to.
- **C***: the total cost in the solution.
- ϵ: the least arcs cost, in this homework it's the smallest movement cost.

	Time Ccomplexity	Space Complexity
BFS	$O(b^s)$	$O(b^s)$
UCS	$O(b^{\frac{C*}{\epsilon}})$	$O(b^{\frac{C*}{\epsilon}})$
A*	Depends on the heuristic function	Depends on the heuristic function

TABLE I: Time and Space Comlexity Comparison

	BFS	UCS	A*
Completeness	Complete	Complete	Complete
Optimality	Optimal only if each movement cost is the same	Optimal	Optimal

TABLE II: Completeness and Optimality Comparison

- **Task 2**

1. **frontier_node** will expand in 8 directions.
2. It should be considered that diagonal length is longer than the side length. So I need change the **movement_cost**.
3. Change the definition of heuristic function.

After add 4 directions, I need to pay attention to the distance of **path** from the wall and the number of turns.

My approach to avoiding obstacles is to treat the area around the original obstacle as not traversable.

To meet the requirement of less turns, I add the cost of steering to **new_cost**. The specific parameters are discussed in the next section.

- **Task 3**

I chose to use the originally selected points as the control points in the Bezier curve and split them into many small sections to build the curve separately. In this way, the route is actually the same as the route of $Task\_2$, only the trajectory is smoother. This is the laziest way in my view.

B. Results

This section consists of screenshots taken during the test.

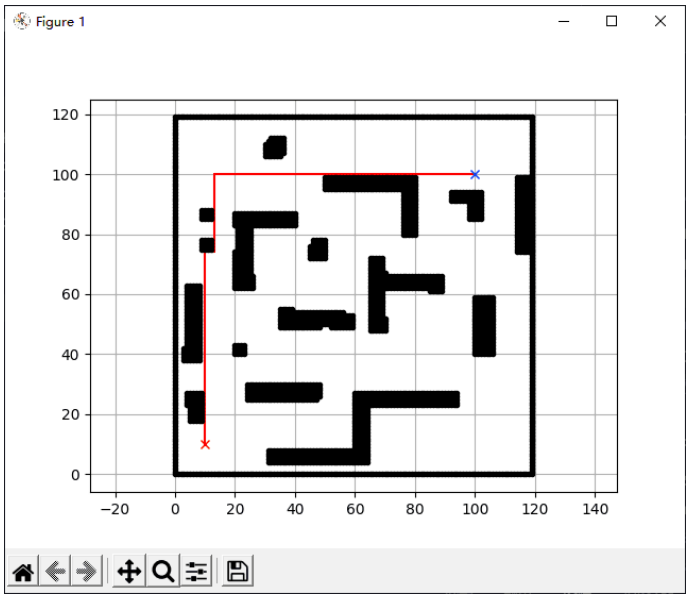


FIG. 3: Result of task 1

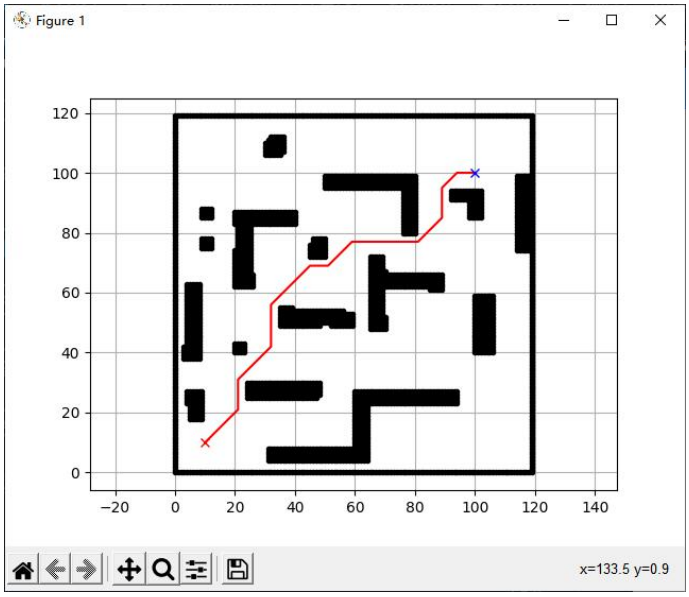


FIG. 4: Result of task 2

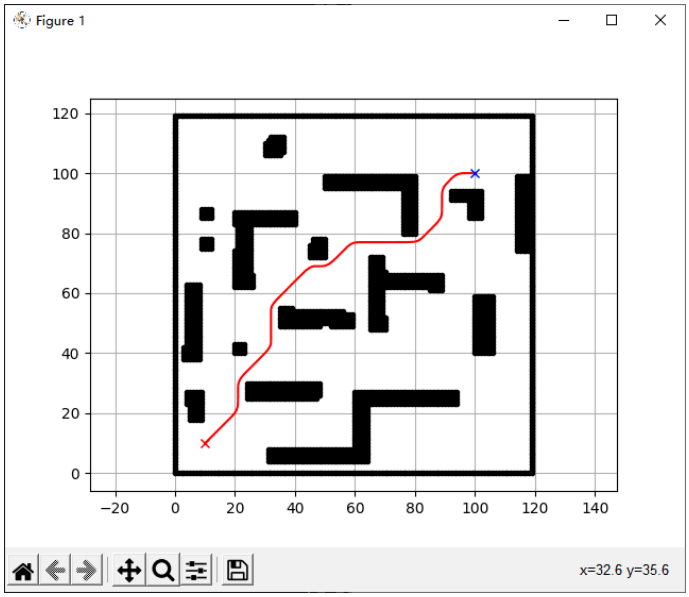


FIG. 5: Result of task 3

III. ANALYSIS

This part will analyze my code and algorithm.

```
    # core process
    direction_x = np.array([-1, 0, 1, 0])
    direction_y = np.array([0, 1, 0, -1])
    while (not frontier.empty()):
        frontier_node = frontier.get()[1]

        if (frontier_node[0] == goal_pos[0] and frontier_node[1] == goal_pos[1]):
            expand_node = goal_pos
            path = [np.array(goal_pos)]
            while (expand_node[0] != start_pos[0]) or (expand_node[1] != start_pos[1]):
                came_from_i = came_from[expand_node[0]][expand_node[1]]
                next_node_x = expand_node[0] - direction_x[came_from_i]
                next_node_y = expand_node[1] - direction_y[came_from_i]
                next_node = [next_node_x, next_node_y]
                path = np.vstack((path, next_node))
                expand_node = next_node
            return path

        for i in range(0, 4):
            temp_node = [frontier_node[0] + direction_x[i],
                         frontier_node[1] + direction_y[i]]

            if (not world_map[temp_node[0]][temp_node[1]]):

                temp_cost = g[frontier_node[0]][frontier_node[1]]+1

                if (g[temp_node[0]][temp_node[1]] == 0) or (temp_cost < g[temp_node[0]][temp_node[1]]):

                    g[temp_node[0]][temp_node[1]] = temp_cost
                    priority = temp_cost + heuristic_estimate(temp_node[0], temp_node[1])
                    frontier.put((priority, temp_node))
                    came_from[temp_node[0]][temp_node[1]] = i
```

Above is a core code block in **Task_1.py**. It corresponds to the **Algorithm 3**. From line 7 to line 17 is the path construction part. The rest is searching part.

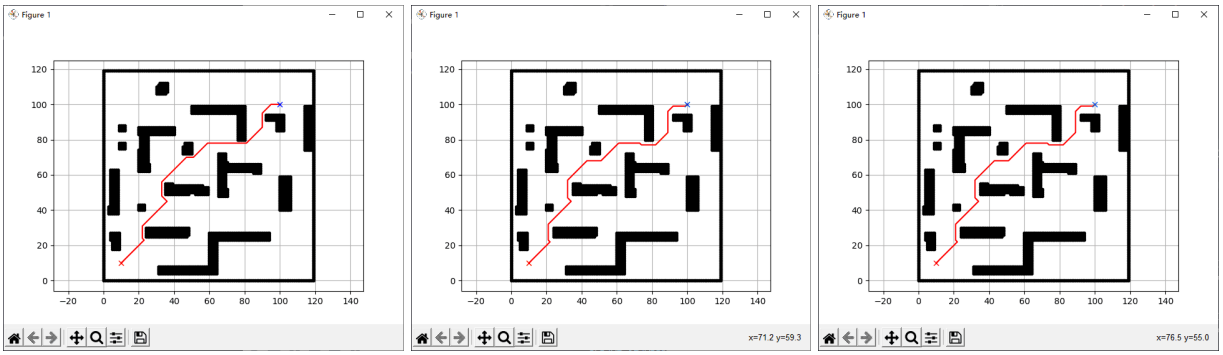
Diagonal movements are very simple. The changes on the coordinate map are actually (-1,1), (1,1), (1,-1), (-1, -1) four vectors. I added these vectors in **direction_x** and **direction_y**.

In accordance with my ideas in the second section, the next question is how to keep its distance from obstacles.

```
    direction_x = np.array(
        [-1, 0, 0, 1,
         1, 1, -1, -1,
         -2, 0, 2, 0,
         1, 1, -1, -1,
         -2, -2, -2, 2,
         2, -2, 2, -2])
    direction_y = np.array(
        [0, -1, 1, 0,
         1, -1, -1, 1,
         0, 2, 0, -2,
         -2, 2, -2, 2,
         -1, -1, 1, 1,
         -2, 2, 2, -2])
    world_map_copy = np.zeros_like(world_map)

    for i in range(120):
        for j in range(120):
            if world_map[i][j] == 1:
                for k in range(0, 20):
                    if i+direction_x[k] <= 119 and i+direction_x[k] >= 0 and j+direction_y[k] <= 119 and j+
                        direction_y[k] >= 0:
                        world_map_copy[i+direction_x[k]
                                       ][j+direction_y[k]] = 1
```

Above is a core code block in **Task_2.py**. I tried adding different ranges of obstacles, range(0,4), range(0,8), range(0,12), range(0,20), range(0,24) to avoid danger.



range(0,8), too close range(0,20), ideal range(0,24), also ideal

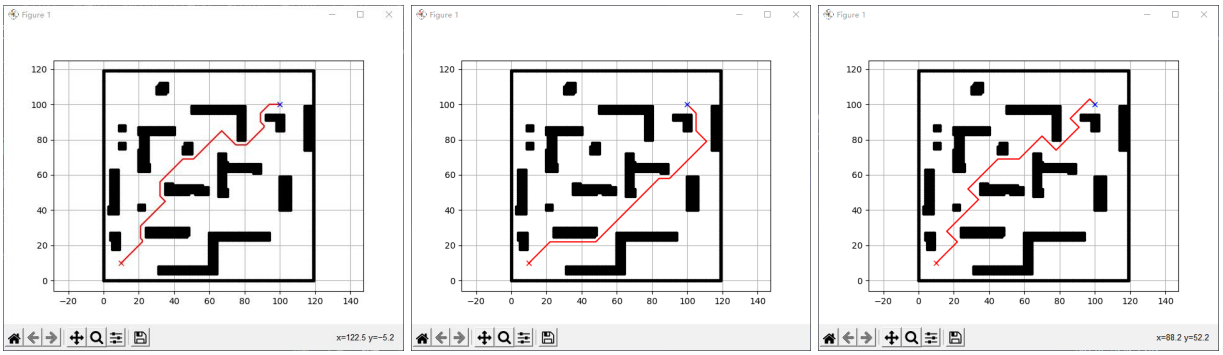
FIG. 6: Comparisons of different ranges of added obstacles

As can be seen from FIG. 6, the effect is better when the range is range(0,20) and range(0,24). Compared to range(0,20), range(0,24) adds 4 corners of a 2m × 2m square. So I ended up using range(0,20).

Still there are some unnecessary turns in the path.

Here is my failed idea. I tried to add a big **if** sentence to reconstruct the path when detecting possible turns. The key conditon is that after adding the vectors of the former turn and the latter turn, in x axis and y axis, one is zero, the other is same as the direction of straight path. I planned four variables to record and determine but this idea doesn't work.

Then I see "Adding the cost of steering" in **1-HW1_Assignment.pdf** as a sort of clue. Choose to add **steer_cost** to **temp_node_cost**.



steer_cost=0.1, bad steer_cost=1, bad steer_cost=1.5, bad

FIG. 7: Comparisons of different steer_cost

Finally I choose to set the **steer_cost** as **0.6**. This acting seems to be a loophole, because if the robot is in a different map, this extra cost could force the expanding process to stay moving straight in early stages. When the **frontier** reaches the **goal_pos**, some turns in the path are still unnecessary, or maybe the path is longer.

Finally comes task 3.

I adopted two functions from this blog.

```
def B_nx(n, i, x):
    if i > n:
        return 0
    elif i == 1:
        return n*x**((1-x)**(n-1))
    elif i == 0:
        return (1-x)**n
    return B_nx(n-1, i, x)*(1-x)+B_nx(n-1, i-1, x)*x
def get_value(p, t): # t range [0,1]
    sumx = 0.
    sumy = 0.
    length = len(p)-1
    for i in range(0, len(p)):
        sumx += (B_nx(length, i, t) * p[i][0])
        sumy += (B_nx(length, i, t) * p[i][1])
    return sumx, sumy
```

Take the nodes I found in **Task_2.py** as the control points of the Bezier curve, then get many points, and then connect them to get a smooth curve. Unfortunately, if I use global control points, the running time is too long.

```
            path = []
            one_axis = np.linspace(0, 1, 11)
            while (expand_node[0] != start_pos[0]) or (expand_node[1] != start_pos[1]):
                expand_node_i = came_from[expand_node[0]][expand_node[1]]
                next_node_x = expand_node[0] - direction_x[expand_node_i]
                next_node_y = expand_node[1] - direction_y[expand_node_i]
                next_node = [next_node_x, next_node_y]
                # update square_path
                square_path.append(next_node)
                # update expand_node
                expand_node = next_node

                if (len(square_path) >= 9):
                    for i in range(0, 11):
                        smooth_node_x, smooth_node_y = get_value(
                            square_path, one_axis[i])
                        path.append([smooth_node_x, smooth_node_y])
                    square_path = []

            # after the while loop
            for i in range(0, 10):
                smooth_node_x, smooth_node_y = get_value(
                    square_path, one_axis[i])
                path.append([smooth_node_x, smooth_node_y])

            return path
```

So I make a segment of the curve every 9 nodes. The constructing logic is shown in the code block above.

Acutually my Bezier curve is also a polygonal line of many points, and their turns are so small that they can be ignored. My **square_path** has 120 nodes in total, while **path**, i.e. the final curve, contains 154 nodes.

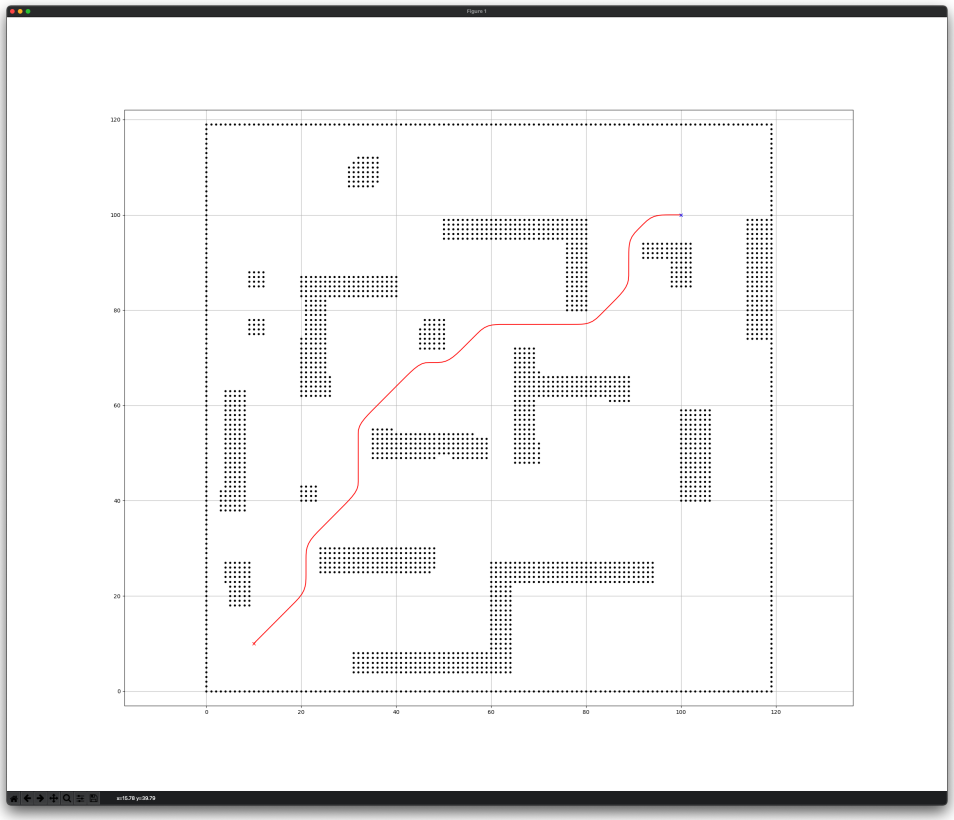


FIG. 8: Result of task 3, enlarged image

This curve is still smooth in my 27 inches display, that's amazing.

IV. DISCUSSION & CONCLUSION

The goal of this homework was to a path-planning framework for a service robot in a room using A* algorithm. Through this assignment, I reviewed some search algorithms such as BFS, UCS.

In this homework the robot does not actually have the function of the real system. My model only record the resulting state change in the simulator. We watched a video about DARPA Urban Challenge during the class. That challenge tested the ability of driverless cars to cope with complex situations. In reality although we have satellite maps with building and road information, obstacles such as vehicles and pedestrians can only be seen when approached. If a route doesn't work, we will need to reroute it.

Hopefully my homework could be of service for model verification in the early stages of design.