AI 3603 Artificial Intelligence: Principles and Techniques

By: 519030910100 and 519030910105

HW1: Search Algorithm

October 3, 2021

Contents

I. Introduction 1

A. Purpose 1

B. Equipment 1

C. Procedure 1

II. Procedure Diagrams 2

III. Experiment Data 3

A. Basic A* Algorithm 3

B. Improved A* Algorithm 3

C. Hybrid A* Algorithm 3

IV. Discussion 5

A. Strengths 5

B. Weaknesses 6

C. Trade-offs on Issues 6

V. Conclusion 7

A. Appendix for Codes 8

1. A* algorithm 8

2. Improved A* algorithm 12

3. Hybrid A* algorithm 16

I. INTRODUCTION

A. Purpose

The goal of today's lab is to develop a path planning framework for a service robot in an unknown environment using A* algorithm. This lab will combine CoppeliaSim and the algorithm code well and can lead to great agent movements based on an unknown map. A* algorithm is one type of search algorithm that is widely used in the artificial intelligence field and it can often lead to the optimal solution at a high speed. Thus, with the help of this lab, we want to understand the mechanisms of A* algorithm.

In this lab, we will control a simulated agent and take it to the goal position using three kinds of A* algorithm programs as the back-end code. Firstly, we managed to control the agent move forward, backward, left, and right. As long as the robot enters the gray block, the mission is considered successful. Next, we developed the BFS method to make sure we can control the agent correctly. Then, with the help of general cost and heuristics cost, we improved the BFS method into A* method. After that, we improved the initial A* algorithm in three parts:

- Possibility of moving towards the upper left, upper right, bottom left, bottom right.
- The distance between the robot and the obstacles to avoid a collision.
- The cost of steering to reduce the frequency of turning.

At last, we learnt Hybrid-A* algorithm and try to realize it.

B. Equipment

There is a minimal amount of equipment to be used in this lab. The few requirements are listed below:

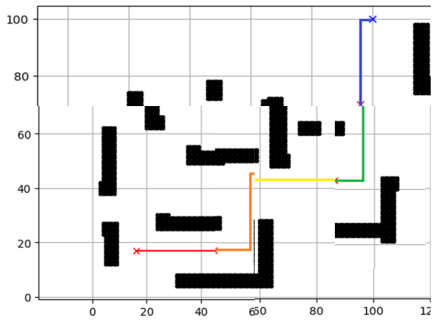
- Coppeliasim EDU V4.2.0 rev5
- Windows environment:Python 3.6, Numpy 1.19.5 and matplotlib 3.1.1.
- Visual Studio Code V1.60

C. Procedure

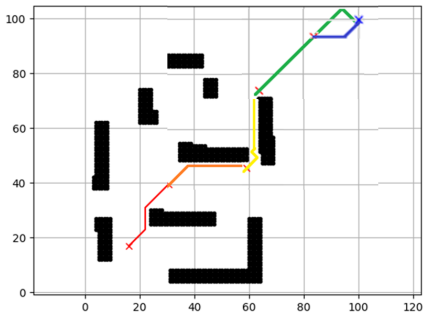
1. Start the Coppeliasim EDU. Open the 5-Search.ttt scene.
2. Start VS code. Open three main codes.
3. Run the codes in the terminal. Observe the trajectory of the agent.
4. Record the walk of the agent and plot it.
5. Stop the simulation, examine the graphs, and make a summary.

II. PROCEDURE DIAGRAMS

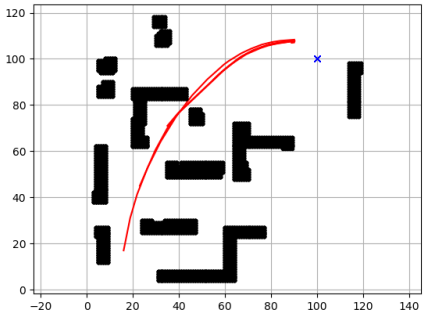
This section consists of screenshots taken during the laboratory procedure. The three figures presented below are the results of A*, improved A*, and hybrid A* algorithm.



(a) A* algorithm result.



(b) Improved A* algorithm result.



(c) Hybrid A* algorithm result.

FIG. 1: Period of path planned by A*, improved A* and hybrid A*.

III. EXPERIMENT DATA

This section will consist of the important code and the main idea of all the algorithms which displays our comprehension of these algorithms to meet the requirements of the lab.

A. Basic A* Algorithm

At the very beginning, we encapsulate the current state of the robot into a class called a node. Each node includes the current position of the robot, its backward cost, the sum of backward cost and forward cost, and its parent node. We also define a method to compare the cost of two nodes.

As for the heuristic function, we calculate the Manhattan Distance from the current position to the goal position. Also, we need to record the step from the start position to the current position as the backward cost. Combining these two cost, we can get the priority of every node.

At the core of the A* Algorithm, we build a priority queue called open set and a set called close-set. For the previous one, we push every searched node into the priority queue which sorts all the nodes by their total cost. And once a position has been searched, we will push it into the close-set to avoid repeated searches.

For each step of the robot, we first pop the top node of the open set as the current node. Then, move the robot forward, backward, left, and right if there is no obstacle and the new position isn't in the close-set. After that, we will push the new position into the close-set.

Finally, when the robot reaches the goal zone (a 17×17 square), the search will stop. With the help of the attribute 'parent', we can easily find the whole path from the start position to the goal position.

B. Improved A* Algorithm

Based on the A* algorithm, we add four more directions so that the robot can make a 45-degree turn. Since the robot can walk diagonally, it is inappropriate to use Manhattan Distance as the heuristic cost. Thus, we select Euclidean Distance as the new heuristic cost. For the same reason, we need to modify the backward cost, for directions like forwarding, backward, left and right, we just update their backward cost by plus one, and for directions like upper left, upper right, bottom left and bottom right, we need to update their backward cost by plus root 2.

To prevent the robot from walking along the wall, in addition to detecting whether there are obstacles in the current position, we will also detect 8 nearby positions. Once there are obstacles in these positions, the current position will not be added to the path.

To reduce the frequency of turning, we give more penalty for every turn, this penalty will be added to the backward cost. The method to judge whether the robot turn is just to check whether the current position and the first two positions on the path are in the same straight line. The mathematical formula is

$$(y_3 - y_1) \times (x_2 - x_1) - (y_2 - y_1) \times (x_3 - x_1) = 0$$

C. Hybrid A* Algorithm

Based on the paper of hybrid A* algorithm, we add **"angle"**, one important factor related to the orientation of the agent. Every turn of the agent will be based on the previous angle and the agent is only allowed to turn a small angle each time.

What's more, we notice that if each step of the agent is as small as possible, we can get a smoother path. To allow the agent to move a small step each time, we no longer force the agent to walk on the integer point. Instead, the point on the path will almost be in the type **"float"**, which is more flexible and closer to reality. But as an old saying goes, every coin has two sides. Since the agent walk on a small step, we need to expand more nodes each time, which leads to more time consumption.

These methods result in a smooth path and slight turning angles. Learning from the paper about the Hybrid A* algorithm, we build a similar "Potential Field". It is a field that sets up the cost for points near the obstacle. The cost is the addition of multiplication of different weights of the barriers near this point. Suppose the map has n rows and m columns. $map[i][j] = 1$ defines there is an obstacle in (i, j) and $map[i][j] = 0$ defines no obstacle in (i, j). Thus, the Potential Field cost is:

$$P - Cost_{i,j} = \Sigma_{i,j} map[i][j] \times weight[i][j]$$

This cost can be well taken into the calculation of total cost in the A* algorithm and leads to well-turning situations in the narrow path between obstacles. The figure below shows the Potential Field we set.

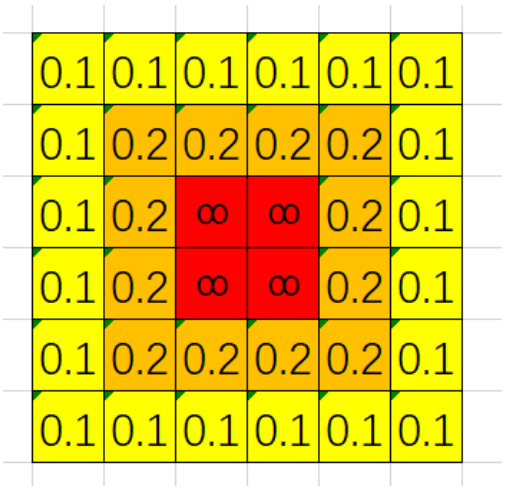


FIG. 2: One example to show the presentation of Potential Field.

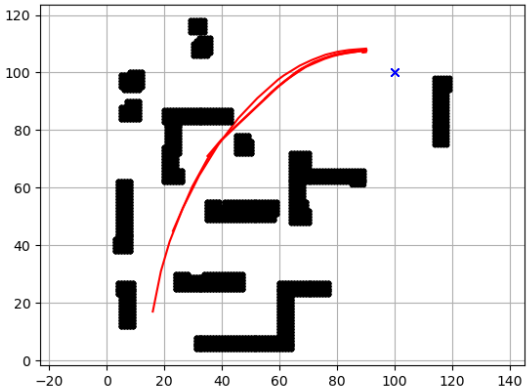


FIG. 3: The result of the Potential Field in A*.

IV. DISCUSSION

The goal of this lab was to apply the A* algorithm in the simulation system to run the agent well. By pressing different combinations of the cost and search methods, the agent on the software is to act in different ways using these algorithms. There was not a Q&A requirement for this lab.

A. Strengths

1. **Effective Fringe: priority queue and set**

   We use the priority as the fringe of the open set, which can significantly reduce the time complexity from $O(n^2)$ to $O(n \log n)$. Also, we use the set as the fringe of the close-set, which can avoid adding repetitive elements to reduce time complexity.

2. **Encapsulation by Class**

   We encapsulate the current state of the robot into a class called a node, which helps build the priority queue. The data structure **heapq** can just use the compare method of the node as the principle. Also, we can easily find the previous position by the attribute "parent" and the previous angle by the attribute "angle".

3. **Comprehensive Backward Cost**

   The total cost $f(n) = g(n)+h(n)$ mainly depends on the backward cost $g(n)$ and the forward cost $h(n)$. When we set $h(n)$ as the Manhattan distance, it is fixed and has nothing to do with the total cost. Anyway, the frequency of turning, the possibility of moving to different directions, and the collision steps can be taken into consideration of our $g(n)$, which is more comprehensive.

4. **Nice Turning Calculation Method**

   One important factor of $g(n)$ is the frequency of turning. To determine whether the agent is turning, we use the cross product. The key property of the cross product is that when three points are in a line, the cross product of these two vectors that are formed by these points is zero.

5. **Intuitive Visual Interface**

   The procedure diagrams are combined with each algorithm result. Each step is colored by different colors so that we can directly recognize the planned path and it will help us analyze the formulation and steps of the algorithm. By the way, we can easily obtain information about the frequency of turning and the distance from the obstacle.

6. **Least Number of Turns with Memorability**

   When we try to get the current_pos with function **"controller.get_robot_pos()"** , we find that the coordinates of the position returned are always integers. However, the agent will stop in restricted steps based on the API, leading to non-integer coordinates and an extra turn. Thus, we directly get the last position in our path and set it as the starting point. We call this method "Memorability".

7. **Introducing the Potential Field**

   We read the paper about hybrid A*, and build a similar potential field in our model. We just consider points near the obstacle, the nearer the point is, the more backward cost it will obtain. Thus, if the agent wants to move near the obstacle, it will not be banned, but it still needs to pay more backward costs.

B. Weaknesses

1. **Bad Time Consumption**

   Although we use priority queue and set to reduce time complexity, to achieve more requirements like the least number of turns, we need more time to compute the heuristic function and backward cost. Also, to keep the distance between the robot and the obstacles to avoid a collision, we need to detect 35 more positions than the basic A* algorithm.

2. **No Rotational Mechanical Factors**

   If we want to make this agent act like a real human, we should consider the angular velocity and linear velocity to make the turning method smoother like a man. However, in our work, we don't consider the rotational mechanical factors for the reason that the factors are hard to compute and the limitation of simulation software.

C. Trade-offs on Issues

1. **Step Size Setting**

   To smooth the path, we are supposed to set the length of each step as smaller as possible. Actually, the smaller each step is, the smoother the path will be. However, we also need to consider real-time consumption. If we set it too small, we will expand a large number of nodes when planning the path, which leads to terrible time complexity. Thus, we need to balance these two factors and try to find a suitable step size by experience.

2. **Parameters of the Potential Field**

   To keep the distance between the agent and the obstacle, we build a potential field around the obstacle. But the question is, how many penalties should we assign to each point based on the distance between the current position to the obstacle?

   Through a lot of experiments, we sum up a basic principle. **The penalty on the Potential Field needs to be similar to the cost of turning**. The potential field will not work if the penalty is too small. And if the penalty is too large, the agent will always avoid obstacles by detouring long distances.

V. CONCLUSION

Nowadays we should notice that driverless technology has made great progress with the help of new training and searching algorithms. Algorithms have played a more and more important role in helping robots to find correct routes. A preliminary analysis of this homework project is of great significance to our understanding of the high-tech frontier technology.

In this project, we design one type of A* algorithm and improve it in directions, smoothness, efficiency, and some factors that are not easy to be quantified in modeling. We try to find linear approximation and some other direct features by setting exact values. We create some important factors like angles, float numbers, and length of path steps. At last, we test our algorithms, present their result figures, and analyze their strengths and weaknesses. Such exploration can provide some useful and reasonable results which can be applied to real situations.

1. **Task 1: A* algorithm**

   We provide each initial status with the correct (may not be optimal) path.

2. **Task 2: Improved A* algorithm**

   We refined the original route in more directions, further distance from blocks, and shorter length of paths.

3. **Task 3: Hybrid A* algorithm**

   We refined the original route in more directions, smoother paths, perfect Potential Field, and nice pictures presented.

All in all, this laboratory gave us an insight into how algorithms work in reality and we hope to be able to apply them to the following labs.

Appendix A: Appendix for Codes

1. A* algorithm

```python
import DR20API
import numpy as np
# the minimun heap
from heapq import *
import matplotlib.pyplot as plt
### START CODE HERE ###
# This code block is optional. You can define your utility function and class in this block if necessary.

def general_cost(current_node, x, y):
    if not current_node.parent:
        return current_node.g_cost + 1
    v1 = [x-current_node.current_pos[0], y-current_node.current_pos[1]]
    v2 = [x-current_node.parent.current_pos[0], y-current_node.parent.current_pos[1]]
    v3 = v1[0]*v2[1] - v1[1]*v2[0]
    if(v3 == 0):
        return current_node.g_cost + 1
    return current_node.g_cost + 3

# The heuristics_cost is cauculated by Manhantan Distance
def heuristics_cost(current_node, goal_node):
    # Euclidean Distance
    # return ((current_node[1]-goal_node[1])**2 + (current_node[0]-goal_node[0])**2)**0.5

    # Manhantan Distance
    return np. sum(np. abs(current_node-goal_node))

def total_cost(current_node, goal_node):
    return heuristics_cost(current_node, goal_node)

class Node:
    def __init__(self, current_pos, parent, cost, g_cost):
        self.current_pos = current_pos
        self.parent = parent
        self.cost = cost
        self.g_cost = g_cost

    # define how to compare two nodes
    def __lt__(self, other):
        return self.cost < other.cost

###  END CODE HERE  ###

def A_star(current_map, current_pos, goal_pos):
    """
    Given current map of the world, current position of the robot and the position of the goal,
    plan a path from current position to the goal using A* algorithm.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1
        indicating obstacles.
```

```
    current_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    path -- A N*2 array representing the planned path by A* algorithm.
    """

    ### START CODE HERE ###

    # define the open_set and the close_set
    open_set, close_set = [],  set()

    # record the step from start to the current (general_cost)
    # step = 0

    # push the start_node into the minimun heap
    heappush(open_set, Node(current_pos, None, heuristics_cost(current_pos,goal_pos), 0))


    # pop node from the open_set
    while (open_set):
        node = heappop(open_set)

        # update the step
        # step += 1

        # if node locates in the goal, we can stop and find a path
        if (reach_goal(node.current_pos, goal_pos)):
            break


        # direct = [(-1,1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)]

        # if there is no obstacle and the next_point is still on the map, push it into the heap
        for next_x, next_y in [(node.current_pos[0]-1,node.current_pos[1]), (node.current_pos[0]+1,node.
            current_pos[1]), (node.current_pos[0],node.current_pos[1]-1), (node.current_pos[0],node.
            current_pos[1]+1)]:
            # flag = True
            # for n_x, n_y in direct:
                # if(current_map[next_x+n_x][next_y+n_y]==1):
                    # flag = False
                    # break
            # if(flag == False):
                # continue
            if (0 < next_x < 119 and 0 < next_y < 119 and current_map[next_x][next_y]==0 and (next_x,
                next_y) not in close_set):
                g_cost = general_cost(node,next_x,next_y)
                heappush(open_set, Node(np.array([next_x, next_y]), node, heuristics_cost(np.array([
                    next_x, next_y]), goal_pos) + g_cost, g_cost))
                # nodes in close_set will not be searched again
                close_set.add((next_x,next_y))

    path = [[node.current_pos[0], node.current_pos[1]]]

    # find the whole path through the parent_pos
    while (node.parent):
        path.append([node.parent.current_pos[0],node.parent.current_pos[1]])
```

```
        node = node.parent

    path = path[::-1]
    # print(path)
    obstacles_x, obstacles_y = [], []
    for i in  range(120):
        for j in  range(120):
            if current_map[i][j] == 1:
                obstacles_x.append(i)
                obstacles_y.append(j)

    path_x, path_y = [], []
    for path_node in path:
        path_x.append(path_node[0])
        path_y.append(path_node[1])

    plt.plot(path_x, path_y, "-r")
    plt.plot(current_pos[0], current_pos[1], "xr")
    plt.plot(goal_pos[0], goal_pos[1], "xb")
    plt.plot(obstacles_x, obstacles_y, ".k")
    plt.grid(True)
    plt.axis("equal")
    plt.show()
    ###  END CODE HERE  ###
    return path

def reach_goal(current_pos, goal_pos):
    """
    Given current position of the robot,
    check whether the robot has reached the goal.

    Arguments:
    current_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    is_reached -- A bool variable indicating whether the robot has reached the goal, where True
        indicating reached.
    """

    ### START CODE HERE ###

    if ( abs(current_pos[0] - goal_pos[0])<=5 and  abs(current_pos[1] - goal_pos[1])<=5):
        is_reached = True
    else:
        is_reached = False

    ###  END CODE HERE  ###
    return is_reached

if __name__ == '__main__':
    # Define goal position of the exploration, shown as the gray block in the scene.
    goal_pos = [100, 100]
    controller = DR20API.Controller()

    # Initialize the position of the robot and the map of the world.
    current_pos = controller.get_robot_pos()
```

```
    current_map = controller.update_map()

    # Plan-Move-Perceive-Update-Replan loop until the robot reaches the goal.
    while not reach_goal(current_pos, goal_pos):
        # Plan a path based on current map from current position of the robot to the goal.
        path = A_star(current_map, current_pos, goal_pos)
        # Move the robot along the path to a certain distance.
        controller.move_robot(path)
        # Get current position of the robot.
        current_pos = controller.get_robot_pos()
        # Update the map based on the current information of laser scanner and get the updated map.
        current_map = controller.update_map()

    # Stop the simulation.
    controller.stop_simulation()
    # plt.show()
```

Listing 1: **7-A-star.py**

2. Improved A* algorithm

```python
import DR20API
import numpy as np
# the minimun heap
from heapq import *
import matplotlib.pyplot as plt

### START CODE HERE ###
# This code block is optional. You can define your utility function and class in this block if necessary.

# def general_cost(current_node, start_node):
#     return 0

def general_cost1(current_node):
    return current_node.g_cost + 1

def general_cost2(current_node):
    return current_node.g_cost + 2**0.5


# The heuristics_cost is cauculated by Manhantan Distance
def heuristics_cost(current_node, goal_node):
    # Euclidean Distance
    return ((current_node[1]-goal_node[1])**2 + (current_node[0]-goal_node[0])**2)**0.5

    # Manhantan Distance
    # return np.sum(np.abs(current_node-goal_node))

# def total_cost(current_node, goal_node):
#     return heuristics_cost(current_node, goal_node)

class Node:
    def __init__(self, current_pos, parent, cost, g_cost):
        self.current_pos = current_pos
        self.parent = parent
        self.cost = cost
        self.g_cost = g_cost

    # define how to compare two nodes
    def __lt__(self, other):
        return self.cost < other.cost

###  END CODE HERE  ###

def Improved_A_star(current_map, current_pos, goal_pos):
    """
    Given current map of the world, current position of the robot and the position of the goal,
    plan a path from current position to the goal using A* algorithm.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1 indicating obstacles.
    current_pos -- A 2D vector indicating the current position of the robot.
```

```
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    path -- A N*2 array representing the planned path by A* algorithm.
    """

    ### START CODE HERE ###

    # define the open_set and the close_set
    open_set, close_set = [],  set()

    # push the start_node into the minimun heap
    heappush(open_set, Node(current_pos, None, heuristics_cost(current_pos,goal_pos), 0))

    # pop node from the open_set
    while (open_set):
        node = heappop(open_set)

        # if node locates in the goal, we can stop and find a path
        if (reach_goal(node.current_pos, goal_pos)):
            break

        # up, down, left, right
        direct1 = [(node.current_pos[0]-1,node.current_pos[1]), (node.current_pos[0]+1,node.current_pos
            [1]), (node.current_pos[0],node.current_pos[1]-1), (node.current_pos[0],node.current_pos
            [1]+1)]
        # NW, NE, SW, SE
        direct2 = [(node.current_pos[0]-1,node.current_pos[1]-1), (node.current_pos[0]+1,node.
            current_pos[1]+1), (node.current_pos[0]+1,node.current_pos[1]-1), (node.current_pos[0]-1,
            node.current_pos[1]+1)]

        # if there is no obstacle and the next_point is still on the map, push it into the heap
        for next_x, next_y in direct1:
            # cost for turn
            turn_cost = 0

            # judge weather near the obstacle
            flag = current_map[next_x][next_y] + current_map[next_x+1][next_y] + current_map[next_x-1][
                next_y] + current_map[next_x][next_y+1] + current_map[next_x][next_y-1] \
                   + current_map[next_x+1][next_y+1] + current_map[next_x+1][next_y-1] + current_map[
                       next_x-1][next_y+1] + current_map[next_x-1][next_y-1]

            if (0 < next_x < 119 and 0 < next_y < 119 and flag==0 and (next_x, next_y) not in close_set)
                :
                # judge three points collinear
                if node.parent and (next_y - node.parent.current_pos[1])*(node.current_pos[0] - node.
                    parent.current_pos[0]) - (node.current_pos[1] - node.parent.current_pos[1])*(next_x
                     - node.parent.current_pos[0]) != 0:
                    turn_cost += 2
                heappush(open_set, Node(np.array([next_x, next_y]), node, heuristics_cost(np.array([
                    next_x, next_y]), goal_pos) + general_cost1(node) + turn_cost, general_cost1(node)
                    + turn_cost))
                # nodes in close_set will not be searched again
                close_set.add((next_x,next_y))
        for next_x, next_y in direct2:
            # cost for turn
            turn_cost = 0
```

```
98
                # judge weather near the obstacle
100             flag = current_map[next_x][next_y] + current_map[next_x+1][next_y] + current_map[next_x-1][
                    next_y] + current_map[next_x][next_y+1] + current_map[next_x][next_y-1] \
                       + current_map[next_x+1][next_y+1] + current_map[next_x+1][next_y-1] + current_map[
                           next_x-1][next_y+1] + current_map[next_x-1][next_y-1]
102
                if (0 < next_x < 119 and 0 < next_y < 119 and flag==0 and (next_x, next_y) not in close_set)
                    :
104                 # judge three points collinear
                    if node.parent and (next_y - node.parent.current_pos[1])*(node.current_pos[0] - node.
                        parent.current_pos[0]) - (node.current_pos[1] - node.parent.current_pos[1])*(next_x
                         - node.parent.current_pos[0]) != 0:
106                     turn_cost += 2
                    heappush(open_set, Node(np.array([next_x, next_y]), node, heuristics_cost(np.array([
                        next_x, next_y]), goal_pos) + general_cost2(node) + turn_cost, general_cost2(node)
                        + turn_cost))
108                 # nodes in close_set will not be searched again
                    close_set.add((next_x,next_y))
110

112     path = [[node.current_pos[0], node.current_pos[1]]]

114     # find the whole path through the parent_pos
        while (node.parent):
116         path.append([node.parent.current_pos[0],node.parent.current_pos[1]])
            node = node.parent
118
        path = path[::-1]
120     print(path)


122
        # Visualize the map and path.
124     obstacles_x, obstacles_y = [], []
        for i in  range(120):
126         for j in  range(120):
                if current_map[i][j] == 1:
128                 obstacles_x.append(i)
                    obstacles_y.append(j)
130
        path_x, path_y = [], []
132     for path_node in path:
            path_x.append(path_node[0])
134         path_y.append(path_node[1])

136     plt.plot(path_x, path_y, "-r")
        plt.plot(current_pos[0], current_pos[1], "xr")
138     plt.plot(goal_pos[0], goal_pos[1], "xb")
        plt.plot(obstacles_x, obstacles_y, ".k")
140     plt.grid(True)
        plt.axis("equal")
142     plt.show()

144     ###  END CODE HERE  ###
        return path
146
def reach_goal(current_pos, goal_pos):
```

```
    """
    Given current position of the robot,
    check whether the robot has reached the goal.

    Arguments:
    current_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    is_reached -- A bool variable indicating whether the robot has reached the goal, where True
        indicating reached.
    """

    ### START CODE HERE ###

    if ( abs(current_pos[0] - goal_pos[0])<=8 and  abs(current_pos[1] - goal_pos[1])<=8):
        is_reached = True
    else:
        is_reached = False

    ###  END CODE HERE  ###
    return is_reached

if __name__ == '__main__':
    # Define goal position of the exploration, shown as the gray block in the scene.
    goal_pos = [100, 100]
    controller = DR20API.Controller()

    # Initialize the position of the robot and the map of the world.
    current_pos = controller.get_robot_pos()
    current_map = controller.update_map()

    # Plan-Move-Perceive-Update-Replan loop until the robot reaches the goal.
    while not reach_goal(current_pos, goal_pos):
        # Plan a path based on current map from current position of the robot to the goal.
        path = Improved_A_star(current_map, current_pos, goal_pos)
        # Move the robot along the path to a certain distance.
        controller.move_robot(path)
        # Get current position of the robot.
        current_pos = controller.get_robot_pos()
        # Update the map based on the current information of laser scanner and get the updated map.
        current_map = controller.update_map()

    # Stop the simulation.
    controller.stop_simulation()
```

Listing 2: **8-Improved-A-star.py**

3. Hybrid A* algorithm

```python
import DR20API
import numpy as np
# the minimun heap
from heapq import *
import matplotlib.pyplot as plt
import math

### START CODE HERE ###
# This code block is optional. You can define your utility function and class in this block if necessary
    .


def general_cost(current_node):
    return current_node.g_cost + 0.2

# def general_cost2(current_node):
#     return current_node.g_cost + 2**0.5


# The heuristics_cost is cauculated by Manhantan Distance
def heuristics_cost(current_node, goal_node):
    # Euclidean Distance
    return ((current_node[1]-goal_node[1])**2 + (current_node[0]-goal_node[0])**2)**0.5

# def total_cost(current_node, goal_node):
#     return heuristics_cost(current_node, goal_node)

class Node:
    def __init__(self, current_pos, angle, layer, parent, cost, g_cost):
        self.x = current_pos[0]
        self.y = current_pos[1]
        self.angle = angle
        self.parent = parent
        self.cost = cost
        self.g_cost = g_cost

        # define the deepest search layer
        self.layer = layer

    # define how to compare two nodes
    def __lt__(self, other):
        return self.cost < other.cost

###  END CODE HERE  ###

# def Hybrid_A_star(current_map, current_pos, goal_pos):
def Hybrid_A_star(current_map, current_pos, goal_pos, angle, total):
    """
    Given current map of the world, current position of the robot and the position of the goal,
    plan a path from current position to the goal using A* algorithm.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1
```

```
        indicating obstacles.
    current_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    path -- A N*2 array representing the planned path by A* algorithm.
    """
    # print(total)
    ### START CODE HERE ###

    # define the open_set and the close_set
    open_set, close_set = [],  set()

    # push the start_node into the minimun heap
    heappush(open_set, Node(current_pos, angle, 0, None, heuristics_cost(current_pos,goal_pos), 0))

    # pop node from the open_set
    while (open_set):
        node = heappop(open_set)

        # if node locates in the goal, we can stop and find a path
        current_pos = np.array([node.x, node.y])
        # if (node.layer > 200 or reach_goal(current_pos, goal_pos)):
        if (reach_goal(current_pos, goal_pos)):
            break

        if(current_pos[0] >=110 or current_pos[1] >=110):
            continue

        # possible steering angle
        steering = [-0.03*math.pi, 0, 0.03*math.pi]
        cost_steering = [0.2, 0, 0.2]

        # possible steering angle
        # steering = [-0.4*math.pi, -0.2*math.pi,  0,  0.2*math.pi, 0.4*math.pi]
        # cost_steering = [2, 1, 0, 1, 2]

        # current position of the robot
        x, y = node.x, node.y

        # if there is no obstacle and the next_point is still on the map, push it into the heap
        for i, angle in  enumerate(steering):
            # cost for turn
            steering_cost = cost_steering[i]

            # new angle
            angle += node.angle

            # calculate the next position of the robot according to the turn
            # cx, cy represent the continue position of the robot
            # dx, dy represent the discrete position of the robot
            next_cx, next_cy =  round(x + 0.25*math.cos(angle),2),  round(y + 0.25  *math.sin(angle),2)
            next_dx, next_dy =  int(next_cx),  int(next_cy)

            # judge weather near the obstacle
            # 这里现在flag直接用来作为cost记录
            # flag = current_map[next_dx][next_dy] + current_map[next_dx+1][next_dy] + current_map[
```

```
                next_dx-1][next_dy] + current_map[next_dx][next_dy+1] + current_map[next_dx][next_dy-1]
                 \
            flag = current_map[next_dx+2][next_dy+1] + current_map[next_dx+2][next_dy-1]    \
                   + current_map[next_dx+2][next_dy]  + current_map[next_dx][next_dy+2]  \
                   + current_map[next_dx+1][next_dy+2] + current_map[next_dx-1][next_dy+2]  \
                   + current_map[next_dx+2][next_dy+2] + current_map[next_dx-1][next_dy+1] \
                   + current_map[next_dx-1][next_dy] + current_map[next_dx-1][next_dy-1] \
                   + current_map[next_dx][next_dy-1] + current_map[next_dx+1][next_dy-1]

            # New definition of flag
            flag *= 2

            flag +=  current_map[next_dx+3][next_dy+1] + current_map[next_dx+3][next_dy-1]    \
                   + current_map[next_dx+3][next_dy]  + current_map[next_dx][next_dy+3]  \
                   + current_map[next_dx+1][next_dy+3] + current_map[next_dx-1][next_dy+3]  \
                   + current_map[next_dx+2][next_dy+3] + current_map[next_dx-1][next_dy-2] \
                   + current_map[next_dx-2][next_dy] + current_map[next_dx-2][next_dy-1] \
                   + current_map[next_dx][next_dy-2] + current_map[next_dx+1][next_dy-2] \
                   + current_map[next_dx+2][next_dy-2] + current_map[next_dx+3][next_dy-2] \
                   + current_map[next_dx+3][next_dy+2] + current_map[next_dx+3][next_dy+3] \
                   + current_map[next_dx-2][next_dy-2] + current_map[next_dx-2][next_dy+1] \
                   + current_map[next_dx-2][next_dy+2] + current_map[next_dx-2][next_dy+3]

            flag *= 0.1

            obs = current_map[next_dx][next_dy] + current_map[next_dx+1][next_dy] + current_map[next_dx
                ][next_dy+1] + current_map[next_dx+1][next_dy+1]

            # 然后flag==0就可以排除掉了，变成该位置不为障碍
            if (0 < next_cx < 110 and 0 < next_cy < 110 and not obs and (next_cx, next_cy) not in
                close_set):
                heappush(open_set, Node(np.array([next_cx, next_cy]), angle, node.layer+1, node,
                    heuristics_cost(np.array([next_cx, next_cy]), goal_pos) + flag + general_cost(node)
                     + steering_cost, general_cost(node) + steering_cost))
                # heappush(open_set, Node(np.array([next_cx, next_cy]), angle, node.layer+1, node,
                    heuristics_cost(np.array([next_cx, next_cy]), goal_pos) + flag + general_cost(node)
                     + steering_cost, general_cost(node) + steering_cost))
                # nodes in close_set will not be searched again
                close_set.add((next_cx,next_cy))

    path = [[node.x, node.y]]

    # find the whole path through the parent_pos
    while (node.parent):
        path.append([node.parent.x,node.parent.y])
        node = node.parent

    path = path[::-1]
    # path[0][1] += 1
    # print(path)


    # Visualize the map and path.
    obstacles_x, obstacles_y = [], []
    for i in  range(120):
        for j in  range(120):
            if current_map[i][j] == 1:
```

```
                obstacles_x.append(i)
                obstacles_y.append(j)

    path_x, path_y = [], []
    for path_node in path:
        path_x.append(path_node[0])
        path_y.append(path_node[1])

    plt.plot(path_x, path_y, "-r")
    # plt.plot(current_pos[0], current_pos[1], "xr")
    plt.plot(goal_pos[0], goal_pos[1], "xb")
    plt.plot(obstacles_x, obstacles_y, ".k")
    plt.grid(True)
    plt.axis("equal")
    # plt.show()
    plt.savefig(r"D:\College\5\AI\AI3603_HW1\result4\filename{}.png". format(total))
    # print(path)
    # print(annngle)
    ###  END CODE HERE  ###
    return path

def reach_goal(current_pos, goal_pos):
    """
    Given current position of the robot,
    check whether the robot has reached the goal.

    Arguments:
    current_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    is_reached -- A bool variable indicating whether the robot has reached the goal, where True
        indicating reached.
    """

    ### START CODE HERE ###

    if ( abs(current_pos[0] - goal_pos[0])<=10 and  abs(current_pos[1] - goal_pos[1])<=10):
        is_reached = True
    else:
        is_reached = False

    ###  END CODE HERE  ###
    return is_reached

if __name__ == '__main__':
    # Define goal position of the exploration, shown as the gray block in the scene.
    goal_pos = [100, 100]
    controller = DR20API.Controller()
    total = 0

    # Initialize the position of the robot and the map of the world.
    current_pos = controller.get_robot_pos()
    current_map = controller.update_map()

    # Plan-Move-Perceive-Update-Replan loop until the robot reaches the goal.
    while not reach_goal(current_pos, goal_pos):
```

```
        # Plan a path based on current map from current position of the robot to the goal.
        total += 1
        # Get current orientation of the robot.
        current_ori = controller.get_robot_ori()
        path = Hybrid_A_star(current_map, current_pos, goal_pos, current_ori, total)
        # Move the robot along the path to a certain distance.
        controller.move_robot(path)
        # Get current position of the robot.
        # current_pos = controller.get_robot_pos()
        current_pos = controller.get_robot_pos()
        # current_pos = np.array(path[-1])
        # This current_pos2 is to avoid the
        current_pos = controller.get_robot_pos()
        # Update the map based on the current information of laser scanner and get the updated map.
        current_map = controller.update_map()

    # Stop the simulation.
    controller.stop_simulation()

###  END CODE HERE  ###
```

Listing 3: **9-Hybrid-A-star.py**