电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

实验报告

Experimental Report

报 告 题 目 基于 A*路径规划实现机器人最短路径至目标点

学　　院 机械与电气工程学院

专　　业 机器人工程

学　　号 2021040902007

作 者 姓 名 经彭宇

指 导 教 师 张鑫

目 录

第一章 任务场景

随着智能移动机器人在各个领域的应用不断增加，如自动驾驶车辆、无人机、仓储机器人等，路径规划成为了其中至关重要的一环。在复杂的环境中，机器人需要能够快速、高效地找到从起点到目标点的最优路径，并避开障碍物，以确保安全性和效率性。因此需要设计并实现一个基于 A*算法的路径规划器，用于解决智能移动机器人的路径规划问题。

任务首先生成地图生成，设计一个虚拟环境，包括起点、目标点和障碍物。生成障碍物地图，用于模拟真实环境中的障碍物。其次实现 A*算法的路径规划功能，能够在障碍物地图中找到起点到目标点的最优路径，确保生成的路径不与障碍物相交。此外可视化展示，用 matplotlib 等工具将地图、起点、目标点、障碍物和最优路径进行可视化展示。接着调整启发函数的权重和机器人的运动模型，优化路径规划的效果和性能。最后结果分析讨论算法在不同环境下的适用性和局限性，提出改进和优化的建议。

第二章 算法原理及流程图

2.1 总体思路

首先把起点加入待考察的节点组成的表 openList。重复如下过程：遍历 openList ，查找 F 值最小的节点，把它作为当前要处理的节点。把这个节点移到已经考察过的节点组成的表 closeList。对当前方格的 8 个相邻方格：如果它是不可抵达的或者它在 closeList 中，忽略它；如果它不在 openList 中，则把它加入 openList ，并且把当前方格设置为它的父节点，记录该方格的从初始状态经由状态 n 到目标状态的代价估计 f，在状态空间中从初始状态到状态 n 的实际代价 g，从状态 n 到目标状态的最佳路径的估计代价 h。如果它已经在 openList 中，检查这条路径是否更好，用 g 值作参考，更小的 g 值表示这是更好的路径。如果 g 值更小，把该节点的父节点设置为当前方格，并重新计算它的 g 和 h 值。停止搜索的情况有两种：把终点加入到了 openList 中，此时路径已经找到了；查找终点失败，并且 openList 是空的，此时没有路径。保存路径。使用回溯的方法，从终点开始，每个方格沿着父节点移动直至起点，最终搜索到的路径为最短路径。

2.2 实现方法

（1）预处理

首先把地图栅格化，把每一个方格的中心称为节点，把搜索区域简化为 2 维数组。数组的每一项代表一个格子，它的状态就是可走和不可走。通过计算出从 A 到目标点需要走过哪些方格，就找到了路径。一旦路径找到了，便从一个方格的中心移动到另一个方格的中心，直至到达目的地。

（2）开始搜索

把搜寻区域简化为一组可以量化的节点后，便是查找最短路径。在 A* 中，从起点开始，检查其相邻的方格，然后向四周扩展，直至找到目标。从起点 A 开始，定义 A 为父节点，并把它加入到 openList 中，父节点 A 周围共有 8 个节点，定义为子节点。将子节点中可达的或者可走的放入 openList 中，节点 A 离自身距离为 0，路径完全确定，将其移入 closeList 中单步移动代价采取 Manhattan 计算方式

$$d = |x_1 - x_2| + |y_1 - y_2|$$

即把横向和纵向移动一个节点的代价定义为 10，斜向移动代价为 14。

接着需要去选择节点 A 相邻的子节点中移动代价 f 最小的节点，采用移动代价评价函数

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

f (n)是从初始状态经由状态 n 到目标状态的代价估计，g (n)是在状态空间中从初始状态到状态 n 的实际代价，h (n)是从状态 n 到目标状态的最佳路径的估计代价。以此类推，分别计算当前 openList 中余下的 7 个子节点的移动代价 f，从中挑选最小代价节点 F，移到 closeList 中。

（3）继续搜索

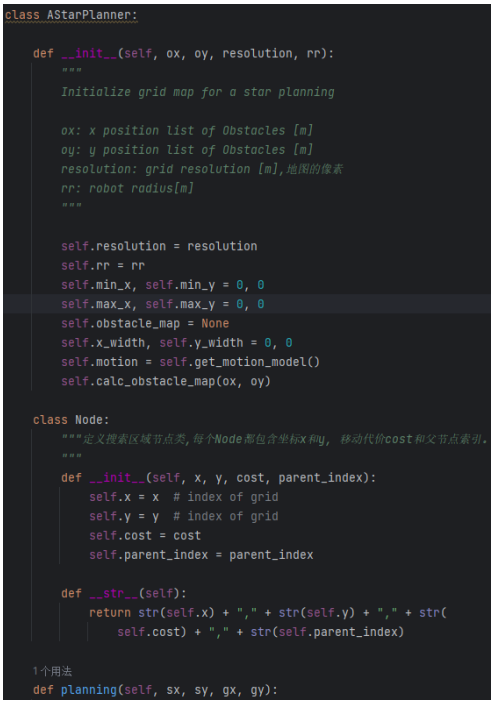
选择完后，检查所有与它相邻的子节点，忽略不可走的节点、以及忽略已经存在于 closeList 的节点；如果方格不在 openList 中，则把它们加入到 openList 中，并把它们作为节点的子节点。如果某个相邻的节点 X 已经在 openLlist 中，则检查这条路径是否更优。如果没有，不做任何操作。否则把 X 的父节点设为当前方格 ，然后重新计算 X 的 f 值和 g 值。依次类推，不断重复。一旦搜索到目标节点 T，完成路径搜索，结束算法。

（4）确定实际路径

完成路径搜索后，从终点开始，向父节点移动，直至被带回到了起点，形成搜索后的路径。

2.3 代码思路

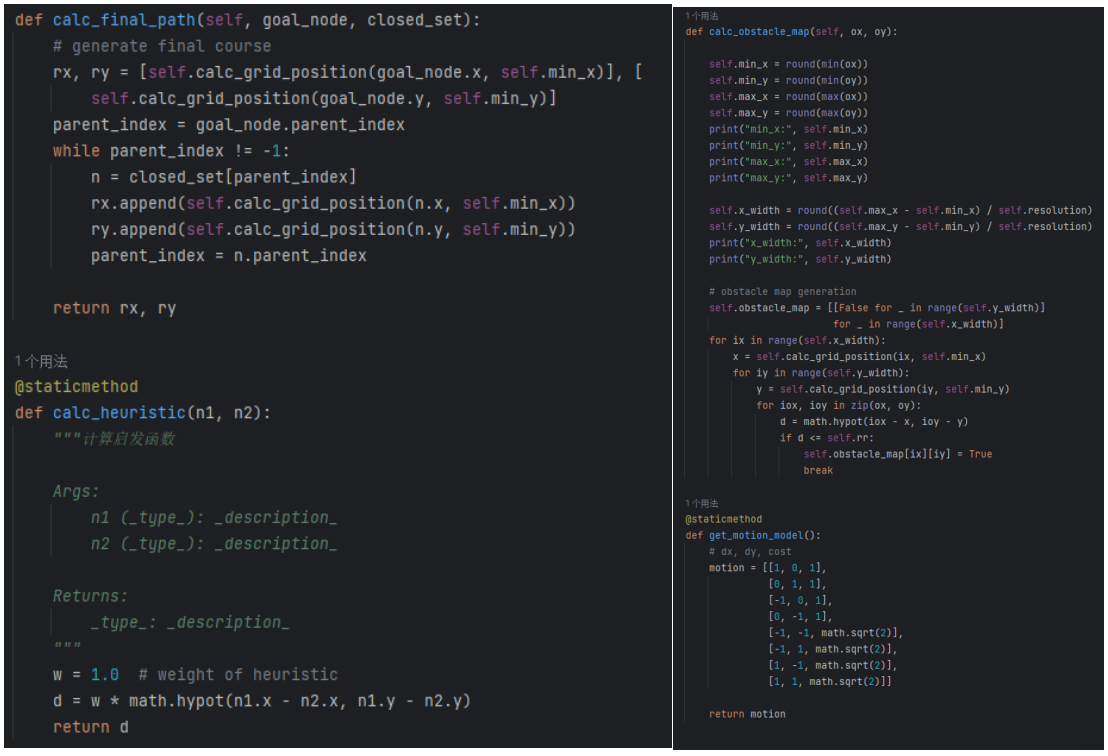
定义了一个 AStarPlanner 类，用于进行 A*路径规划。init 用于初始化地图和机器人参数，包括障碍物列表、地图分辨率、机器人半径等。Node 内部类定义了搜索区域的节点，包括节点的坐标、移动代价和父节点索引等信息。planning 方法是 A*路径规划的主要实现，接收起始点和目标点坐标，并输出最优路径的坐标集合。



```
class AStarPlanner:

    def __init__(self, ox, oy, resolution, rr):
        """
        Initialize grid map for a star planning

        ox: x position list of Obstacles [m]
        oy: y position list of Obstacles [m]
        resolution: grid resolution [m],地图的像素
        rr: robot radius[m]
        """

        self.resolution = resolution
        self.rr = rr
        self.min_x, self.min_y = 0, 0
        self.max_x, self.max_y = 0, 0
        self.obstacle_map = None
        self.x_width, self.y_width = 0, 0
        self.motion = self.get_motion_model()
        self.calc_obstacle_map(ox, oy)

    class Node:
        """定义搜索区域节点类,每个Node都包含坐标x和y, 移动代价cost和父节点索引。
        """
        def __init__(self, x, y, cost, parent_index):
            self.x = x  # index of grid
            self.y = y  # index of grid
            self.cost = cost
            self.parent_index = parent_index

        def __str__(self):
            return str(self.x) + "," + str(self.y) + "," + str(
                self.cost) + "," + str(self.parent_index)

    1个用法
    def planning(self, sx, sy, gx, gy):
```

图 2-1 创建类

定义了一些函数：calc_final_path 方法用于生成最终路径。calc_heuristic 方法用于计算启发函数来估计当前节点到目标节点的成本。verify_node 方法用于检查节点是否安全。calc_obstacle_map 方法用于生成障碍物地图。get_motion_model 方法用于定义机器人的运动模型，即允许的运动方向和代价。



```
def calc_final_path(self, goal_node, closed_set):
    # generate final course
    rx, ry = [self.calc_grid_position(goal_node.x, self.min_x)], [
        self.calc_grid_position(goal_node.y, self.min_y)]
    parent_index = goal_node.parent_index
    while parent_index != -1:
        n = closed_set[parent_index]
        rx.append(self.calc_grid_position(n.x, self.min_x))
        ry.append(self.calc_grid_position(n.y, self.min_y))
        parent_index = n.parent_index

    return rx, ry

1个用法
@staticmethod
def calc_heuristic(n1, n2):
    """计算启发函数

    Args:
        n1 (_type_): _description_
        n2 (_type_): _description_

    Returns:
        _type_: _description_
    """
    w = 1.0  # weight of heuristic
    d = w * math.hypot(n1.x - n2.x, n1.y - n2.y)
    return d
```

```
1个用法
def calc_obstacle_map(self, ox, oy):

    self.min_x = round(min(ox))
    self.min_y = round(min(oy))
    self.max_x = round(max(ox))
    self.max_y = round(max(oy))
    print("min_x:", self.min_x)
    print("min_y:", self.min_y)
    print("max_x:", self.max_x)
    print("max_y:", self.max_y)

    self.x_width = round((self.max_x - self.min_x) / self.resolution)
    self.y_width = round((self.max_y - self.min_y) / self.resolution)
    print("x_width:", self.x_width)
    print("y_width:", self.y_width)

    # obstacle map generation
    self.obstacle_map = [[False for _ in range(self.y_width)]
                         for _ in range(self.x_width)]
    for ix in range(self.x_width):
        x = self.calc_grid_position(ix, self.min_x)
        for iy in range(self.y_width):
            y = self.calc_grid_position(iy, self.min_y)
            for iox, ioy in zip(ox, oy):
                d = math.hypot(iox - x, ioy - y)
                if d <= self.rr:
                    self.obstacle_map[ix][iy] = True
                    break

1个用法
@staticmethod
def get_motion_model():
    # dx, dy, cost
    motion = [[1, 0, 1],
              [0, 1, 1],
              [-1, 0, 1],
              [0, -1, 1],
              [-1, -1, math.sqrt(2)],
              [-1, 1, math.sqrt(2)],
              [1, -1, math.sqrt(2)],
              [1, 1, math.sqrt(2)]]

    return motion
```

图 2-2 函数

定义了 main 函数用于执行路径规划实验：首先定义了起始点、目标点、栅格分辨率、机器人半径和障碍物位置。利用 matplotlib 绘制了栅格地图、起始点、目标点和最终路径。创建 AStarPlanner 对象并调用 planning 方法进行路径规划。最后展示路径规划结果。

2.4 流程图

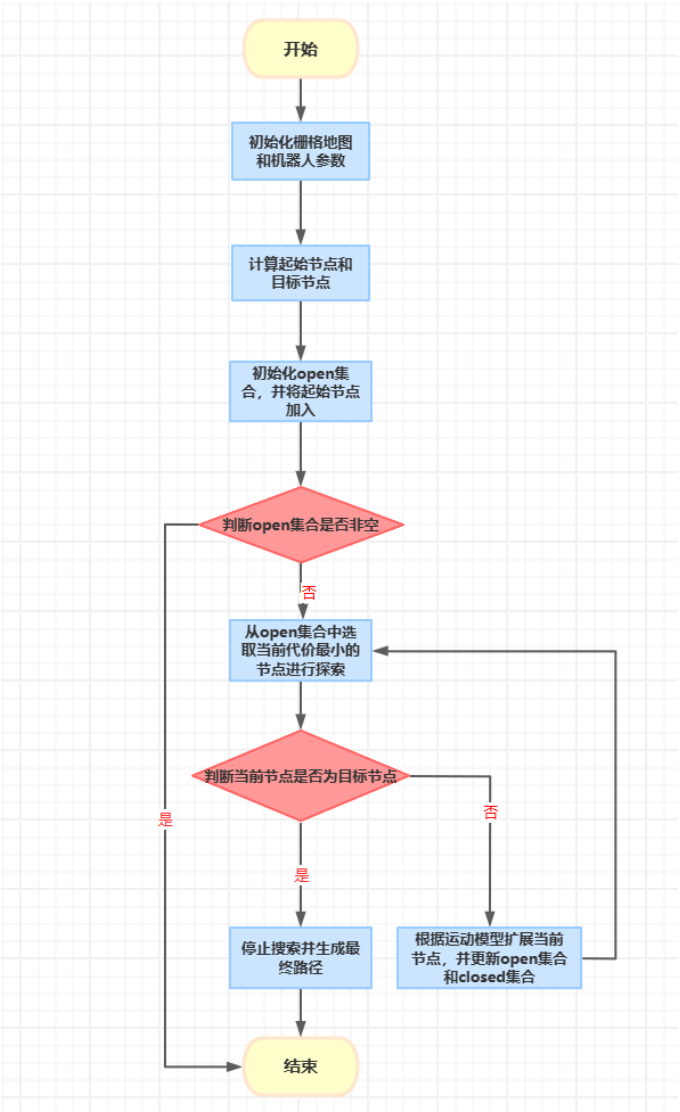


图 2-3 流程图

第三章 代码与结果

3.1 代码

```
import math
import matplotlib.pyplot as plt
show_animation = True

class AStarPlanner:

    def __init__(self, ox, oy, resolution, rr):

        self.resolution = resolution
        self.rr = rr
        self.min_x, self.min_y = 0, 0
        self.max_x, self.max_y = 0, 0
        self.obstacle_map = None
        self.x_width, self.y_width = 0, 0
        self.motion = self.get_motion_model()
        self.calc_obstacle_map(ox, oy)

    class Node:
        """定义搜索区域节点类,每个Node 都包含坐标x 和y, 移动代价cost 和父节点索
引。
        """
        def __init__(self, x, y, cost, parent_index):
            self.x = x  # index of grid
            self.y = y  # index of grid
            self.cost = cost
            self.parent_index = parent_index

        def __str__(self):
            return str(self.x) + "," + str(self.y) + "," + str(
                self.cost) + "," + str(self.parent_index)

    def planning(self, sx, sy, gx, gy):
        start_node = self.Node(self.calc_xy_index(sx, self.min_x),
                               self.calc_xy_index(sy, self.min_y), 0.0, -1)
        goal_node = self.Node(self.calc_xy_index(gx, self.min_x),
                              self.calc_xy_index(gy, self.min_y), 0.0, -1)

        open_set, closed_set = dict(), dict()
        open_set[self.calc_grid_index(start_node)] = start_node

        while 1:
            if len(open_set) == 0:
                print("Open set is empty..")
                break

            c_id = min(
                open_set,
                key=lambda o: open_set[o].cost +
self.calc_heuristic(goal_node, open_set[o]))
            current = open_set[c_id]
```

```
            # show graph
            if show_animation:  # pragma: no cover
                plt.plot(self.calc_grid_position(current.x, self.min_x),
                         self.calc_grid_position(current.y, self.min_y),
"xc")
                # for stopping simulation with the esc key.
                plt.gcf().canvas.mpl_connect('key_release_event', lambda
event: [exit(0) if event.key == 'escape' else None])
                if len(closed_set.keys()) % 10 == 0:
                    plt.pause(0.001)

            # 通过追踪当前位置 current.x 和 current.y 来动态展示路径寻找
            if current.x == goal_node.x and current.y == goal_node.y:
                print("Find goal")
                goal_node.parent_index = current.parent_index
                goal_node.cost = current.cost
                break

            # Remove the item from the open set
            del open_set[c_id]

            # Add it to the closed set
            closed_set[c_id] = current

            # expand_grid search grid based on motion model
            for i, _ in enumerate(self.motion):
                node = self.Node(current.x + self.motion[i][0],
                                 current.y + self.motion[i][1],
                                 current.cost + self.motion[i][2], c_id)
                n_id = self.calc_grid_index(node)

                # If the node is not safe, do nothing
                if not self.verify_node(node):
                    continue

                if n_id in closed_set:
                    continue

                if n_id not in open_set:
                    open_set[n_id] = node  # discovered a new node
                else:
                    if open_set[n_id].cost > node.cost:
                        # This path is the best until now. record it
                        open_set[n_id] = node

        rx, ry = self.calc_final_path(goal_node, closed_set)

        return rx, ry

    def calc_final_path(self, goal_node, closed_set):
        # generate final course
        rx, ry = [self.calc_grid_position(goal_node.x, self.min_x)], [
            self.calc_grid_position(goal_node.y, self.min_y)]
        parent_index = goal_node.parent_index
        while parent_index != -1:
            n = closed_set[parent_index]
            rx.append(self.calc_grid_position(n.x, self.min_x))
            ry.append(self.calc_grid_position(n.y, self.min_y))
            parent_index = n.parent_index
```

```
        return rx, ry

    @staticmethod
    def calc_heuristic(n1, n2):
        w = 1.0  # weight of heuristic
        d = w * math.hypot(n1.x - n2.x, n1.y - n2.y)
        return d

    def calc_grid_position(self, index, min_position):
        pos = index * self.resolution + min_position
        return pos

    def calc_xy_index(self, position, min_pos):
        return round((position - min_pos) / self.resolution)

    def calc_grid_index(self, node):
        return (node.y - self.min_y) * self.x_width + (node.x -
self.min_x)

    def verify_node(self, node):
        px = self.calc_grid_position(node.x, self.min_x)
        py = self.calc_grid_position(node.y, self.min_y)

        if px < self.min_x:
            return False
        elif py < self.min_y:
            return False
        elif px >= self.max_x:
            return False
        elif py >= self.max_y:
            return False

        # collision check
        if self.obstacle_map[node.x][node.y]:
            return False

        return True

    def calc_obstacle_map(self, ox, oy):

        self.min_x = round(min(ox))
        self.min_y = round(min(oy))
        self.max_x = round(max(ox))
        self.max_y = round(max(oy))
        print("min_x:", self.min_x)
        print("min_y:", self.min_y)
        print("max_x:", self.max_x)
        print("max_y:", self.max_y)

        self.x_width = round((self.max_x - self.min_x) /
self.resolution)
        self.y_width = round((self.max_y - self.min_y) /
self.resolution)
        print("x_width:", self.x_width)
        print("y_width:", self.y_width)

        # obstacle map generation
        self.obstacle_map = [[False for _ in range(self.y_width)]
```

```
                        for _ in range(self.x_width)]
        for ix in range(self.x_width):
            x = self.calc_grid_position(ix, self.min_x)
            for iy in range(self.y_width):
                y = self.calc_grid_position(iy, self.min_y)
                for iox, ioy in zip(ox, oy):
                    d = math.hypot(iox - x, ioy - y)
                    if d <= self.rr:
                        self.obstacle_map[ix][iy] = True
                        break

    @staticmethod
    def get_motion_model():
        # dx, dy, cost
        motion = [[1, 0, 1],
                  [0, 1, 1],
                  [-1, 0, 1],
                  [0, -1, 1],
                  [-1, -1, math.sqrt(2)],
                  [-1, 1, math.sqrt(2)],
                  [1, -1, math.sqrt(2)],
                  [1, 1, math.sqrt(2)]]

        return motion


def main():
    print(__file__ + " start!!")

    # start and goal position
    sx = 10.0  # [m]
    sy = 10.0  # [m]
    gx = 45.0  # [m]
    gy = 50.0  # [m]
    grid_size = 2.0  # [m]
    robot_radius = 1.0  # [m]

    # set obstacle positions
    ox, oy = [], []
    for i in range(-10, 60):
        ox.append(i)
        oy.append(-10.0)
    for i in range(-10, 60):
        ox.append(60.0)
        oy.append(i)
    for i in range(-10, 61):
        ox.append(i)
        oy.append(60.0)
    for i in range(-10, 61):
        ox.append(-10.0)
        oy.append(i)
    for i in range(-10, 40):
        ox.append(20.0)
        oy.append(i)
    for i in range(0, 40):
        ox.append(40.0)
        oy.append(60.0 - i)

    if show_animation:  # pragma: no cover
```

```
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "og")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")

    a_star = AStarPlanner(ox, oy, grid_size, robot_radius)
    rx, ry = a_star.planning(sx, sy, gx, gy)

    if show_animation:  # pragma: no cover
        plt.plot(rx, ry, "-r")
        plt.pause(0.001)
        plt.show()


if __name__ == '__main__':
    main()
```

3.2 结果

经过 A*路径规划算法计算后，机器人成功找到起始点到目标点的最优路径，设置三组实验，最终路径分别如下图所示：

（1）出发点（10,10） 目标点（45,50）

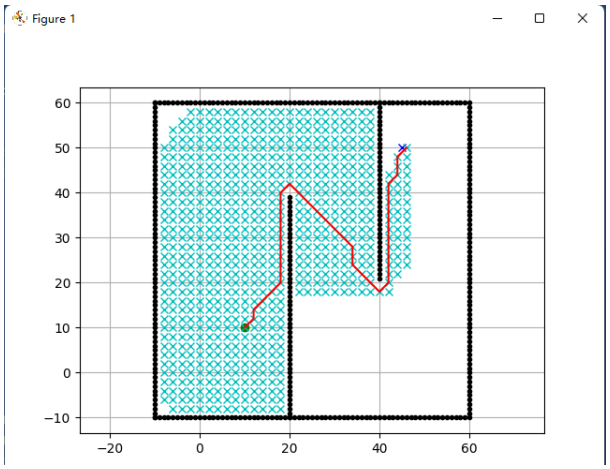


图 3-1 结果 1

（2）出发点（55,50） 目标点（10,15）

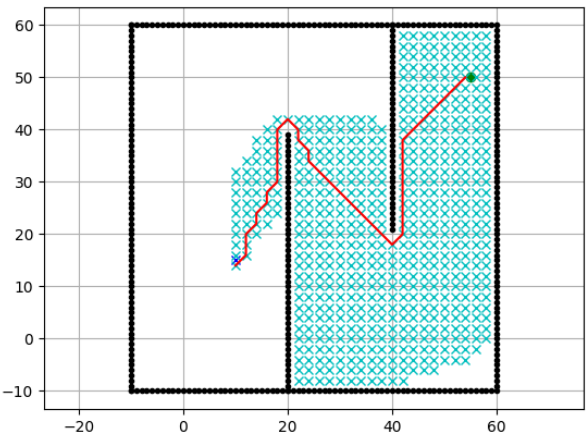


图 3-2 结果 2

（3）出发点（55,50） 目标点（25,30）

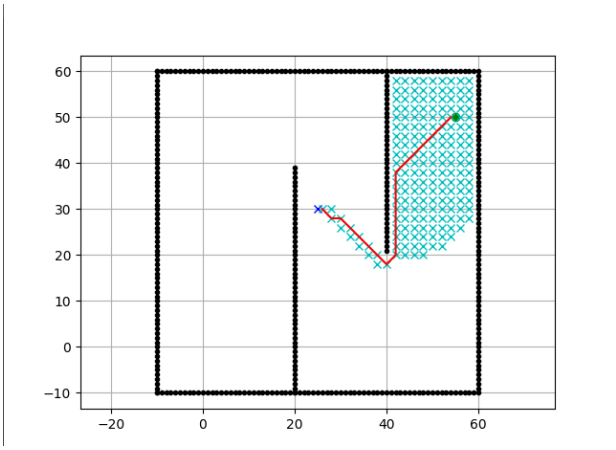


图 3-3 结果 3

第四章 总结与改进

整体来说实现了 A*路径规划算法在栅格地图中的应用，通过定义节点、计算启发函数、生成障碍物地图等步骤，成功找到起始点到目标点的最优路径，并在地图中展示了路径规划的过程。

结果表明 A*算法在栅格地图路径规划中具有良好的性能和应用前景，但对代码的实现还有改进的地方：

1.数据结构优化

使用更高效的数据结构来表示地图和节点信息，如使用 numpy 数组来表示障碍物地图。考虑使用优先队列来代替字典实现的 open 集合，以提高搜索效率。

2.参数调优：

调整启发函数的权重参数，根据实际情况优化路径规划的效果。调整机器人的运动模型和代价，根据实际机器人的运动特性进行调优。

3. 可视化优化：

添加更多的可视化功能，如显示搜索过程中的每一步扩展的节点。