



南開大學
Nankai University

人工智能技术实验 实验报告

实验名称: A*算法的实现

姓名: 张恒硕

学号: 2212266

专业: 智能科学与技术

人工智能学院

2024 年 10 月

目录

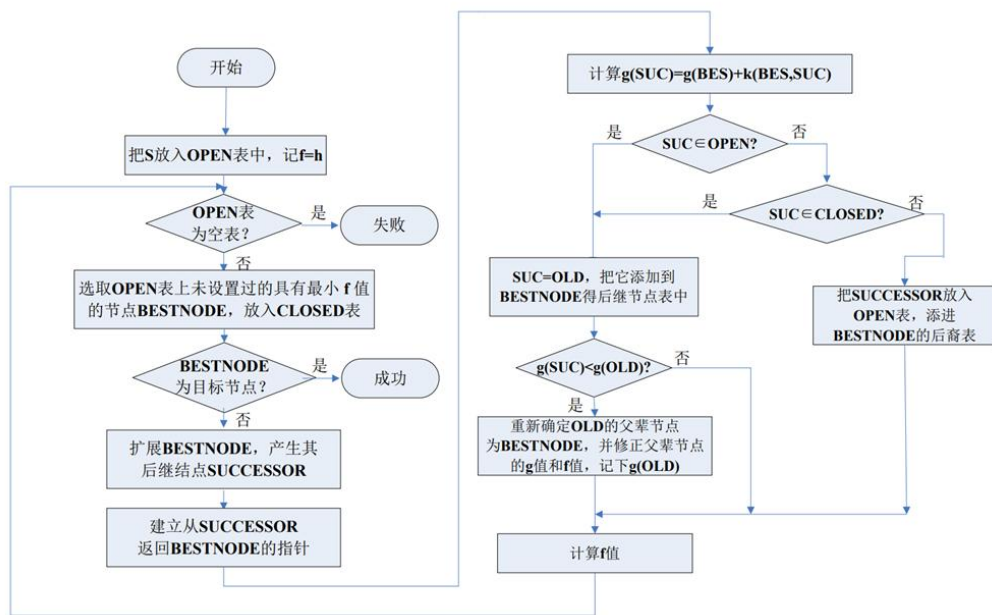
目录	2
一、 问题简述	3
二、 实验目的	4
三、 实验内容	4
四、 编译环境	4
五、 实验步骤与分析	4
1. 节点类	4
2. 地图类	6
3. 输入信息检查	7
4. 迭代过程与构建路径	9
5. 绘图	12
6. 主函数与主程序	16
六、 实验结果	19
1. A*与 Dijkstra	19
2. 可视化展示	26
七、 分析总结	27
1. 成果总结	27
2. 问题分析与改进思路	27

一、问题简述

A*算法一种典型的启发式搜索算法，是静态路网中求解最短路径最有效的方法之一。启发式搜索又称为有信息搜索，它是利用问题拥有的启发信息来引导搜索，达到减少搜索范围、降低问题复杂度的目的，通常拥有更好的性能。A*算法关键部分为启发函数：

$$F = G + H$$

其中，G 为从起点 A 移动到目前方格的代价，H 是该方格到终点 B 的估算成本，F 为寻找下一个遍历节点的依据。



伪代码：

初始化 open_list 和 close_list;

* 将起点加入 open_list 中，并设其移动代价为 0;

* 如果 open_list 不为空，则从 open_list 中选取移动代价最小的节点 n:

* 如果节点 n 为终点，则：

* 从终点开始逐步追踪 parent 节点，一直达到起点；

* 返回找到的结果路径，算法结束；

* 如果节点 n 不是终点，则：

* 将节点 n 从 open_list 中移除，并加入 close_list 中；

* 遍历节点 n 所有的邻近节点：

* 如果邻近节点 m 在 close_list 中，则：

* 跳过，选取下一个邻近节点

* 如果邻近节点 m 在 open_list 中，则：

- * 计算起点经 n 到 m 的 g 值
- * 如果此时计算出来的 g 值小于原来起点经 m 的原父节点到 m 的 g 值:
 - * 更新 m 的父节点为 n ，重新计算 m 的移动代价 $f=g+h$
 - * 否则，跳过
- * 如果邻近节点 m 不在 `open_list` 也不在 `close_list` 中，则：
 - * 设置节点 m 的 `parent` 为节点 n
 - * 计算节点 m 的移动代价 $f=g+h$
 - * 将节点 m 加入 `open_list` 中

二、实验目的

1. 掌握图搜索算法思路 and 流程，理解无信息搜索与有信息搜索的区别；
2. 对于启发式搜索，理解 A*算法估值函数的选取对算法性能的影响；
3. 对于启发式搜索，理解 A*算法求解流程和搜索顺序；
4. 深入理解对图形化界面设计。

三、实验内容

1. 实现 Dijkstra 算法 ($H=0$)，以带有障碍物的二维地图为基础，寻找到达目标点的最短路径；
2. 实现 A*算法，以带有障碍物的二维地图为基础，寻找到达目标点的最短路径；
3. 对 Dijkstra 算法和 A*算法的性能进行比较分析；
4. 对实验进行图形化界面设计，展示搜索过程和最优路径。

四、编译环境

Pycharm: python3.11+matplotlib 包+tkinter 包+numpy 包。

五、实验步骤与分析

以下按照代码实现的逻辑逐步分析，与实际代码的顺序略有不同，具体请以 `A_star.py` 文件为准。另外，在比较 Dijkstra 算法时，可以参加 `Dijkstra.py` 文件。

1. 节点类

● 内容与原理

首先定义节点类，对应格子信息。一个节点有横纵坐标值、起点到节点的实际代价 G 、节点到终点的估计代价 H 、总代价 F ($loss$)、父节点等信息。另外重构 “`==`”，使其表示两个横纵坐标都相同的格子是同一个格子。

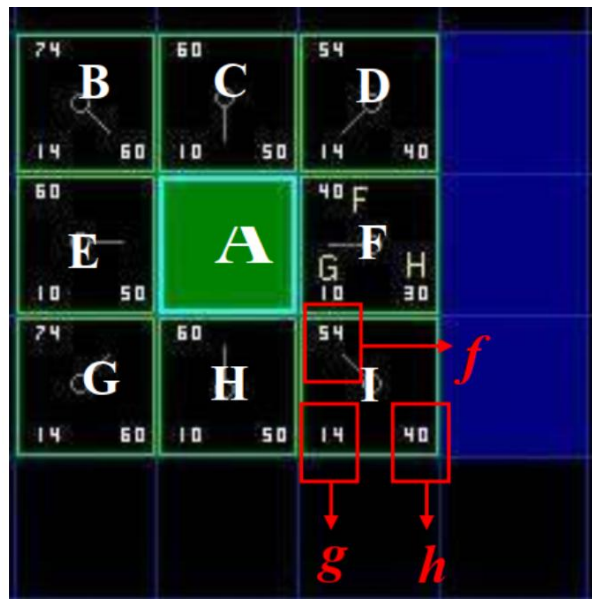


图 5.1 节点类

● 代码

节点类

class Node:

def __init__(self, x, y):

self.x, self.y = x, y

self.G = 0 # 起点到当前节点的实际代价（欧式距离）

self.H = 0 # 当前节点到终点的估计代价（曼哈顿距离）

self.parent = None # 储存父节点

重载==

def __eq__(self, other):

if other and self:

return self.x == other.x and self.y == other.y

else:

return not (self or other)

```
# 节点总代价

def loss(self):

    return self.G + self.H
```

- 节点的横纵坐标值。
- 节点的代价值，由起点到当前节点的实际代价（欧式距离）G 和当前节点到重点的估计代价（曼哈顿距离）H 组成，loss 函数可以返回节点的总代价 F，即 G+H。
- 节点的父节点，标记路径中节点的上一个节点。
- 重载 “==”，表示两个横纵坐标都相同的节点是同一节点。

2. 地图类

● 内容与原理

地图类将给定的地图矩阵、起点、终点信息导入，使每个格子都是一个节点。另定义返回节点信息和判断是否在开闭表的函数。

● 代码

```
# 地图类

class Map:

    def __init__(self, in_map, target_position):

        self.r = len(in_map)

        self.c = len(in_map[0])

        self.map = []

        for i in range(self.r):

            self.map.append([])

            for j in range(self.c):

                self.map[i].append(Node(i, j))
```

```

        self.map[i][j].H = abs(target_position.x - i) + abs(target_position.y - j)

def Getnode(self, node):

    return self.map[node.x][node.y]

def in_open_list(self, pos):

    return any(open_list_pos == pos for open_list_pos in open_list)

def in_close_list(self, pos):

    return any(close_list_pos == pos for close_list_pos in close_list)

```

- 导入包含 0、1 信息的地图矩阵和起点、终点，0 代表通路，1 代表障碍物。提取矩阵的行列。
- 初始化二维列表，将地图信息以节点的形式输入，并计算各节点的 H 值。
- 定义返回节点信息的函数，其用于提取地图中某个节点的具体信息。
- 定义判断是否在开闭表中的函数。

3. 输入信息检查

● 内容与原理

在算法的主体部分中，首先对输入信息进行核查，如果起点、终点是无法通行的障碍物，则弹窗警告。

● 代码

```

# 检查起点、终点

def check():

    flag = 1

    if in_map[start_position.x][start_position.y] == 1:

        root = tkinter.Tk()

        root.withdraw()

        messagebox.showwarning("输入不合法", "起点为障碍！")

```

```
root.destroy()

flag = 0

return flag

if in_map[target_position.x][target_position.y] == 1:

    root = tkinter.Tk()

    root.withdraw()

    messagebox.showwarning("输入不合法", "终点为障碍！")

    root.destroy()

    flag = 0

    return flag
```

- 首先判断起点的输入地图信息是否为 0，不是则退出程序，并弹窗警告。

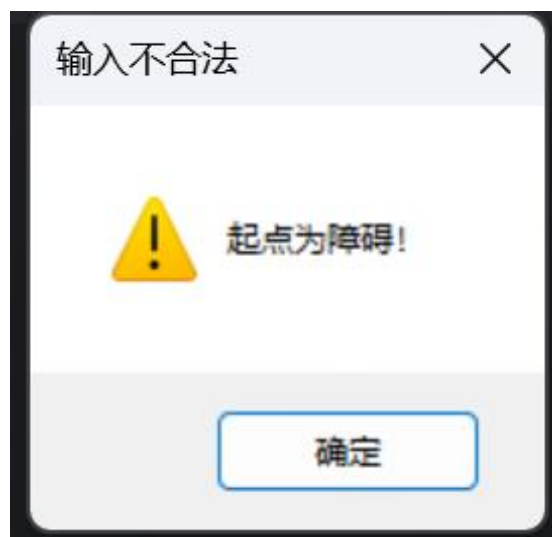


图 5.3-1 起点为障碍物

- 其次判断终点的输入地图信息是否为 0，不是则退出程序，并弹窗警告。

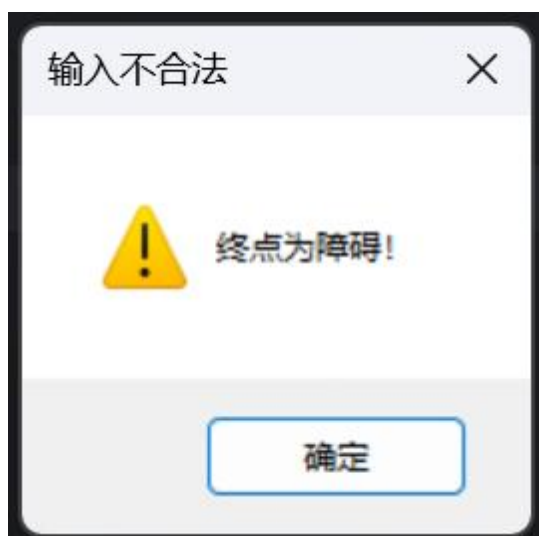


图 5.3-2 终点为障碍物

4. 迭代过程与构建路径

● 内容与原理

在每一次探索的迭代过程中，首先判断是否能继续寻找，即开表是否为空，不能则终止并弹窗。如果非空，在开表中找到代价最小的节点作为下一个节点，判断其是否是终点，是则终止。不是终点的话，将该节点从开表移入闭表，遍历周围八个点，将不在开闭表中的点加入开表，然后在八邻域中进行比较，找到当前最短路径。为了实时更新绘图，需要不断构建实时路径。

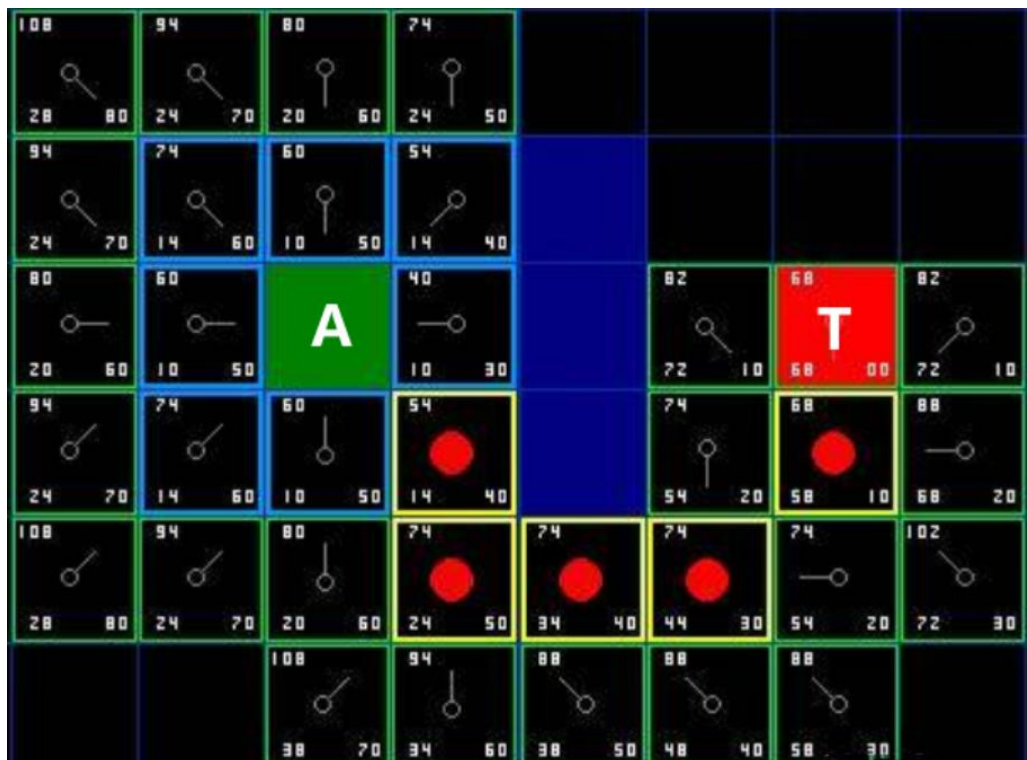


图 5.4-1 路径迭代过程

- 代码

```
# 迭代

def update(frame):

    if not open_list:                                     # 如果 open_list 空，则停止动画

        animate.event_source.stop()

        root = tkinter.Tk()

        root.withdraw()

        messagebox.showwarning("没有路径", "没有从起点到终点的路径！")

        root.destroy()

    return lines

current_position = min(open_list, key=lambda elem: map.Getnode(elem).loss())

if current_position == target_position:                 # 如果到达终点，则停止动画

    animate.event_source.stop()

    path(current_position)

    return lines

open_list.remove(current_position)

close_list.append(current_position)

# 遍历邻居

for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]:

    ni, nj = current_position.x + dx, current_position.y + dy

    # 在范围内并且不是障碍物

    if 0 <= ni < map.r and 0 <= nj < map.c and in_map[ni][nj] == 0:
```

```

        new_G = map.Getnode(current_position).G + (1.414 if (dx != 0 and dy != 0)
else 1)

    # 不在两个表内，加入开表

    if not map.in_close_list(Node(ni, nj)) and not map.in_open_list(Node(ni, nj)):

        open_list.append(Node(ni, nj))

        map.map[ni][nj].parent = current_position

        map.map[ni][nj].G = new_G

    # 在开表并且更近，更新

    elif map.in_open_list(Node(ni, nj)) and map.map[ni][nj].G > new_G:

        map.map[ni][nj].parent = current_position

        map.map[ni][nj].G = new_G

    path(current_position)

    return lines

# 构建路径

def path(current_position):

    # 回溯并反转

    path = []

    while current_position:

        path.append(current_position)

        current_position = map.Getnode(current_position).parent

    path.reverse()

    # 转存

```

```

path_set = set()

for pos in path:

    path_set.add((pos.x, pos.y))

Draw(map, path_set)

add_legend(ax)

```

- 如果起点、终点并不相连，则在迭代一定次数后，开表将没有可以探索的点，此时终止运行并弹窗警告。

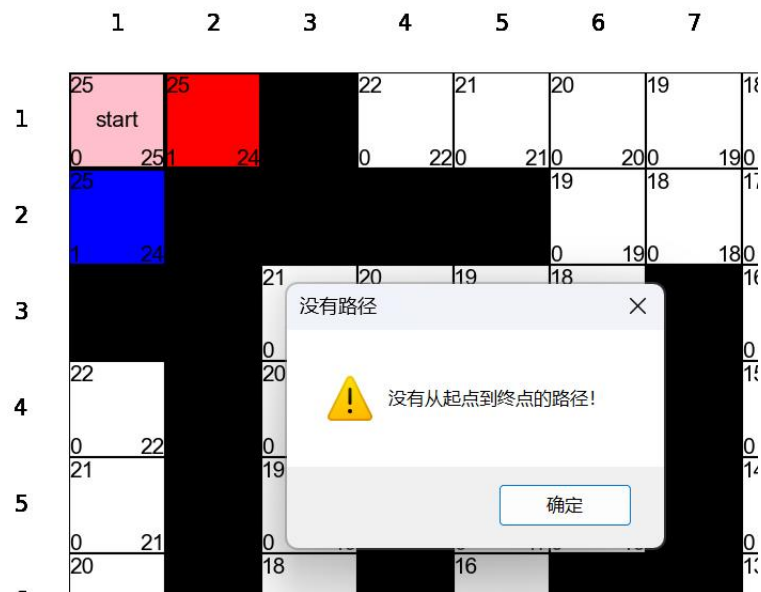


图 5.4-2 起点、终点间没有通路

- 如果到达终点，则终止，并画出最终路径。
- 其他情况则在八邻域中寻找更近的路径，画出迭代过程。
- 每次更新图像，首先调用构建路径的函数，由当前节点往回找到当前的完整路径，并将路径翻转过来，放到列表里。

5. 绘图

● 内容与原理

绘制实时图像，并给出图例。

● 代码

```
# 绘图
```

```
def Draw(map, path_set):
```

```
    ax.clear()
```

```
    # 设置等比例 x、y 轴范围，关闭坐标轴，翻转图像
```

```
    ax.set_xlim(0, map.c)
```

```
    ax.set_ylim(0, map.r)
```

```
    ax.set_aspect('equal')
```

```
    ax.set_axis_off()
```

```
    plt.gca().invert_yaxis()
```

```
    # 设置字体
```

```
    font = FontProperties(size=10, family='Arial')
```

```
    # 绘制格子
```

```
    for i in range(map.r):
```

```
        for j in range(map.c):
```

```
            # 坐标轴
```

```
            ax.text(-0.5, i + 0.5, str(i + 1), ha='center', va='center')
```

```
            ax.text(j + 0.5, -0.5, str(j + 1), ha='center', va='center', rotation=0)
```

```
            if in_map[i][j] == 1:
```

```
# 障碍物格子填黑
```

```
                rect = patches.Rectangle((j, i), 1, 1, linewidth=1, edgecolor='black',
```

```
                facecolor='black')
```

```
            else:
```

```

        if (i, j) in path_set:

# 路径格子填红

        rect = patches.Rectangle((j, i), 1, 1, linewidth=2, edgecolor='black',
facecolor='red')

        elif map.in_open_list(Node(i, j)) and (i, j) not in path_set:

# open_list 格子填绿

        rect = patches.Rectangle((j, i), 1, 1, linewidth=1, edgecolor='black',
facecolor='green')

        elif map.in_close_list(Node(i, j)) and (i, j) not in path_set:

# close_list 格子填蓝

        rect = patches.Rectangle((j, i), 1, 1, linewidth=1, edgecolor='black',
facecolor='blue')

        else:

# 非路径非障碍物格子填白

        rect = patches.Rectangle((j, i), 1, 1, linewidth=1, edgecolor='black',
facecolor='white')

        ax.add_patch(rect)

        node = map.map[i][j]

        # 显示代价

        ax.text(j, i + 0.9, round(node.G, 1), horizontalalignment='left',
verticalalignment='center', fontproperties=font)

        ax.text(j + 1, i + 0.9, node.H, horizontalalignment='right',

```

```

verticalalignment='center', fontproperties=font)

        ax.text(j, i + 0.15, round(node.loss(), 1), horizontalalignment='left',
verticalalignment='center', fontproperties=font)

# 起点和终点

    rect = patches.Rectangle((start_position.y, start_position.x), 1, 1, linewidth=2,
edgecolor='black', facecolor='pink')

    ax.add_patch(rect)

    ax.text(start_position.y + 1 / 2, start_position.x + 1 / 2, 'start',
horizontalalignment='center', verticalalignment='center', fontproperties=font)

    rect = patches.Rectangle((target_position.y, target_position.x), 1, 1, linewidth=2,
edgecolor='black', facecolor='yellow')

    ax.add_patch(rect)

    ax.text(target_position.y + 1 / 2, target_position.x + 1 / 2, 'end',
horizontalalignment='center', verticalalignment='center', fontproperties=font)

# 图例

def add_legend(ax):

    start_patch = patches.Patch(color='pink', label='Start')

    end_patch = patches.Patch(color='yellow', label='End')

    path_patch = patches.Patch(color='red', label='Path')

    open_patch = patches.Patch(color='green', label='Open List')

    closed_patch = patches.Patch(color='blue', label='Closed List')

    obstacle_patch = patches.Patch(color='black', label='Obstacle')

```

```

# 添加到图像底部

ax.legend(handles=[start_patch, end_patch, path_patch, open_patch, closed_patch,
obstacle_patch],

          loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True,
shadow=True, ncol=6)

```

- 绘制坐标轴，设置字体。
- 遍历所有行列，标明坐标轴的行列数，并为每一个格子上色、赋值。
- ◆ 颜色视格子的属性而定，其中起点为粉，终点为黄，路径为红，开
表点为绿，闭表点为蓝，障碍物格子为黑，其他格子为白。



图 5.5-1 图例

- ◆ 赋值中，一个格子的左下角为 G 值，右下角为 H 值，左上角为总代
价值。

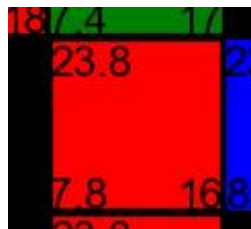


图 5.5-2 格子表数示例

- 在图像底部添加如上图例。

6. 主函数与主程序

● 内容与原理

信息的导入和函数的执行。

● 代码

```

# A*算法

def SearchPath(in_map, start_position, target_position):

    ~

```



```

# 主函数

start_time = time.time()

flag = check()

if flag == 0:

    sys.exit()

map = Map(in_map, target_position)

open_list.append(start_position)

fig, ax = plt.subplots(figsize=(map.c * 0.8, map.r * 0.8))

lines = []

animate = animation.FuncAnimation(fig, update, frames=np.arange(0, 100),
interval=200, blit=False)

plt.show()

end_time = time.time()

root = tkinter.Tk()

root.withdraw()

messagebox.showwarning("找到路径", f"成功找到解，用时{end_time - start_time}s")

root.destroy()

# 主程序

if __name__ == "__main__":

    in_map = [

        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

```

```

[0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0],

[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0],

[0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0],

[0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0],

[0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0],

[0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],

[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],

[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

]

start_position = Node(0, 0)

target_position = Node(12, 13)

path_set = set()

open_list = []

close_list = []

SearchPath(in_map, start_position, target_position)

```

- A*算法函数执行部分，首先判断输入信息的合法性，其后构建地图，然后设定图像动画。在指定窗口大小时，设定为根据输入矩阵的规模而变化，保证单个格子的大小不变。在算法运行完成后，展示完最后一个图像并关掉图像后，会弹窗显示代码运行时间。

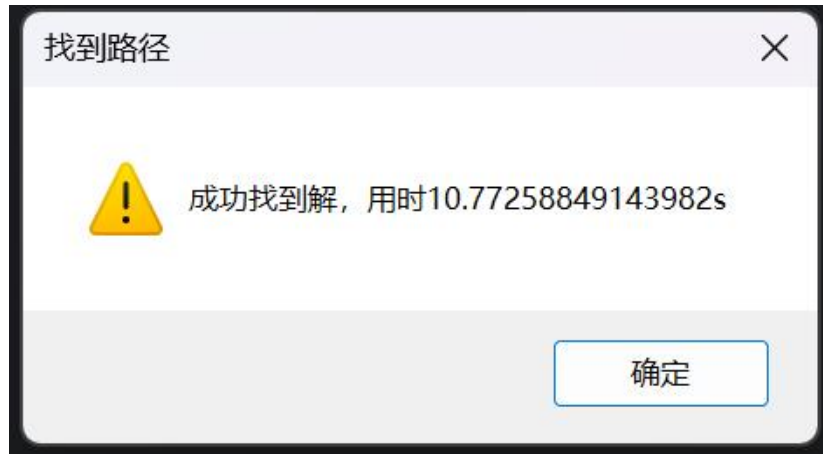
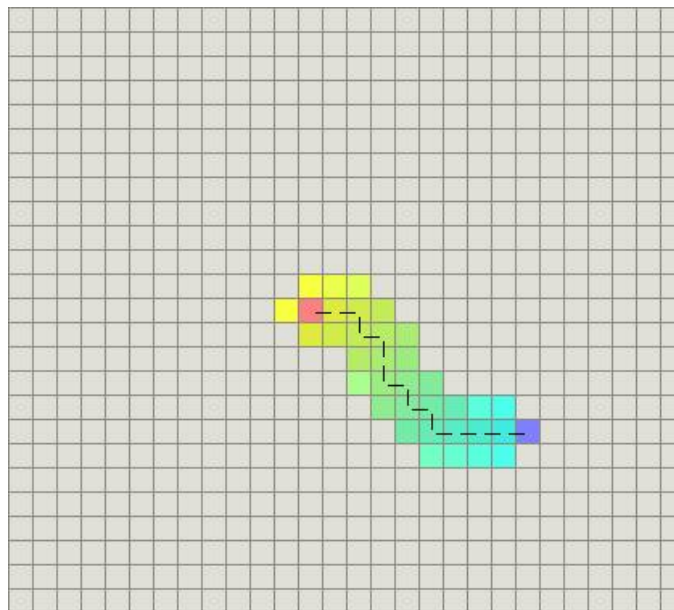


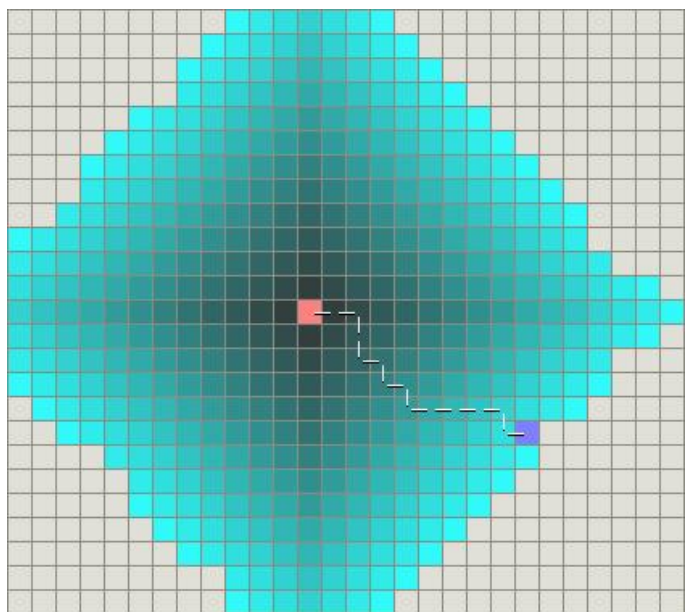
图 5.6 代码运行时间

- 主程序部分给定输入的矩阵信息并指定起点、终点，初始化开闭表，执行 A*算法函数。

六、实验结果

1. A*与 Dijkstra





图组 6.1 A*算法和 Dijkstra 算法的搜索原理

A*算法和 Dijkstra 算法都是用于寻找图中两点间最短路径的算法，前者是一种启发式搜索算法，使用了两种成本度量。当启发式部分 H 恒为 0 时，退化为后者。前者比后者快，但可能受 H 度量的影响，找不到最优解；而后者虽然效率低，但能保证找到全局最优解。以下对二者进行实验比较：

表 1 A*与 Dijkstra 的对比 1

输 入 信 息 1	<div><div>in_map = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0], [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0], [0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0], [0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0], [0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0], [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],</div></div>
--------------	---

[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],

[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],

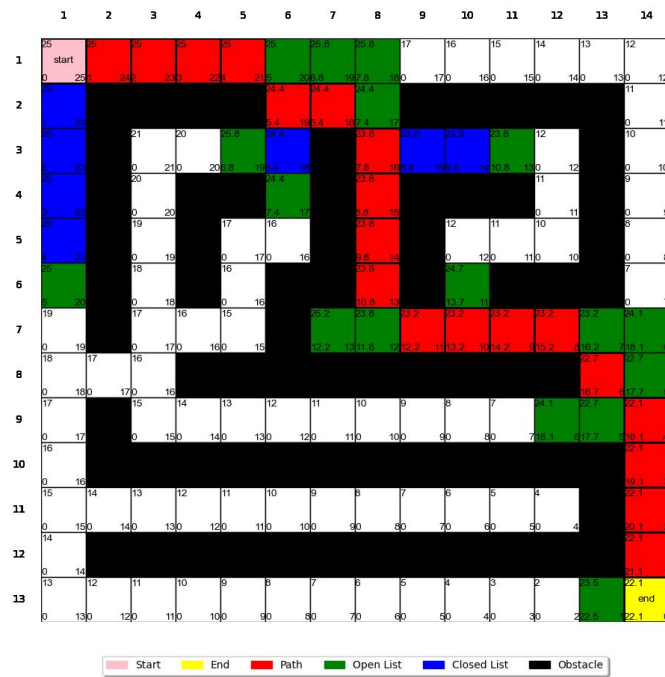
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

]

start_position = Node(0, 0)

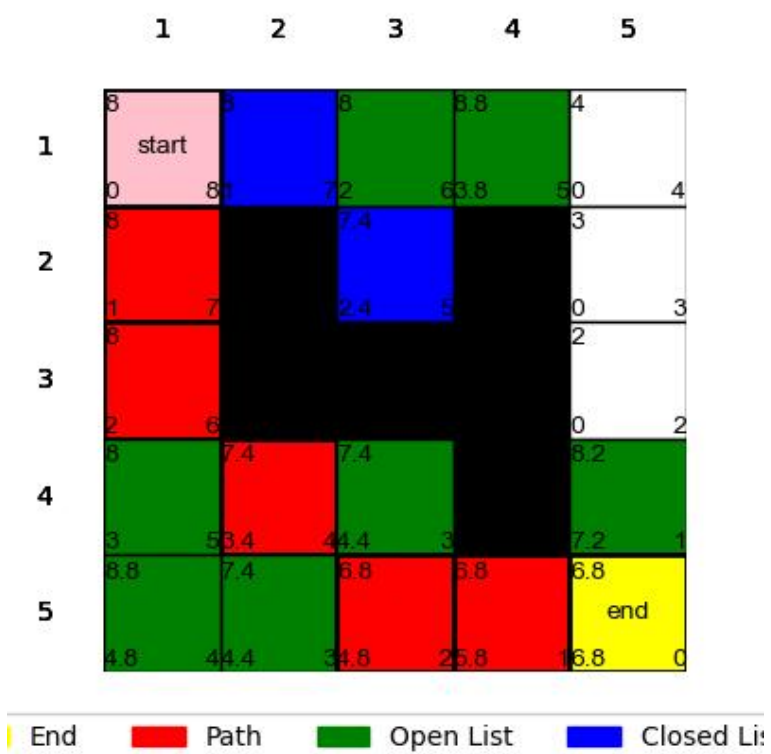
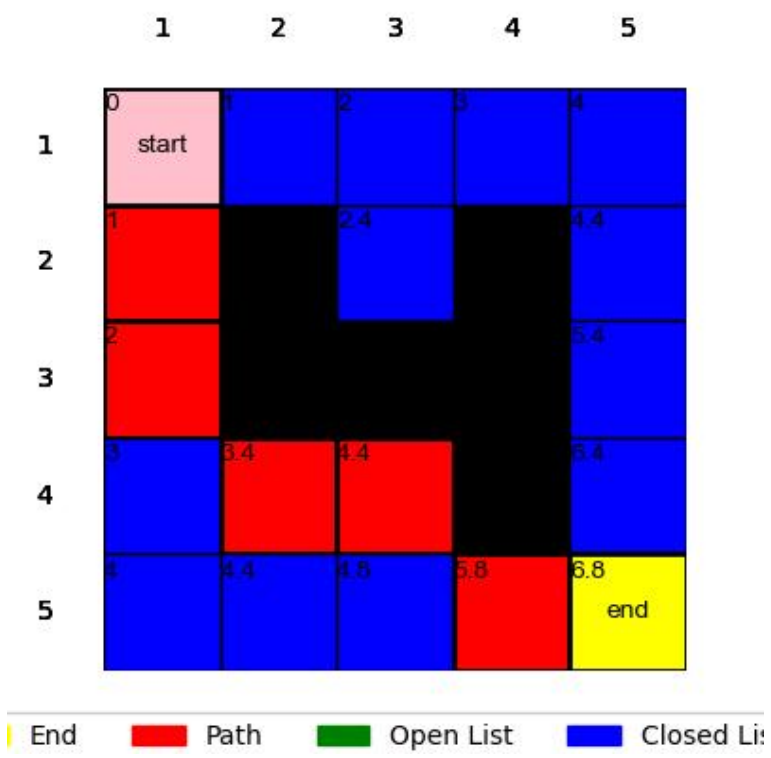
target_position = Node(12, 13)

A*



13.67s

<p>Dijkstra</p>	<div data-bbox="470 264 1134 936"> </div> <p>39.78s</p>	
<p>输入信息 2</p>	<pre> in_map = [[0, 0, 0, 0, 0], [0, 1, 0, 1, 0], [0, 1, 1, 1, 0], [0, 0, 0, 1, 0], [0, 0, 0, 0, 0],] start_position = Node(0, 0) target_position = Node(4, 4) </pre>	

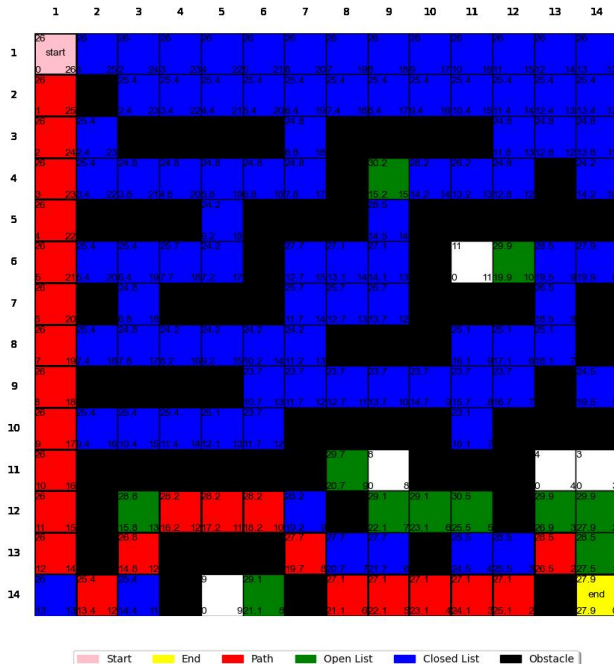
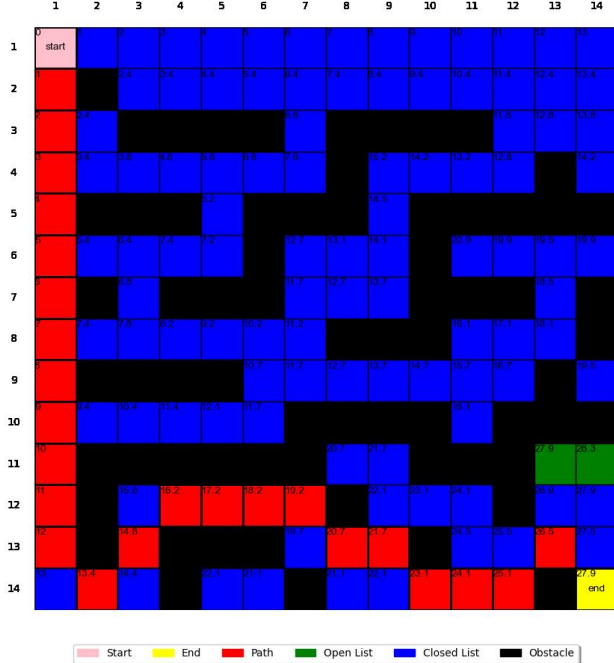
<p>A*</p>	<div data-bbox="383 212 1157 963"> <div> <div>12345</div> <div> <div>1</div> <div>2</div> <div>3</div> <div>4</div> <div>5</div> </div> </div>  <div> <div>End</div> <div>Path</div> <div>Open List</div> <div>Closed Li</div> </div> </div>	<p>5.36s</p>
<p>Dijkstra</p>	<div data-bbox="383 1003 1157 1758"> <div> <div>12345</div> <div> <div>1</div> <div>2</div> <div>3</div> <div>4</div> <div>5</div> </div> </div>  <div> <div>End</div> <div>Path</div> <div>Open List</div> <div>Closed Li</div> </div> </div>	<p>7.53s</p>
<p>分析：对比上面两组实验可以发现，作为启发式的 A*算法要比 Dijkstra 算法运行更快、遍历更少，验证了上述结论。从最终找到的路径来看，二者在这两个问题中找到的路径都有一定差别，但没有本质上的区别，实际代价是一致的。</p>		

以下测试在 H 与实际值不符时的情况，即从起点出发，没有一直向终点方

向行进的道路的情况。在这种情况下，A*算法的效率将大大下降。

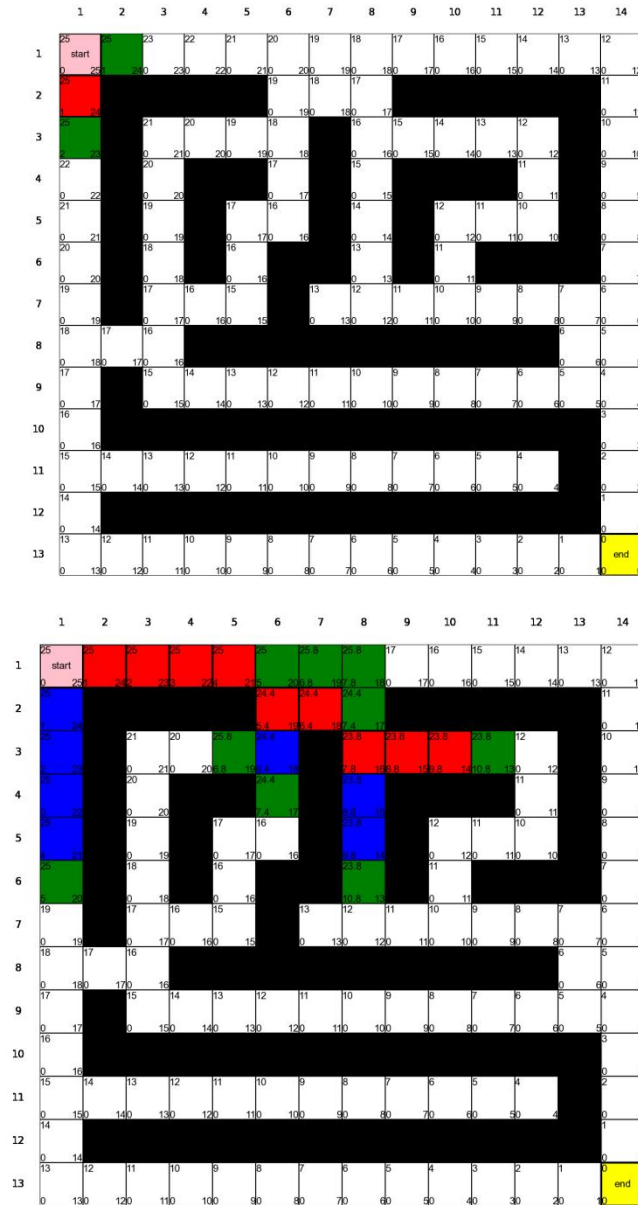
表 2 A*与 Dijkstra 的对比 2

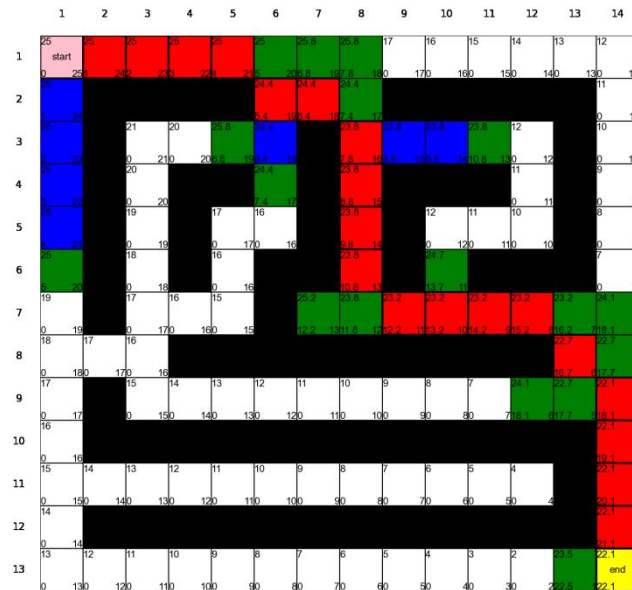
输 入 信 息	<pre>in_map = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0], [0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0], [0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1], [0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1], [0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0], [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0], [0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]]</pre> <pre>start_position = Node(0, 0) target_position = Node(13, 13)</pre>
------------	--

<p>A*</p>	 <p>14x14 grid visualization showing the A* search process. The grid includes a Start cell (pink), End cell (yellow), Path (red), Open List (green), Closed List (blue), and Obstacle cells (black). The search process is shown with numerical values on the grid cells.</p>	<p>42.36s</p>
<p>Dijkstra</p>	 <p>14x14 grid visualization showing the Dijkstra search process. The grid includes a Start cell (pink), End cell (yellow), Path (red), Open List (green), Closed List (blue), and Obstacle cells (black). The search process is shown with numerical values on the grid cells.</p>	<p>43.73s</p>
<p>分析：对比以上这组实验的结果可以发现，面对这种情况，A*算法的效率被大大限制，其经历了反复探索胡同区域的试错阶段，最终耗时很久才找到终点。与Dijkstra 算法相比，其启发式的优点被磨平了，二者访问的格子数相差不多，运行时间也基本一致，结果也一样。</p>		

2. 可视化展示

由于在前面已经展示了程序运行的多种情况的截图，这里仅展示一次运行中的几个不连续截图，这里截取的并不全，下面的图例部分略去了。





图组 6.2 一次运行中的不连续截图

七、分析总结

1. 成果总结

本次实验成功实现了 A*寻路问题的求解和相应的简单可视化程序，并且考虑了一定量的特殊情况，程序的稳定性较高。另将 A*算法与 Dijkstra 算法进行横向对比，对启发性算法带来的改进有了更深入的理解。

2. 问题分析与改进思路

- 可视化与交互性待改进：并未实现可交互的可视化窗口，没有具体的执行按钮。对于初始信息的输入，不能在交互界面上进行指定障碍物、起点、终点等行为，不能进行步进，只能在代码中给定信息，一次运行完。这些并不是实验的重点，因此没有力求完成。
- 运行时间窗口显示逻辑问题：最后的运行时间只能在关闭最后一个窗口后弹窗显示，且时间是代码开始运行到弹窗前，并不是真正的代码运行时间。这是因为没有找到正确的插入代码截止时间点函数的位置，但并不影响。
- 在一定的情况下，A*算法是可能无法找到最优解的，以上实验部分并没有举出相应的例子。