lab4：盲目搜索与启发式搜索

姓名	学号	专业	教学班级
张航悦	19335262	计算机科学与技术	19计科2班

一、 算法原理

盲目搜索

盲目搜索策略都采用固定的规定来选择下一需要被拓展的状态，这些策略不考虑任何与要解决的问题领域相关的信息。其搜索规则不会随着要搜索解决问题的变化而变化。

常见的盲目搜索策略有：BFS，DFS，一致代价搜索，深度受限搜索等。

• 一致代价搜索

在本次实验中，我选择的盲目搜索策略为一致代价搜索。

一致代价搜索是在广度优先搜索上进行拓展的。算法流程为，将边界队列中的节点按路径的成本升序排序，每次选择成本最小的节点进行拓展。具有完备性和最优性。

代码上我们可以利用队列来实现。将队列中的节点按路径代价升序排序，每次取队头进行拓展，当有新的节点入队时就要重新排序。

在本次走迷宫的实验中，由于每步的代价相同，一致代价搜索退化为BFS。

启发式搜索

对于一个具体的问题，构造一个专用于该领域的启发式函数$h(n)$，该函数用于估计从节点n到目标节点的成本。不同的问题其对应的启发式函数也不同。

• A*搜索

A^*算法可理解为一致代价搜索的启发式版本。

在A^*算法中定义了评价函数$f(n)=g(n)+h(n)$。其中$g(n)$初始节点到达节点n的路径成本。$h(n)$是从n节点到达目标节点的成本的启发式估计值。因此$f(n)$是经过节点n从初始节点到达目标节点的路径成本的估计值。

其算法流程为，从初始节点开始，不断查询周围可达节点的状态并计算它们的评价函数$f(n)$值，选择$f(n)$值最小的节点进行拓展（这里就和一致代价搜索很像），并更新边界队列中的节点的$g(n)$值，直到达到目标节点。

• 启发函数

- 可采用的启发函数
  - 曼哈顿距离

    在正方形网格中，允许向4邻域的移动，使用曼哈顿函数。

    $$h(n)=D*(abs(node.x-goal.x)+abs(node.y-goal.y))$$

    其中`node`表示当前节点，`goal`表示目标节点。D的设计是为了距离衡量单位与启发函数相匹配。可以设置为方格间移动的最低代价。

  - 欧式距离

    $$h(n)=D*\sqrt{(node.x-goal.x)^2+(node.y-goal.y)^2}$$

  - 切比雪夫距离

    在正方形网络中，允许向8邻域的移动，使用对角线距离。

    $$h(n)=D*max(abs(node.x-goal.x),abs(node.y-goal.y))$$

- 启发式函数性质
  - 可采纳性

    对于所有节点n，满足$h(n)\leq h^*(n)$时，$h(n)$是可采纳的。可采纳的启发式函数低估了当前节点到目标节点的成本，使得实际成本最小的最优路径能够被选上。

  - 单调性

    对于任意节点n_1和n_2，若

    $$h(n_1)\leq c(n_1\rightarrow n_2)+h(n_2)$$

    则$h(n)$具有一致性/单调性。一致性保证了，A^* 每次选择的到当前节点的路径就是到该节点的最优路径。

  - 总结

    可采纳性在不采用环检测剪枝的条件下，意味着可以检测到最优解。但只要具有一致性，就能在进行环检测之后仍然保持最优性。

算法性能评价

算法性能可从完备性、最优性、时间复杂度、空间复杂度四个角度来评估。

下对比UCS和A^*算法性能。

算法\性能	完备性	最优性	时间复杂度	空间复杂度
UCS	是	是	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^{1+\lfloor C^*/e \rfloor})$
A^*	取决于$h(x)$	取决于$h(x)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^{1+\lfloor C^*/e \rfloor})$

二、 伪代码

- UCS搜索

```
move = [(0, 1), (0, -1), (-1, 0), (1, 0)] //用于表示移动的四个方向
Function USCsearch(maze,s,e):
//input:maze迷宫
//      s起点
//      e终点
//output:是否存在路径

    vis := 初始为0的矩阵 //用于记录当前某点是否被访问过
    q := []
    q.push(s)
    while 1:
        curnode := q.pop()
        if curnode == e then
            return True

        vis[curnode]:= 1
        for i in move do //遍历四个移动方向
            temp_node=curnode+i

            if vis[temp_code]==0 and maze[temp_node]==0 then
                if temp_node in q and temp_node.step<q[temp_node].step do
                    q[temp_node].step:=curnode.step+1

                else q.append(temp_node)
        end
    end

    return False
```

- A^*搜索

  A^*搜索的伪代码和USC搜索的伪代码几乎相同，二者思路相仿。不同点就是USC搜索的队列排序是按step升序排序，但A^*搜索的队列排序是按step+h值升序排序。

```
move = [(0, 1), (0, -1), (-1, 0), (1, 0)] //用于表示移动的四个方向
```

```
Function USCsearch(maze,s,e):
//input:maze迷宫
//      s起点
//      e终点
//output:是否存在路径

    vis := 初始为0的矩阵 //用于记录当前某点是否被访问过
    q := [] //一个优先队列
    q.push(s)
    while 1:
        curnode := q.pop()
        if curnode == e then
            return True

        vis[curnode]:= 1
        for i in move do //遍历四个移动方向
            temp_node=curnode+i

            if vis[temp_code]==0 and maze[temp_node]==0 then
                if temp_node in q and temp_node.step<q[temp_node].step do
                    q[temp_node].step:=curnode.step+1

                else
                    temp_node.h=h(temp_node,e)  //计算h值
                    q.append(temp_node)
        end
    end

    return False
```

三、 关键代码展示

1. 一致代价搜索

- 节点类定义

```
class node:
    def __init__(self, pos, parent=None, step=0):
        self.pos = pos
        self.step = step
        self.parent = parent

    def __lt__(self, other):  # 重定义'<' 以路径代价为指标进行队列排序
        return self.step < other.step
```

- USC搜索

  由于每次加入新的节点，队列都需要进行重新排序，在这里利用一个最小堆来维护实现。

在这里利用了一个二维列表vis记录拓展过的节点位置，未被拓展为0，已拓展为1，在搜索的过程中若该位置已为1则不拓展，相当于实现了**环检测**。

```
def USCsearch(maze, maze2, s, e):
    """
    USC搜索
```

```
    :param maze: 原本迷宫
    :param s: 起点
    :param e: 终点
    :return:
    """
    count = 1  # 记录时间复杂度
    length = 1  # 记录空间复杂度
    row = len(maze)
    col = len(maze[0])
    vis = [[0] * col for i in range(row)]  # 标记该位置是否已访问过 未访问0
    q = []
    maze_2 = copy.deepcopy(maze)
    heapq.heappush(q, s)
    while 1:
        curnode = heapq.heappop(q)

        if curnode.pos == e.pos:
            print('find')
            print('时间复杂度为%d' % count)
            print('空间复杂度为%d' % length)
            print('距离为%d' % curnode.step)
            return curnode

        vis[curnode.pos[0]][curnode.pos[1]] = 1
        maze2[curnode.pos[0]][curnode.pos[1]] = 0.6  # 用于记录探索过的点
        count += 1

        for i in range(4):  # 遍历四个方向移动
            temp_x = curnode.pos[0] + move[i][0]
            temp_y = curnode.pos[1] + move[i][1]
            # 若该点在界内且 有路可走 且此处未被访问过
            if temp_x >= 0 and temp_x < row and temp_y >= 0 and temp_y < col
and \
                    maze[temp_x][temp_y] != 1 and vis[temp_x][temp_y] != 1:
                flag = 1
                # 若该点已在边界队列中 但还未拓展 更新其成本
                for NODE in q:
                    if NODE.pos[0] == temp_x and NODE.pos[1] == temp_y and
NODE.step > curnode.step + 1:
                        NODE.step = curnode.step + 1
                        NODE.parent = curnode
                        heapq.heapify(q)
                        flag = 0
                        break
                # 若该点不在边界队列中 计算其成本 入队
                if flag:
                    temp = node([temp_x, temp_y], curnode, curnode.step + 1)
                    heapq.heappush(q, temp)
                    if len(q) > length:
                        length = len(q)  # 空间复杂度更新

```

- 画图

在这里我利用热力图来可视化搜索的过程。热力图是基于数值矩阵进行绘制的，不同的数值会被赋予不同的颜色，而这正好与本实验中迷宫的表示相同。将迷宫边界、迷宫通路、探索过的点赋予不同的数值即可利用热力图完成可视化的过程，十分便捷。

热力图绘制调用了python的`seaborn`库，需要下载安装。

1. 绘制迷宫

   迷宫边界值为1，通路值为0

```
def drawmaze(maze):
    maze2 = np.array(maze)
    sns_plot = sns.heatmap(maze2, cbar=False, cmap="Blues")
    plt.xticks([])
    plt.yticks([])
    plt.title('Maze')
    plt.show()
```

2. 绘制探索过的点

   迷宫边界值为1，被探索过的点的值为0.6

```
def drawexlored(maze):
    maze2 = np.array(maze)
    sns_plot = sns.heatmap(maze2, cbar=False, cmap="Blues")
    plt.xticks([])
    plt.yticks([])
    plt.title('Explored')
    plt.show()
```

3. 绘制迷宫通路

   通路值赋为3。在`node`节点中定义了`parent`的属性，用于记录节点的父节点，从最后一个探索节点倒序遍历即可得到迷宫的通路。

```
def drawroute(curnode, maze):
    node = curnode
    while node != None:
        temp_x = node.pos[0]
        temp_y = node.pos[1]
        maze[temp_x][temp_y] = 3
        node = node.parent
    sns_plot = sns.heatmap(maze, cbar=False, cmap="Blues")
    plt.xticks([])
    plt.yticks([])
    plt.title('Path')
    plt.show()
```

2. A^*搜索

其中A^*搜索的代码与USC搜索大部分相同，下只展示不同的部分。

- 节点类定义

  A^*搜索的节点类定义与USC几乎相同，只是增加了启发式函数的预估值这一子属性。

```python
class node:
    def __init__(self, pos, parent=None, step=0, h=0):
        self.pos = pos
        self.step = step
        self.parent = parent
        self.h = h  # 启发函数h(n)

    def __lt__(self, other):  # 重定义'<' 以f(n)=g(n)+h(n)为指标进行队列排序
        return self.step + self.h < other.step + other.h
```

- 启发式函数

```python
def heuristic(start, goal, n):
    """
    启发式函数
    :param start: 当前点位置
    :param goal: 目标点
    :param n: 决定采用何种启发式函数
    :return:
    """
    if n == 1:  # 曼哈顿距离
        return abs(start[0] - goal[0]) + abs(start[1] - goal[1])
    if n == 2:  # 欧式距离
        return np.sqrt((start[0] - goal[0]) ** 2 + (start[1] - goal[1]) **
2)
```

- A^*搜索

```python
def Astarsearch(maze, maze2, s, e):
    count = 1  # 记录时间复杂度
    length = 1  # 记录空间复杂度
    row = len(maze)
    col = len(maze[0])
    vis = [[0] * col for i in range(row)]  # 标记该位置是否已访问过 未访问0
    q = []
    maze_2 = copy.deepcopy(maze) # 用于标记哪些点被访问过
    heapq.heappush(q, s)
    while 1:
        curnode = heapq.heappop(q)

        if curnode.pos == e.pos:
            print('find')
            print('时间复杂度为%d' % count)
            print('空间复杂度为%d' % length)
            print('距离为%d' % curnode.step)
            return curnode

        vis[curnode.pos[0]][curnode.pos[1]] = 1
        maze2[curnode.pos[0]][curnode.pos[1]] = 0.6  # 用于记录探索过的点
        count += 1

        for i in range(4):  # 遍历四个方向移动
```

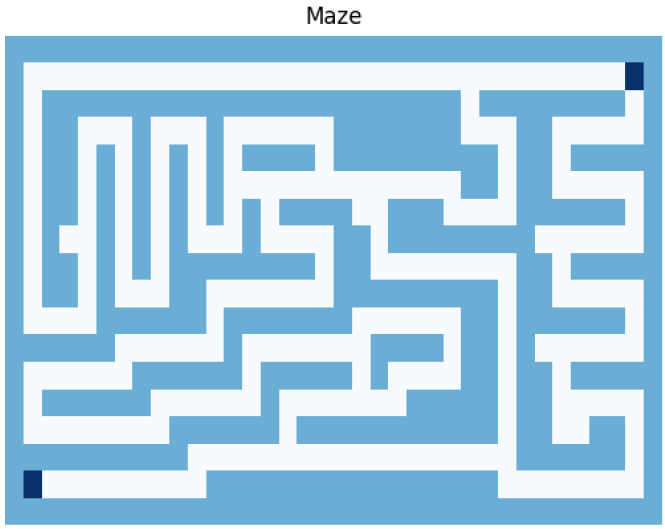
```
            temp_x = curnode.pos[0] + move[i][0]
            temp_y = curnode.pos[1] + move[i][1]

            # 若该点在界内且 有路可走 且此处未被访问过
            if temp_x >= 0 and temp_x < row and temp_y >= 0 and temp_y < col
and \
                    maze[temp_x][temp_y] != 1 and vis[temp_x][temp_y] != 1:
                flag = 1
                # 若该点已在边界队列中 但还未拓展 更新其g(n)值 并更新队列的排序
                for NODE in q:
                    if NODE.pos[0] == temp_x and NODE.pos[1] == temp_y and
NODE.step > curnode.step + 1:
                        NODE.step = curnode.step + 1
                        NODE.parent = curnode
                        heapq.heapify(q)
                        flag = 0
                        break
                # 若该点不在边界队列中 计算其f值 入队
                if flag:
                    temp_h = heuristic([temp_x, temp_y], e.pos, n=1)
                    temp = node([temp_x, temp_y], curnode, curnode.step + 1,
temp_h)
                    heapq.heappush(q, temp)
                    if len(q) > length:
                        length = len(q)  # 更新

```

四、 实验结果及分析

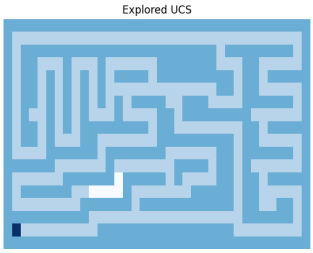
1. 搜索过程对比

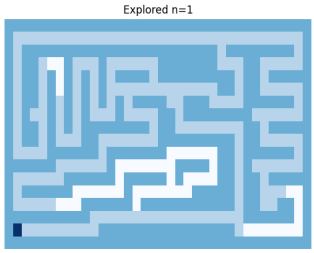
- 迷宫

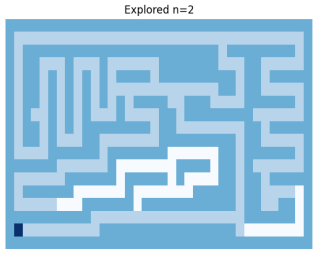
  左下角和右上角深色的点分别代表终点和起点。白色的部分为路，蓝色为墙



- 探索过的点







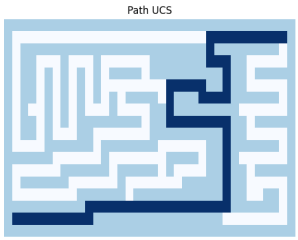
分析：

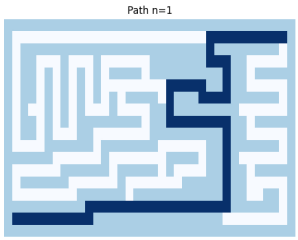
对比上图，白色路部分变蓝的即为探索过的点。

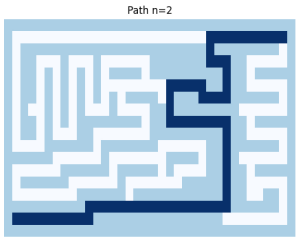
使用A^*算法时，n=1时表示启发式函数为曼哈顿距离，n=2时表示启发式函数为欧氏距离。

可以看见，**UCS**几乎把所有的点都搜索了一遍。**A*算法**搜索过的点数要明显少于UCS，在本实验条件下使用A*算法效果更佳。且使用曼哈顿距离时搜索的点数要少于使用欧氏距离时，得出在本实验下使用曼哈顿距离作为启发函数更佳。

- 最优路径







分析：可以看见三种方法得到的最优路径相同。

2. 算法性能对比

	时间复杂度	空间复杂度	完备性	最优性	运行时间
UCS	275	9	是	是	0.0043398
使用曼哈顿距离为启发函数的A*搜索	224	8	是	是	0.0073568
使用欧式距离为启发函数的A*搜索	231	8	是	是	0.0076195

时间复杂度以边界队列出队一次的时间为指标。空间复杂度以边界队列最大的长度为指标。

- 时间复杂度

其实该时间复杂度的指标可归结为**搜索过的点数**，结果与上面图所展示的相吻合，A^*搜索的搜索效率优于UCS。

但是仅以此作为时间复杂度的指标其实并不是很严谨。计算实际的运行时间，发现其实UCS的运行时间更少。可能是由于输入规模较小，A^*搜索中的启发式函数计算 和 每次更新队列时需要进行$g(n)+h(n)$的计算消耗了大部分时间，导致最终A^*搜索的实际运行时间并没有优于USC搜索。

- 空间复杂度

  发现使用启发式函数的确降低了空间复杂度，但由于地图规模较小，搜索的深度并不大，且每个点可选分枝数并不是很多，所以差距没有明显体现。

- 完备性 最优性

  在本实验下，由于走迷宫每步代价相同，其实UCS退化为了BFS，具有完备性和最优性。

  A^* 搜索的启发式函数无论是采用曼哈顿距离还是欧式距离，都具有单调性，所以具有完备性和最优性。

由于原始迷宫规模较小，没有很明显的地体现出两种算法的差异，于是我将原先的迷宫规模扩大了一倍为35×36，进行实验，结果如下。

输入迷宫为

```
1111111111111111111111111111111111
10000000000000000000000000000000S1
10111111111111111111111101111111101
10110001000100000011111100011000001
10110101010101111011111111011011111
10110101010100000000000011011000001
10110101010101011110011000011111101
10100101010001000011011111110000001
10110101011111111101100000000011011111
10110100011000000011111111011000001
10000111111011111110000011011111101
11111100000010000000111101101000000l
10000001111110111111010000110110111111
10111111000000100000001111011000001
10000000011111101111111111011001101
11111111110000000000000000011111101
10000000001111111111111111000000001
10000000001000100000100011110001001
10000000000000000000000000000000001
10111111111111111111111101111111101
10110001000100000011111100011000001
10110101010101111011111111011011111
10110101010100000000000011011000001
10110101010101011110011000011111101
10100101010001000011011111110000001
10110101011111111101100000000011011111
10110100011000000011111111011000001
10000111111011111110000011011111101
11111100000010000000111101101000000l
10000001111110111111010000110110111111
10111111000000100000001111011000001
10000000011111101111111111011001101
11111111110000000000000000011111101
1E000000000111111111111111000000001
11111111111111111111111111111111111
```

结果如下：

	时间复杂度	空间复杂度	运行时间
UCS	3878	122	0.0803378
使用曼哈顿距离为启发函数的A*搜索	560	21	0.0108347
使用欧式距离为启发函数的A*搜索	2317	127	0.0756226

可以看见将输入规模调大后，A^*搜索的优势显现了出来。特别是以曼哈顿距离为启发式函数的A^*搜索大幅降低了时间复杂度和空间复杂度，时间复杂度和空间复杂度约降低为了UCS的$1/5$，运行时间降低为了UCS的$1/8$。可以看到如果启发式函数取的不好，可能效果比一般的无信息搜索效率还要低下，因此启发式函数不仅要满足可采纳性和一致性，还需要让它尽可能不要过于松弛。

五、 思考题

- 对不同策略优缺点，适用场景的理解和认识。

1. **一致代价搜索**

一致代价搜索是BFS加上最优化的思想，能确保第一次到某个点一定是沿着最优的路径搜索到的，具有最优性。在解离根节点比较近，层数较浅时，可以较快地得到答案。但由于其需要维护边界队列，经常需要保存指数级的节点数量，消耗大量内存，空间复杂度高。

适用于搜索树层数较浅的情况下。

2. **迭代加深**

迭代加深搜索就像以BFS的思想写DFS。搜索过程为首先深度优先搜索k层，若没有找到可行解，再深度优先搜索k+1层，直到找到可行解或达到限制的层数为止。

迭代加深本身是一个DFS，具有空间复杂度小的优点。此外它克服了DFS可能运行时间非常长，或存在无限路径时无限运行下去的缺点。由于深度是从小到大逐渐增大的，所以当搜索到结果时可以保证搜索深度是最小的，具备了完备性和最优性（在每个动作的成本一致的情况下）。但由于每次提高深度限制时，都要重复 重头开始DFS搜索，效率较低，时间复杂度是呈指数级增加的。

因此适用于当搜索树非常深，但是我们能确定答案一定在浅层节点的情况下。

3. **双向搜索**

双向搜索适用于已知起点和终点位置状态的情况，从起点和终点两个方向同时开始搜索，可以明显提高单向BFS的搜索效率，将搜索深度变为了$d/2$层。缺点是需要维护两个边界队列，空间复杂度较高，而且必须要知道终点位置才能使用。

4.A^***搜索**

结合了BFS和Dijkstra算法的优点：在进行启发式搜索提高算法效率的同时，基于评估函数可以保证找到一条最优路径。适合用于解决最短路径问题。但其难点在于启发式函数的设计与控制上，若设置不好，反而可能会找不到最优解，导致搜索效率下降。

5. IDA^***搜索**

为A^*搜索与迭代加深算法的结合。迭代加深是以深度为限制，而IDA^*是以$f(n)$值为限制。其优点在于不用将节点的$f(n)$值排序，选择最小的进行拓展，只需要判断估值是否大于阈值决定是否拓展。此外不用维护开启和关闭队列保存节点，空间消耗较小。其缺点是不具有最优性。