第二次实验报告

姓名：刘佳凡
学号：10205501403

实验环境

anaconda3（Python 3.8.8）　vscode编辑器

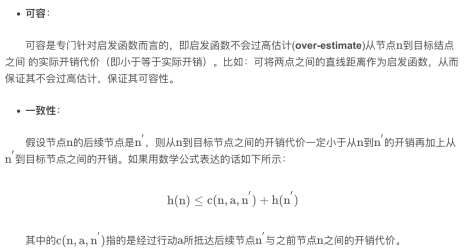
实验目的

两道算法题都用A*算法实现，每道题有5个测试输入，你需要用你的算法去预测对应输出

实验过程

A*算法

- A* 不一定的点在于：不仅考虑了探测点到出发点的步长即**真实cost：g(n)**，又考虑探测点到最终目标点的步长即**未来cost：h(n)**。A*算法通过`f(n) = g(n) + h(n)`来计算每个节点的优先级。
- 为了保证A*算法最终结果是最优的我们的启发式函数设计需要满足两点：1.可容的(**admissible heuristic**) 2.一致的**(consistency)**



- **可容**：

可容是专门针对启发函数而言的，即启发函数不会过高估计(**over-estimate**)从节点n到目标结点之间 的实际开销代价（即小于等于实际开销）。比如：可将两点之间的直线距离作为启发函数，从而保证其不会过高估计，保证其可容性。

- **一致性**：

假设节点n的后续节点是$n^{'}$，则从n到目标节点之间的开销代价一定小于从n到$n^{'}$的开销再加上从$n^{'}$到目标节点之间的开销。如果用数学公式表达的话如下所示：

$$h(n) \leq c(n, a, n^{'}) + h(n^{'})$$

其中的$c(n, a, n^{'})$指的是经过行动a所抵达后续节点$n^{'}$与之前节点n之间的开销代价。

Q1: 森林宝石的秘密通道

启发式函数设计

在理解完题目之后，通过移动空位周围的宝石来改变宝石的位置，移动的每一步代价都是相同的，所以本道题的关键在于设计启发式函数，在上面提到我们的启发式函数需要满足以上两种特性。在这里我一共尝试了四种：

1. **曼哈顿距离**`D(I, J)=|XI-XJ|+|YI-YJ|`是标准的启发式函数，本题就是遍历这个矩阵中每一个点再与目标位置累计计算并相加

```
for i in self.state_matrix.flatten():
    dx = abs(np.where(self.state_matrix == i)[0][0] - np.where(target_state == i)
[0][0])
    dy = abs(np.where(self.state_matrix == i)[1][0] - np.where(target_state == i)
[1][0])
    h += dx + dy
```

2. **对角线距离** `D(I, J)=D*max(|XI-XJ|,|YI-YJ|)`

```
h += math.sqrt(2)*max(dx,dy)
```

3. **更复杂精确的对角线距离** `D2 * h_diagonal(n) + D * (h_straight(n) - 2*h_diagonal(n))`

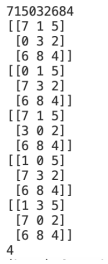
   这里计算h_diagonal(n)：沿着斜线可以移动的步数；h_straight(n)：曼哈顿距离；然后合并这两项，让所有的斜线步都乘以根号2，剩下的所有直线步(注意这里是曼哈顿距离的步数减去2倍的斜线步数)。

```
h += (dx+dy)+(math.sqrt(2)-2)*min(dx,dy)
```

4. **欧几里得距离**：这里需要计算距离的平方，代价会大一点

```
h += math.sqrt(dx^2 + dy^2)
```

⚠️⚠️：发现四种启发式函数只有用欧式距离来算第四个是错误的，结果应该为3但输出4，打印出路径如下：



⚠️⚠️原因：第一步根据启发式函数算出来的结果有两个选择，算出来h都是2，所以这两条路具有相同的f(n)值的时候，它们都会被搜索，即都会被加入到priority_q队列中，进一步排查程序发现，是自己的程序逻辑写的有点问题，因为一开始的逻辑是专门设置一个变量n来记录走的步数，每循环一遍则n+1，但这样存在一个问题在于，若算出多个f值相同的next state，就会导致priority_q队列写入很多，从而导致步数出现错误，如上图所示，所以代码逻辑需要修改成，在统一批次的next states里，他们的步数都是相同的，只有当进入下一个状态步数才会➕1

解题思路

状态类Forest，需要传入**state_matrix**(当前的状态矩阵)和**g**(当前的真实代价)，内部包含一下几个重要部分：

```
self.state_matrix          # 每个时刻的状态矩阵
self.f = self.g + self.h  # f值计算
def count_h(self)          # 用启发式函数来计算具体的h值，这里用到上面👆四种
def find_next(self)        # 先找到空白键的坐标，然后分别x+-1，y+-1，看下一个状态是否超过范围，从而交换
    if i-1 >= 0:  上移
    if i+1 <= 2:  下移
    if j-1 >= 0:  左移
    if j+1 <= 2:  右移
def __lt__(self, other)    # 重定义_lt_比较方法，使得按f值从小到大排列
```

⚠️：因为这里是将其存储为3*3的矩阵，所以下标i、j的范围都在0-2之间

应用函数Forest_A_start，先初始化优先级队列，然后将初始状态入队，然后先去出队首开始循环**直至当前状态的启发式函数为0**，否则弹出队首作为当前状态，先通过调用状态类的**find_nex()**函数来找到下一步可能存在的状态列表，然后遍历这个列表，如果下一个状态**已经访问过了则跳过**，没有则加入到队列中，主要在这里需要传入的真实代价则是前一个代价➕1，因为在这里我们设定的就是每走一步则代价+1，然后最后再返回最终走的步数。

```
def Forest_A_start(start_state):
    priority_q = PriorityQueue()
    expanded_state_set = []  # 已经访问过的状态
    state = Forest(start_state, 0)
    priority_q.put(state)
    present_state = priority_q.get()#取出f值最小的状态,取队首元素的值并将其弹出
    real_cost = 0
    while present_state.count_h() != 0: #设置中止条件为当h启发式函数为0
        expanded_state_set.append(str(present_state.state_matrix.flatten()))
        next_states = present_state.find_next()
        for i in range(len(next_states)):
            if str(next_states[i].flatten()) in expanded_state_set:  #跳过已经扩展过的路径
                continue
            #这里由于每一步代价均相同，真实代价为1即可，最终的真实代价即可代表步数
            real_cost = present_state.g + 1
            state_next = Forest(next_states[i], real_cost)
            priority_q.put(state_next)
        present_state = priority_q.get() #获取f值最小的开始
    print(real_cost)
```

⚠️：这这里因为我们设计的g值是每走一步代价+1，所以我们最终所需要打印出的移动的最少步骤则就是我们最后present_state的g值，即real_cost

⚠️：关于这里的主函数设计，因为一开始在题目中例子给到如下解释，所以在这里所有的处理都是基于矩阵数组，即一开始就将输入转换为3*3的数组`array = np.array(list(input()), dtype=int).reshape(3, 3)`

例：150732684
即：
初始状态为
1 5 0
7 3 2
6 8 4

本题结果

序号	输入	输出
1	135720684	1
2	105732684	1
3	015732684	2
4	135782604	1
5	715032684	3

杰克的金字塔探险

首先在这道题中需要注意的点为：**1.** 总是从塔顶1号出发到塔底 **2.** 最后的若出现多个相同长度路径需要依次输出

启发式函数设计

在金字塔探险这一问题中，我专门用一个(N+1)*(N+1)维的数组来存储路径⚠️：这里用多加一维是为了在后续找路径时可以直接使用房间的房号，从而避免再做-1运算。

```
    rooms_map = np.zeros((N + 1, N + 1), dtype=int)
    for i in range(M):
        #因为输入矩阵是这样的：[[1 2 1] [1 3 4] [2 4 3][3 4 2][3 5 1][4 5 2][5 1 5]]
        rooms_map[rooms[i][0], rooms[i][1]] = rooms[i][2]
```

但本题中**启发函数设计比较困难**，因为我们无法得知每一条路径具体长度，除此之外启发函数必须小于等于真实值才能满足一开始说的两个性质。所以总体来说有两种思路：

1. **第一种设计：**直接统一设置为一个常数，一开始我尝试了直接将其设为为 0，那相当于只考虑了真实代价，也就退化成了UCS 算法。所以后面将其改为 0 到 1 之间的小数，这里使用了0.2，是因为根据题目当中给出的例子可以看到路径代价最小为1。
2. **第二种设计：**这里启发式函数是以从该点出发的最短路径长度作为启发式函数的值，这一定是满足可容性的，具体思路就是先遍历路径矩阵，找到该位置下的最短路径长度，若找到了则设置为h，否则为0

```
    def count_h(self,N,M,rooms):
        if Pyramid == N:  # 已经到山下了
            return
        mini = 0
        for i in range(M): #先找到第一条符合条件的路径
            if rooms[i][0] == self.state:
                mini = rooms[i][2]
                break
        for i in range(M): #找到最短路径
            if rooms[i][0] == self.state:
                if rooms[i][2] < mini:
                    mini = rooms[i][2]
        self.h = mini
```

解题思路

状态类Pyramid：传入参数需要有当前所在位置input_state和当前真实代价 g，类中包含以下部分：

```
self.state_matrix          # 该时刻状态即位置
self.f = self.g + self.h   # f值计算
def count_h(self)          # 计算启发式函数
def find_next(self)        # 直接遍历我们的rooms，找到下一个可走房间加入到列表next_rooms中
def __lt__(self, other)    # 重定义_lt_比较方法，使得按f值从小到大排列
```

应用函数Pyramid_A_Start：开始部分与第一题一样，初始化优先级队列，注意这里对类函数一开始传参为：`Pyramid(1, 0)`因为是从1号房间开始往下走，**初始的真实代价为0**。然后对输入的路径开始存储，然后开始循环，这里的中止条件为队列为空或者找到了k条路径。在循环内部的逻辑为如果已经**到达N房间**，则将这条路径上的代价存储到cost_set结果集中。接着开始寻找下一条可走房间路径，更新真实代价，这里的**真实代价函数即为状态转移时的路径长度**，找到将下一个状态入队列。在最后查看结果集是否是有K个结果，不足则补上

```
    while not priority_q.empty() and len(cost_set) < K:
        State = priority_q.get()
        if State.state == N:  # 到底了
            cost_set.append(State.g)
        next_rooms = State.find_next(M, rooms) #寻找下一个
        for i in range(len(next_rooms)):
            new_g = State.g + rooms_map[State.state, next_rooms[i]]  # 更新真实代价g
            next_room = Pyramid(next_rooms[i], new_g)
            #next_room.count_h() #第一种启发式
            next_room.count_h(N,M,rooms) #第二种启发式
            priority_q.put(next_room)
    if len(cost_set) < K: #查看是否一共找到了K条路径
        for i in range(K - len(cost_set)):
            cost_set.append(-1)
```

⚠️：关于主函数的设计，根据在题目当中所给的输入格式，先讲每一行用split按空格分割，然后再用**迭代器**处理`list(map(int, str))`，并将每一行路径其转换为list加入到总列表rooms中

实验结果

序号	输入	输出
1	5 6 4 1 2 1 1 3 1 2 4 2 2 5 2 3 4 2 3 5 2	3 3 -1 -1
2	6 9 4 1 2 1 1 3 3 2 4 2 2 5 3 3 6 1 4 6 3 5 6 3 1 6 8 2 6 4	4 5 6 7
	7 12 6	

3	1 2 1 1 3 3 2 4 2 2 5 3 3 6 1 4 7 3 5 7 1 6 7 2 1 7 10 2 6 4 3 4 2 4 5 1	5 5 6 6 7 7
4	5 8 7 1 2 1 1 3 3 2 4 1 2 5 3 3 4 2 3 5 2 1 4 3 1 5 4	4 4 5 -1 -1 -1 -1
5	6 10 8 1 2 1 1 3 2 2 4 2 2 5 3 3 6 3 4 6 3 5 6 1 1 6 8 2 6 5 3 4 1	5 5 6 6 6 8 -1 -1

实验心得

本次实验运用到了python的**PriorityQueue**优先级队列和 **A*算法**来解决两个具体的应用场景。深刻体会了 A*算法，在有些应用场景下其效率要比普通 BFS，DFS 等快，且h满足 admissible 能保证最终结果一定是正确的

在具体解题过程中一开始遇到的困难是对题目输入数据的处理和整体思维构建，但在梳理清楚之后A算法逻辑就发现比较好上手了，也可以感受到，A算法既考虑过去又考虑未来的巧妙，但其实第一题是偏向贪心的搜索方法，只要用贪心法即可达到最优解；第二题是偏向 UCS 的搜索方法，而在本题目所使用的 A算法本质上也可理解为UCS