实验二：Astar算法

10215501412 彭一珅

一、实验目的

熟悉A*算法的原理，并在具体问题上应用。

二、实验环境

- windows11操作系统
- python3.9

三、实验过程

题目简介：

Q1：在遥远的冰雪王国中，存在一个由9个神秘冰块组成的魔法魔方。在这个魔法魔方中，有8块冰雪魔块，每块都雕刻有1-8中的一个数字（每个数字都是独特的）。魔方上还有一个没有雕刻数字的空冰块，用0表示。你可以滑动与空冰块相邻的冰块来改变魔块的位置。传说，当冰雪魔块按照一个特定的顺序排列（设定为1 3 5 7 0 2 6 8 4）时，魔法魔方将会显露出通往一个隐秘冰宫的冰霜之道。
现在，你站在这个神秘的魔方前，试图通过最少的滑动步骤将冰雪魔块排列成目标顺序。为了揭开这一神秘，你决定设计一个程序来从初始状态成功找到冰霜之道。
请注意只能使用A*算法。
输入示例：一行九个数字，用0表示空冰块，其他数字代表从左到右，从上到下的冰雪魔块上的雕刻数字。例如：150732684表示初始状态为
1 5 0
7 3 2
6 8 4
输出示例：仅一行，该行只有一个数字，表示从初始状态到目标状态需要的最少移动次数（测试数据已确保都能到达目标状态）。

Q2：在一个神秘的王国里，有一个名叫杰克的冒险家，他对宝藏情有独钟。传说在那片广袤的土地上，有一座名叫金字塔的奇迹，隐藏着无尽的财富。杰克决定寻找这座金字塔，挖掘隐藏在其中的宝藏。 金字塔共有N 个神秘的房间，其中1号房间位于塔顶，N 号房间位于塔底。在这些房间之间，有先知们预先设计好的M条秘密通道。这些房间按照它们所在的楼层顺序进行了编号。杰克想从塔顶房间一直探险到塔底，带走尽可能多的宝藏。然而，杰克对寻宝路线有着特别的要求：
(1) 他希望走尽可能短的路径，但为了让探险更有趣和挑战性，他想尝试K条不同的较短路径。
(2) 他希望在探险过程中尽量节省体力，所以在选择通道时，他总是从所在楼层的高处到低处。
现在问题来了，给你一份金字塔内房间之间通道的列表，每条通道用 (X_i, Y_i, D_i) 表示，表示房间X_i 和房间Y_i 之间有一条长度为D_i 的下行通道。你需要计算出杰克可以选择的K条最短路径的长度，以便了解他在探险过程中的消耗程度。

Q1 冰雪魔方的冰霜之道

问题分析：

该问题是8-puzzle问题，可以通过A*搜索来解决。对于每个包含9个数码的状态，可以进行的移动有4种，即将数码0与其上下左右四个方向的数字交换位置。如果数码0的位置在3*3的网格边缘或角落，则只有3或2种移动方式。为了使用A*算法来解决这个问题，最重要的是启发式函数的定义。

算法实现：

在python代码的实现中，将每个状态用一个字符串表示，这个字符串就像输入用例一样，是由0-8这9个数字组成的，表示每个冰块的位置。在下面定义的启发式函数中，输入target和state分别表示目标状态，即"135702684"，和当前状态。启发式函数的返回值dist表示当前状态转移到目标状态预计需要的步数。

根据启发式函数的定义，这个估计值需要小于等于实际从当前状态转移到目标状态所需要的代价。在这里尝试了两种启发式函数的实现。对于注释中的这个函数，返回值是当前状态字符串中与目标状态字符串不同的数字个数，也就是对于8-puzzle来讲，有几个puzzle在错误的位置。但是这种方法不能很好地反映puzzle位置错误的程度，因此改用了第二个启发式函数，也就是根据曼哈顿距离来计算当前状态与目标状态的偏差。

```
# def heuristic(target, state):
#     return sum(1 for a, b in zip(state, target) if a != b)

def heuristic(target,state):
    dist = 0
    for i in range(1, 9):
      # 计算出每个puzzle的横纵坐标，并计算相对于目标状态的曼哈顿距离
        current_row, current_col = state.index(str(i)) // 3, state.index(str(i)) % 3
        target_row, target_col = target.index(str(i)) // 3, target.index(str(i)) % 3
        dist += abs(current_row - target_row) + abs(current_col - target_col)
    return dist
```

以启发式函数为依托，A*算法的实现需要以下几个步骤：

- 从open_list优先级队列中弹出具有最小代价的状态，这是当前要扩展的状态。
- 如果当前状态等于目标状态，说明已经找到了最短路径，返回移动步数。
- 否则，将当前状态添加到closed_set集合，用于记录已经访问过的状态。
- 生成当前状态的所有可能操作，得到新的状态。
- 对于每个新状态，如果它不在closed_set集合中，计算该状态的代价（已经走过的步数cost加上启发式函数的值），然后将其添加到open_list优先级队列。

如果open_list为空，但仍未找到目标状态，说明不可能达到目标状态，返回-1，一旦找到目标状态，则open_list的第二个值cost则为当前走过的步数。

```
def a_star_search(target, initial_state):
  # 将closed_set初始化为空，open_list里的元素，第一个值是初始状态，第二个值是走到初始状态的步数，即0步
    open_list = [(initial_state, 0)]
    closed_set = set()
  # 当优先级队列不为空，始终取出cost+h最小的状态进行扩展
    while open_list:
        state, cost = open_list.pop(0)
        # 返回当前步数
        if state == target:
            return cost
        if state not in closed_set:
            closed_set.add(state)
            # 扩展状态的方法是找到与0相邻的几个数字的index
```

```
            zero_idx = state.index("0")
            neighbors = [(zero_idx - 1, zero_idx), (zero_idx + 1, zero_idx),
                         (zero_idx - 3, zero_idx), (zero_idx + 3, zero_idx)]
            for a, b in neighbors:
              # 如果这些index超出了0-9，则说明空块0在边缘或角落位置
                if 0 <= a < 9 and 0 <= b < 9:
                    new_state = list(state)
                    # 交换0与相邻的块
                    new_state[a], new_state[b] = new_state[b], new_state[a]
                    new_state = "".join(new_state)
                    open_list.append((new_state, cost + 1))
        # 排序的方式是按照步数cost+启发式函数h
        open_list.sort(key=lambda x: x[1] + heuristic(target, x[0]))
    return -1
```

Q2 杰克的金字塔探险

问题分析：

该问题是k-shortest-path问题，需要寻找有向图中从源节点到目的节点的K条不同的最短路径。该问题可以结合A*算法和Dijkstra算法来解决。其中路径长度由启发式函数引导，并根据路径累计的权重来计算。

算法实现：

使用python中的字典来表示有向图graph，graph的key是图中边的起点，而value是一个列表，这个列表中的每个元素分别是边的目的节点、边的权重。

```
N, M, K = map(int, input().split())
graph = {}  # 图的数据结构，以邻接列表表示
graph2 = {}
for _ in range(M):
    X, Y, D = map(int, input().split())
    if X not in graph:
        graph[X] = []
    graph[X].append((Y, D))
    if Y not in graph2:
        graph2[Y] = []
    graph2[Y].append((X, D))
```

启发式函数通过dijkstra算法来计算，对于图中的每个节点，启发式函数的值应该是节点与目的节点的距离估算值，并且要小于等于真实的距离，因此采用dijkstra算法，求出每个节点距离目的节点的最短距离，放在字典distances里。上面一段代码中记录的graph2，表示了与要求的图graph中每条边都方向相反的图，因此可以根据graph2，以目的节点end为dijkstra算法的源节点，计算出每个节点相对目的节点的最短距离。

```
def dijkstra(graph, end):
    # 使用Dijkstra算法计算从start到end的最短距离
    # distances = {node: float('inf') for node in graph}
    distances = {node: float('inf') for node in range(1,end+1)}

    distances[end] = 0
    priority_queue = [(0, end)]

```

```
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        # 如果此时的距离比之前计算出的距离远，那么忽略
        if current_distance > distances[current_node]:
            continue
        # 将当前节点的可达节点，及与当前节点相关的计算距离加入到优先级队列里面，如果节点没有出边，则忽略
        if current_node in graph:
            for neighbor, weight in graph[current_node]:
                distance = current_distance + weight

                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))
        else:
            continue

    return distances
```

在此基础上，就可以实现A*算法了。这道题的实现步骤与上一题基本类似，只是需要扩展出k条路径才可以结束，而不是一旦出现一条到达目的状态的路径就返回，因此需要用k_shortest_paths来记录所有的路径的长度。

还有一点不同是，在这个问题中应用了heapq，可以将一个列表转化为最小堆，每次出队都是队列里最小的值，可以省去每次扩展完排序的步骤，但是也引发了另一个改变，就是最小堆的元组第一个值必须是f=g+h，也就是当前走过的路径weight总和与当前节点到目标节点最短路径的长度相加，因此，在扩展一个新的状态后，f值必须减去上一个状态的g，再加上扩展到这个状态所经过的边的weight以及这个节点与目的节点的距离。

```
def astar(graph, start, end, H):
    open_set = [(H[start], start, [])]  # 优先级队列，(f, node, path)
    k_shortest_paths = []  # 存储K短路路径

    while open_set:
        f, current, path = heapq.heappop(open_set)

        if current == end:
            k_shortest_paths.append((f, path + [current]))

        if len(k_shortest_paths) == K:
            break

        if current in graph:
            for neighbor, cost in graph[current]:
                h = H[current]
                g = f - h + cost  # 重新计算路径长度
                h = H[neighbor]  # 启发式估计函数值
                heapq.heappush(open_set, (g + h, neighbor, path + [current]))
        else:
            continue

    return k_shortest_paths
```

四、实验结果

Q1

input:

> 135720684

output:

> 1

input:

> 105732684

output:

> 1

input:

> 015732684

output:

> 2

input:

> 135782604

output:

> 1

input:

> 715032684

output:

> 3

Q2

input:

> 5 6 4
> 1 2 1
> 1 3 1
> 2 4 2
> 2 5 2

```
3 4 2
3 5 2
```

output:

```
3
3
-1
-1
```

input:

```
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
```

output:

```
4
5
6
7
```

input:

```
7 12 6
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
```

output:

```
5
5
6
6
```

```
7
7
```

input:

```
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
```

output:

```
4
4
5
-1
-1
-1
-1
```

input:

```
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
```

output:

```
5
5
6
6
6
8
-1
-1
```