当代人工智能实验二--A*算法

--10215501435 杨茜雅

实验题目:

Q1:冰雪魔方的冰霜之道

题目描述

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 135702684)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。

输入格式:

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。

输出格式:

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:杰克的金字塔探险

题目描述

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。

输入格式:

第一行三个用空格分开的整数 NM,K。

第二行到第 M+1 行，每行有是三个空格分开的整数 X ;YD，描述了一条下坡的路

输出格式:

共 K 行。

在第 i 行输出第 i 短的路线长度，如果不存在就输出-1。如果出现多条相同长度的路线，务必全部依次输出

简单介绍一下 A*算法：

A* 算法是一种在图形遍历中寻找从开始节点到结束节点路径的算法。它被广泛用于各种应用中，比如计算机科学中的路径查找问题，也被用于游戏中的 NPC（非玩家角色）移动计算等。这种算法通过每一步都选择一个预测的最优解来保证效率，同时它也是非常实用的，因为它总能找到一个解（如果解存在的话）。

A*算法通过下面这个函数来计算每个节点的优先级。

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

关键概念：

●**G Cost（实际代价）：**从起始点到当前节点的实际代价。
●**H Cost（启发式/预测代价）：**从当前节点到目标的估计代价，这也就是 A *算法的启发函数，是 A*算法的关键。
●**F Cost：**G Cost 与 H Cost 的和，表示为 F = G + H。是节点 n 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。

A*算法在运算过程中，每次从优先队列中选取 f(n)值最小（优先级最高）的节点作为下一个待遍历的节点。另外，A*算法使用两个集合来表示待遍历的节点，与已经遍历过的节点，这通常称之为 open_set 和 close_set。

完整的A*算法描述如下：

```
* 初始化open_set和close_set；
* 将起点加入open_set中，并设置优先级为0（优先级最高）；
* 如果open_set不为空，则从open_set中选取优先级最高的节点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_set中删除，并加入close_set中；
        * 遍历节点n所有的邻近节点：
            * 如果邻近节点m在close_set中，则：
                * 跳过，选取下一个邻近节点
            * 如果邻近节点m也不在open_set中，则：
                * 设置节点m的parent为节点n
                * 计算节点m的优先级
                * 将节点m加入open_set中
```

启发函数

启发函数会影响 A*算法的行为。
●在极端情况下，当启发函数 h(n)始终为 0，则将由 g(n)决定节点的优先级，此时算法就退

化成了 Dijkstra 算法。
●如果 h(n)始终小于等于节点 n 到终点的代价，则 A*算法保证一定能够找到最短路径。但是当 h(n)的值越小，算法将遍历越多的节点，也就导致算法越慢。
●如果 h(n)完全等于节点 n 到终点的代价，则 A*算法将找到最佳路径，并且速度很快。可惜的是，并非所有场景下都能做到这一点。因为在没有达到终点之前，我们很难确切算出距离终点还有多远。
●如果 h(n)的值比节点 n 到终点的代价要大，则 A*算法不能保证找到最短路径，不过此时会很快。
●在另外一个极端情况下，如果 h()n 相较于 g(n)大很多，则此时只有 h(n)产生效果，这也就变成了最佳优先搜索。

由上面这些信息我们可以知道，通过调节启发函数我们可以控制算法的速度和精确度。因为在一些情况，我们可能未必需要最短路径，而是希望能够尽快找到一个路径即可。这也是 A*算法比较灵活的地方。

对于网格形式的图，有以下这些启发函数可以使用：

●如果图形中只允许朝上下左右四个方向移动，则可以使用曼哈顿距离（Manhattan distance）。
●如果图形中允许朝八个方向移动，则可以使用对角距离。
●如果图形中允许朝任何方向移动，则可以使用欧几里得距离（Euclidean distance）。

Q1：冰雪魔方的冰霜之道

这个问题可以看作是一个 **8-puzzle** 问题，其中启发函数 H 可以**使用曼哈顿距离（每块到目标位置的距离之和）**来估计。A* 算法将通过最小化每个步骤的成本（G）和预测的剩余距离（H）来找到解决方案。

算法设计思路：

- 初始化 open_set
- 将起点加入 open_set 中，并且将 open_set 设置为堆
    - 如果节点 n 是终点，则：
        - 输出需要移动它的步数
    - 如果节点 n 不是终点，则：
        - 将节点 n 从 open_set 中删除，加入 visited 中
        - 遍历节点 n 的所有邻近节点
            - 如果邻近节点在 visited 中，跳过
            - 如果邻近节点也不在 open_set 中：
                - 将 m 加入 open_set，根据节点 m 的 f 值进行堆排序，加入 open_set 这个堆中。

选取启发函数：

由于魔法魔方中的冰块只能有四个移动方向，所以我使用曼哈顿距距离作为 h(n)。
实验中计算曼哈顿距离的核心部分是 **manhattan_distance(board)函数。**
让我来详细解释这个函数是如何工作的：
●初始化 distance 为 0，这是用来累积所有数字块到其应该在的位置的距离。
●遍历整个棋盘，对于棋盘上的每一个位置（共 9 个，因为是 3x3 的棋盘），做以下操作：忽略 0（也就是空块）。
●对于非 0 的数字块，找出这个块应该在的正确位置 correct_pos。这里使用了 correct_pos = (board[i] - 1) % 8 来计算，这是因为目标状态是(1, 3, 5, 7, 0, 2, 6, 8, 4)，在这个序列中，每个数字块的值与它应该在的索引有特定的关系。
●计算这个数字块当前的位置和它应该在的位置之间的距离。这是通过 abs(i // 3 - correct_pos // 3) + abs(i % 3 - correct_pos % 3)来计算的。这里，i // 3 和 i % 3 可以分别得到数字块当前位置的行和列，而 correct_pos // 3 和 correct_pos % 3 则可以得到数字块应该在的行和列。
●返回累积的距离。

```
# 定义曼哈顿距离函数，计算当前状态到目标状态的距离
def manhattan_distance(board):
    distance = 0
    for i in range(9): # 对于棋盘上的每一个位置
        if board[i] != 0: # 0是空块，不需要计算距离
            # 根据题目的目标状态计算每个块应该在的位置
            correct_pos = (board[i] - 1) % 8
            # 计算当前块和应该在的位置的距离，并累加
            distance += abs(i // 3 - correct_pos // 3) + abs(i % 3 - correct_pos % 3)
    return distance
```

h(n)函数选取完毕，接下来选择移动的次数 g 作为我们的 g(n)，然后在堆中我们将 g 值与 h 值相加作为 f 值进行堆排序。

首先，定义一个 namedtuple 类型的变量，自定义类名为 State，拥有 board, zero_idx, g, h 属性。f 用来记录 f 值也同时作为堆排序的依据，board 用于记录每一个宝石的位置，zero_idx 用于记录 0 的位置，g 用于记录走过的步数，h 用于记录启发函数的值。

```
State = namedtuple("State", ["f", "board", "zero_idx", "g", "h"])
```

将初始状态命名为 initial_state 并且输入到 open_set 中然后进行 heapify, 定义 visited 集合表示已经访问过的状态。

```
# 定义主解决函数
def solve(initial_board):
    # 目标状态
    goal_board = (1, 3, 5, 7, 0, 2, 6, 8, 4)
    # 评估初始状态到目标状态的代价
    f = manhattan_distance(initial_board)
    # 创建初始状态
    initial_state = State(f, initial_board, initial_board.index(0), 0, manhattan_distance(initial_board))
# 使用一个列表作为优先队列存放待扩展的状态
    open_set = [initial_state]
    heapq.heapify(open_set)# 转化为最小堆结构，使得代价最小的状态能够被优先扩展
    visited = set()# 用于记录已经访问过的状态
```

当 open_set 不为空时，每次从 open_set 里面 pop out f 值最小的状态，如果 pop 出来的状态与最终状态相同，就直接返回 g 值。
否则，对原图空位旁边的宝石进行上下左右的四个方向的变换。在这里，由于变换的方式只和空位（0）有关，所以我们可以等效于移动 0 的位置，然后将移动的位置和 0 进行交换，得到新的状态。如果状态不在 visited 集合中，即没有被访问过，使用 heappush 方法插入到堆中，再对其使用 f 值进行排序。此处的 f 值等于 g 值 + h 值，g 值为移动步数，h 值为启发函数，即曼哈顿距离。

```python
    while open_set:  # 当待扩展的状态列表不为空时
        current_state = heapq.heappop(open_set) # 取出代价最小的状态
# 如果当前状态为目标状态，返回到达该状态所需的步数
        if current_state.board == goal_board:
            return current_state.g
        visited.add(current_state.board) # 标记当前状态为已访问
       # 获取空块的位置
        x, y = current_state.zero_idx // 3, current_state.zero_idx % 3
         # 定义四个可能的移动方向
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_x, new_y = x + dx, y + dy
            # 判断移动方向是否有效（是否超出棋盘边界）
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_idx = new_x * 3 + new_y
                # 获取新的棋盘状态
                new_board = list(current_state.board)
                new_board[current_state.zero_idx], new_board[new_zero_idx] = new_board[new_zero_idx], new_board[current_state.zero_idx]
                new_board = tuple(new_board)
                # 如果新的状态没有被访问过
                if new_board not in visited:
                    # 计算新状态的评估代价
                    f = current_state.g + 1 + manhattan_distance(new_board)
                    new_state = State(f, new_board, new_zero_idx, current_state.g + 1, manhattan_distance(new_board))
                    # 创建新的状态并加入到待扩展的状态列表中
                    heapq.heappush(open_set, new_state)
```

对于每一个输入，我们使用 python 里面的 map 方法先转化为 iterable 的元素，再将其转化为元组，然后对其应用以上的 A* 算法进行搜索。

```python
if __name__ == "__main__":
    # 定义几个测试的初始状态
    input_str = ["135720684", "105732684", "015732684", "135782604", "715032684"]
    for i in range(len(input_str)):
        # 将字符串转换为元组格式
        initial_board = tuple(map(int, input_str[i]))
         # 输出从初始状态到目标状态所需的步数
        print(solve(initial_board))
```

重要的点：

●**启发式函数**：代码中用到的启发式函数是曼哈顿距离（manhattan_distance）。对于每个非空的冰块，计算它与其应在的位置之间的曼哈顿距离，并将这些距离加起来。这为每个状态提供了一个估计到目标状态的代价。
●**评估函数 f**: f 值是到达当前状态的实际代价 g 和从当前状态到目标状态的估计代价 h 的总和。优先队列（使用 heapq 实现）按照 f 值的大小来存放状态，确保代价最小的状态被优先考虑。
●**状态表示和转换**：每个状态由冰块的当前排列、空块的位置、实际代价、估计代价和 f 值组成。当移动一个与空块相邻的冰块时，状态会发生改变，此时新的状态可能会被加入到待考虑的状态列表中。
●**避免重复状态**：visited 集合用于记录已经被考虑过的状态。这是为了防止算法陷入循环或

者多次考虑相同的状态。
●**搜索策略**：代码从初始状态开始，不断地尝试移动与空块相邻的冰块，每次都选择代价估计最小的状态进行扩展，直到找到目标状态为止。
●**终止条件**：当 open_set 为空时，意味着所有可能的状态都已经被考虑过了，但仍然没有找到到达目标状态的方法。在此代码中，这种情况不会发生，因为 8 数码问题总是有解的（至少对于给定的目标状态）。
代码文件中也都写了很详细的注释。

对应测试输入的输出：

总的来看：
每一行对应输入为老师给的测试样例

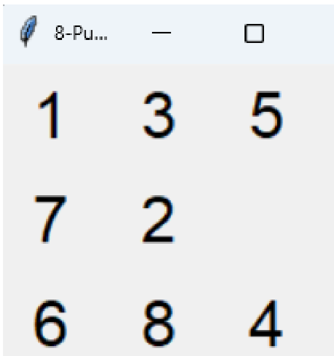
```
# 定义几个测试的初始状态
input_str = ["135720684", "105732684", "015732684", "135782604", "715032684"]
```

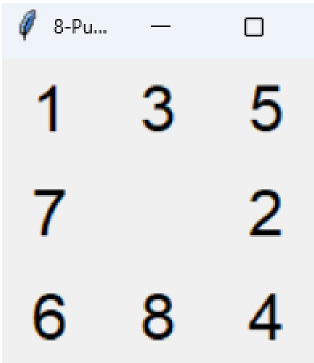
```
1
1
2
1
3
```

另外，我单独对每个输入进行了**更加详尽的可视化**，这些都可以在 **visual_Q1.py** 中看见。

1、"135720684"　移动一次

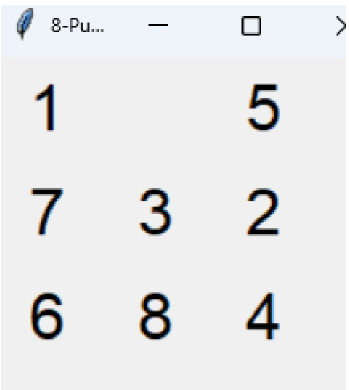
```
Step 0:
(1, 3, 5)
(7, 2, 0)
(6, 8, 4)

Step 1:
(1, 3, 5)
(7, 0, 2)
(6, 8, 4)
```

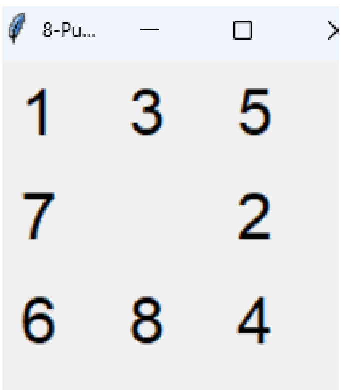




2、"105732684"　移动一次

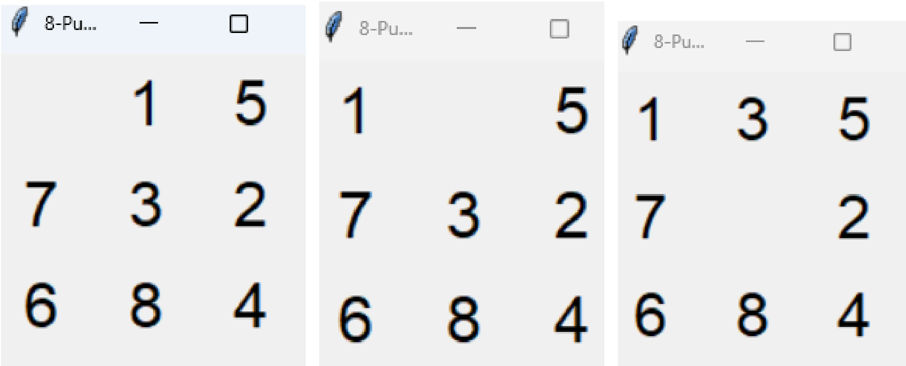
```
Step 0:
(1, 0, 5)
(7, 3, 2)
(6, 8, 4)

Step 1:
(1, 3, 5)
(7, 0, 2)
(6, 8, 4)
```





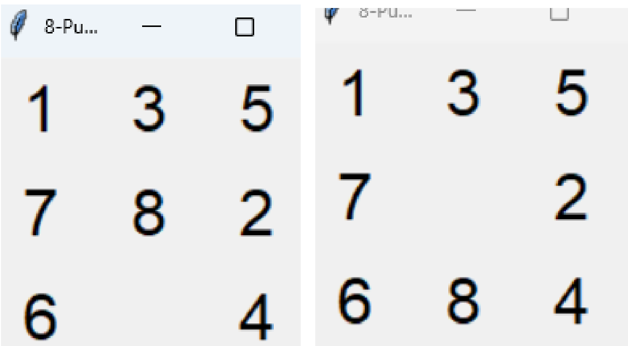
3、"015732684" 移动两次

```
Step 0:
(0, 1, 5)
(7, 3, 2)
(6, 8, 4)

Step 1:
(1, 0, 5)
(7, 3, 2)
(6, 8, 4)

Step 2:
(1, 3, 5)
(7, 0, 2)
(6, 8, 4)
```



4、"135782604" 移动一次

```
Step 0:
(1, 3, 5)
(7, 8, 2)
(6, 0, 4)

Step 1:
(1, 3, 5)
(7, 0, 2)
(6, 8, 4)
```



5、"715032684" 移动三次

```
Step 0:
(7, 1, 5)
(0, 3, 2)
(6, 8, 4)

Step 1:
(0, 1, 5)
(7, 3, 2)
(6, 8, 4)

Step 2:
(1, 0, 5)
(7, 3, 2)
(6, 8, 4)

Step 3:
(1, 3, 5)
(7, 0, 2)
(6, 8, 4)
```

Q2：杰克的金字塔探险

这个问题可以看作是一个**基于启发式搜索的 K 条最短路径**的问题，特定于从房间 1 到房间 N 的路径搜索。其中使用启发式函数来估计从当前房间到目标房间的距离。使用 A*算法，通过**最小化每条路径的成本（g 值）和启发式估计（h 值）**来找到 K 条最短路径的长度。

从实现 A* 算法的树搜索入手，为了准确地计算 f 值，我们需要选取合适的 g 值和 h 值。
有关 g 值的选取很简单，只要计算当前节点距离起始节点所总共移动过的代价就可以了。
关于启发函数的选取，我的思路如下：
每一行输入，我们可以得到 x_i 与 y_i 的距离。并且题目中说到：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。由于我们的路径是从 1 号到 N 号，并且不能走回头路，所以可以这样想：**如果 x_i < y_i ，那么路径是下降的，符合题意；如果 x_i > y_i，那么就不将这条边填充到邻接列表里**。启发函数的选取原则就是**小于等于达到目标所需要的真实代价**，所以无论 y_i 是否为目标，我们都可以**将这一条小路径上面的 cost 值作为启发函数的值**。
定义一个邻接表 adj_list，用于存储每一行输入里面，所到达的目标节点，以及需要消耗的代价。那么对于任意一个起始点，将其作为下标输入到邻接表中，都可以返回它的目标和代价。若某个下标所对应的值为空，则返回 0。我们就把这里的代价作为启发函数的值。

```python
# 启发式函数定义：返回一个给定房间到其相邻房间的最小代价。
def heuristic(room, adj_list):
    return min(adj_list[room], default=0)
```

首先定义一个 namedtuple，存储当前节点的 f 值，位置，g 值，h 值。f 值用来作为堆排序的依据。

```python
# 定义一个命名元组，表示搜索状态，其中f是预测的总代价，room表示当前房间，g是已知代价，h是启发式函数的值。
State = namedtuple("State", ["f", "room", "g", "h"])
```

定义一些参数。设置起始位置为 1，目标状态为 N. 定义 visited 集合为空集，如果一个节点被访问过就加入这个集合。paths 作为所有可行路径的长度集合。

```python
start_room = 1 # 起始房间，固定为1
goal_room = N # 目标房间，固定为N
visited = set()  # 创建一个集合用于存储已经访问过的状态
paths = [] # 用于存储找到的路径长度
```

由于初始状态的 g 值为 0，那么将其到任意下一个节点的 cost 作为启发函数的值，也就是 f 值。定义初始状态，将 open_set 作为一个容纳所有状态的列表，并首先加入初始状态，然后进行 heapify。

```python
# 计算起始状态的代价
f = heuristic(start_room, adj_list)[1]
# 创建初始状态对象
initial_state = State(f, start_room, 0, heuristic(start_room, adj_list))
open_set = [initial_state] # 创建优先队列，初始为起始状态
heapq.heapify(open_set)  # 转为堆结构
```

然后我们就可以逐个计算与初始节点相邻的节点的 f 值，并且加入堆中，进行堆排序。每一次排序之后都取堆顶的值也就是最小值，跳到这个节点并且遍历其相邻节点，计算出 f 值，再重新加入堆中重新进行排序，直到到达终点。

由于这里有一个路径长度为 K 的限制，所以循环条件中，既要让 open_set 不为空，也要让 paths 的长度小于 K。若 open_set 为空，且 paths 的长度小于 K 了，就在循环的外面补上 -1。

如果已经到达了终点，就将其 g 值加入到 path 列表中，然后跳过循环的剩余部分。跳过循环的剩余部分是不想将 State(f, goal_room, g, 0) 加入到 open_set 中，使得多条相同长度的路径所得到的结果，也就是长度，可以得到输出。

```python
# 当优先队列不为空且未找到足够的路径时，继续搜索
while open_set and len(paths) < K:
    current_state = heapq.heappop(open_set) # 弹出代价最小的状态
    # 如果当前房间是目标房间，将其代价加入到路径列表
    if current_state.room == goal_room:
        paths.append(current_state.g)
        continue
    visited.add(current_state) # 将当前状态加入到已访问集合
```

如果没有到达终点，就查看它的邻接节点。如果它的邻接节点的 State 不在 open_set 中，就跳入下一节点，同时更新 State 的各个元素，放入 open_set 中。

这里需要注意的是，如果 next_room 没有后继节点，adj_list[next_room] 应该是 0. 所以 heuristic(next_room, adj_list) 为 0，就不是一般情况下 adj_list 取下标返回的元组类型了。所以这里需要分类讨论一下，**如果是元组，就取 [1] 下标；若不是就加 0**。

然后我们就可以更新状态，然后放进 open_set 中了。

```python
# 遍历当前房间的相邻房间
for next_room, cost in adj_list[current_state.room]:
    # 如果这个相邻房间的状态未被访问过，将其加入优先队列
    if State(f, next_room, current_state.g + cost, heuristic(next_room, adj_list)) not in visited:
        # 如果启发式函数返回一个元组，取其第二个值作为h
        if isinstance(heuristic(next_room, adj_list), tuple):
            f = current_state.g + cost + heuristic(next_room, adj_list)[1]
        else:
            f = current_state.g + cost
        new_state = State(f, next_room, current_state.g + cost, heuristic(next_room, adj_list))
        heapq.heappush(open_set, new_state)
```

如果 paths 的长度小于 K，就在最后全部补上 -1. 最后返回 paths

```python
# 如果未找到足够的K个路径，将-1填充至paths
if len(paths) < K:
    paths.extend([-1] * (K-len(paths)))
return paths
```

重要的点：

●启发式函数 (heuristic):

选取启发式函数的逻辑已经写过，它和邻接列表是基于输入数据的特定性质而选择的。

●k_shortest_paths 主要逻辑:

这个函数的目的是找到从房间 N 到房间 1 的 K 个最短路径。

使用了一个启发式搜索的变种，类似于 A*算法，但是有一些重要的区别。最主要的区别是这个函数试图找到多条路径，而不仅仅是一条最短路径。

状态中的 f 值是该状态的预测总代价，g 是从起始房间到当前房间的已知代价，h 是从当前房间到目标房间的估计代价。

当找到目标房间时，将当前代价 g 添加到 paths 列表中。如果找到的路径数量达到了 K，搜索将停止。

否则，对于当前房间的每一个相邻房间，都会生成一个新的状态并添加到优先队列中。

如果在搜索过程中没有找到足够的路径，会使用-1 来填充结果列表，直到其长度达到 K。

主函数逻辑:

对于每组输入数据，首先解析出 N（房间数）、M（边的数目）和 K（需要的路径数目）。

使用邻接列表 adj_list 表示图。对于每一条边，将其加入邻接列表。

调用 k_shortest_paths 函数，并将结果打印出来。

需要注意的是:

X_i 和 Y_i 之间是一条 X_i 指向 Y_i 的单向通道，**题目要求只能走下行通道，但给出的 X_i 不一定总小于 Y_i，也就是说，如果出现 X_i 大于 Y_i 的情况，则需要去掉!!** 对此我的做法是，**在填充邻接列表的时候进行判断，只有 x_i 小于 y_i 才能成功填充。**

```python
# 填充邻接列表
for line in lines[1:]:
    x, y, d = map(int, line.split())
    if x < y:
        #x, y = y, x
        adj_list[x].append((y, d))
```

代码文件中也都写了很详细的注释。

对应测试输入的输出：

总的来看:

每一列对应输入为老师给的测试样例

```python
# 测试数据
input_strs = ["5 6 4\n1 2 1\n1 3 1\n2 4 2\n2 5 2\n3 4 2\n3 5 2",
              "6 9 4\n1 2 1\n1 3 3\n2 4 2\n2 5 3\n3 6 1\n4 6 3\n5 6 3\n1 6 8\n2 6 4",
              "7 12 6\n1 2 1\n1 3 3\n2 4 2\n2 5 3\n3 6 1\n4 7 3\n5 7 1\n6 7 2\n1 7 10\n2 6 4\n3 4 2\n4 5 1",
              "5 8 7\n1 2 1\n1 3 3\n2 4 1\n2 5 3\n3 4 2\n3 5 2\n1 4 3\n1 5 4",
              "6 10 8\n1 2 1\n1 3 2\n2 4 2\n2 5 3\n3 6 3\n4 6 3\n5 6 1\n1 6 8\n2 6 5\n3 4 1"]
```

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

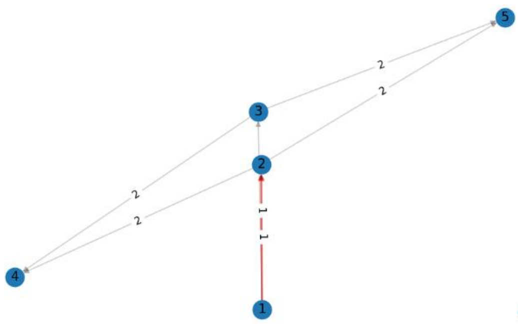
另外，我单独对每个输入进行了**更加详尽的可视化**，这些都可以在 **visual_Q2.py** 中看见。

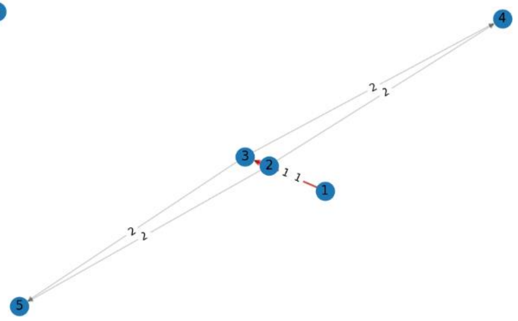
五个样例就不一一展示了，**以样例一二为例展示一下搜索过程**。

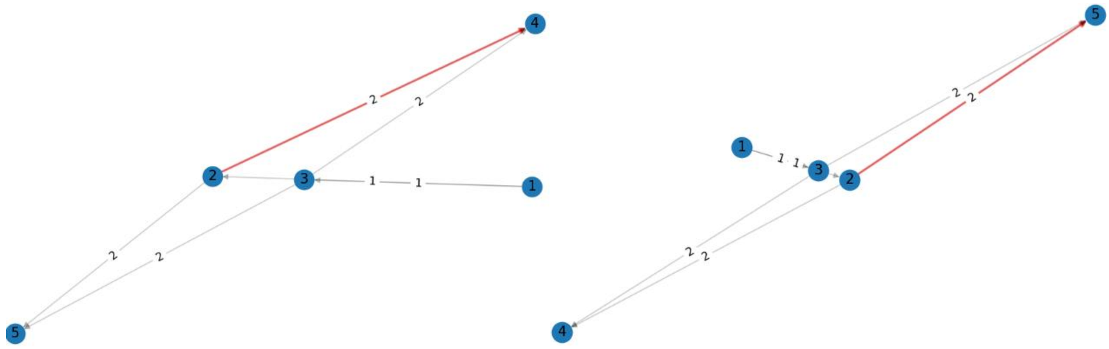
1、
```
input_strs = ["5 6 4\n1 2 1\n1 3 1\n2 4 2\n2 5 2\n3 4 2\n3 5 2"]
```

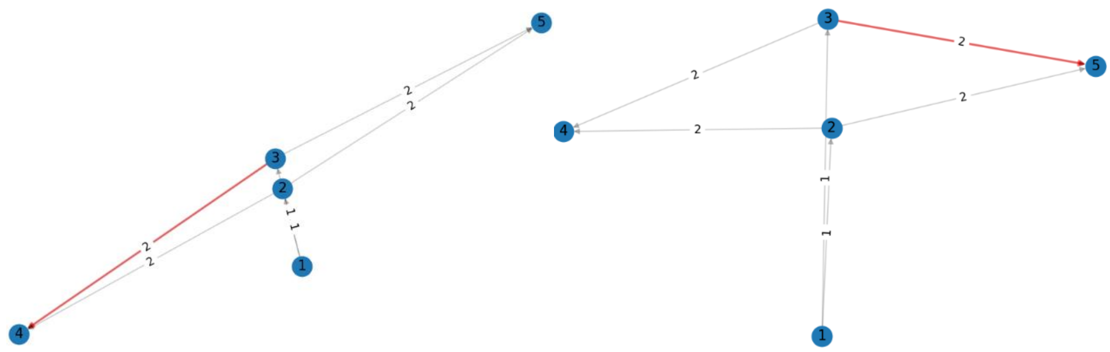
结果：路线 1-2-5 和 1-3-5 的长度都为 3，1-2-4 和 1-3-4 都无法抵达房间五

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





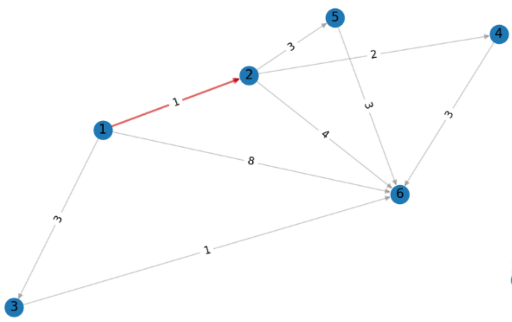


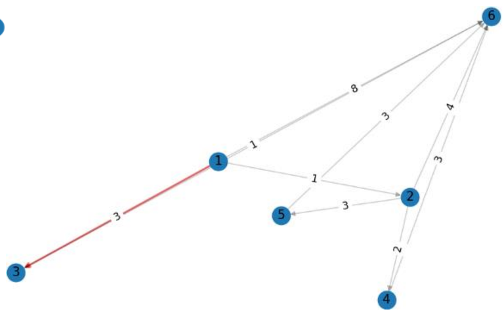


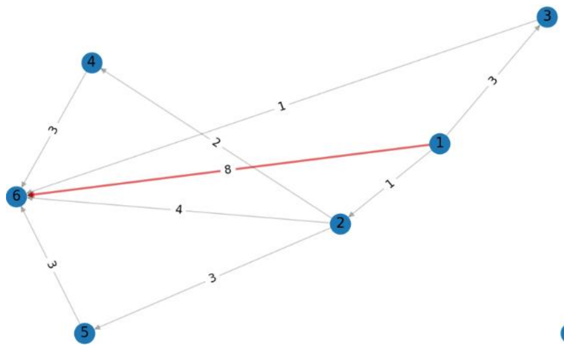
2、
```
input_strs = ["6 9 4\n1 2 1\n1 3 3\n2 4 2\n2 5 3\n3 6 1\n4 6 3\n5 6 3\n1 6 8\n2 6 4"]
```

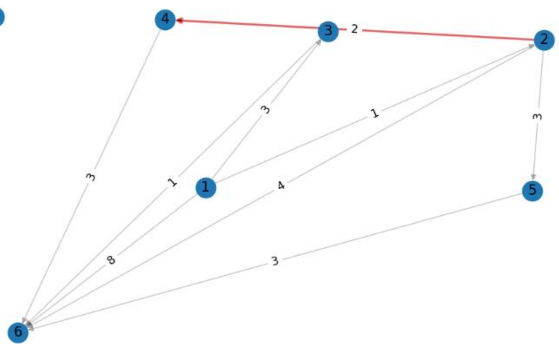
结果：路线 1-3-6 长度为 4；路线 1-2-6 长度为 5；路线 1-2-4-6 长度为 6；路线 1-2-5-6 长度为 7。

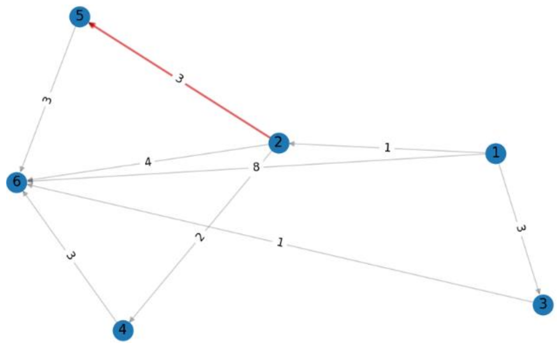
```
4
5
6
7
```

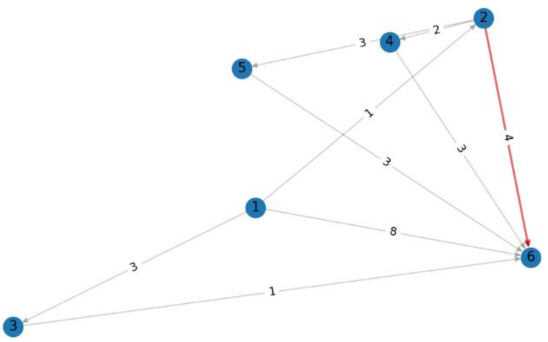


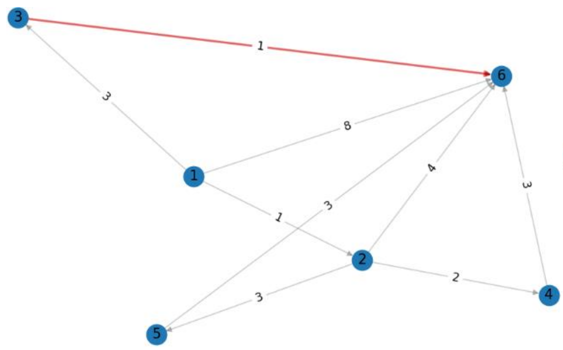


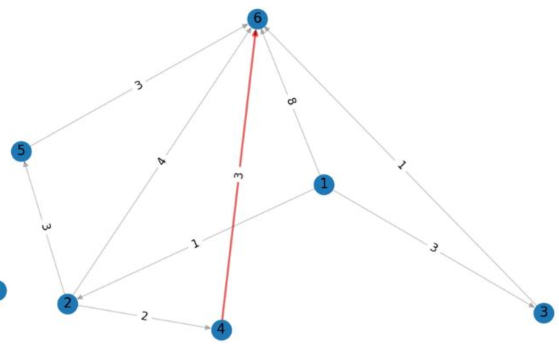


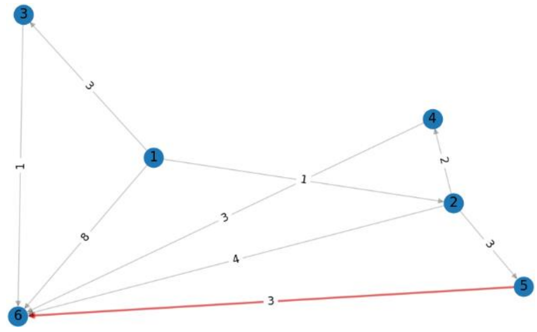












遇到的问题及解决方法：

Q1:冰雪魔方的冰霜之道

遇到的问题：

1、在状态搜索过程中，可能会存在重复扩展的状态，这导致了不必要的计算，使得程序运行时间增加。

解决措施：

1、通过使用 visited 集合来记录已经访问过的状态，避免了状态的重复扩展。

Q2:杰克的金字塔探险

遇到的问题：

1、通道信息存储问题：如何有效地存储房间间通道的信息，使得在搜索时可以快速获取。

2、我一开始并没有完全理解题目中“**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**“的含义，与助教交流以后才知道：X_i 和 Y_i 之间是一条 X_i 指向 Y_i 的单向通道，题目要求只能走下行通道，但给出的 X_i 不一定总小于 Y_i，**所以遇到 X_i 大于 Y_i 的情况应该直接舍弃这条路径**。一开始我并没有舍弃这种路径，凑巧的是题目给出的五个测试样例中 x_i 确实都小于 y_i，只有输入输出样例中有一个 515（唯一一个 x_i 大于 y_i 的情况），让我误以为我做对了。但是当我运行输入输出样例时发现运行结果为 5、5、6 而正确结果应该为 5、6、8，才发现自己做错了。

解决措施：

1、考虑使用邻接矩阵或邻接表的方式来存储通道信息。这样在搜索时可以快速地根据房间号获取与其相连的其他房间。对于大型金字塔，邻接表更为高效，因为它只存储了实际存在的通道，减少了存储空间。

2、既然 x_i 大于 y_i 的情况需要直接舍弃，那我就在主函数的部分作出改动。修改前的代码是这样的，我没有舍去 x_i 大于 y_i 的情况，而是将两者调换，即改动了路径的方向（这块也是理解错了…破坏了单向原则）。如果要直接删去这个路径的话，直接不把它加入邻接列表即可。我特别将检测输入输出样例的代码放在 Q2_test.py 文件中，运行可以看到 5 6 8 的结果。

```python
# 填充邻接列表
for line in lines[1:]:
    x, y, d = map(int, line.split())
    if x > y:
        x, y = y, x
    adj_list[x].append((y, d))
```

```python
# 填充邻接列表
for line in lines[1:]:
    x, y, d = map(int, line.split())
    if x < y:
        #x, y = y, x
        adj_list[x].append((y, d))
```

（左为修改前，输出结果为 5 5 6；右为修改后，输出结果为 5 6 8）