实验二：A*算法

10215501433 仲韦萱 实验报告

实验目的：

了解、熟悉并使用A*算法完成两道算法题：冰雪魔方的冰霜之道、 杰克的金字塔探险

实验环境：

在**VSCode**中编写和运行Python代码

实验过程

A*算法阐述：

A*算法是一种常用的启发式搜索算法，用于在图形或网络中找到最短路径。它结合了**广度优先搜索**和**最佳优先搜索**的特点，通过评估函数来选择下一步最有希望达到目标的节点。

其组成包括：

1. **开放列表**：A*算法使用一个开放列表来存储待扩展的节点。初始时，只有起始节点在开放列表中。在搜索过程中，从开放列表中选择最优的节点进行扩展。
2. **关闭列表**：A*算法使用一个关闭列表来存储已经扩展过的节点，以避免重复扩展相同的节点。
3. **启发函数**：A*算法使用一个启发函数来评估节点的优先级。启发函数估计从当前节点到目标节点的代价。这个启发函数是A*算法的关键部分，它决定了搜索的方向。*
4. **估价函数**：A*算法使用一个估价函数来评估节点的优先级。估价函数由两部分组成：g(n)表示从起始节点到当前节点的实际代价，h(n)表示从当前节点到目标节点的启发式估计。节点的优先级通过 f(n) = g(n) + h(n) 来计算，其中 f(n) 是节点的综合评估值。
5. **选择下一个节点**：从开放列表中选择具有最低 f(n) 值的节点作为下一个要扩展的节点。这意味着A*算法会根据节点的实际代价和启发式估计值来优先考虑最有希望达到目标的节点。
6. **邻居节点**：对选定的节点进行扩展，生成其相邻的未访问节点。计算并更新这些节点的 f(n) 值，并将它们添加到开放列表中。
7. **重复步骤**：重复选择下一个节点和扩展节点的过程，直到找到目标节点或开放列表为空。如果找到目标节点，则可以通过回溯路径来获取最短路径。

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。

解答思路：

这道题可以看成经典的使用A*算法解决的8数码问题。9个神秘方块我们可以假想成**九宫格**（如下图，表格中的数字是每个方块的位置索引）。

0	1	2
3	4	5
6	7	8

本道题中我把每次的位置记为一个状态：
并且题目中"滑动与空冰块相邻的冰块来改变魔块的位置"，由于空冰块0的附近有多个魔块，且数量不一定（2、3、4个都有可能），所以我假设**滑动空冰块0改变魔块的状态**。通过移动其已花费的步数和当前状态和目标状态九宫格中相异位置的个数来综合判断具体移动位置。

- **开放列表**：使用优先队列来实现。优先队列是一种特殊的队列，它根据元素的优先级进行排序，每次从队列中取出优先级最高的元素。通过put()方法将状态对象加入队列中。在每次循环中，使用get()方法从队列中取出优先级最高的状态对象。
- **关闭列表**：通常情况下，关闭列表用于存储已经访问过的状态，以避免重复访问。在上述通过优先队列来实现开放列表的前提下，已经能够确保不会重复访问同一个状态，因此没有必要显式地定义关闭列表。
- **启发函数**：$h(n)$已经确定的步数：即从起始状态到目前状态已经花费的步数。
- **估价函数**：$f(n) = g(n) + h(n), g(n:)$当前状态和目标状态九宫格中相异位置的个数。
- **选择下一个节点**：根据优先级高低选择下一个合适的状态
- **邻居节点**：对空冰块可行的移动方向生成状态并记入邻居节点（状态）
- **重复步骤**：重复选择下一个节点和邻居节点的过程，直到找到目标节点或开放列表为空。如果找到目标节点，则可以通过回溯路径来获取最短路径。

实现代码：

首先导入需要的包,本题中需要使用优先级队列：

```
from queue import PriorityQueue
```

根据题目要求定义目标顺序：

```
target_state = [1, 3, 5, 7, 0, 2, 6, 8, 4]
```

定义State类表示魔方的状态，其中包含以下四个属性：

- puzzle：当前状态，是一个列表，其中的元素表示魔方从上到下、从左到右的方块编号。
- parent：当前状态的父状态，用于记录搜索过程中状态之间的转移关系。
- move：从父状态到当前状态的移动步骤，如"up"、"down"、"left"、"right"。
- blank：空冰块0对应的位置。

```
class State:
def __init__(self, puzzle, parent=None, move=None):
        self.puzzle = puzzle
        self.parent = parent
        self.move = move
        if self.puzzle:
            self.blank = self.puzzle.index(0)
```

获取可行移动的邻居状态列表

- 定义了每个方向上的移动步骤，如第一排（位置序号为“0，1，2”）不能上移；第一列（位置序号为“0，3，6”）不能左移……根据序号特点定义移动方向和相应偏移量和移动条件，然后通过判断空位置是否满足移动条件来确定每个方向上是否可以进行移动，如果可以移动，就将当前状态和移动步骤传给State类的构造函数，创建一个新的状态对象，并将其添加到neighbors列表，其中注意的是，由于转换移动对象为空冰块，所以邻居状态只需要将空冰块和其移动方向上对应的冰块交换。

```
def get_neighbors(self):
        neighbors = []
        moves = {
            'up': -3,
            'down': 3,
            'left': -1,
            'right': 1
        }
        for move, direction in moves.items():
            if (self.blank % 3 == 0 and move == 'left') or \
                (self.blank % 3 == 2 and move == 'right') or \
                (self.blank < 3 and move == 'up') or \
                (self.blank > 5 and move == 'down'):
                continue
            new_puzzle = list(self.puzzle)
            new_puzzle[self.blank], new_puzzle[self.blank + direction] =
new_puzzle[self.blank + direction], new_puzzle[self.blank]
            neighbors.append(State(new_puzzle, self, move))
        return neighbors
```

计算启发函数：

- $h(n)$已经确定的步数。通过递归遍历状态的父节点来计算步数，直到遍历到根节点或没有父节点为止。

```
def get_cost(self):
        cost = 0
        parent = self.parent
        while parent:
            cost += 1
            parent = parent.parent
        return cost
```

计算 代价g(n)：

- 当前状态和目标状态九宫格中相异位置的个数，其中g(n)遍历了魔方中的所有方块，如果当前方块不在目标位置上，就将计数器加1。最终返回计数器的值。

```
def misplaced_tiles(self):
        count = 0
        for i in range(9):
            if self.puzzle[i] != target_state[i]:
                count += 1
        return count
```

定义状态的优先级：

- 用于在A*算法中对状态进行排序。它通过启发式函数和代价函数的和来计算状态的优先级。这里使用了错误位置的方块数量作为启发式函数，将从初始状态到当前状态的步数作为代价函数。

```
def __lt__(self, other):
        return (self.misplaced_tiles() + self.get_cost()) <
(other.misplaced_tiles() + other.get_cost())
```

判断当前状态是否为目标状态

```
def is_goal(self):
    return self.puzzle == target_state
```

计算从初始状态到目标状态的步数

```
def steps_to_goal(self):
    return self.get_cost()
```

调用上述基础函数执行A*算法：

- 创建了一个起始状态对象，并将其添加到优先队列中。然后，循环从队列中取出状态对象，并依次生成其可行的邻居状态对象并添加到优先队列中。如果当前状态是目标状态，则输出移动序列和最优策略的步数，并结束搜索。

```
def solve(puzzle):
    start_state = State(puzzle)
    queue = PriorityQueue()
    queue.put(start_state)

    while not queue.empty():
        current_state = queue.get()

        if current_state.is_goal():
            print(current_state.steps_to_goal())
            break

        for neighbor in current_state.get_neighbors():
            queue.put(neighbor)
```

然后进行测试：

- 读取测试文件，这里注意读取测试的数据，包括去除换行符、转换成int列表等。

```
with open('1_test.txt', 'r') as f:
    tests = f.readlines()
for test in tests:
    puzzle_ = test.strip()#直接使用有换行符，这里做一步处理
    puzzle = list(puzzle_)
    puzzle = [int(x) for x in puzzle]
    solve(puzzle)
```

测试用例以及测试结果：

测试用例（1_test.txt）：

1_test.txt

```
135720684
105732684
015732684
135782604
715032684
```

输出结果：

```
PS C:\Users\47517\Desktop\大三上\当代人工智能\code\2>
1
1
2
1
3
```

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：

- 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。
- 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。

解题思路

这道题可以看成经典的使用A*算法解决的K-最短路问题。这类问题中常用曼哈顿距离、欧几里得距离、最小生成树等作为启发式函数，但是对于该题都不是很适用。

本道题中我把杰克的探险比喻成从初始节点到目标节点（**目标节点 > 初始节点 ,保证下坡路** ），需要找到k个最小代价的路径的过程。那么启发式函数应该是当前节点到目标节点的最短代价（这部分用 Dijkstra 算法实现，**但仅使用于确定启发函数，最终的实现还是A*算法**）。最后通过输入的代价D和到目标节点的最短代价来综合确定最短路径。

- **开放列表**：使用最小堆来实现优先队列open_set。每个节点都有一个 f_score 表示该节点到目标节点的估计距离和该节点到起始节点的实际距离之和。在每一次迭代中，我们从 open_set 中弹出 f_score 最小的节点，然后将其邻居节点加入 open_set 中.
- **关闭列表**：通常情况下，关闭列表用于存储已经访问过的状态，以避免重复访问。但本题我没有显式的使用关闭列表，因为本题意为找最短路径，我使用了path来记录最短路径的访问节点，所以相当于close_set作用。
- **启发函数**：$h(n)$当前节点到目标节点的最短代价：我使用Dijkstra 算法实现，在这里我把节点位置互换，即把目标节点当作起始，从它出发寻找到各个节点的最短路径代价，并存在数组dis中作为启发式函数。
- **估价函数**：$f(n) = g(n) + h(n), g(n)$：输入的D代价。
- **选择下一个节点**：根据f_score对soen_set中节点的优先级进行排序，每次会从堆中弹出距离起始节点最近的节点。
- **邻居节点**：根据输入用graph（邻接表）来保存各个节点的关系和代价，，因此每次寻找邻居节点只需要在graph中寻找
- **重复步骤**：重复根据优先级选择下一个节点和扩展邻居节点以及f_score的过程，直到到达目标节点（找到一条最短路径）且找到k条最短路径。

实现代码

首先导入本道题需要的包，需要使用最小堆来实现优先队列：

```
import heapq
from queue import PriorityQueue
```

计算启发式函数：当前节点到目标节点的最短路径代价。该函数使用了Dijkstra算法和优先队列来实现。注意将节点位置互换，即起始位置为目标节点，计算到各个节点的最短路径代价。实现流程：

- 创建一个空字典dis，用于存储目标节点到图中每个节点的距离；创建一个空集合visited，用于记录已经访问过的节点；创建一个优先队列pq，用于存储待处理的节点。优先队列中的元素是一个二元组，包含节点的距离和节点编号。
- 初始化距离数组dis，将目标节点到除了自身以外的所有节点的距离设置为无穷大，将目标节点的距离设置为0。将目标节点加入优先队列pq，距离为0。

```
def heuristic(goal, graph):
    dis = {}  # 节点到图中每个节点的距离
    visited = set()  # 记录已访问过的节点
    pq = PriorityQueue()  # Dijkstra 算法的优先队列
    # 初始化距离数组，除了起始节点外都设置为无穷大
    for i in range(len(graph)):
        dis[i] = float('inf')
    dis[goal] = 0
    pq.put((0, goal))
```

进入循环，直到优先队列为空：从优先队列中取出距离最小的节点，记为dist, curr_node。

- 如果当前节点已经访问过，则跳过该节点。
- 将当前节点标记为已访问，更新邻居节点的距离，对于当前节点的每个邻居节点neighbor，计算新的距离new_dist，即当前节点的距离加上当前节点到邻居节点的边权重。如果新的距离new_dist小于邻居节点的当前距离，则更新邻居节点的距离为新的距离，并将邻居节点加入优先队列。
- 循环结束后，返回字典dis，其中包含了目标节点到图中每个节点的最短距离。

```
    while not pq.empty():
```

```
        # 从优先队列中获取距离最小的节点
        dist, curr_node = pq.get()
        if curr_node in visited:
            continue
        visited.add(curr_node)

        # 更新邻居节点的距离
        for neighbor, cost in graph[curr_node]:
            new_dist = dis[curr_node] + cost
            if new_dist < dis[neighbor]:
                dis[neighbor] = new_dist
                pq.put((new_dist, neighbor))
    return dis
```

计算k条最短路径：

- 定义函数shortest_k_paths；创建一个空的邻接表graph，用于存储图的结构。邻接表是一个长度为n的列表，每个元素都是一个空列表，用于存储与该节点相邻的节点和对应的边权重。创建另一个空的邻接表graph_，用于计算启发式函数时使用。它和graph的结构相同，但边的方向相反。**graph_是需要传入用Dijkstra算法计算启发式函数的，因为前面提到了该函数是把节点位置互换，目标节点是起始点，所以需要创建正反方向的关系邻接表；但在计算最短k条最短路的时候因为要求只能走“下坡路”，所以邻接表设置成单向的避免走“回头路”**

```
def shortest_k_paths(n, m, k, edges):
    # 构建图的邻接表表示
    graph = [[] for _ in range(n)]
    graph_ = [[] for _ in range(n)]
    for x, y, d in edges:
        graph[x-1].append((y-1, d))
        graph_[x-1].append((y-1, d))
        graph_[y-1].append((x-1, d))
```

- 初始化起始节点start_node，它包含了起始节点的f_score、g_score、节点编号和路径。这里初始的f_score和g_score都为0，路径只包含起始节点。
- 创建一个优先队列open_set，用于存储待处理的节点。将起始节点加入到优先队列中。
- 创建一个空列表k_shortest_paths，用于存储找到的k条最短路径的长度。
- 调用启发式函数heuristic计算启发式函数，并将结果保存在字典dis中。

```
    start_node = (0, 0, 0, [0])  # (f_score, g_score, node, path)
    open_set = [start_node]
    k_shortest_paths = []
    dis = heuristic(n-1,graph_)
```

进入主循环，当优先队列不为空且找到的最短路径数量小于k时执行以下操作：

- 从优先队列中弹出具有最低f_score的节点，如果当前节点是目标节点（即current_node == n-1），将g_score添加到k_shortest_paths中，表示找到了一条最短路径；否则，遍历当前节点的所有邻居节点（如果邻居节点已经在路径中，跳过该邻居节点。）
- 计算邻居节点的tentative_g_score，即当前节点的g_score加上邻居节点与当前节点的边权重。计算邻居节点的f_score，即tentative_g_score加上邻居节点到目标节点的最短距离（根据启发式函数计算）。
- 将邻居节点加入优先队列，并更新路径为当前节点的路径加上邻居节点。

```
    while open_set and len(k_shortest_paths) < k:
        # 从优先队列中弹出具有最低 f_score 的节点
        _, g_score, current_node, path = heapq.heappop(open_set)

        # 检查是否到达目标节点
        if current_node == n - 1:
            k_shortest_paths.append(g_score)

        else:
            # 检查当前节点的所有邻居
            for neighbor, distance in graph[current_node]:
                # 如果邻居节点已经在，则跳过
                if neighbor in path:
                    continue

                # 计算邻居节点的 tentative_g_score
                tentative_g_score = g_score + distance

                # 将邻居节点加入优先队列，并记录路径
                f_score = tentative_g_score + dis[neighbor]
                heapq.heappush(open_set, (f_score, tentative_g_score, neighbor,
[neighbor]))
```

有可能可走的路径本身不足k条：

- 为加以区分当找到的最短路径数量小于k时，将缺少的路径补充为"-1"。最后返回列表k_shortest_paths，其中包含了从起始节点到目标节点的k条最短路径的长度。

```
    while(len(k_shortest_paths) < k):
        k_shortest_paths.extend(["error"] * (k - len(k_shortest_paths)))
    return k_shortest_paths
```

然后进行测试：

- 注意观察测试文件，这次多个测试用例以空格隔开，且每个测试用例的第一行包含了参数：目标节点、边数和k。所以首先，使用 split('\n\n') 将读取到的字符串按照两个换行符 \n\n 分割成多个测试用例，并将这些测试用例存储在 test_cases 列表中。

```
with open('2_test.txt', 'r') as file:
    test_cases = file.read().split('\n\n')
```

- 接下来，使用 enumerate(test_cases) 遍历 test_cases 列表中的每个测试用例，并使用 i 记录当前测试用例的索引，使用 test_case 获取当前测试用例的内容。在每个测试用例中，首先使用 split('\n') 将测试用例内容按照换行符 \n 分割成多行，并将这些行存储在 lines 列表中。
- 在第一行中，使用 map(int, lines[0].split()) 将该行按照空格分割成多个字符串，并使用 map 函数将这些字符串转换为整数类型。然后，使用 N, M, K = ... 将这些整数分别赋值给变量 N, M, K，使用 for line in lines[1:] 遍历除第一行外的其他行，即遍历测试用例中的边的信息。
- 在每个边的行中，使用 map(int, line.split()) 将该行按照空格分割成多个字符串，并使用 map 函数将这些字符串转换为整数类型。然后，使用 X, Y, D = ... 将这些整数分别赋值给变量 X, Y, D，这些变量表示边的起始节点、目标节点和距离。将其作为一个元组加入到 edges 列表中
- 调用 shortest_k_paths(N, M, K, edges) 函数，并打印最后的结果。

```
for i, test_case in enumerate(test_cases):
    print(test_case)
    lines = test_case.split('\n')
    N, M, K = map(int, lines[0].split())
    edges = []
    for line in lines[1:]:
        X, Y, D = map(int, line.split())
        edges.append((X, Y, D))
    paths = shortest_k_paths(N, M, K, edges)
    for k in paths:
        print(k)
    print()
```

测试用例及测试结果

测试用例（2_test.txt）以及输出结果：

```
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
```

```
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
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
```

```
2 4 1
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
```

```
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
```

问题&解决：

1. 实现过程中，发现启发式比较好理解但是需要寻找最适合题目的，不是常见的那几种距离度量方式。
2. Q2的邻接表，一个是寻找邻居节点需要，一个是反向 Dijkstra 算法需要，刚开始只建立了正反向都有的graph，并且设立了closed_set，把每次选择过的节点都放进去，这样导致在第一条路中出现过的节点后续都不再访问且还会从走回头路，致使结果错误——后来发现，查找邻居节点的邻接表必须是单向的，且所谓的关闭列表只需要满足当条路径不再访问重复节点就可以，记录节点的path就具备这个功能，所以使用path代替关闭列表。