# 环境要求

本次实验中用到的IDE：Dataspell，Pycharm

本次实验的系统：windows11

用到的库：heapq

# 使用方法

pip install heapq

python Alg1.py

python Alg2.py

# 实验过程

## 3.1实验目的

本次实验需要利用A\*算法解决两个问题，问题一是八数码问题，问题二是K最短路径问题。

## 3.2实验思路

首先明确A\*算法的原理。A\* 是建立在 Dijkstra 和BFS 基础上的启发式遍历搜索算法，在路径规划时不仅要考虑自身与最近节点位置的距离（Dijkstra 实现），还需要考虑自身位置与目标点的距离（BFS 实现）。

A\*算法通过下面这个函数来计算每个节点的优先级。

f(n)是节点n的综合优先级。当选择下一个要遍历的节点时，我们需要选取综合优先级最高的节点。g(n) 是节点n距离起点的代价。h(n)是节点n距离终点的预计代价，这也就是A\*算法的启发函数。

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。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，分别为open\_set和close\_set。

A\*算法的思路可以理解如下：

\* 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

接下来考虑启发式函数。

在极端情况下，当启发函数h(n)始终为0，则将由g(n)完全决定节点的优先级，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。

在本次实验中，最重要的是**启发式函数的选择**。一般来说，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。曼哈顿距离可以直接应用到问题一中，而问题二中只涉及到上下的移动，所以我们将曼哈顿距离进行简化，直接将层与层之间的垂直距离，设置为启发式函数。

不过值得注意的是，A\*算法的准确程度和启发式函数有很大的关系。A\*算法在最短路径搜索中可以给出最短路径的准确长度，但前提是使用“一致的启发式函数”以及满足一定的条件：

1. 一致性：如果一个启发式函数是一致的，那么A\*算法能够确保找到的路径是最短路径。一致性的含义是，对于每个节点n和它的后继节点n'以及从n到n'的边e，满足不等式：**h(n) <= cost(e) + h(n')**

其中，h(n) 是从节点n到目标节点的启发式估计，cost(e)是边e的实际代价。这意味着启发式估计值不会高估从n到目标节点的实际最短路径的代价。如果启发式函数是一致的，A\*算法将在有限时间内找到最短路径。

2. 完备性：A\*算法也要求搜索空间是有限的，且包含了最短路径。如果搜索空间无限大或者不包含最短路径，A\*算法可能无法给出最短路径的准确长度。

3. 可列状态空间：A\*算法通常用于问题领域中的可列状态空间。也就是所有状态是有限的。

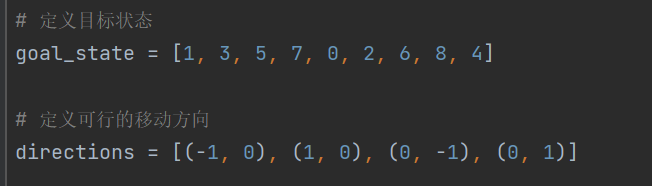
针对题目二，如果我们想得到完全准确的答案，除非我们可以找到完美的启发式函数，否则最好的解决方式是将h(n)设为0，将A\*算法退化为Dijkstra算法。我对于第二题给出的解决方法是：依旧保留垂直距离作为启发式函数，但是同时我要将g（n）的值（就像使用Dijkstra算法）进行更新，这样储存的是真实的路径长度，但是从思路上又运用了A\*算法。只是为了通过测试而存储真实距离。

## 3.3代码实现

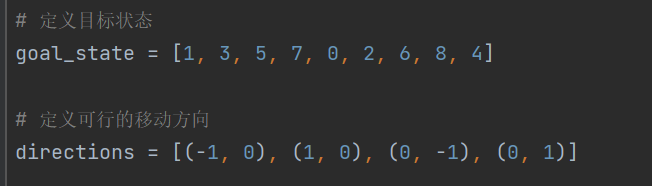
### 3.3.1 冰雪魔方的冰霜之道

1. 导入heapq库：用于构建和管理优先队列。

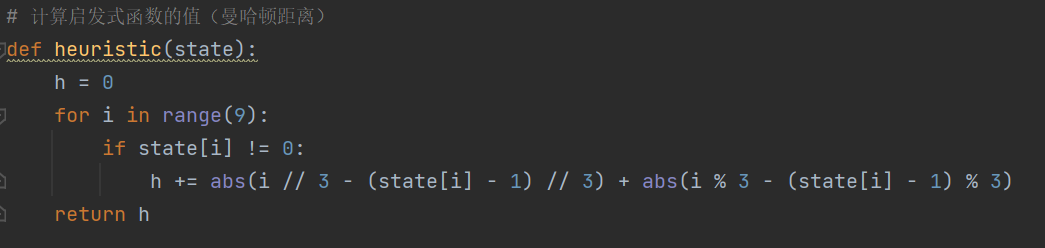
2. 定义目标状态 `goal\_state`：表示拼图的目标状态，0表示空位置。



3. 定义可行的移动方向 `directions`：每个元组表示一个可行的移动方向，分别是上、下、左、右。



4. 定义启发式函数 `heuristic`：这个函数计算了当前状态和目标状态之间的曼哈顿距离，作为启发式函数的值。曼哈顿距离是从每个数字到它在目标状态中应该所在的位置的水平和垂直距离的绝对值总和。



1. 定义 `is\_valid\_state` 函数：这个函数用于判断当前状态是否是可解的。通过计算逆序数的个数，如果是偶数个，则是可解的。

关于逆序数：在八数码问题中，每个状态由一个三行三列的方格组成，其中有八个方格被标记为数字1到8，还有一个空白方格。通过交换空白方格和相邻数字方格的位置，可以将一个状态转化为另一个状态。移动方块的过程中，如果一个方块从上往下（或从左往右）移动了，就称它的逆序数加一。

证明：对于每次拼图块的移动，它都会使得逆序数增加偶数或者奇数。举一个简单的例子，如果把一个拼图块从第 i 个位置移动到第 j 个位置，那么：

- 如果 i < j，只有下面两种情况：

- state[i] > state[j]，此时逆序数会增加 1

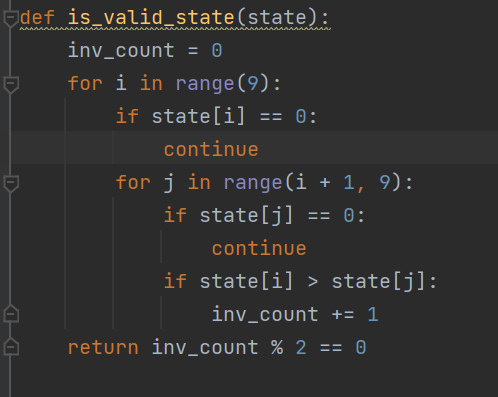
- state[i] < state[j]，此时逆序数不变

- 如果 i > j，只有下面两种情况：

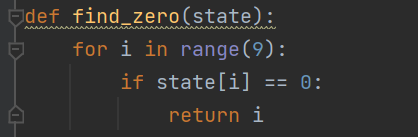
- state[i] > state[j]，此时逆序数不变

- state[i] < state[j]，此时逆序数会减少 1

因此，每次移动都会使得逆序数增加偶数或者奇数。同时，由于初始状态的逆序数是固定的，因此如果一个状态的逆序数为奇数，那么它无法通过移动拼图块的方式变成有序状态，反之，如果逆序数为偶数，那么它可以通过移动拼图块的方式变成有序状态。（详情可见Reference4）



6. 定义 `find\_zero` 函数：用于找到当前状态中的空位置（数字0的位置）。



1. 定义 `solve\_puzzle` 函数：这是A\*算法的主要求解函数。

a. 首先，检查初始状态是否可解，如果不可解，返回-1表示无解。

b. 创建一个空的集合 visited 用于记录已经访问过的状态。

c. 创建一个优先队列 pq 用于存储待扩展的状态，每个状态包括启发式函数值、步数和状态本身。

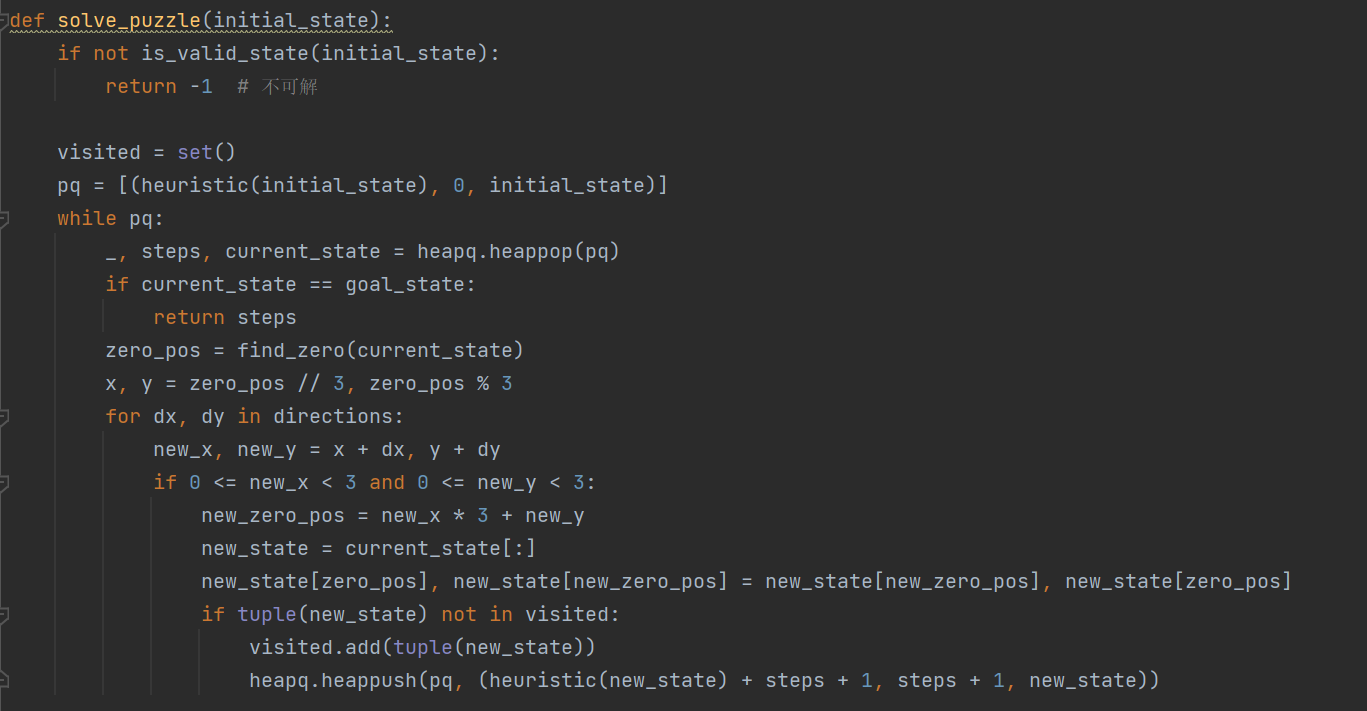
d. 进入主循环，直到队列为空或者找到目标状态为止：

- 弹出队列中启发式函数值最小的状态（函数值越小优先度越高）。

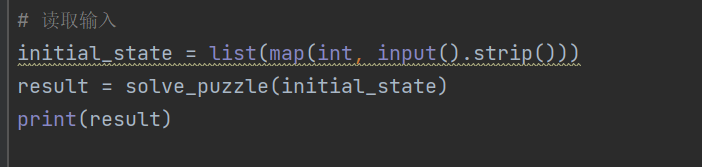
- 如果当前状态等于目标状态，返回已经经过的步数作为答案。

- 否则，找到当前状态中的空位置，然后尝试四个方向的移动，生成新的状态。

- 检查新状态是否之前未访问过，如果是则加入到 visited 中，并将新状态加入队列，更新步数和启发式函数值。



8. 读取输入：通过读取输入确立问题的开始状态并进行存储，之后调用 solve\_puzzle 函数来解决八数码问题，然后输出结果，即从初始状态到目标状态所需的最短步数。

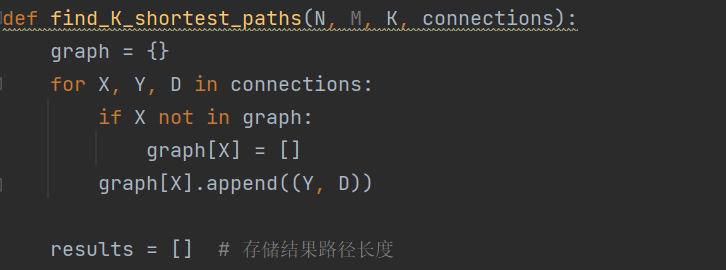


### 3.3.2 杰克的金字塔探险

1. 导入heapq库：用于构建和管理优先队列

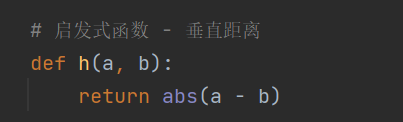
2. 定义 `find\_K\_shortest\_paths` 函数：这个函数是主要的解决方法。它接受四个参数：节点总数 N，边的总数 M，要找到的前K短路径数 K，以及边的连接信息 connections。

3. 创建图数据结构： graph来表示图的连接关系。遍历 connections 列表，将连接信息按节点X 分组，然后将(Y, D) 添加到对应节点 X 的值中，表示从节点 X 到节点 Y 的边权重为 D。



4. 初始化结果列表 results：这个列表将用于存储找到的前K短路径的长度。

5. 定义启发式函数 h(a, b)：定义了一个启发式函数，这里使用的是垂直距离 h(a, b) = abs(a - b)。启发式函数用于估计从当前节点到目标节点的距离。



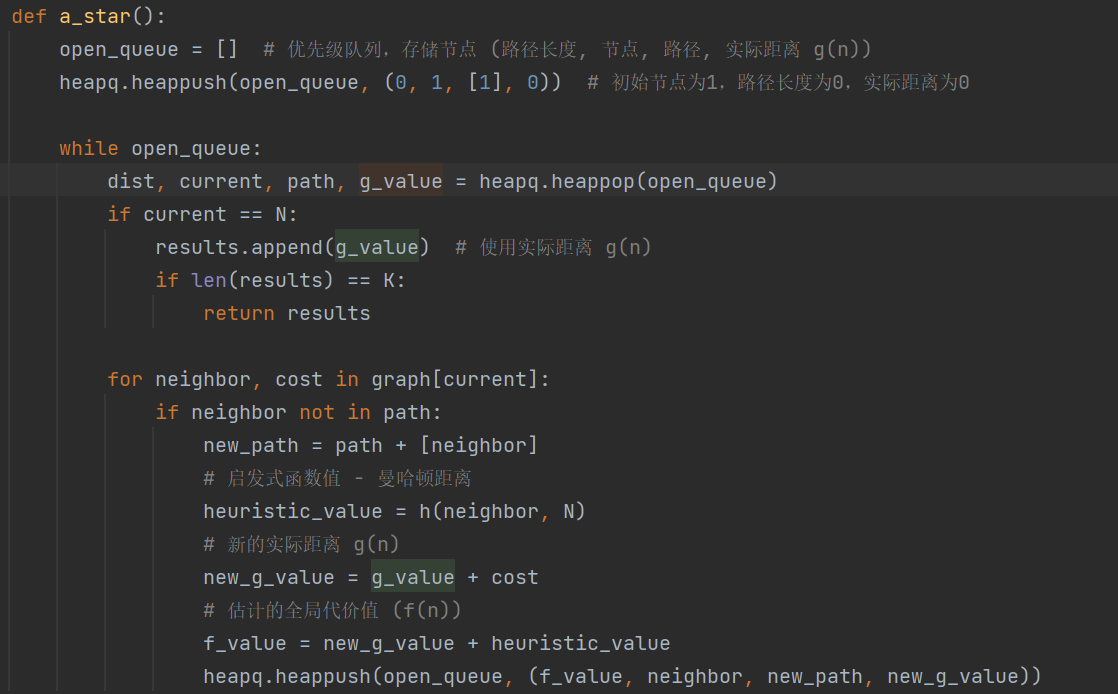
6. 定义A\*算法 a\_star：A\*搜索算法。它使用优先级队列 open\_queue 来存储节点，每个节点包括路径长度、当前节点、路径、实际距离 g(n)。

a. 初始化队列，将初始节点1加入队列，路径长度为0，实际距离 g\_value 为0。

b. 进入主循环，直到队列为空。在每次迭代中，弹出队列中具有最小估计距离的节点。

c. 如果当前节点是目标节点 N，则将找到的路径的实际距离 g\_value 添加到 results 列表中，并检查是否已经找到了K条路径。如果找到了K条路径，返回结果列表。

d. 对于当前节点，遍历与之相邻的节点，计算新的路径，启发式估计值 heuristic\_value，以及新的实际距离 new\_g\_value。然后，计算新的全局代价值 f\_value，将这个节点和路径添加到优先级队列中。



7. 调用 `a\_star` 函数：这一步执行A\*搜索，寻找前K短路径。

8. 对结果列表 `results` 进行排序：将找到的最短路径按长度进行排序。

9. 循环打印前K短路径的长度：如果找到的路径数少于K，将剩余的路径设为-1。

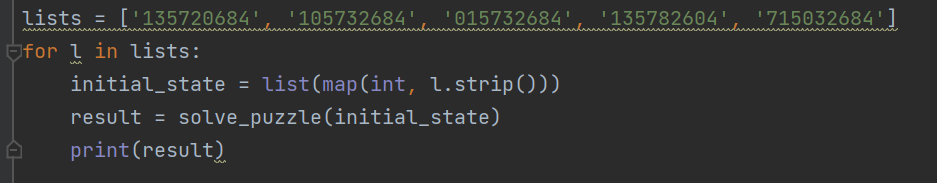
10. 读取输入：从标准输入读取节点总数 N，边的总数 M，以及连接信息 connections。

11. 调用 `find\_K\_shortest\_paths` 函数：使用输入的参数寻找前K短路径，并将结果打印输出。

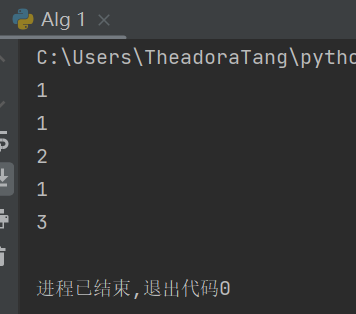
# 实验结果

## 冰雪魔方的冰霜之道

测试代码：

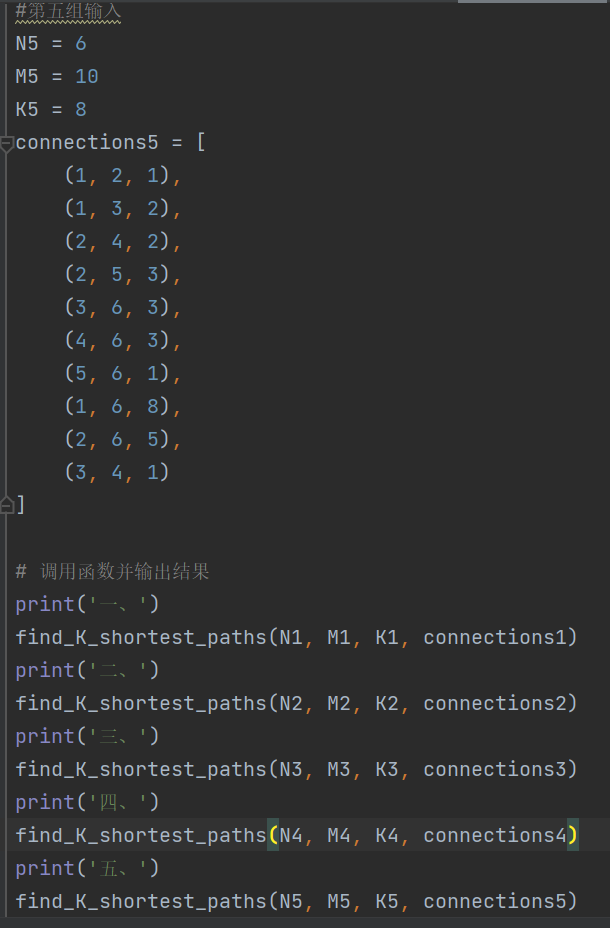


测试结果：

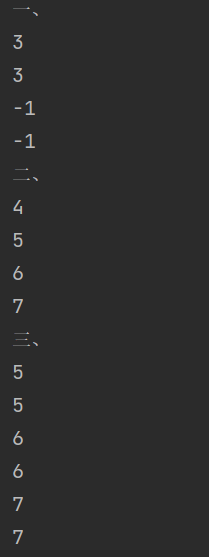
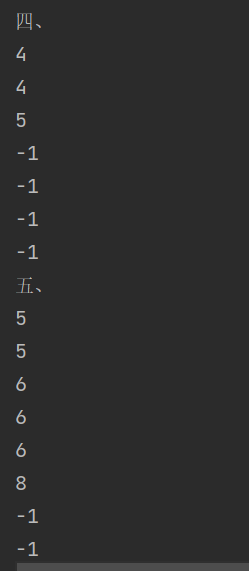


## 杰克的金字塔探险

测试代码（节选）：



测试结果：

# 总结与思考

## 遇到的问题

1. 首先是选择启发式函数。在搜索资料的过程中，我了解到曼哈顿距离非常适合作为启发式函数的原因，因为在上下左右移动的情况下，曼哈顿距离可以非常准确地估计从当前位置到目标位置的最小距离。而且曼哈顿距离的计算非常高效，只需要进行简单的绝对值运算，而不需要复杂的几何计算。最重要的是曼哈顿距离是一种一致的启发式函数。这意味着它满足启发式搜索算法的一致性要求，可以保证A算法找到的路径是最短路径。我也尝试了其他的一些函数，但是效果一般。第二问的垂直距离也是参考了曼哈顿距离，经过简化得到的。但是我认为目前的启发式函数还是不是很好，还需要进行进一步的参数调优。
2. 目前的代码内存消耗比较大，因为A\*算法使用一个优先级队列（通常是堆）来管理候选节点，这可能会导致内存消耗较大，特别是在解决大规模问题时。后续应该进行资源的优化管理，现在的代码还没有进行优化。
3. 有些问题可能没有解，或者初始状态无法达到目标状态。最开始我没有考虑到这个问题，所以在后续的时候才增添代码，避免算法陷入无限循环或错误。
4. 在最开始进行问题二实验的过程中，我误认为层与层之间是可以往返行走的，也就是假如从3可以走到4，那么从4也可以走到3，所以导致有的结果错误。后来发现题目中明确表示了是“下行”，所以对graph进行了调整。

## 对本次实验的总结

本次实验的原理和实验内容都相对简单，所以实验过程比较顺利，通过本次实验，我对A\*算法有了更加深刻的了解。我了解了如何使用A\*算法来解决实际问题，包括寻找最短路径和解决八数码问题。相信未来在解决路径规划问题非常有用。同时我也了解了启发式函数在A\*算法中的作用，以及如何选择适当的启发式函数以提高算法的效率和准确性。