一、实验内容

1.分别用广度优先搜索策略、深度优先搜索策略和启发式搜索算法（至少两种）求解八数码问题;

2.分析估价函数对启发式搜索算法的影响;

3.探究讨论各个搜索算法的特点。

二、实验设备

1. 实验设备：笔记本

2. 平台：Python Pycharm

三、实验步骤

- 1. 随机生成一个八数码问题分布，设计一个可解的目标状态（要求棋盘9个位置都不同）。

- 2. 分别用广度优先搜索策略、深度优先搜索策略和至少两种启发式搜索算法求解八数码问题。

- 3. 分析估价函数对启发式搜索算法的影响。

- 4.探究讨论各个搜索算法的特点。

- *扩展选做题：从初始状态到目标状态的变换，符合什么规律才可解（提示参考：逆序数）

四、分析说明（包括结果图表分析说明，主要核心代码及解释）

深度优先算法和广度优先算法共同用到的函数：

```
def getDirection(self): # 返回当前可用的移动方向

def getEmptyPos(self):# 查找空格的位置，返回其坐标

def generateSubStates(self): # 生成所有可能的子状态
```

1. 广度优先搜索（BFS）

广度优先搜索是一种遍历图存储结构的算法。它以队列作为核心，从始结点开始，寻找一步到达的合法可行点，并加入队列，然后依次对队列中的结点执行寻找操作，直至队列为空。

思路：

A.使用两个列表：openTable（存储待访问状态）和closeTable（存储已访问状态）。

B.初始状态被添加到openTable中，开始循环。

C.从openTable中取出节点，生成其子状态并检查是否匹配目标状态。

D.如果找到目标状态，回溯生成路径并返回。

如果未找到目标状态，继续将子状态加入openTable进行下一轮搜索。

搜索深度受限制，以避免无尽搜索。

i. 定义状态函数

初始化:

参数解释:

State：表示当前八数码的状态。它存储拼图中各个位置的数字和空格（使用空格符号表示）；

directionFlag：表示当前状态上一次移动的方向，以便在生成子状态时避免向相反方向的移动（回退）。例如，如果上一次移动是向左，生成子状态时就不会再向右移动;

Parent：用于记录当前状态的父状态。通过保存与其父状态的链接，可以在找

到目标状态后进行路径回溯，从而构造出完整的移动路径。

```
def __init__(self, state, directionFlag=None, parent=None):



    self.state = state

    self.direction = ['up', 'down', 'right', 'left']

    if directionFlag:

        self.direction.remove(directionFlag)  # 移除反方向，防止回退

    self.parent = parent

    self.symbol = ' '  # 确保空格符号的一致性
```

ii. 广度优先搜索算法实现：

```
def solve(self, max_depth=1000):

    #使用广度优先搜索解决八数码问题

    openTable = []  # 开放列表，存储待访问的节点

    closeTable = []  # 关闭列表，存储已访问的节点

    openTable.append(self)  # 将初始状态加入开放列表

    depth = 0  # 当前搜索深度


    while len(openTable) > 0 and depth < max_depth:

        n = openTable.pop(0)  # 从开放列表中取出第一个节点

        closeTable.append(n)  # 将此节点加入关闭列表

        subStates = n.generateSubStates()  # 生成子状态

        path = []
```

```
        for s in subStates:
            if (s.state == State.answer).all():  # 如果子状态是目标状态
                path.append(s)
                while s.parent:  # 回溯路径
                    path.append(s.parent)
                    s = s.parent
                path.reverse()  # 反转路径
                closeTable.extend(openTable)  # 将开放列表中的节点加入关闭列表
                return path, closeTable

        openTable.extend(subStates)  # 将子状态加入开放列表
        depth += 1  # 增加搜索深度

    return None, closeTable  # 如果没有找到目标状态，返回None
```

2. 深度优先搜索(DFS)

深度优先搜索算法是一种通过从起始节点开始，不断向下探索直到无法继续为止，然后回溯到前一节点，再继续探索其他未遍历的路径，直到遍历完整个图或树的算法。

思路：

A. 使用两个列表：openTable来存储待扩展的状态，closeTable来存储已扩展的状态,path列表用于存储从初始状态到目标状态的路径;

B. 将当前状态添加到openTable中，检查当前状态是否已经是目标状态，如果是直接将其加入路径并返回；

C. 当openTable不为空时，进入循环：从openTable中取出最后一个节点n，并将其移入closeTable；如果当前节点的深度达到10，则跳过它，以限制搜索的深度。

D.生成当前节点的所有可能子状态（通过调用generateSubStates方法）

遍历所有生成的子状态：对于每一个子状态(s)，检查它是否是目标状态。如果是，将其加入路径并进行回溯，将其父节点逐步加入路径直到回到初始状态。然后将路径反转，使得路径顺序从初始状态到目标状态。

F. 将生成的子状态添加到openTable中，准备下一轮的扩展。

G. 如果循环结束后仍未找到目标状态，则返回None，表示没有可行的路径。

搜索深度受限制，以避免无尽搜索。

i. 定义状态函数

参数解释：

State：表示当前八数码的状态。它存储拼图中各个位置的数字和空格（使用空格符号表示）；

directionFlag：表示当前状态上一次移动的方向，以便在生成子状态时避免向相反方向的移动（回退）。例如，如果上一次移动是向左，生成子状态时就不会再向右移动；

Parent：用于记录当前状态的父状态。通过保存与其父状态的链接，可以在找到目标状态后进行路径回溯，从而构造出完整的移动路径。

Depth:表示当前状态在搜索树中的深度。初始状态的深度默认为1，随着状态的

扩展，深度会增加。深度信息可以用于控制搜索的深度限制（如代码中提到的10的深度限制），从而避免过度扩展。

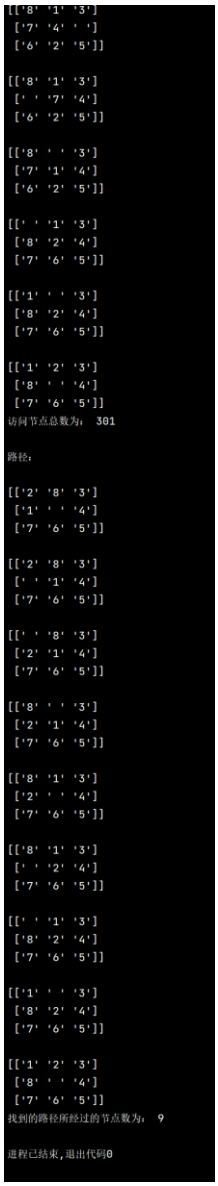
```
def __init__(self, state, directionFlag=None, parent=None, depth=1):
    # 初始化状态
    # state是State类的属性，表示3x3的八数码矩阵
    self.state = state
    # 存储可移动的方向
    self.direction = ['up', 'down', 'right', 'left']
    # 当directionFlag有值时，从方向数组中移除该值，避免回退移动
    if directionFlag:
        self.direction.remove(directionFlag)
        # 存放当前节点的父节点，用于路径回溯
    self.parent = parent
    # 表示八数码中的空格，用于空格的定位
    self.symbol = ' '
    # 表示当前节点的深度，初始节点深度设为1
    self.depth = depth
```

实验结果：

初始状态：283104765

目标状态：123804765

输出结果：



ii. 广度优先搜索算法实现：

```
def solve(self):

    openTable = []  # 存放待扩展的节点（状态）

    closeTable = []  # 存放已扩展的节点

    path = []  # 存放最终路径


    # 将当前状态加入到openTable

    openTable.append(self)
```

```
# 检查初始状态是否为目标状态
if (openTable[0].state == openTable[0].answer).all():
    path.append(openTable[0])  # 直接返回路径
    return path, closeTable

    # 当openTable不为空时
while len(openTable) > 0:
    n = openTable.pop()  # 取出openTable的最后一个节点
    closeTable.append(n)  # 将节点移入closeTable
    if n.depth == 10:
        continue  # 深度达到限制时，跳过当前节点

    # 扩展当前节点，生成子状态
    subStates = n.generateSubStates()

    for s in subStates:
        # 检查子节点是否为目标状态
        if (s.state == s.answer).all():
            path.append(s)  # 将找到的状态加入路径
            while s.parent and s.parent != originState:  # 回溯到根节点
                path.append(s.parent)
                s = s.parent
            path.reverse()  # 翻转路径，使其从根节点到目标节点
```

```
                # 返回路径和扩展的节点
                return path, closeTable

                # 将扩展的子节点加入openTable
        openTable.extend(subStates)
    return None, None  # 如果无法找到目标状态，返回None
```

实验结果：

初始状态：283104765

目标状态：123804765

输出结果：

```
[['2' '8' '3']
 ['7' ' ' '4']
 ['6' '1' '5']]

[['2' '8' '3']
 ['7' '1' '4']
 ['6' '5' ' ']]

[['1' '2' '3']
 ['8' ' ' '4']
 ['7' '6' '5']]
访问节点次数为： 26

路径：

[['2' '8' '3']
 ['1' ' ' '4']
 ['7' '6' '5']]

[['2' ' ' '3']
 ['1' '8' '4']
 ['7' '6' '5']]

[[' ' '2' '3']
 ['1' '8' '4']
 ['7' '6' '5']]

[['1' '2' '3']
 [' ' '8' '4']
 ['7' '6' '5']]

[['1' '2' '3']
 ['8' ' ' '4']
 ['7' '6' '5']]
找到的路径所经过的次数为： 5

进程已结束,退出代码0
```

初始状态：283104765

目标状态：132408765

输出结果：

```
[['1' '3' '2']
 ['4' ' ' '8']
 ['7' '6' '5']]
访问节点次数为： 922

路径：

[['2' '8' '3']
 ['1' ' ' '4']
 ['7' '6' '5']]

[['2' '8' '3']
 ['1' '4' ' ']
 ['7' '6' '5']]

[['2' '8' ' ']
 ['1' '4' '3']
 ['7' '6' '5']]

[['2' ' ' '8']
 ['1' '4' '3']
 ['7' '6' '5']]

[[' ' '2' '8']
 ['1' '4' '3']
 ['7' '6' '5']]

[['1' '2' '8']
 [' ' '4' '3']
 ['7' '6' '5']]

[['1' '2' '8']
 ['4' ' ' '3']
 ['7' '6' '5']]

[['1' '2' '8']
 ['4' '3' ' ']
 ['7' '6' '5']]

[['1' '2' ' ']
 ['4' '3' '8']
 ['7' '6' '5']]

[['1' ' ' '2']
 ['4' '3' '8']
 ['7' '6' '5']]

[['1' '3' '2']
 ['4' ' ' '8']
 ['7' '6' '5']]
找到的路径所经过的次数为： 11

进程已结束,退出代码0
```

3.启发式搜索（曼哈顿距离和错位计数）

启发式算法是相对于最优化算法提出的。一个问题的最优算法求得该问题每个实例的最优解。启发式算法可以这样定义：一个基于直观或经验构造的算法，

在可接受的花费（指计算时间和空间）下给出待解决组合优化问题每一个实例的一个可行解，该可行解与最优解的偏离程度一般不能被预计。

思路：

A. 定义一个目标状态TARGET，表示完成的八数码排列;创建一个EightPuzzle类，其中包含了构造函数来初始化起始状态、目标状态和可能的移动方向。

B. 从当前状态出发，可通过空白格的移动生成的新状态。通过确定空白格的位置，可以考虑上下左右的移动，生成所有合法的后续状态。

C. 在选择下一个搜索状态时，代码计算两个启发式值：**曼哈顿距离和错位计数。**

曼哈顿距离计算每个数字当前的位置到其目标位置的绝对行和列差值之和，得到整个状态的曼哈顿距离。而错位计数则计算当前状态中每个数字与其目标位置不相同的数量，返回错位的总数。

D. 为了优化搜索过程，使用**优先队列**来管理待探索的状态。每个状态都包含其总代价（步数加上启发式评估值），从中优先选择总代价最低的状态进行扩展。

E. 在一个循环中不断从优先队列中取出代价最低的状态，检查该状态是否为目标状态。如果是，程序则结束并返回所需的步数和路径。如果不是，生成所有可能的新状态，并将尚未访问过的状态加入队列。

F. 在找到目标状态后，程序通过记录每个状态的父状态来重建从起始状态到目标状态的具体路径。最终输出的路径显示如何一步步达到目标状态。

I.初始化函数

```
# 定义目标状态

TARGET = "123456780"  # 目标状态，'0' 代表空白

def __init__(self, start):

    self.start = start  # 初始状态

    self.target = TARGET  # 目标状态

    # 定义移动方向（右、下、左、上）

    self.directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
```

li.获取下一个状态

```
def get_next_states(self, state):

    # 获取从当前状态可以移动到的所有下一个状态

    blank_pos = self.get_blank_position(state)  # 找到空白格的位置

    row, col = divmod(blank_pos, 3)  # 将一维索引转换为二维坐标（行，列）

    next_states = []  # 存储可到达的下一状态


    for dr, dc in self.directions:

        # 遍历所有可能的移动方向

        new_row, new_col = row + dr, col + dc  # 计算新空白位置

        # 确保新位置仍在范围内

        if 0 <= new_row < 3 and 0 <= new_col < 3:

            new_blank_pos = new_row * 3 + new_col  # 新空白的索引

            new_state = list(state)  # 转换为列表以便交换

            # 交换当前空白与新的空白位置的数字
```

```
            new_state[blank_pos], new_state[new_blank_pos] =
new_state[new_blank_pos], new_state[blank_pos]
            next_states.append(''.join(new_state))  # 将列表转换为字符串并添加到结果列
表

    return next_states  # 返回所有可能的下一状态
```

Iii.曼哈顿距离启发式函数

```
# 曼哈顿距离启发式函数
def heuristic_manhattan(self, state):
    distance = 0  # 初始化总距离
    for index, value in enumerate(state):
        if value != '0':  # 跳过空白格
            target_index = int(value) - 1  # 目标位置（索引）
            target_row, target_col = divmod(target_index, 3)  # 目标位置的行和列
            current_row, current_col = divmod(index, 3)  # 当前数字的位置的行和列
            # 计算曼哈顿距离
            distance += abs(target_row - current_row) + abs(target_col - current_col)
    return distance  # 返回总的曼哈顿距离
```

Iv.错位计数启发式函数

```
# 错位计数启发式函数
def heuristic_misplaced(self, state):
    count = 0  # 初始化错位计数
    for index, value in enumerate(state):
```

```
        if value != '0' and value != str(index + 1):  # 跳过空白格并计算错位
            count += 1  # 如果当前值不在正确的位置上，计数加一
    return count  # 返回错位计数
```

V.A*算法

```
# A* 搜索算法实现
def a_star(self, heuristic_func):
    priority_queue = []  # 使用优先队列来管理状态
    # 初始化优先队列，f(n) = g(n) + h(n)，g(n) = 步数
    heapq.heappush(priority_queue, (0 + heuristic_func(self.start), 0, self.start))  # (f(n), g(n), state)

    visited = set()  # 记录已访问的状态
    parent_map = {}  # 记录每个状态的父状态，用于追踪路径

    while priority_queue:
        # 从优先队列中取出代价最低的状态
        estimated_cost, steps, current = heapq.heappop(priority_queue)

        # 如果当前状态是目标状态
        if current == self.target:
            return steps, self.reconstruct_path(parent_map, current)  # 返回步数和路径

        visited.add(current)  # 将当前状态标记为已访问
```

```
        # 生成所有可到达的下一状态
        for next_state in self.get_next_states(current):
            if next_state not in visited:  # 如果未被访问过
                g_n = steps + 1  # 更新步数
                f_n = g_n + heuristic_func(next_state)  # 计算新的f(n)
                # 将新状态加入优先队列
                heapq.heappush(priority_queue, (f_n, g_n, next_state))
                parent_map[next_state] = current  # 记录路径（父子关系）

    return -1, []  # 如果无法达到目标状态，返回-1和空路径
```

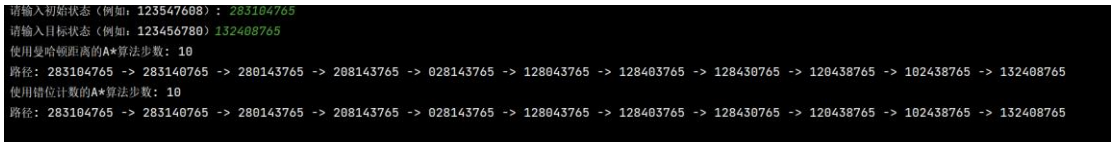
Vi.回溯路径并翻转

```
def reconstruct_path(self, parent_map, current):
    # 根据父状态重建从初始状态到目标状态的路径
    path = []#无关路径输出过多，只需要一个正确路径
    while current in parent_map:  # 追踪路径直到到达起始状态
        path.append(current)  # 添加当前状态到路径
        current = parent_map[current]  # 移动到父状态
    path.append(self.start)  # 添加初始状态
    path.reverse()  # 因为我们是从目标状态开始追踪的，所以需要反转路径
    return path  # 返回从初始状态到目标状态的路径
```

初始状态：283104765

目标状态：132408765

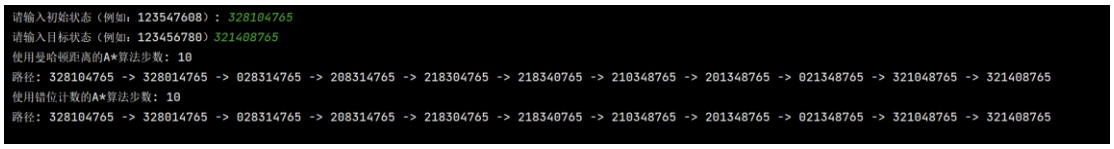
输出结果：



初始状态：328104765

目标状态：321408765

输出结果：



4.探究讨论各个搜索算法的特点

BFS:层层推进，优先访问当前深度的所有节点，然后再访问下一层，空间复杂度较高，适用于寻找最短路径的问题，例如最小步数的迷宫解法；

DFS:沿着一个路径深入探索，直到达到终点或无路可走，然后回溯，空间复杂度相对较低，适合路径不需要是最短的情况，或是需要遍历所有可能路径的场景；

启发式搜索：结合启发式信息，引导搜索向最佳路径前进。路径是否最短取决于估值函数的选择（如曼哈顿距离和错位计数）。

5.分析估价函数对启发式搜索算法的影响

I.错位计数计算更快，适合效率要求高的场景；复杂的估价函数可能提供更精确的估计，但计算开销较高，影响搜索过程的实时性。

Ii.与错位计数相比，曼哈顿距离通常在复杂度较高的状态空间中表现更好，因为它考虑了每个数字的具体位置，而不是单纯的数量统计；

Iii.错位计数能够快速计算且具有一定的有效性，但它缺乏对数字具体位置的考

虑，通常比曼哈顿距离效率低。在空间复杂度较高的情况下使用错位计数作为启发式函数可能会导致更多的节点被扩展，从而延长搜索时间。

Iv.在对实时性要求更高的应用中，错位计数可能更为合适，但在求解质量要求高的情况下，曼哈顿距离是更好的选择。

6.从初始状态到目标状态的变换，符合什么规律才可解（提示参考：逆序数）

初始状态和目标状态的逆序对奇偶性一致时，可解

原因：逆序的奇偶将所有的状态分为了两个等价类，同一个等价类中的状态都可相互到达。当左右移动空格时，逆序不变。当上下移动空格时，相当于将一个数字向前（或向后）移动两格，跳过的这两个数字要么都比它大（小），逆序可能±2；要么一个较大一个较小，逆序不变。

五、总结心得

1.在实验过程中，选择启发式搜索算法得到的路径优于BFS和DFS，且启发式搜索算法的时间空间开销较小；

2.广度优先搜索让我意识到它在寻找最短路径时的重要性。它通过逐层扩展的方式确保了找到最优解，但BFS要使用大量结点空间，空间复杂度较高；

3.深度优先搜索则展现出了其在内存占用上的优势，特别是在访问路径较深而宽度相对较小的情况中，它显得尤为高效。但是在实验中，DFS不能很好地找到最短路径，因此需要根据实际情况选择合适的算法；

4.启发式搜索算法能够显著减少搜索的时间，尤其在面对复杂问题时表现突出，

5.在实现DFS时，我忘记设置最大深度，导致程序进入死循环，即没有退出条件；

6.在实现启发式搜素算法时，没添加限制条件，导致每一步试探都输出，没有输出正确答案，后面添加条件以后输出显示正常；

附录（所有代码）

BFS搜索算法：

```
import numpy as np



class State:

    def __init__(self, state, directionFlag=None, parent=None):



        self.state = state

        self.direction = ['up', 'down', 'right', 'left']

        if directionFlag:

            self.direction.remove(directionFlag)  # 移除反方向，防止回退

        self.parent = parent

        self.symbol = ' '  # 确保空格符号的一致性



    def getDirection(self):

        #获取当前状态可以移动的方向

        return self.direction



    def getEmptyPos(self):
```

```
        # 获取空格的位置
        position = np.where(self.state == self.symbol)
        return position[0][0], position[1][0]  # 返回空格位置的行和列

    def generateSubStates(self):
        #生成当前状态的所有子状态
        if not self.direction:
            return []
        subStates = []
        boarder = len(self.state) - 1  # 边界限制
        row, col = self.getEmptyPos()  # 获取空格位置

        # 向左移动
        if 'left' in self.direction and col > 0:
            s = self.state.copy()
            s[row, col], s[row, col - 1] = s[row, col - 1], s[row, col]
            news = State(s, directionFlag='right', parent=self)
            subStates.append(news)

        # 向上移动
        if 'up' in self.direction and row > 0:
            s = self.state.copy()
            s[row, col], s[row - 1, col] = s[row - 1, col], s[row, col]
```

```
            news = State(s, directionFlag='down', parent=self)
            subStates.append(news)

        # 向右移动
        if 'right' in self.direction and col < boarder:
            s = self.state.copy()
            s[row, col], s[row, col + 1] = s[row, col + 1], s[row, col]
            news = State(s, directionFlag='left', parent=self)
            subStates.append(news)

        # 向下移动
        if 'down' in self.direction and row < boarder:
            s = self.state.copy()
            s[row, col], s[row + 1, col] = s[row + 1, col], s[row, col]
            news = State(s, directionFlag='up', parent=self)
            subStates.append(news)

        return subStates

    def solve(self, max_depth=1000):
        #使用广度优先搜索解决八数码问题
        openTable = []  # 开放列表，存储待访问的节点
```

```
closeTable = []  # 关闭列表，存储已访问的节点
openTable.append(self)  # 将初始状态加入开放列表
depth = 0  # 当前搜索深度

while len(openTable) > 0 and depth < max_depth:
    n = openTable.pop(0)  # 从开放列表中取出第一个节点
    closeTable.append(n)  # 将此节点加入关闭列表
    subStates = n.generateSubStates()  # 生成子状态
    path = []

    for s in subStates:
        if (s.state == State.answer).all():  # 如果子状态是目标状态
            path.append(s)
            while s.parent:  # 回溯路径
                path.append(s.parent)
                s = s.parent
            path.reverse()  # 反转路径
            closeTable.extend(openTable)  # 将开放列表中的节点加入关闭列表
            return path, closeTable

    openTable.extend(subStates)  # 将子状态加入开放列表
    depth += 1  # 增加搜索深度
```

```
        return None, closeTable  # 如果没有找到目标状态，返回None


if __name__ == '__main__':
    # 定义八数码初始状态
    originState = State(np.array([[2, 8, 3], [1, ' ', 4], [7, 6, 5]]))
    # 确定八数码的目标状态
    State.answer = np.array([[1, 2, 3], [8, ' ', 4], [7, 6, 5]])
    s1 = State(state=originState.state)
    path, closeTable = s1.solve()

    # 输出访问的节点
    print('访问的结点有：')
    for node in closeTable:
        print(node.state)
        print('')
    print(State.answer)
    print('访问节点次数为：', len(closeTable) + 1)

    # 输出找到的路径
    if path:
        print('\n路径：')
        for node in path:
```

```
            print('')
            print(node.state)
        print('找到的路径所经过的次数为：', len(path))
    else:
        print('无法找到路径')
```

DFS搜索算法：

```
import numpy as np


# DFS算法
# 设置深度上限为10
class State:
    def __init__(self, state, directionFlag=None, parent=None, depth=1):
        # 初始化状态
        # state是State类的属性，表示3x3的八数码矩阵
        self.state = state
        # 存储可移动的方向
        self.direction = ['up', 'down', 'right', 'left']
        # 当directionFlag有值时，从方向数组中移除该值，避免回退移动
        if directionFlag:
            self.direction.remove(directionFlag)
            # 存放当前节点的父节点，用于路径回溯
        self.parent = parent
```

```
        # 表示八数码中的空格，用于空格的定位
        self.symbol = ' '
        # 表示当前节点的深度，初始节点深度设为1
        self.depth = depth


    def getDirection(self):
        # 返回当前可用的移动方向
        return self.direction


    def showInfo(self):
        # 显示当前状态的信息
        for i in range(3):
            for j in range(3):
                print(self.state[i, j], end=' ')
            print('\n')  # 换行
        print('->')
        return


    def getEmptyPos(self):
        # 查找空格的位置，返回其坐标
        position = np.where(self.state == self.symbol)
        return position
```

```
    def generateSubStates(self):
        # 生成所有可能的子状态
        if not self.direction:
            return []  # 如果没有可用方向，返回空列表
        subStates = []  # 存放生成的子状态
        border = len(self.state) - 1  # 矩阵的边界
        row, col = self.getEmptyPos()  # 获取空格的位置

        # 向上移动
        if 'up' in self.direction and row > 0:
            s = self.state.copy()  # 复制当前状态
            temp = s.copy()  # 再次复制以便于交换
            s[row, col] = s[row - 1, col]  # 与上方元素交换
            s[row - 1, col] = temp[row, col]
            news = State(s, directionFlag='down', parent=self, depth=self.depth + 1)
# 创建新状态
            subStates.append(news)  # 添加到子状态列表

        # 向下移动
        if 'down' in self.direction and row < border:
            s = self.state.copy()
            temp = s.copy()
```

```
            s[row, col] = s[row + 1, col]
            s[row + 1, col] = temp[row, col]
            news = State(s, directionFlag='up', parent=self, depth=self.depth + 1)
            subStates.append(news)

            # 向左移动
        if 'left' in self.direction and col > 0:
            s = self.state.copy()
            temp = s.copy()
            s[row, col] = s[row, col - 1]
            s[row, col - 1] = temp[row, col]
            news = State(s, directionFlag='right', parent=self, depth=self.depth + 1)
            subStates.append(news)

            # 向右移动
        if 'right' in self.direction and col < border:
            s = self.state.copy()
            temp = s.copy()
            s[row, col] = s[row, col + 1]
            s[row, col + 1] = temp[row, col]
            news = State(s, directionFlag='left', parent=self, depth=self.depth + 1)
            subStates.append(news)
```

```
        return subStates  # 返回所有生成的子状态

    def solve(self):
        openTable = []  # 存放待扩展的节点（状态）
        closeTable = []  # 存放已扩展的节点
        path = []  # 存放最终路径

        # 将当前状态加入到openTable
        openTable.append(self)
        # 检查初始状态是否为目标状态
        if (openTable[0].state == openTable[0].answer).all():
            path.append(openTable[0])  # 直接返回路径
            return path, closeTable


            # 当openTable不为空时
        while len(openTable) > 0:
            n = openTable.pop()  # 取出openTable的最后一个节点
            closeTable.append(n)  # 将节点移入closeTable
            if n.depth == 10:
                continue  # 深度达到限制时，跳过当前节点

            # 扩展当前节点，生成子状态
            subStates = n.generateSubStates()
```

```
            for s in subStates:
                # 检查子节点是否为目标状态
                if (s.state == s.answer).all():
                    path.append(s)  # 将找到的状态加入路径
                    while s.parent and s.parent != originState:  # 回溯到根节点
                        path.append(s.parent)
                        s = s.parent
                    path.reverse()  # 翻转路径，使其从根节点到目标节点
                    # 返回路径和扩展的节点
                    return path, closeTable

                    # 将扩展的子节点加入openTable
            openTable.extend(subStates)
        return None, None  # 如果无法找到目标状态，返回None


if __name__ == '__main__':
    # 定义八数码初始状态
    originState = State(np.array([[2, 8, 3], [1, ' ', 4], [7, 6, 5]]))
    # 确定八数码的目标状态
    State.answer = np.array([[1, 2, 3], [8, ' ', 4], [7, 6, 5]])
```

```python
# 创建一个状态对象s1
s1 = State(state=originState.state)

# 进行求解，得到路径和访问的节点
path, closeTable = s1.solve()
print('访问的节点有：')
for node in closeTable:
    print(node.state)  # 打印每个访问的状态
    print('')

print(State.answer)  # 打印目标状态
print('访问节点总数为：', len(closeTable) + 1)  # 输出访问节点的总数

# 只显示在path数组中的矩阵
if path:
    print('\n路径：')
    for node in path:
        print('')
        print(node.state)  # 打印路径中的每个状态
    print('找到的路径所经过的节点数为：', len(path))  # 输出路径的节点数
else:
    print('无法找到路径')  # 如果未能找到路径，输出提示信息
```

启发式搜索（曼哈顿距离和错位计数）

```
import heapq



# 定义目标状态

TARGET = "123456780"  # 目标状态，'0' 代表空白


class EightPuzzle:

    def __init__(self, start):

        self.start = start  # 初始状态

        self.target = TARGET  # 目标状态

        # 定义移动方向（右、下、左、上）

        self.directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]



    def get_blank_position(self, state):

        # 获取空白格（'0'）的位置

        return state.index('0')  # 返回'0'的索引


    def get_next_states(self, state):

        # 获取从当前状态可以移动到的所有下一个状态

        blank_pos = self.get_blank_position(state)  # 找到空白格的位置

        row, col = divmod(blank_pos, 3)  # 将一维索引转换为二维坐标（行，列）

        next_states = []  # 存储可到达的下一状态
```

```
        for dr, dc in self.directions:
            # 遍历所有可能的移动方向
            new_row, new_col = row + dr, col + dc  # 计算新空白位置
            # 确保新位置仍在范围内
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_blank_pos = new_row * 3 + new_col  # 新空白的索引
                new_state = list(state)  # 转换为列表以便交换
                # 交换当前空白与新的空白位置的数字
                new_state[blank_pos], new_state[new_blank_pos] =
new_state[new_blank_pos], new_state[blank_pos]
                next_states.append(''.join(new_state))  # 将列表转换为字符串并添加到结
果列表

        return next_states  # 返回所有可能的下一状态

    # 曼哈顿距离启发式函数
    def heuristic_manhattan(self, state):
        distance = 0  # 初始化总距离
        for index, value in enumerate(state):
            if value != '0':  # 跳过空白格
                target_index = int(value) - 1  # 目标位置（索引）
                target_row, target_col = divmod(target_index, 3)  # 目标位置的行和列
                current_row, current_col = divmod(index, 3)  # 当前数字的位置的行和
列
```

```
                # 计算曼哈顿距离
                distance += abs(target_row - current_row) + abs(target_col -
current_col)
        return distance  # 返回总的曼哈顿距离

    # 错位计数启发式函数
    def heuristic_misplaced(self, state):
        count = 0  # 初始化错位计数
        for index, value in enumerate(state):
            if value != '0' and value != str(index + 1):  # 跳过空白格并计算错位
                count += 1  # 如果当前值不在正确的位置上，计数加一
        return count  # 返回错位计数

    # A* 搜索算法实现
    def a_star(self, heuristic_func):
        priority_queue = []  # 使用优先队列来管理状态
        # 初始化优先队列，f(n) = g(n) + h(n)，g(n) = 步数
        heapq.heappush(priority_queue, (0 + heuristic_func(self.start), 0, self.start))  #
(f(n), g(n), state)

        visited = set()  # 记录已访问的状态
        parent_map = {}  # 记录每个状态的父状态，用于追踪路径

        while priority_queue:
```

```
            # 从优先队列中取出代价最低的状态
            estimated_cost, steps, current = heapq.heappop(priority_queue)

            # 如果当前状态是目标状态
            if current == self.target:
                return steps, self.reconstruct_path(parent_map, current)  # 返回步数和路径

            visited.add(current)  # 将当前状态标记为已访问

            # 生成所有可到达的下一状态
            for next_state in self.get_next_states(current):
                if next_state not in visited:  # 如果未被访问过
                    g_n = steps + 1  # 更新步数
                    f_n = g_n + heuristic_func(next_state)  # 计算新的f(n)
                    # 将新状态加入优先队列
                    heapq.heappush(priority_queue, (f_n, g_n, next_state))
                    parent_map[next_state] = current  # 记录路径（父子关系）

        return -1, []  # 如果无法达到目标状态，返回-1和空路径

    def reconstruct_path(self, parent_map, current):
        # 根据父状态重建从初始状态到目标状态的路径
        path = []#无关路径输出过多，只需要一个正确路径
        while current in parent_map:  # 追踪路径直到到达起始状态
```

```
            path.append(current)  # 添加当前状态到路径
            current = parent_map[current]  # 移动到父状态
        path.append(self.start)  # 添加初始状态
        path.reverse()  # 因为我们是从目标状态开始追踪的，所以需要反转路径
        return path  # 返回从初始状态到目标状态的路径

# 示例使用
if __name__ == "__main__":
    start_state = input("请输入初始状态（例如：123547608）: ")  # 手动输入初始状态
    TARGET=input("请输入目标状态（例如：123456780）")
    puzzle = EightPuzzle(start_state)  # 创建八数码问题实例

    # 使用曼哈顿距离启发式
    steps, path = puzzle.a_star(puzzle.heuristic_manhattan)
    print("使用曼哈顿距离的A*算法步数:", steps)  # 输出步数
    print("路径:", ' -> '.join(path))  # 输出路径

    # 使用错位计数启发式
    steps, path = puzzle.a_star(puzzle.heuristic_misplaced)
    print("使用错位计数的A*算法步数:", steps)  # 输出步数
    print("路径:", ' -> '.join(path))  # 输出路径
```