**《人工智能》实验报告**

实验名称 A\*算法解决八数码问题

学生姓名 黄鸿展

学　　　　号 202230441138

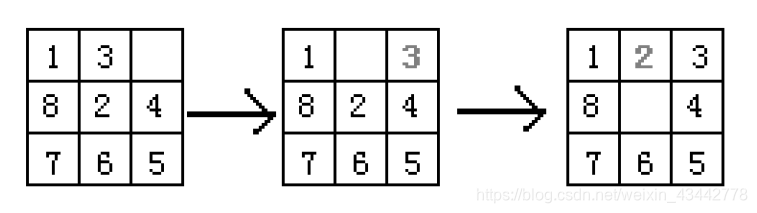
学生班级 计算机科学与技术2班

实验日期 2024年11月7日

实验成绩

一、实验内容

本实验旨在通过实现A\*算法来解决经典的八数码问题。八数码问题是一个经典的搜索问题，目标是通过一系列移动将一个3x3的数字拼图转换为一个目标状态，如下图所示。拼图中的数字1到8各有一个，而0代表一个空格，可以向上、下、左、右移动。



而A\*算法是一种在图形平面上，有多个节点的路径中，寻找一条从起始点（source）到目标点（goal）的最短遍历路径的算法。它使用启发式方法来计算图中的节点，从而减少实际计算的节点数量，提高搜索效率。

二、实验过程

1．核心思想

A\*算法的启发函数为,其中，是从初始状态经由状态n到目标状态的代价估计， 是在状态空间中从初始状态到状态n的实际代价，是从状态n到目标状态的最佳路径的估计代价。是启发函数中很重要的一项，它是对当前状态到目标状态的最小代价 的一种估计，且需要满足

也就是说h(n)是h\*(n)的下界，这一要求保证了Astar算法能够找到最优解。

A\*算法的步骤如下：

初始化：将起点加入开放表(OpenList)，设置起始点的g值为0，f值为h值。

循环：当开放列表不为空时，执行以下步骤：

1. 从开放列表中选择具有最小f值的节点。
2. 如果该节点是目标节点，则搜索完成，回溯路径。
3. 否则，将该节点从开放表移至封闭表(ClosedList)，扩展其邻居节点。
4. 对于每个邻居节点，如果它不在封闭列表中，计算其g值和f值，如果f值比已知的值小，则更新其路径和g、f值，并将其加入开放列表。

结束：如果开放列表为空，搜索失败，没有找到目标节点。

2. 算法设计

首先定义：

g值（实际代价）

g值表示从初始状态到当前状态的代价。在八数码问题中，这个代价就是移动次数。计算方法：如果当前状态的previous属性不是None（即当前状态不是初始状态），则当前状态的g值是前一个状态的g值加1。如果是初始状态，则g值为0。

h值（启发式估计代价）

h值是启发式函数，用于估计从当前状态到目标状态的最小代价。在八数码问题中，使用曼哈顿距离作为h值的估计，即每个数字到其目标位置的行和列距离之和。

计算方法：h值是所有数字与其目标位置之间的曼哈顿距离之和。find\_position方法用于找到每个数字在当前状态中的位置。

f值（总代价估计）

f值是g值和h值的和，表示从初始状态通过当前状态到达目标状态的总代价。

a\_star\_solver函数实现了A\*算法的主要逻辑：

1. 初始化状态：创建一个 PuzzleGrid 对象，表示初始状态。
2. 检查初始状态是否可解（通过计算逆序数）。
3. 设置开放集和封闭集：

open\_set 用于存放待扩展的状态，closed\_set 用于存放已经扩展的状态。

1. 将初始状态添加到 open\_set 中。
2. 主循环：

当 open\_set 不为空时，执行以下步骤：

* 1. 排序：根据每个状态的 F值（F=G + H）对 open\_set 进行排序。
  2. 取出当前状态：从 open\_set 中取出 F 值最小的状态（即当前状态）。
  3. 显示当前状态：输出当前状态及其 F、G、H 值。
  4. 检查目标状态：

如果当前状态的H值为 0，表示已找到目标状态，则：

输出找到解决方案的消息和移动次数。

显示完整的解决路径，返回。

* 1. 扩展状态：

将当前状态添加到 closed\_set中，表示它已经被扩展。

扩展当前状态，生成所有可能的后续状态（将空白格移动到相邻位置）。

对于每个新状态，进行以下检查：

使用 state\_exists函数检查新状态是否在open\_set或closed\_set中。

如果新状态不在两个集合中，记录扩展次数并将其添加到open\_set。

如果新状态在closed\_set中且F值低于已存状态，则更新closed\_set及open\_set。

如果新状态在 open\_set 中且其 F 值低于已存的状态，则更新 open\_set。

3. 具体实现

PuzzleGrid类:用于表示八数码问题的状态。

属性：

previous：指向前一个状态的引用，用于回溯路径。

target\_state：目标状态，即解决状态。

state：当前状态。

g、h、f：分别表示从起始状态到当前状态的实际代价、估计代价和总代价。

方法：

calculate\_costs()：计算当前状态的g、h和f值。

calculate\_g()：计算从起始状态到当前状态的移动次数。

calculate\_h()：使用曼哈顿距离计算启发式函数h。

calculate\_f()：计算总代价f = g + h。

display()：显示当前状态和代价。

display\_solution\_path()：回溯并显示整个解决方案路径。

find\_position()：查找特定值在状态中的位置。

find\_blank()：查找空白（0）的位置。

expand\_positions()：生成当前状态所有可能的后继状态。

move\_blank()：实现空白格的移动。

state\_exists：状态检查

state\_exists函数用于检查某个状态是否已存在于开放列表或封闭列表中，以避免重复处理。

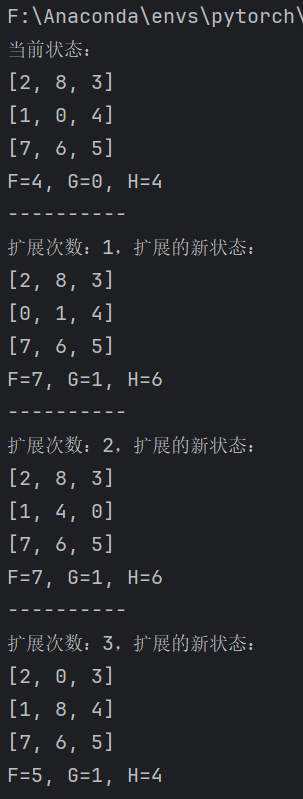
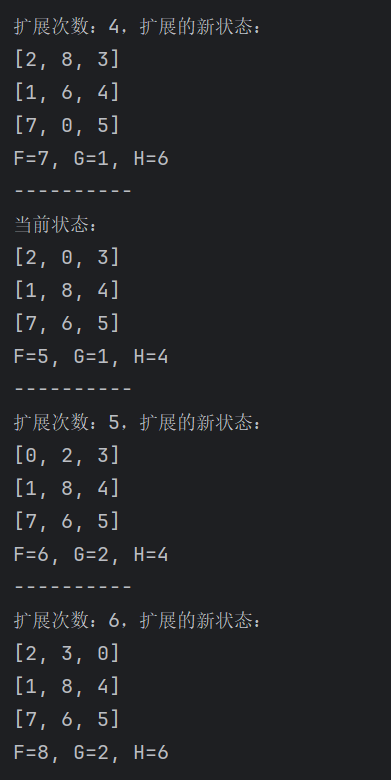
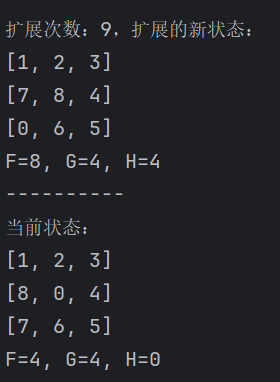
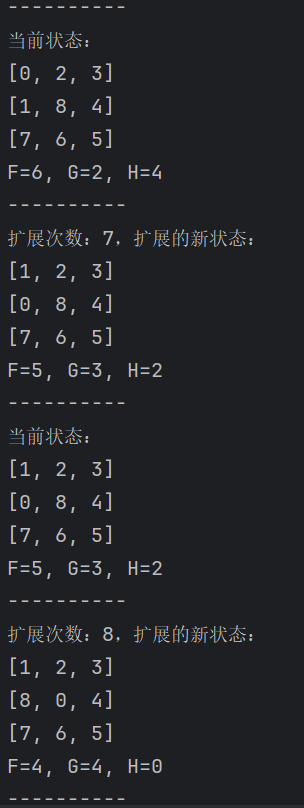
is\_solvable：解的可行性检查

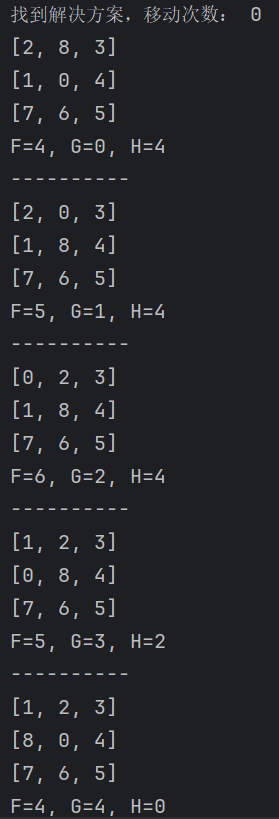
is\_solvable函数通过计算初始状态和目标状态的逆序数的奇偶性来判断问题是否有解。如果奇偶性相同，则问题有解。

count\_inversions：逆序数计算

count\_inversions函数用于计算一个状态的逆序数，即有多少对数字的顺序与它们在排序后应有的顺序相反

三、实验结果



用树状图表示如下：



遍历9次，而最终解用红色路线标注，可以看到A\*算法每次对启发函数值最小的状态进行扩展，进行搜索（去掉重复），再选出最优解。

四、实验心得

通过本次实验，我深入理解了A算法的原理和实现过程。A算法是一种有效的启发式搜索算法，它通过评估每个节点的代价来寻找最优解。在八数码问题中，A\*算法能够高效地找到从初始状态到目标状态的最短路径。实验过程中，我学会了如何设计和实现一个搜索问题的状态空间，以及如何通过启发式函数来优化搜索过程。此外，我也体会到了算法的可行性检查对于问题解决的重要性。

五、附录（关键代码）

|  |
| --- |
| import copy class PuzzleGrid:     def \_\_init\_\_(self, state):         self.previous = None  *# 用于追踪父节点*         self.target\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]           self.state = state  *#*         self.find\_blank()           self.calculate\_costs()  *# 计算当前状态的G、H、F值*     def calculate\_costs(self):         self.calculate\_h()  *# 计算H值（启发值）*         self.calculate\_g()  *# 计算G值（从起始状态到当前状态的步数）*         self.calculate\_f()  *# 计算F值（总成本）*     def calculate\_g(self):         *# 如果有父节点，步数加一；否则步数为0*         if self.previous is not None:             self.g = self.previous.g + 1         else:             self.g = 0     def calculate\_h(self):         *# 计算每个数字到目标位置的曼哈顿距离之和作为H值*         self.h = 0         for i in range(3):             for j in range(3):                 target\_value = self.target\_state[i][j]                 current\_position = self.find\_position(target\_value)                 self.h += abs(current\_position[0] - i) + abs(current\_position[1] - j)     def calculate\_f(self):         *# F值为G值和H值之和*         self.f = self.g + self.h     def display(self):         for row in self.state:             print(row)         print(f"F={self.f}, G={self.g}, H={self.h}")         print("-" \* 10)     def display\_solution\_path(self):         *# 显示从起始状态到达当前状态的路径*         path = []         path.append(self)  *# 添加当前状态到路径中*         node = self.previous  *# 从父节点开始*         while node:             path.append(node)  *# 逐步向上添加父节点*             node = node.previous         path.reverse()  *# 反转路径以从起始到目标显示*         for state in path:             state.display()  *# 显示每个状态*     def find\_position(self, value):         *# 查找数字在棋盘中的位置*         for i in range(3):             if value in self.state[i]:                 j = self.state[i].index(value)  *# 找到数字的列索引*                 return [i, j]  *# 返回行列坐标*     def find\_blank(self):         *# 找到空白（0）的位置*         self.blank\_position = self.find\_position(0)     def expand\_positions(self):         *# 扩展当前状态，生成可能的后续状态*         i, j = self.blank\_position  *# 获取空白格的位置*         expanded\_states = []         *# 检查四个方向是否可以移动*         if j > 0:  *# 向左移动*             expanded\_states.append(self.move\_blank(0, -1))         if j < 2:  *# 向右移动*             expanded\_states.append(self.move\_blank(0, 1))         if i > 0:  *# 向上移动*             expanded\_states.append(self.move\_blank(-1, 0))         if i < 2:  *# 向下移动*             expanded\_states.append(self.move\_blank(1, 0))         return expanded\_states       def move\_blank(self, row\_offset, col\_offset):         *# 移动空白格，并返回新的状态*         new\_state = copy.deepcopy(self.state)  *# 深度复制当前状态*         target\_value = new\_state[self.blank\_position[0] + row\_offset][self.blank\_position[1] + col\_offset]         new\_state[self.blank\_position[0]][self.blank\_position[1]] = target\_value          new\_state[self.blank\_position[0] + row\_offset][self.blank\_position[1] + col\_offset] = 0          return new\_state  *# 返回新的状态* def state\_exists(state, state\_list):     *# 检查给定状态是否在状态列表中*     for index, existing\_state in enumerate(state\_list):         if existing\_state.state == state.state:  *# 若状态相同*             return [True, index]  *# 返回存在标记和索引*     return [False, -1]  def count\_inversions(array):     *# 计算逆序数*     inversions = 0     for i in range(len(array)):         if array[i] != 0:              for j in range(i):                 if array[j] > array[i]:                      inversions += 1     return inversions  def is\_solvable(initial\_state, target\_state):     *# 判断初始状态是否可解*     inv\_count\_initial = count\_inversions([num for row in initial\_state for num in row])  *# 获取初始状态的逆序数*     inv\_count\_target = count\_inversions([num for row in target\_state for num in row])  *# 获取目标状态的逆序数*     return (inv\_count\_initial % 2) == (inv\_count\_target % 2)  *# 判断逆序数的奇偶性* def a\_star\_solver(initial\_state):     *# A\*算法求解八数码问题*     open\_set = []  *# 用于存储待扩展的状态*     closed\_set = []  *# 用于存储已扩展的状态*     puzzle = PuzzleGrid(initial\_state)  *# 创建初始状态的PuzzleGrid对象*     if not is\_solvable(initial\_state, puzzle.target\_state):  *# 检查是否可解*         print("该八数码问题无解，请检查输入。")         return     open\_set.append(puzzle)  *# 将初始状态添加到open\_set*     move\_count = 0  *# 记录移动次数*     while open\_set:  *# 当还有待扩展的状态时*         open\_set.sort(key=lambda x: x.f)  *# 根据F值排序*         current\_state = open\_set.pop(0)  *# 取出F值最小的状态*         if current\_state.h == 0:  *# 如果H为0，表示找到解决方案*             print("找到解决方案，移动次数：", move\_count)             current\_state.display\_solution\_path()  *# 显示解决路径*             return         closed\_set.append(current\_state)           for new\_state in current\_state.expand\_positions():  *# 扩展当前状态*             new\_puzzle = PuzzleGrid(new\_state)               new\_puzzle.previous = current\_state               new\_puzzle.calculate\_costs()               in\_open = state\_exists(new\_puzzle, open\_set)               in\_closed = state\_exists(new\_puzzle, closed\_set)               if in\_closed[0] and new\_puzzle.f < closed\_set[in\_closed[1]].f:  *# 如果在closed且F值更小*                 closed\_set[in\_closed[1]] = new\_puzzle  *# 更新closed\_set*                 open\_set.append(new\_puzzle)  *# 添加到open\_set*                 move\_count += 1             if in\_open[0] and new\_puzzle.f < open\_set[in\_open[1]].f:  *# 如果在open且F值更小*                 open\_set[in\_open[1]] = new\_puzzle  *# 更新open\_set*                 move\_count += 1             if not in\_open[0] and not in\_closed[0]:  *# 如果不在两个集合中*                 open\_set.append(new\_puzzle)  *# 添加到open\_set*                 move\_count += 1 *# 使用示例* initial\_configuration = [[2, 8, 3], [1, 0, 4], [7, 6, 5]]   *# 初始配置* *#initial\_configuration = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]* *#initial\_configuration = [[5, 3, 4], [6, 7, 2], [0, 1, 8]]* a\_star\_solver(initial\_configuration)  *# 调用A\*算法求解* |