

**人工智能导论-实验报告**

**题目：十五数码问题求解算法及性能比较**

院 系 软件学院

专业班级

姓 名

学 号

指导教师 沈刚

2024年 12月 13 日

**目录**

**1 引言**

1.1问题背景和研究意义

1.1.1问题背景

1.1.2研究意义

1.2实验目的和实验内容

1.2.1实验目的

1.2.2实验内容

**2问题的表示和求解算法**

2.1 15 数码问题的状态表示方法

2.2 15 数码问题的求解算法

**3实验设计**

3.1 实验目标和评价指标

3.2 设计不同难度级别的15数码问题实例

3.2.1 低难度实例

3.2.2 高难度实例

3.3 选择并实现不同的搜索算法

3.3.1 不同的启发函数

3.3.2 实验方法

**4实验结果**

4.1 使用的不同方法及其特点

4.2 实验结果的展示

**5实验分析与讨论**

5.1 性能比较

5.2 定性和定量分析

5.3 结果可视化

5.3.1 性能比较可视化

5.3.2 求解过程可视化

**6结论**

6.1 实验结果总结

6.2 改进思考与建议

**7参考文献**

**8附录**

8.1 实验代码

8.1.1 A\_star.py

8.1.2 DFS.py

8.1.3 BFS.py

8.2 实验数据表和图表

1. **引言**

**1.1问题背景和研究意义**

1.1.1问题描述

8 数码问题是一个老少咸宜的益智游戏：在一个方盒内装有 8 块方形积木，每一

块都标有从 1 到 8 的数字。由于只有空白周围的方块可以以移动（即与空白交换

位置），改变数字的布局往往需要很多步骤。现在，给你一个初始的放置方式，

如何用最少的步骤将其恢复成目标形式？

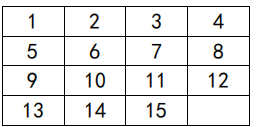
 

图 1-1 数码游戏 图 1-2 15数码游戏

现在为了增加挑战性和复杂度，我们将8数码问题升级为15数码问题，即在一个4x4的方盒中，有15块标有数字的积木和一个空白格，如图1-1、图1-2所示。此问题的求解变得更加困难，需要更高级的策略和算法。因此，本研究旨在设计一个C程序，实现一种快速且有效的算法，用于求解15数码问题，即如何用最少的步骤将给定的初始布局转换成目标形式。

**1.1.2 研究意义**

15数码问题是计算机科学和人工智能领域的经典研究课题，具有重要的理论和实际意义。首先，它是启发式搜索算法的典型应用，通过研究15数码问题，可以深入理解A\*算法及其启发函数设计对效率的影响，为路径规划和智能问题求解提供指导。其次，该问题的状态空间复杂性高达约10^{13}个状态，研究如何在庞大的搜索空间中快速找到解，有助于优化搜索算法和剪枝策略。此外，15数码问题涉及组合数学中的逆序数理论，可用于分析问题的可达性，拓展数学建模应用。在教育领域，15数码问题直观易懂，可作为教学工具帮助学生掌握状态空间搜索和算法设计。通过研究该问题，不仅可以提升解决复杂问题的能力，还能为实际问题（如机器人路径规划）提供有效的解决思路，具有深远的学术与实践价值。

**1.2 实验目的和实验内容**

1.2.1 实验目的

研究和比较 15 数码问题不同求解方法的性能，包括算法中的时间/空间复杂度和解决问题的效率

1.2.2 实验内容

1. 实现 15 数码问题的表示和求解算法，包括状态表示、初始状态生成、搜索算法、解的判断等。

2. 选择并实现至少三种不同的启发函数形式，使用 A\*算法实现。

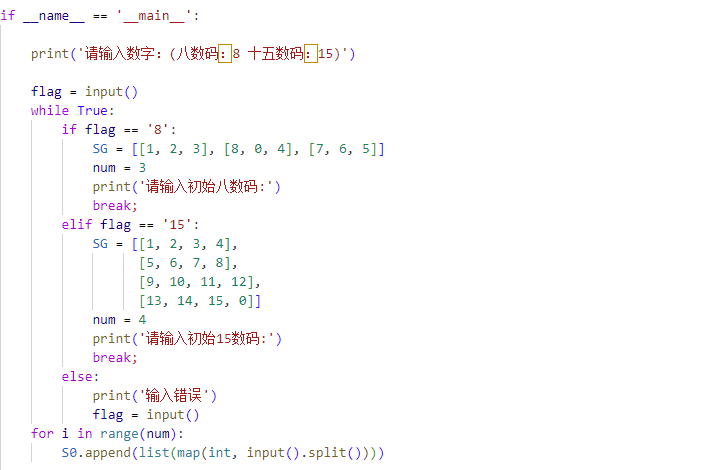
3. 设计实验案例，划分等价类，生成不同难度级别的 15 数码问题实例。

4. 在不同启发函数下，对实例进行求解，记录求解时间和步数。

**2. 问题的表示和求解算法**

2.1 15数码问题的状态表示方法

15数码问题的状态可以通过一个4x4的矩阵来表示，其中15个单元格包含从1到15的数字，剩下一个单元格为空，表示可以移动的空间。



**2.2 15数码问题的求解算法**

**A\*算法**

A\*算法是一种求解最短路径最有效的直接搜索方法，也是许多其他问题的常用启发式算法。

它的启发函数为**f(n)=g(n)+h(n)**,其中，f(n) 是从初始状态经由状态n到目标状态的代价估计，g(n) 是在状态空间中从初始状态到状态n的实际代价，h(n) 是从状态n到目标状态的最佳路径的估计代价。h(n)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h\*(n)的一种估计，且需要满足 **h(n)**

**<=h‘(n)**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。

读入初始状态和目标状态，并计算初始状态评价函数值f；

初始化两个open表和closed表，将初始状态放入open表中

如果open表为空，则查找失败；

否则：

① 在open表中找到评价值最小的节点，作为当前结点，并放入closed表中；

② 判断当前结点状态和目标状态是否一致，若一致，跳出循环；否则跳转到③；

③ 对当前结点，分别按照上、下、左、右方向移动空格位置来扩展新的状态结点，并计算新扩展结点的

评价值f并记录其父节点；

④ 对于新扩展的状态结点，进行如下操作：

A．新节点既不在open表中，也不在closed表中，则添加进OPEN表；

B．新节点在open表中，则计算评价函数的值，取最小的。

C．新节点在closed表中，则计算评价函数的值，取最小的。

⑤ 把当前结点从open表中移除；

**注：计算距离函数(启发函数h(n))**：

1. 采用曼哈顿距离的计算方法，计算每一个位置的数据与它理论位置的横纵标和与纵坐标距离之和。

2. 采用欧氏距离的计算方法，计算每一个位置的数据与它理论位置直线距离之和。

**3. 实验设计**

**3.1 实验目标和评价指标**

本实验的主要目标是研究并比较不同求解方法在解决15数码问题时的性能。为了实现这一目标，需要定义具体的评

价指标，这些指标将用于量化和比较各种算法的有效性。主要评价指标包括：

1. 搜索时长：从算法开始到找到最终解的总耗时。这是衡量算法效率的主要指标之一。

2. 中间状态数（检测节点数）：在达到最终解决方案的过程中，算法生成并考察的总状态数量。中间状态数反映了算法的空间复

杂度，即算法在寻找解决方案的过程中所需处理和存储的信息量。

3. 最优路径长度：从初始状态到达目标状态所需的最小移动次数。这反映了算法找到解决方案的优化程度。

**3.2 设计不同难度级别的15数问题实例**

3.2.1 低难度实例

在低难度实例中，初始状态通过对目标状态进行5到10步随机合法移动得到。这些移动较少，因此解决起来相对容

易。

初始状态： 5 1 2 4 9 6 3 8 13 15 10 11 0 14 7 12

目标状态：1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数

E

最终搜索路径为：

------ 0 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 15, 10, 11]

[0, 14, 7, 12]

------ 1 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 15, 10, 11]

[14, 0, 7, 12]

------ 2 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 0, 10, 11]

[14, 15, 7, 12]

------ 3 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 10, 0, 11]

[14, 15, 7, 12]

------ 4 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 10, 7, 11]

[14, 15, 0, 12]

------ 5 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 10, 7, 11]

[14, 0, 15, 12]

------ 6 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[13, 10, 7, 11]

[0, 14, 15, 12]

------ 7 --------

[5, 1, 2, 4]

[9, 6, 3, 8]

[0, 10, 7, 11]

[13, 14, 15, 12]

------ 8 --------

[5, 1, 2, 4]

[0, 6, 3, 8]

[9, 10, 7, 11]

[13, 14, 15, 12]

------ 9 --------

[0, 1, 2, 4]

[5, 6, 3, 8]

[9, 10, 7, 11]

[13, 14, 15, 12]

------ 10 --------

[1, 0, 2, 4]

[5, 6, 3, 8]

[9, 10, 7, 11]

[13, 14, 15, 12]

------ 11 --------

[1, 2, 0, 4]

[5, 6, 3, 8]

[9, 10, 7, 11]

[13, 14, 15, 12]

------ 12 --------

[1, 2, 3, 4]

[5, 6, 0, 8]

[9, 10, 7, 11]

[13, 14, 15, 12]

------ 13 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 10, 0, 11]

[13, 14, 15, 12]

------ 14 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 10, 11, 0]

[13, 14, 15, 12]

------ 15 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 10, 11, 12]

[13, 14, 15, 0]

采用欧式距离计算启发函数

搜索最优路径长度为 15

搜索时长为 11.239624738693237 s

共检测节点数为 1127

3.2.2 高难度实例

高难度实例通过对目标状态执行41步的随机合法移动来生成。这个实例更难解决，需要更多时间和资源。

初始状态：11 9 4 15 1 3 0 12 7 5 8 6 13 2 10 14

目标状态：1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

请输入初始15数码:

11 9 4 15

1 3 0 12

7 5 8 6

13 2 10 14

选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数

E

最终搜索路径为：

------ 0 --------

[11, 9, 4, 15]

[1, 3, 0, 12]

[7, 5, 8, 6]

[13, 2, 10, 14]

------ 1 --------

[11, 9, 4, 15]

[1, 3, 12, 0]

[7, 5, 8, 6]

[13, 2, 10, 14]

------ 2 --------

[11, 9, 4, 0]

[1, 3, 12, 15]

[7, 5, 8, 6]

[13, 2, 10, 14]

------ 3 --------

[11, 9, 0, 4]

[1, 3, 12, 15]

[7, 5, 8, 6]

[13, 2, 10, 14]

------ 4 --------

[11, 9, 12, 4]

[1, 3, 0, 15]

[7, 5, 8, 6]

[13, 2, 10, 14]

------ 5 --------

[11, 9, 12, 4]

[1, 3, 8, 15]

[7, 5, 0, 6]

[13, 2, 10, 14]

------ 6 --------

[11, 9, 12, 4]

[1, 3, 8, 15]

[7, 5, 6, 0]

[13, 2, 10, 14]

------ 7 --------

[11, 9, 12, 4]

[1, 3, 8, 0]

[7, 5, 6, 15]

[13, 2, 10, 14]

------ 8 --------

[11, 9, 12, 4]

[1, 3, 0, 8]

[7, 5, 6, 15]

[13, 2, 10, 14]

------ 9 --------

[11, 9, 0, 4]

[1, 3, 12, 8]

[7, 5, 6, 15]

[13, 2, 10, 14]

------ 10 --------

[11, 0, 9, 4]

[1, 3, 12, 8]

[7, 5, 6, 15]

[13, 2, 10, 14]

------ 11 --------

[0, 11, 9, 4]

[1, 3, 12, 8]

[7, 5, 6, 15]

[13, 2, 10, 14]

------ 12 --------

[1, 11, 9, 4]

[0, 3, 12, 8]

[7, 5, 6, 15]

[13, 2, 10, 14]

------ 13 --------

[1, 11, 9, 4]

[7, 3, 12, 8]

[0, 5, 6, 15]

[13, 2, 10, 14]

------ 14 --------

[1, 11, 9, 4]

[7, 3, 12, 8]

[5, 0, 6, 15]

[13, 2, 10, 14]

------ 15 --------

[1, 11, 9, 4]

[7, 3, 12, 8]

[5, 6, 0, 15]

[13, 2, 10, 14]

------ 16 --------

[1, 11, 9, 4]

[7, 3, 0, 8]

[5, 6, 12, 15]

[13, 2, 10, 14]

------ 17 --------

[1, 11, 0, 4]

[7, 3, 9, 8]

[5, 6, 12, 15]

[13, 2, 10, 14]

------ 18 --------

[1, 0, 11, 4]

[7, 3, 9, 8]

[5, 6, 12, 15]

[13, 2, 10, 14]

------ 19 --------

[1, 3, 11, 4]

[7, 0, 9, 8]

[5, 6, 12, 15]

[13, 2, 10, 14]

------ 20 --------

[1, 3, 11, 4]

[7, 6, 9, 8]

[5, 0, 12, 15]

[13, 2, 10, 14]

------ 21 --------

[1, 3, 11, 4]

[7, 6, 9, 8]

[5, 2, 12, 15]

[13, 0, 10, 14]

------ 22 --------

[1, 3, 11, 4]

[7, 6, 9, 8]

[5, 2, 12, 15]

[13, 10, 0, 14]

------ 23 --------

[1, 3, 11, 4]

[7, 6, 9, 8]

[5, 2, 12, 15]

[13, 10, 14, 0]

------ 24 --------

[1, 3, 11, 4]

[7, 6, 9, 8]

[5, 2, 12, 0]

[13, 10, 14, 15]

------ 25 --------

[1, 3, 11, 4]

[7, 6, 9, 8]

[5, 2, 0, 12]

[13, 10, 14, 15]

------ 26 --------

[1, 3, 11, 4]

[7, 6, 0, 8]

[5, 2, 9, 12]

[13, 10, 14, 15]

------ 27 --------

[1, 3, 11, 4]

[7, 0, 6, 8]

[5, 2, 9, 12]

[13, 10, 14, 15]

------ 28 --------

[1, 3, 11, 4]

[7, 2, 6, 8]

[5, 0, 9, 12]

[13, 10, 14, 15]

------ 29 --------

[1, 3, 11, 4]

[7, 2, 6, 8]

[5, 9, 0, 12]

[13, 10, 14, 15]

------ 30 --------

[1, 3, 11, 4]

[7, 2, 0, 8]

[5, 9, 6, 12]

[13, 10, 14, 15]

------ 31 --------

[1, 3, 0, 4]

[7, 2, 11, 8]

[5, 9, 6, 12]

[13, 10, 14, 15]

------ 32 --------

[1, 0, 3, 4]

[7, 2, 11, 8]

[5, 9, 6, 12]

[13, 10, 14, 15]

------ 33 --------

[1, 2, 3, 4]

[7, 0, 11, 8]

[5, 9, 6, 12]

[13, 10, 14, 15]

------ 34 --------

[1, 2, 3, 4]

[0, 7, 11, 8]

[5, 9, 6, 12]

[13, 10, 14, 15]

------ 35 --------

[1, 2, 3, 4]

[5, 7, 11, 8]

[0, 9, 6, 12]

[13, 10, 14, 15]

------ 36 --------

[1, 2, 3, 4]

[5, 7, 11, 8]

[9, 0, 6, 12]

[13, 10, 14, 15]

------ 37 --------

[1, 2, 3, 4]

[5, 7, 11, 8]

[9, 6, 0, 12]

[13, 10, 14, 15]

------ 38 --------

[1, 2, 3, 4]

[5, 7, 0, 8]

[9, 6, 11, 12]

[13, 10, 14, 15]

------ 39 --------

[1, 2, 3, 4]

[5, 0, 7, 8]

[9, 6, 11, 12]

[13, 10, 14, 15]

------ 40 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 0, 11, 12]

[13, 10, 14, 15]

------ 41 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 10, 11, 12]

[13, 0, 14, 15]

------ 42 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 10, 11, 12]

[13, 14, 0, 15]

------ 43 --------

[1, 2, 3, 4]

[5, 6, 7, 8]

[9, 10, 11, 12]

[13, 14, 15, 0]

采用欧式距离计算启发函数

搜索最优路径长度为 43

搜索时长为 127.77426552772522 s

共检测节点数为 3431727

在设计实验时，可以使用这些具体实例来评估不同算法在解决15数码问题上的性能。通过对这些不同难度级别的实

例进行测试，可以更准确地衡量和比较各种求解方法的时间和空间效率。

**3.3 选择并实现不同的搜索算法**

**3.3.1 不同的启发函数**

在本实验中，我们将专注于使用A\*搜索算法，并采用三种不同的启发式函数来评估算法的效率和有效性。以下是我

们将实现和比较的三种启发式函数：

1. 欧式距离（**Euclidean Distance**）：欧式距离启发函数是15数码问题中一种估计当前位置到目标状态的代价的启发式方法。其核心思想是通过计算每个数字方块当前位置与目标位置之间的欧式距离，并将这些距离求和，得到启发值h(n)。

2. 曼哈顿距离（**Manhattan Distance**）：此方法计算每个数字方块从其当前位置到目标位置的格子数总和。对于每 个方块，其曼哈顿距离是其在行和列上的距离之和。曼哈顿距离考虑了方块需要移动的实际距离，因此比汉明距离提供了更精确的启发式估计。

3. 线性冲突（**Linear Conflict**）：这种方法在曼哈顿距离的基础上增加了额外的考虑因素。如果两个方块在达到其 最终位置之前需要交换位置，那么这被认为是一个线性冲突。例如，如果两个方块在同一行或同一列中并且它们的目标位置也在这一行或列中，但它们的顺序是错误的，那么每对线性冲突将增加两步到总曼哈顿距离中。

这使得启发式估计更加接近实际的最小步数。

**3.3.2 其他搜索算法**

**1.DFS算法**

深度优先搜索算法（**Depth-First-Search，DFS**）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所在边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节 点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

对于八数码问题，没有已经存在的路径供我们遍历，需要我们从初始状态向下延伸（也就是上下左右移动）才能构造出类似的树。

代码思路：

（DFS不一定能找到最优解。因为深度界限的原因，找到的解可能在最优解和深度界限之间。）

设置OPEN表，每扩展一个节点，将该节点添加到OPEN表首部（insert(0)，且按照左上下右的反向循环）。另外，为了尽可能保证程序有解，在深度优先算法的基础上，添加了深度限制，即在扩展节点之前，计算当前节点的深度，如果深度大于规定的阈值，则不扩展该节点。



1. BFS算法

广度优先搜索算法（**Breadth-First-Search，BFS**），是一种图形搜索算法。简单的说，BFS是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。BFS是一种盲目搜索法，目的是系统地展开并检查图中的所有节点，以找寻结果。在应用BFS算法进行八数码问题搜索时需要open和closed两个表。首先将初始状态加入open队列，然后进行出队操作并放入closed中，对出队的状态进行扩展（所谓扩展也就是找出其上下左右移动后的状态），将扩展出的状态加入队列，然后继续循环出队-扩展-入队的操作，直到找到解为止。

通过队列实现：入队-扩展-出队

由于BFS是一层一层找的，所以一定能找到解，并且是最优解。虽然能找到最优解，但它的盲目性依然是一个很大的缺点。从上面的遍历树状图中，每一层都比上一层元素更多，且是近似于指数型的增长。也就是说，深度每增加一，这一层的搜索速度就要增加很多。使用哈希数值进行是否达到终点的判断。



**4.实验结果**

4.1使用的不同方法及其特点

基于上述结果，我们可以得出每种方法的特点：

1. 欧式距离（**Euclidean Distance**）：

特点：计算当前状态中与目标状态不匹配的方块数量。简单易计算，但不考虑实际步数，可能不够精确。

优点：实现简单，计算速度快。

缺点：对于更复杂的布局，可能无法提供足够有效的指导。

2. 曼哈顿距离（**Manhattan Distance**）：

特点：计算每个数字方块从其当前位置到目标位置的格子数总和。更准确地反映了方块的实际移动距离。

优点：相对准确，适用于大多数情况。

缺点：计算量比汉明距离大，可能导致搜索速度略慢。

3. 线性冲突（**Linear Conflict**）：

特点：在曼哈顿距离的基础上增加了额外的线性冲突考量，考虑了特定方块之间的相互影响。

优点：提供了更接近实际步数的估计，可以在某些情况下加速解的发现。

缺点：计算复杂度更高，对于一些简单布局可能是过度优化。

4.2实验结果的展示

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 启发函数计算方法 | 搜索最优路径长度 | 搜索时长 | 共检测节点数 |
| 低难度 | 欧式距离 | 10 | 10.80940s | 55 |
| 低难度 | 曼哈顿距离 | 10 | 7.70280s | 68 |
| 低难度 | 线性冲突 | 10 | 10.06034s | 12 |
| 高难度 | 欧式距离 | 43 | 127.74654s | 34311717 |
| 高难度 | 曼哈顿距离 | 45 | 40.62714s | 292615 |
| 高难度 | 线性冲突 | 41 | 13.98486s | 13438 |

**5. 实验分析与讨论**

**5.1 性能⽐较**

在实验中，我们对比了三种不同启发式函数（欧拉距离、曼哈顿距离和线性冲突）在解决15数码问题时的性能表现。下面是对这些方法的性能比较和分析：

1. 欧拉距离：在低难度实例中表现适中，但在高难度实例中无法在合理时间内找到解决方案（时间很长）。 欧拉距离作为一种简单的启发式函数，在处理复杂问题时效率较低。
2. 曼哈顿距离： 在低难度实例中效率高，求解时间和中间状态数均优于欧拉距离。 在高难度实例中能够找到解决方案，但求解时间较长，生成的中间状态数量多。
3. 线性冲突： 在低难度实例中的表现与曼哈顿距离相似。 在高难度实例中表现最优，求解时间和中间状态数都显著优于其他方法。

**5.2 定性和定量分析**

定量分析：通过实验数据，我们可以看到，尤其在高难度实例中，线性冲突的求解时间和中间状态数明显低于曼哈顿距离和欧拉距离。这表明线性冲突在处理复杂情况时更高效。

定性分析：汉明距离由于其简单性，在面对复杂情况时往往不足以提供有效的启发。曼哈顿距离，虽然比欧拉距离更准确，但在处理特定的线性冲突情况时效率不高。线性冲突通过在曼哈顿距离的基础上增加额外的考量，能够更好地指导搜索过程，尤其是在复杂的实例中。

**5.3不同算法的比较**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 启发函数计算方法 | 搜索最优路径长度 | 搜索时长 | 检测节点数 |
| 情况1 | 深度优先 | 39 | 160.2545s | 1342460 |
| 情况2 | 广度优先 | 15 | 11.1330s | 107285 |
| 情况3 | A\* | 21 | 0.3512s | 2116 |

通过对比分析，可以看出 A星算法的搜索时长和检测节点数明显小于另外两种方法，可见启发式信息对于搜索过程的重要性；

DFS有界深度优先算法的算法性能差异较大，设置不同的最深深度得到的结果有一定的差异，一般设置 较小可能不会在此深度找到结果，较大会造成内存爆炸的现象，所以通过该方法进行搜索较为困难，对 于任务较为复杂的情况，很难快速求解。

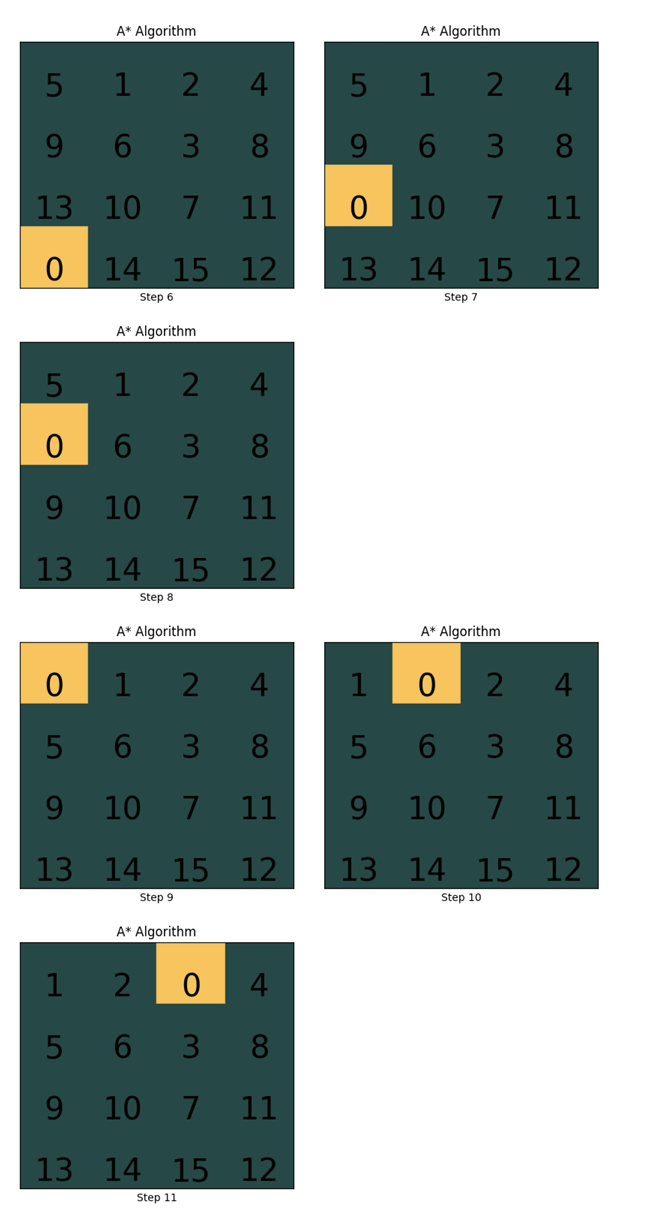
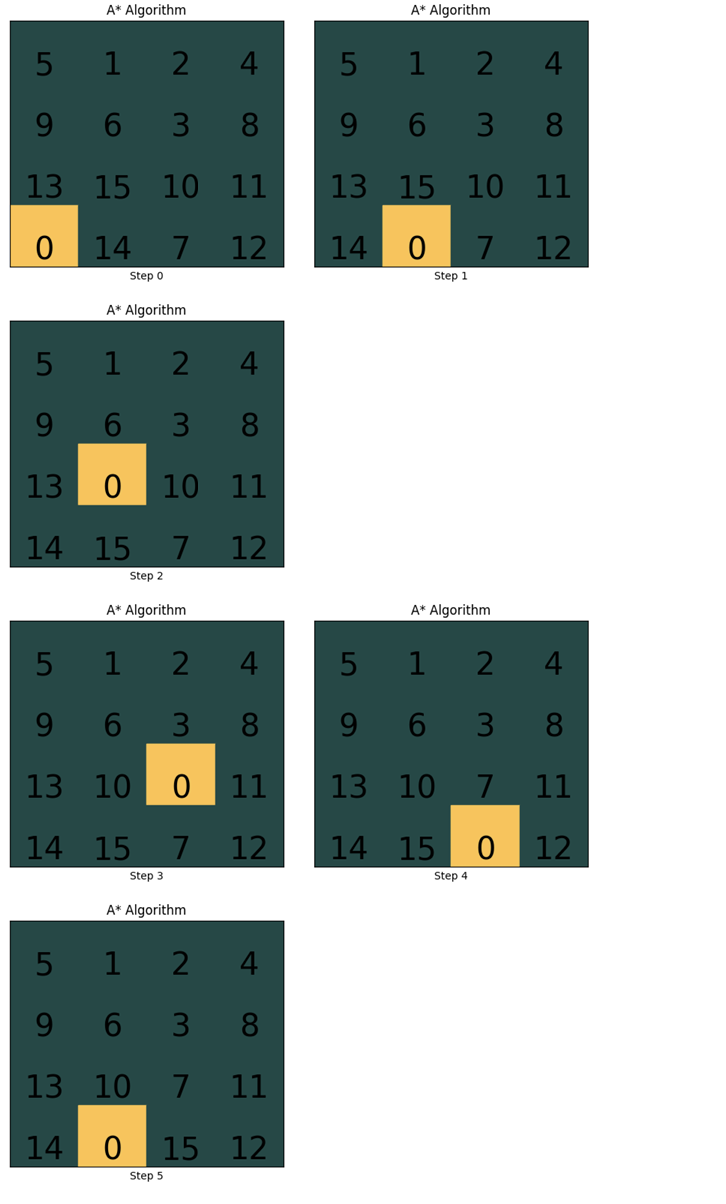
BFS广度优先算法，针对较为简单问题，基本可以以最短路径给出答案，但同时搜索时间和搜索节点数 一定会比启发式搜索多一些，针对复杂问题，很难给出答案，每扩展一层，都会以指数的形式增加待扩 展节点的数量，很难得出答案。

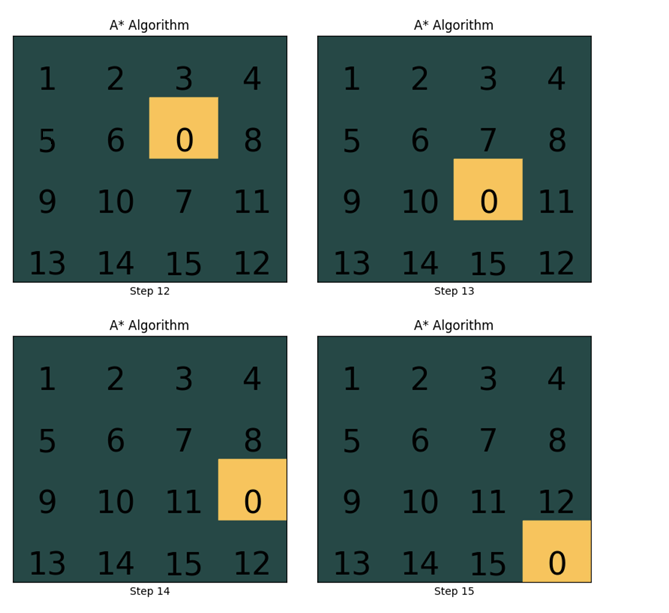
综上所述，与深度优先和广度优先算法相比，启发式搜索算法有很强的优越性，一般情况下要尽可能去 寻找启发函数，添加到代码中辅助进行算法的训练，尽可能缩短程序运行时间，提高程序效率。

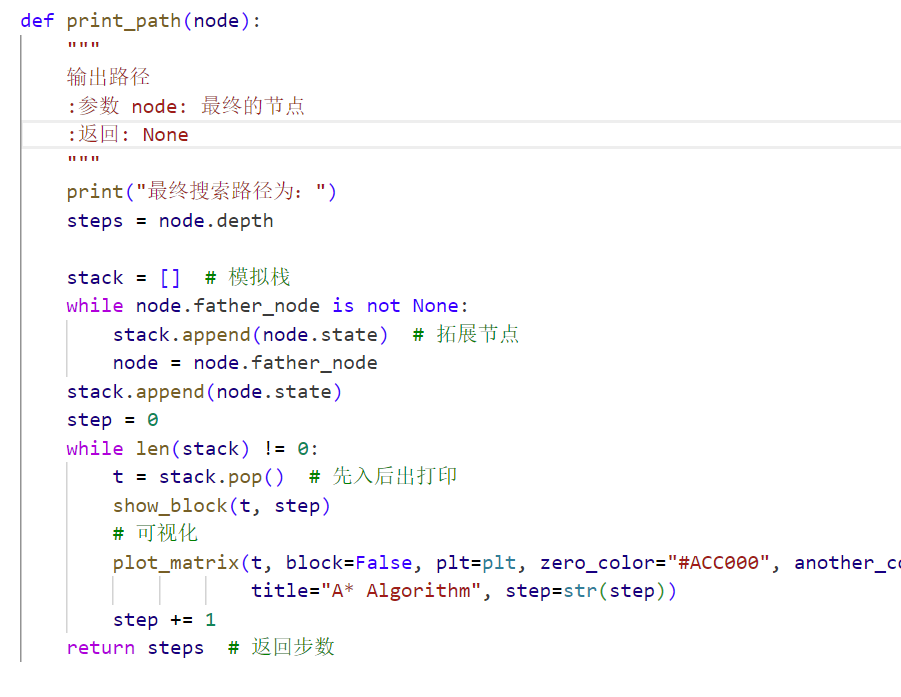
**5.4 结果可视化**

**5.4.1 求解过程可视化**

运用模拟栈的方式将最终的搜索路径输出







**6. 结论**

**6.1 实验结果总结**

本次实验对15数码问题的求解进行了深入的分析和比较，主要关注点在于不同启发式函数在A\*搜索算法中的表现。 实验结果显示：

1. 不同启发式函数的表现：在低难度实例中，曼哈顿距离和线性冲突表现出接近的高效率，而欧式距离虽然能够解决问题，但效率稍低。在高难度实例中，欧式距离无法在合理时间内找到解决方案，而线性冲突在求解时间和生成的中间状态数量上均显著优于曼哈顿距离。
2. 效率和复杂度：线性冲突由于其在曼哈顿距离的基础上增加了额外考虑因素，因此在处理更复杂的布局时更为高效。这表明复杂问题求解中，考虑更多因素的启发式函数可能更优。

**6.2 改进思考与建议**

1. 启发式函数的优化：虽然线性冲突在本实验中表现最佳，但仍有进一步优化的空间。可以考虑开发新的启发式函数，或改进现有的函数，使其更精确清晰地反映实际解决问题所需的步骤。

2. 探索其他搜索算法：虽然A\*算法在许多情况下有效，但探索其他搜索算法，如IDA\*或双向搜索，可能在特定情况下更有效。

3. 算法的混合使用：在不同阶段或不同类型的实例中，结合使用不同的启发式函数可能会提高效率。例如，对于简单实例使用曼哈顿距离，在达到一定复杂度后运用线性冲突。

4. 内存和处理优化：进行高难度实例中会生成的大量中间状态，优化内存管理和处理效率是提高算法整体性能的关键。例如，可以采用更高效的数据结构或改进状态存储方法。

5. 并行计算：考虑到15数码问题的计算密集性，利用并行计算资源（如多线程或分布式计算）可能会显著提高求解速度。

总的来说，本实验提供了对15数码问题求解方法的深入理解，并指出了现有方法的优势和局限性。

**7参考文献**

[1]Edelkamp, S., & Schrödl, S. (2012). Heuristic Search: Theory and Applications. Elsevier.

[2] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

[3] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 97 109.

**8 附录**

8.1 实验代码

A\_star.py

import heapq

import copy

import time

import math

import matplotlib.pyplot as plt

S0 = []

SG = []

# 上下左右四个方向移动

MOVE = {'up': [-1, 0],

        'down': [1, 0],

        'left': [0, -1],

        'right': [0, 1]}

# OPEN表

OPEN = []

# 节点的总数

SUM\_NODE\_NUM = 0

# 状态节点

class State(object):

    def \_\_init\_\_(self, depth=0, rest\_dis=0.0, state=None, hash\_value=None, father\_node=None):

        """

        初始化

        :参数 depth: 从初始节点到目前节点所经过的步数

        :参数 rest\_dis: 启发距离

        :参数 state: 节点存储的状态 4\*4的列表

        :参数 hash\_value: 哈希值，用于判重

        :参数 father\_node: 父节点指针

        """

        self.depth = depth

        self.rest\_dis = rest\_dis

        self.fn = self.depth + self.rest\_dis

        self.child = []  # 孩子节点

        self.father\_node = father\_node  # 父节点

        self.state = state  # 局面状态

        self.hash\_value = hash\_value  # 哈希值

    def \_\_lt\_\_(self, other):  # 用于堆的比较，返回距离最小的

        return self.fn < other.fn

    def \_\_eq\_\_(self, other):  # 相等的判断

        return self.hash\_value == other.hash\_value

    def \_\_ne\_\_(self, other):  # 不等的判断

        return not self.\_\_eq\_\_(other)

def cal\_M\_distence(cur\_state):

    """

    计算曼哈顿距离

    :参数 state: 当前状态,4\*4的列表, State.state

    :返回: M\_cost 每一个节点计算后的曼哈顿距离总和

    """

    global num

    M\_cost = 0

    # -15：

    for i in range(num):

        for j in range(num):

            # -8：

            # for i in range(3):

            #     for j in range(3):

            if cur\_state[i][j] == SG[i][j]:

                continue

            number = cur\_state[i][j]

            if number == 0:

                x, y = 3, 3

            else:

                x = number / 4  # 理论横坐标

                y = number - 4 \* x - 1  # 理论的纵坐标

                M\_cost += (abs(x - i) + abs(y - j))

    return M\_cost

def cal\_E\_distence(cur\_state):

    """

    计算曼哈顿距离

    :参数 state: 当前状态,4\*4的列表, State.state

    :返回: M\_cost 每一个节点计算后的曼哈顿距离总和

    """

    E\_cost = 0

    for i in range(num):

        for j in range(num):

            if cur\_state[i][j] == SG[i][j]:

                continue

            number = cur\_state[i][j]

            if number == 0:

                x, y = 3, 3

            else:

                x = number / 4  # 理论横坐标

                y = number - 4 \* x - 1  # 理论的纵坐标

                E\_cost += math.sqrt((x - i) \* (x - i) + (y - j) \* (y - j))

    return E\_cost

def generate\_child(sn\_node, sg\_node, hash\_set, open\_table, cal\_distence):

    """

    生成子节点函数

    :参数 sn\_node:  当前节点

    :参数 sg\_node:  最终状态节点

    :参数 hash\_set:  哈希表，用于判重

    :参数 open\_table: OPEN表

    :参数 cal\_distence: 距离函数

    :返回: None

    """

    if sn\_node == sg\_node:

        heapq.heappush(open\_table, sg\_node)  # heappush(heap,item)建立大小堆

        print('已找到终止状态！')

        return

    global flag, num

    for i in range(0, num):

        for j in range(0, num):

            if sn\_node.state[i][j] != 0:

                continue

            for d in ['left', 'up', 'down', 'right']:  # 四个偏移方向

                x = i + MOVE[d][0]

                y = j + MOVE[d][1]

                if x < 0 or x >= num or y < 0 or y >= num:  # 越界了

                    continue

                state = copy.deepcopy(sn\_node.state)  # 复制父节点的状态

                state[i][j], state[x][y] = state[x][y], state[i][j]  # 交换位置

                h = hash(str(state))  # 哈希时要先转换成字符串

                if h in hash\_set:  # 重复了

                    continue

                hash\_set.add(h)  # 加入哈希表

                # 记录扩展节点的个数

                global SUM\_NODE\_NUM

                SUM\_NODE\_NUM += 1

                depth = sn\_node.depth + 1  # 已经走的距离函数

                rest\_dis = cal\_distence(state)  # 启发的距离函数

                node = State(depth, rest\_dis, state, h, sn\_node)  # 新建节点

                sn\_node.child.append(node)  # 加入到孩子队列

                heapq.heappush(open\_table, node)  # 加入到堆中

                # show\_block(state,depth)

                # print('child')

def plot\_matrix(matrix, block, plt, zero\_color=" ", another\_color=" ", title=" ", step=" "):

    """

    plot\_matrix: 用来画出矩阵；

    matrix为二维列表；

    plt为画笔，应该为：import matplotlib.pyplot as plt

    """

    plt.subplots(figsize=(4, 4))

    plt.title(title)

    plt.xlabel("Step " + step)

    rows = len(matrix)

    columns = len(matrix[0])

    #  -15：

    # plt.xlim(0, 4 \* rows)

    # plt.ylim(0, 4 \* columns)

    # -8:

    plt.xlim(0, num \* rows)

    plt.ylim(0, num \* columns)

    for i in range(rows):

        for j in range(columns):

            if flag == '8':

                if matrix[i][j] != 0:

                    # 画出一个3\*3的矩形，其中左下角坐标为：(3 \* j, 6 - 3 \* i)，并填充颜色， 0和其他的要有区分；

                    plt.gca().add\_patch(plt.Rectangle((3 \* j, 6 - 3 \* i), 3, 3, color=another\_color, alpha=1))

                else:

                    plt.gca().add\_patch(plt.Rectangle((3 \* j, 6 - 3 \* i), 3, 3, color=zero\_color, alpha=1))

                plt.text(3 \* j + 1.5, 7.5 - 3 \* i, str(matrix[i][j]), fontsize=30, horizontalalignment='center')

            if flag == '15':

                if matrix[i][j] != 0:

                    # 画出一个4\*4的矩形,并填充颜色， 0和其他的要有区分；

                    plt.gca().add\_patch(plt.Rectangle((4 \* j, 12 - 4 \* i), 4, 4, color=another\_color, alpha=1))

                else:

                    plt.gca().add\_patch(plt.Rectangle((4 \* j, 12 - 4 \* i), 4, 4, color=zero\_color, alpha=1))

                plt.text(4 \* j + 2, 12.5 - 4 \* i, str(matrix[i][j]), fontsize=30, horizontalalignment='center')

    plt.xticks([])

    plt.yticks([])

    plt.show(block=block)

    plt.pause(0.5)

    plt.close()

def show\_block(block, step):

    print("------", step, "--------")

    for b in block:

        print(b)

def print\_path(node):

    """

    输出路径

    :参数 node: 最终的节点

    :返回: None

    """

    print("最终搜索路径为：")

    steps = node.depth

    stack = []  # 模拟栈

    while node.father\_node is not None:

        stack.append(node.state)  # 拓展节点

        node = node.father\_node

    stack.append(node.state)

    step = 0

    while len(stack) != 0:

        t = stack.pop()  # 先入后出打印

        show\_block(t, step)

        # 可视化

        plot\_matrix(t, block=False, plt=plt, zero\_color="#ACC000", another\_color="#FA4444",

                    title="A\* Algorithm", step=str(step))

        step += 1

    return steps  # 返回步数

def A\_start(start, end, distance\_fn, generate\_child\_fn):

    """

    A\*算法

    :参数 start: 起始状态

    :参数 end: 终止状态

    :参数 distance\_fn: 距离函数，可以使用自定义的

    :参数 generate\_child\_fn: 产生孩子节点的函数

    :返回: 最优路径长度

    """

    root = State(0, 0, start, hash(str(S0)), None)  # 根节点

    end\_state = State(0, 0, end, hash(str(SG)), None)  # 最后的节点

    if root == end\_state:

        print("start == end !")

    OPEN.append(root)

    heapq.heapify(OPEN)  # 成堆

    node\_hash\_set = set()  # 存储节点的哈希值

    node\_hash\_set.add(root.hash\_value)

    while len(OPEN) != 0:

        top = heapq.heappop(OPEN)  # 返回最小值

        if top == end\_state:  # 结束后直接输出路径

            return print\_path(top)

        # 产生孩子节点，孩子节点加入OPEN表

        generate\_child\_fn(sn\_node=top, sg\_node=end\_state, hash\_set=node\_hash\_set,

                          open\_table=OPEN, cal\_distence=distance\_fn)

    print("无搜索路径!")  # 没有路径

    return -1

if \_\_name\_\_ == '\_\_main\_\_':

    print('请输入数字：(八数码：8 十五数码：15)')

    flag = input()

    while True:

        if flag == '8':

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

            num = 3

            print('请输入初始八数码:')

            break;

        elif flag == '15':

            SG = [[1, 2, 3, 4],

                  [5, 6, 7, 8],

                  [9, 10, 11, 12],

                  [13, 14, 15, 0]]

            num = 4

            print('请输入初始15数码:')

            break;

        else:

            print('输入错误')

            flag = input()

    for i in range(num):

        S0.append(list(map(int, input().split())))

    print("选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数")

    method = input()

    time1 = time.time()

    if method == 'E':

        length = A\_start(S0, SG, cal\_E\_distence, generate\_child)

    else:

        length = A\_start(S0, SG, cal\_M\_distence, generate\_child)

    time2 = time.time()

    if length != -1:

        if method == 'E':

            print("采用欧式距离计算启发函数")

        else:

            print("采用曼哈顿距离计算启发函数")

        print("搜索最优路径长度为", length)

        print("搜索时长为", (time2 - time1), "s")

        print("共检测节点数为", SUM\_NODE\_NUM)

DFS.py

import heapq

import copy

import time

S0 = []

SG = []

# 上下左右四个方向移动

MOVE = {'up': [-1, 0],

        'down': [1, 0],

        'left': [0, -1],

        'right': [0, 1]}

# OPEN表

OPEN = []

# 节点的总数

SUM\_NODE\_NUM = 0

# 状态节点

class State(object):

    def \_\_init\_\_(self, depth=0, state=None, hash\_value=None, father\_node=None):

        """

        初始化

        :参数 depth: 从初始节点到目前节点所经过的步数

        :参数 state: 节点存储的状态 4\*4的列表

        :参数 hash\_value: 哈希值，用于判重

        :参数 father\_node: 父节点指针

        """

        self.depth = depth

        self.child = []  # 孩子节点

        self.father\_node = father\_node  # 父节点

        self.state = state  # 局面状态

        self.hash\_value = hash\_value  # 哈希值

    def \_\_eq\_\_(self, other):  # 相等的判断

        return self.hash\_value == other.hash\_value

    def \_\_ne\_\_(self, other):  # 不等的判断

        return not self.\_\_eq\_\_(other)

def generate\_child(sn\_node, sg\_node, hash\_set):

    """

    生成子节点函数

    :参数 sn\_node:  当前节点

    :参数 sg\_node:  最终状态节点

    :参数 hash\_set:  哈希表，用于判重

    :参数 open\_table: OPEN表

    :返回: None

    """

    global flag, num

    for i in range(0, num):

        for j in range(0, num):

            if sn\_node.state[i][j] != 0:

                continue

            for d in ['right', 'down', 'up', 'left']:  # 四个偏移方向

                x = i + MOVE[d][0]

                y = j + MOVE[d][1]

                if x < 0 or x >= num or y < 0 or y >= num:  # 越界了

                    continue

                state = copy.deepcopy(sn\_node.state)  # 复制父节点的状态

                state[i][j], state[x][y] = state[x][y], state[i][j]  # 交换位置

                h = hash(str(state))  # 哈希时要先转换成字符串

                if h in hash\_set:  # 重复则跳过

                    continue

                hash\_set.add(h)  # 加入哈希表

                depth = sn\_node.depth + 1  # 已经走的距离函数

                node = State(depth, state, h, sn\_node)  # 新建节点

                sn\_node.child.append(node)  # 加入到孩子队列

                OPEN.insert(0, node)

def plot\_matrix(matrix, block, plt, zero\_color, another\_color, title=" ", step=" "):

    """

    plot\_matrix: 用来画出矩阵；

    matrix为二维列表；

    plt为画笔，应该为：import matplotlib.pyplot as plt

    """

    plt.subplots(figsize=(4, 4))

    plt.title(title)

    plt.xlabel("Step " + step)

    rows = len(matrix)

    columns = len(matrix[0])

    plt.xlim(0, num \* rows)

    plt.ylim(0, num \* columns)

    for i in range(rows):

        for j in range(columns):

            if flag == '8':

                if matrix[i][j] != 0:

                    # 画出一个3\*3的矩形，其中左下角坐标为：(3 \* j, 6 - 3 \* i)，并填充颜色， 0和其他的要有区分；

                    plt.gca().add\_patch(plt.Rectangle((3 \* j, 6 - 3 \* i), 3, 3, color=another\_color, alpha=1))

                else:

                    plt.gca().add\_patch(plt.Rectangle((3 \* j, 6 - 3 \* i), 3, 3, color=zero\_color, alpha=1))

                plt.text(3 \* j + 1.5, 7.5 - 3 \* i, str(matrix[i][j]), fontsize=30, horizontalalignment='center')

            if flag == '15':

                if matrix[i][j] != 0:

                    # 画出一个4\*4的矩形,并填充颜色， 0和其他的要有区分；

                    plt.gca().add\_patch(plt.Rectangle((4 \* j, 12 - 4 \* i), 4, 4, color=another\_color, alpha=1))

                else:

                    plt.gca().add\_patch(plt.Rectangle((4 \* j, 12 - 4 \* i), 4, 4, color=zero\_color, alpha=1))

                plt.text(4 \* j + 2, 12.5 - 4 \* i, str(matrix[i][j]), fontsize=30, horizontalalignment='center')

    plt.xticks([])

    plt.yticks([])

    plt.show(block=block)

    plt.pause(0.5)

    plt.close()

def show\_block(block, step):

    print("------", step, "--------")

    for b in block:

        print(b)

def print\_path(node):

    """

    输出路径

    :参数 node: 最终的节点

    :返回: None

    """

    print("最终搜索路径为：")

    steps = node.depth

    stack = []  # 模拟栈

    while node.father\_node is not None:

        stack.append(node.state)  # 拓展节点

        node = node.father\_node

    stack.append(node.state)

    step = 0

    while len(stack) != 0:

        t = stack.pop()  # 先入后出打印

        show\_block(t, step)

        # 可视化

        # plot\_matrix(t, block=False, plt=plt, zero\_color="#FFC050", another\_color="#1D4946",

        #             title="DFS", step=str(step))

        step += 1

    return steps  # 返回步数

def DFS(start, end, generate\_child\_fn, max\_depth):

    """

    DFS 算法

    :参数 start: 起始状态

    :参数 end: 终止状态

    :参数 generate\_child\_fn: 产生孩子节点的函数

    :参数 max\_depth: 最深搜索深度

    :返回: 最优路径长度

    """

    root = State(0, start, hash(str(S0)), None)  # 根节点

    end\_state = State(0, end, hash(str(SG)), None)  # 最后的节点

    if root == end\_state:

        print("start == end !")

    OPEN.append(root)  # 放入队列

    node\_hash\_set = set()  # 存储节点的哈希值

    node\_hash\_set.add(root.hash\_value)

    while len(OPEN) != 0:

        # 计数

        global SUM\_NODE\_NUM

        SUM\_NODE\_NUM += 1

        top = OPEN.pop(0)  # 依次出队

        if top == end\_state:  # 结束后直接输出路径

            return print\_path(top)

        if top.depth >= max\_depth:  # 超过深度则回溯

            continue

        # 产生孩子节点，孩子节点加入OPEN表

        generate\_child\_fn(sn\_node=top, sg\_node=end\_state, hash\_set=node\_hash\_set)

    print("在当前深度下没有找到解，请尝试增加搜索深度")  # 没有路径

    return -1

if \_\_name\_\_ == '\_\_main\_\_':

    print('请输入数字：(八数码：8 十五数码：15)')

    flag = input()

    Max\_depth = int(input('搜索深度为：'))

    while True:

        if flag == '8':

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

            num = 3

            print('请输入初始八数码:')

            break;

        elif flag == '15':

            SG = [[1, 2, 3, 4],

                  [5, 6, 7, 8],

                  [9, 10, 11, 12],

                  [13, 14, 15, 0]]

            num = 4

            print('请输入初始15数码:')

            break;

        else:

            print('输入错误')

            flag = input()

    for i in range(num):

        S0.append(list(map(int, input().split())))

    time1 = time.time()

    length = DFS(S0, SG, generate\_child, Max\_depth)

    time2 = time.time()

    if length != -1:

        print("搜索最优路径长度为", length)

        print("搜索时长为", (time2 - time1), "s")

        print("共检测节点数为", SUM\_NODE\_NUM)

BFS.py

import heapq

import copy

import time

import math

import matplotlib.pyplot as plt

# 初始状态

# S0 = [[11, 9, 4, 15],

#       [1, 3, 0, 12],

#       [7, 5, 8, 6],

#       [13, 2, 10, 14]]

# S0 = [[5, 1, 2, 4],

#       [9, 6, 3, 8],

#       [13, 15, 10, 11],

#       [0, 14, 7, 12]]

S0 = []

SG = []

# 上下左右四个方向移动

MOVE = {'up': [-1, 0],

        'down': [1, 0],

        'left': [0, -1],

        'right': [0, 1]}

# OPEN表

OPEN = []

# 节点的总数

SUM\_NODE\_flag = 0

# 状态节点

class State(object):

    def \_\_init\_\_(self, depth=0, state=None, hash\_value=None, father\_node=None):

        """

        初始化

        :参数 depth: 从初始节点到目前节点所经过的步数

        :参数 state: 节点存储的状态 4\*4的列表

        :参数 hash\_value: 哈希值，用于判重

        :参数 father\_node: 父节点指针

        """

        self.depth = depth

        self.child = []  # 孩子节点

        self.father\_node = father\_node  # 父节点

        self.state = state  # 局面状态

        self.hash\_value = hash\_value  # 哈希值

    def \_\_lt\_\_(self, other):  # 用于堆的比较，返回距离最小的

        return self.depth < other.depth

    def \_\_eq\_\_(self, other):  # 相等的判断

        return self.hash\_value == other.hash\_value

    def \_\_ne\_\_(self, other):  # 不等的判断

        return not self.\_\_eq\_\_(other)

def generate\_child(sn\_node, sg\_node, hash\_set, open\_table):

    """

    生成子节点函数

    :参数 sn\_node:  当前节点

    :参数 sg\_node:  最终状态节点

    :参数 hash\_set:  哈希表，用于判重

    :参数 open\_table: OPEN表

    :返回: None

    """

    global flag, num

    if sn\_node == sg\_node:

        open\_table.appent(sg\_node)

        print('已找到终止状态！')

        return

    # -8；

    # for i in range(0, 3):

    #     for j in range(0, 3):

    # -15：

    for i in range(0, num):

        for j in range(0, num):

            if sn\_node.state[i][j] != 0:

                continue

            for d in ['left', 'up', 'down', 'right']:  # 四个偏移方向

                x = i + MOVE[d][0]

                y = j + MOVE[d][1]

                # -8:

                if x < 0 or x >= num or y < 0 or y >= num:  # 越界了

                    # -15：

                    # if x < 0 or x >= 4 or y < 0 or y >= 4:  # 越界了

                    continue

                state = copy.deepcopy(sn\_node.state)  # 复制父节点的状态

                state[i][j], state[x][y] = state[x][y], state[i][j]  # 交换位置

                h = hash(str(state))  # 哈希时要先转换成字符串

                if h in hash\_set:  # 重复则跳过

                    continue

                hash\_set.add(h)  # 加入哈希表

                # # 记录扩展节点的个数

                # global SUM\_NODE\_flag

                # SUM\_NODE\_flag += 1

                depth = sn\_node.depth + 1  # 已经走的距离函数

                node = State(depth, state, h, sn\_node)  # 新建节点

                sn\_node.child.append(node)  # 加入到孩子队列

                open\_table.append(node)

def plot\_matrix(matrix, block, plt, zero\_color="#93C760", another\_color="blue", title=" ", step=" "):

    """

    plot\_matrix: 用来画出矩阵；

    matrix为二维列表；

    plt为画笔，应该为：import matplotlib.pyplot as plt

    """

    plt.subplots(figsize=(4, 4))

    plt.title(title)

    plt.xlabel("Step " + step)

    rows = len(matrix)

    columns = len(matrix[0])

    #  -15：

    # plt.xlim(0, 4 \* rows)

    # plt.ylim(0, 4 \* columns)

    # -8:

    plt.xlim(0, num \* rows)

    plt.ylim(0, num \* columns)

    for i in range(rows):

        for j in range(columns):

            if flag == '8':

                if matrix[i][j] != 0:

                    # 画出一个3\*3的矩形，其中左下角坐标为：(3 \* j, 6 - 3 \* i)，并填充颜色， 0和其他的要有区分；

                    plt.gca().add\_patch(plt.Rectangle((3 \* j, 6 - 3 \* i), 3, 3, color=another\_color, alpha=1))

                else:

                    plt.gca().add\_patch(plt.Rectangle((3 \* j, 6 - 3 \* i), 3, 3, color=zero\_color, alpha=1))

                plt.text(3 \* j + 1.5, 7.5 - 3 \* i, str(matrix[i][j]), fontsize=30, horizontalalignment='center')

            if flag == '15':

                if matrix[i][j] != 0:

                    # 画出一个4\*4的矩形,并填充颜色， 0和其他的要有区分；

                    plt.gca().add\_patch(plt.Rectangle((4 \* j, 12 - 4 \* i), 4, 4, color=another\_color, alpha=1))

                else:

                    plt.gca().add\_patch(plt.Rectangle((4 \* j, 12 - 4 \* i), 4, 4, color=zero\_color, alpha=1))

                plt.text(4 \* j + 2, 12.5 - 4 \* i, str(matrix[i][j]), fontsize=30, horizontalalignment='center')

    plt.xticks([])

    plt.yticks([])

    plt.show(block=block)

    plt.pause(0.5)

    plt.close()

def show\_block(block, step):

    print("------", step, "--------")

    for b in block:

        print(b)

def print\_path(node):

    """

    输出路径

    :参数 node: 最终的节点

    :返回: None

    """

    print("最终搜索路径为：")

    steps = node.depth

    stack = []  # 模拟栈

    while node.father\_node is not None:

        stack.append(node.state)  # 拓展节点

        node = node.father\_node

    stack.append(node.state)

    step = 0

    while len(stack) != 0:

        # # 记录扩展节点的个数

        t = stack.pop()  # 先入后出打印

        show\_block(t, step)

        # 可视化

        plot\_matrix(t, block=False, plt=plt, zero\_color="#FFC050", another\_color="#1D4946",

                    title="BFS", step=str(step))

        step += 1

    return steps  # 返回步数

def BFS(start, end, generate\_child\_fn):

    """

    BFS 算法

    :参数 start: 起始状态

    :参数 end: 终止状态

    :参数 generate\_child\_fn: 产生孩子节点的函数

    :返回: 最优路径长度

    """

    root = State(0, start, hash(str(S0)), None)  # 根节点

    end\_state = State(0, end, hash(str(SG)), None)  # 最后的节点

    if root == end\_state:

        print("start == end !")

    OPEN.append(root)  # 放入队列

    node\_hash\_set = set()  # 存储节点的哈希值

    node\_hash\_set.add(root.hash\_value)

    while len(OPEN) != 0:

        global SUM\_NODE\_flag

        SUM\_NODE\_flag += 1

        top = OPEN.pop(0)  # 依次出队

        if top == end\_state:  # 结束后直接输出路径

            return print\_path(top)

        # 产生孩子节点，孩子节点加入OPEN表

        generate\_child\_fn(sn\_node=top, sg\_node=end\_state, hash\_set=node\_hash\_set,

                          open\_table=OPEN)

    print("无搜索路径!")  # 没有路径

    return -1

if \_\_name\_\_ == '\_\_main\_\_':

    print('请输入数字：(八数码：8 十五数码：15)')

    flag = input()

    while True:

        if flag == '8':

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

            num = 3

            print('请输入初始八数码:')

            break;

        elif flag == '15':

            SG = [[1, 2, 3, 4],

                  [5, 6, 7, 8],

                  [9, 10, 11, 12],

                  [13, 14, 15, 0]]

            num = 4

            print('请输入初始15数码:')

            break;

        else:

            print('输入错误')

            flag = input()

    for i in range(num):

        S0.append(list(map(int, input().split())))

    time1 = time.time()

    length = BFS(S0, SG, generate\_child)

    time2 = time.time()

    if length != -1:

        print("搜索最优路径长度为", length)

        print("搜索时长为", (time2 - time1), "s")

        print("共检测节点数为", SUM\_NODE\_flag)

8.2 实验数据表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 启发函数计算方法 | 搜索最优路径长度 | 搜索时长 | 共检测节点数 |
| 低难度 | 欧式距离 | 10 | 10.80940s | 55 |
| 低难度 | 曼哈顿距离 | 10 | 7.70280s | 68 |
| 低难度 | 线性冲突 | 10 | 10.06034s | 12 |
| 高难度 | 欧式距离 | 43 | 127.74654s | 34311717 |
| 高难度 | 曼哈顿距离 | 45 | 40.62714s | 292615 |
| 高难度 | 线性冲突 | 41 | 13.98486s | 13438 |

图8-1 不同难度下不同启发函数的运行结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 启发函数计算方法 | 搜索最优路径长度 | 搜索时长 | 检测节点数 |
| 情况1 | 深度优先 | 39 | 160.2545s | 1342460 |
| 情况2 | 广度优先 | 15 | 11.1330s | 107285 |
| 情况3 | A\* | 21 | 0.3512s | 2116 |

图8-2 不同算法的搜索性能对比