十五数码问题求解算法及性能比较

华中科技大学 软件学院

摘　要：本论文详细研究并比较了不同算法在求解15数码问题上的性能。首先介绍了15数码问题的背景和研究意义，强调了其在人工智能和计算机科学领域的应用价值。接着，论文描述了15数码问题的状态表示方法，并详细阐述了使用A*搜索算法的求解过程，特别是汉明距离、曼哈顿距离和线性冲突等不同的启发式函数。通过设计实验，比较了这些启发式函数在解决15数码问题时的效率和有效性，展示了不同算法的求解时间、中间状态数和步数。最后，论文对实验结果进行了分析，讨论了不同算法的性能，并给出了对未来研究的建议。

关键词：15数码问题；A*搜索算法；启发式函数；汉明距离；曼哈顿距离；线性冲突；性能比较；算法分析

Fifteen Digital Seeking solutions and comparison

Huazhong University of Sci. and Tec., SSE

Abstract: This thesis studied in detail and compared the performance of different algorithms on solving 15 digital problems. First of all, the background and research significance of the 15 digital problems emphasized its application value in the field of artificial intelligence and computer science. Then, the paper describes the status representation method of the 15 digital problem, and elaborates in detail the process of solving the use of the A*search algorithm, especially the different inspirational functions such as Hanying distance, Manhattan distance and linear conflict. By designing experiments, the efficiency and effectiveness of these inspirational functions in solving 15 digital problems show the solution time, middle status and steps of different algorithms. Finally, the paper analyzed the experimental results, discussed the performance of different algorithms, and gave suggestions for future research.

Key Words: 15 digital problems; a*search algorithm; inspiration function; distance of Hanying; Manhattan distance; linear conflict; performance comparison; algorithm analysis

目 录

1 引言

1.1 问题背景和研究意义

1.1.1 问题背景

8 数码问题，被广泛认为是一种经典的智力游戏，它不仅适合各年龄段的人群，还具有显著的益智作用。这个游戏在一个3x3的方盒内布置有8块正方形积木，每一块积木上都标有从1到8的数字，而剩余的一个空格允许周围的积木移动（即与空白交换位置）。玩家的任务是通过一系列的移动，将初始随机布局的积木恢复成特定的目标形式。这个过程中，只有空白周围的方块可以移动，因此达到目标布局通常需要多个步骤和策略的运用。

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。

图 1-1 八数码问题

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

图 1-2 十五数码问题

1.1.2 研究意义

15数码问题不仅是一种智力游戏，它在计算机科学和人工智能领域具有重要的理论和应用价值。该问题的求解涉及到多个领域的核心概念，包括算法设计、启发式搜索、复杂度分析等。通过开发高效的算法来解决这个问题，我们不仅能够提高解决类似智力游戏的能力，还能够在算法设计和优化方面获得深刻的洞见。

此外，15数码问题的算法研究也对人工智能领域有着直接的影响。高效的算法可以为人工智能系统中的问题求解提供范例，尤其是在路径规划、模式识别和自动决策制定等领域。它还可以作为算法教学和研究的一个有价值的案例，帮助学生和研究人员更好地理解复杂问题求解的策略和方法。

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。在本次编程实现中，这个矩阵用二维数组status[][]来表示，空格用特殊字符（本此实验中为字符'0'）表示。如下文的代码所示：

```
//定义二维数组来存储数据表示某一个特定状态
typedef int status[size][size];

//开始状态与目标状态
status start = {};
status target = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0};
```

2.2 15 数码问题的求解算法

1. **搜索算法**：求解15数码问题通常使用A*搜索算法。A*算法是一种启发式搜索算法，它结合了最优优先搜索的高效性和Dijkstra算法的确保最短路径的特性。它通过维护一个优先队列来选择下一个探索的状态，优先考虑估计成本最低的状态。
2. **启发式函数**：在A*算法中，启发式函数用于估计从当前状态到目标状态的最小成本。常用的启发式函数包括：

- **汉明距离（Hamming Distance）**：这是计算每个方块不在其目标位置的数量。简单来说，就是计数有多少数字不在正确的位置上。

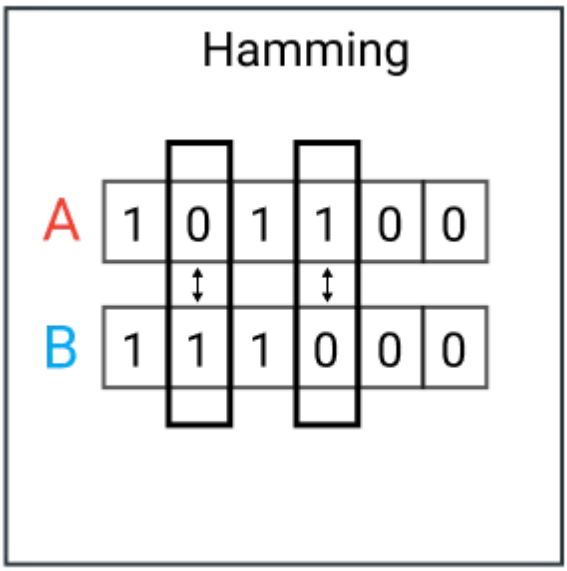

图 2-1 汉明距离表示示意图

- **曼哈顿距离（Manhattan Distance）**：对于每个方块，计算其在网格上从当前位置到目标位置的最少移动次数（只考虑水平和垂直移动）。所有方块的曼哈顿距离之和就是启发式的估计值。计算公式为 $h(n) = |n_x - goal_x| + |n_y - goal_y|)$，其中 (n_x, n_y) 是当前节点的坐标，$(goal_x, goal_y)$ 是目标节点的坐标。

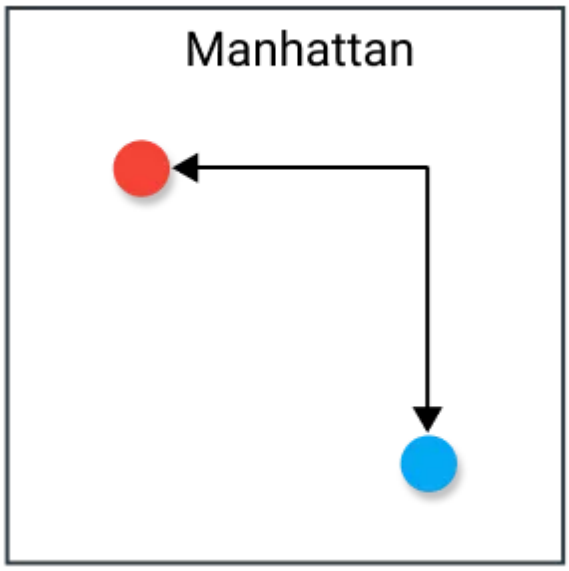

图 2-2 曼哈顿距离表示示意图

- **线性冲突（Linear Conflict）**：这是对曼哈顿距离的一种改进。如果两个方块在同一行或列中，并且都处于彼此的目标路径上，那么它们之间存在线性冲突。对于每对发生冲突的方块，需要额外的移动次数来解决这个冲突。因此，总的启发式估计值是曼哈顿距离加上所有线性冲突的调整值。

- **模式数据库（Pattern Database）**：这是一种预先计算并存储特定模式或方块组合的最优解的方法。对于十五数码问题，可以创建几个模式数据库，每个数据库对应网格上一组特定的方块。然后，将这些数据库中的值相加以得到启发式估计。

  这些启发式函数在实践中的表现取决于具体的问题实例和算法的实现细节。通常，更复杂的启发式函数（如线性冲突和模式数据库）可以提供更准确的估计，但同时也可能需要更多的计算资源。在本次实验中，选择的启发函数为：**汉明距离（Hamming Distance）、曼哈顿距离（Manhattan Distance）和线性冲突（Linear Conflict）。**

3. **状态转换和扩展**：在每一步中，算法探索所有可能的移动（上、下、左、右），生成新的状态。每个新状态都会根据其成本（已走步数+启发式估计成本）加入到优先队列中。

4. **初始状态和目标状态**：在本实验中，初始状态从可以通过Console控制台进行输入，可以针对不同难度的测试样式进行测试；目标状态是数字顺序排列（1到15）和一个空格（0），在代码中已经做好定义。

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. **求解时间**：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**：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. **步数**：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数码问题实例

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。如图3-1、图3-2所示：

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

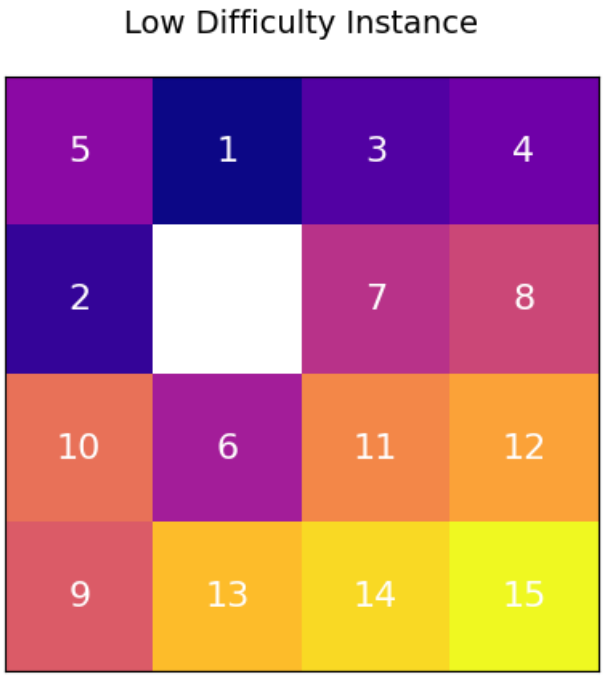

图 3-1 低难度十五数码实例

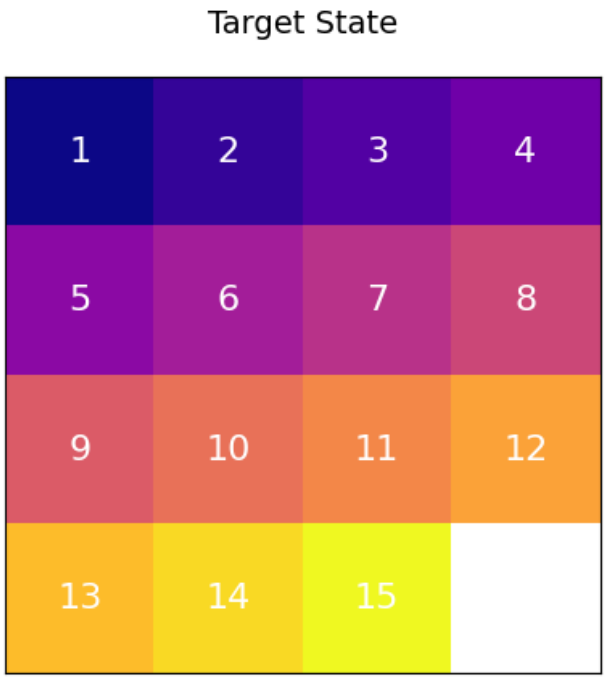

图 3-2 目标状态

其最短步骤如下所示：

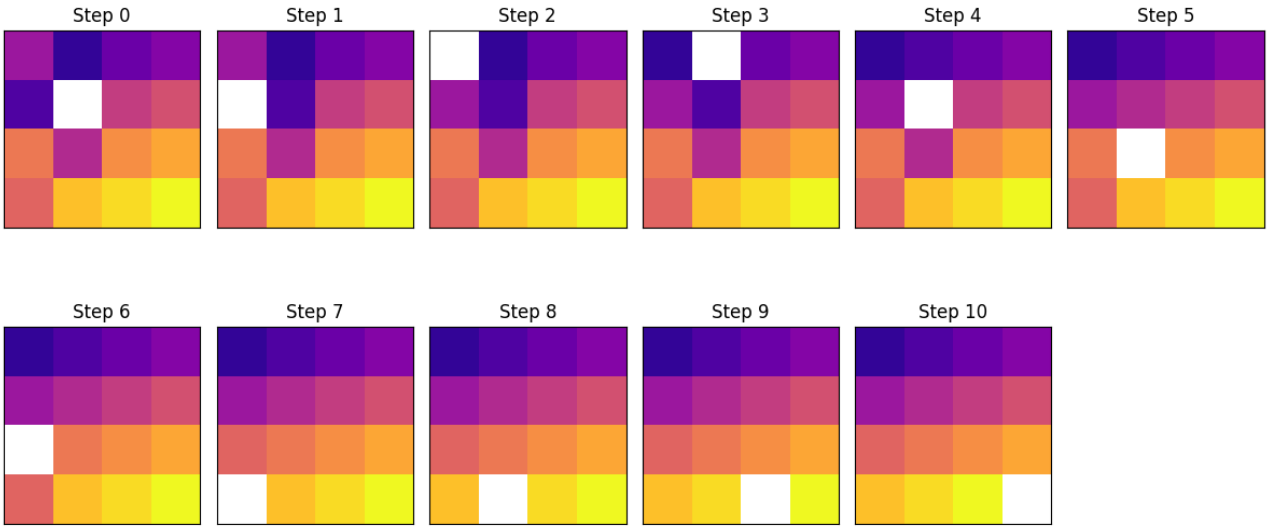

图 3-3 低难度实例解决方法

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

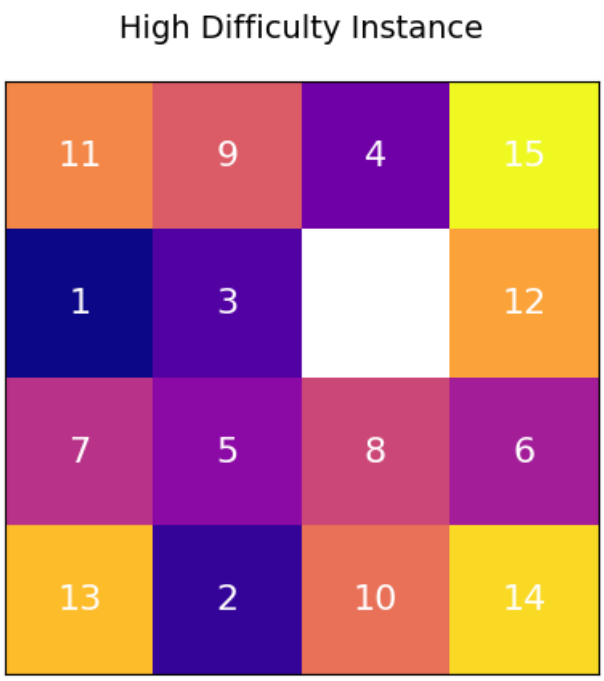

图 3-4 高难度十五数码实例

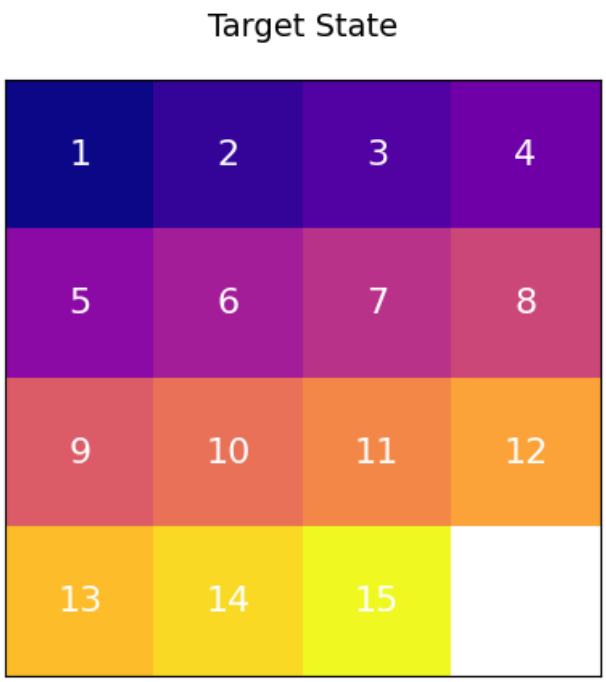

图 3-5 目标状态

其最短步骤如下所示：

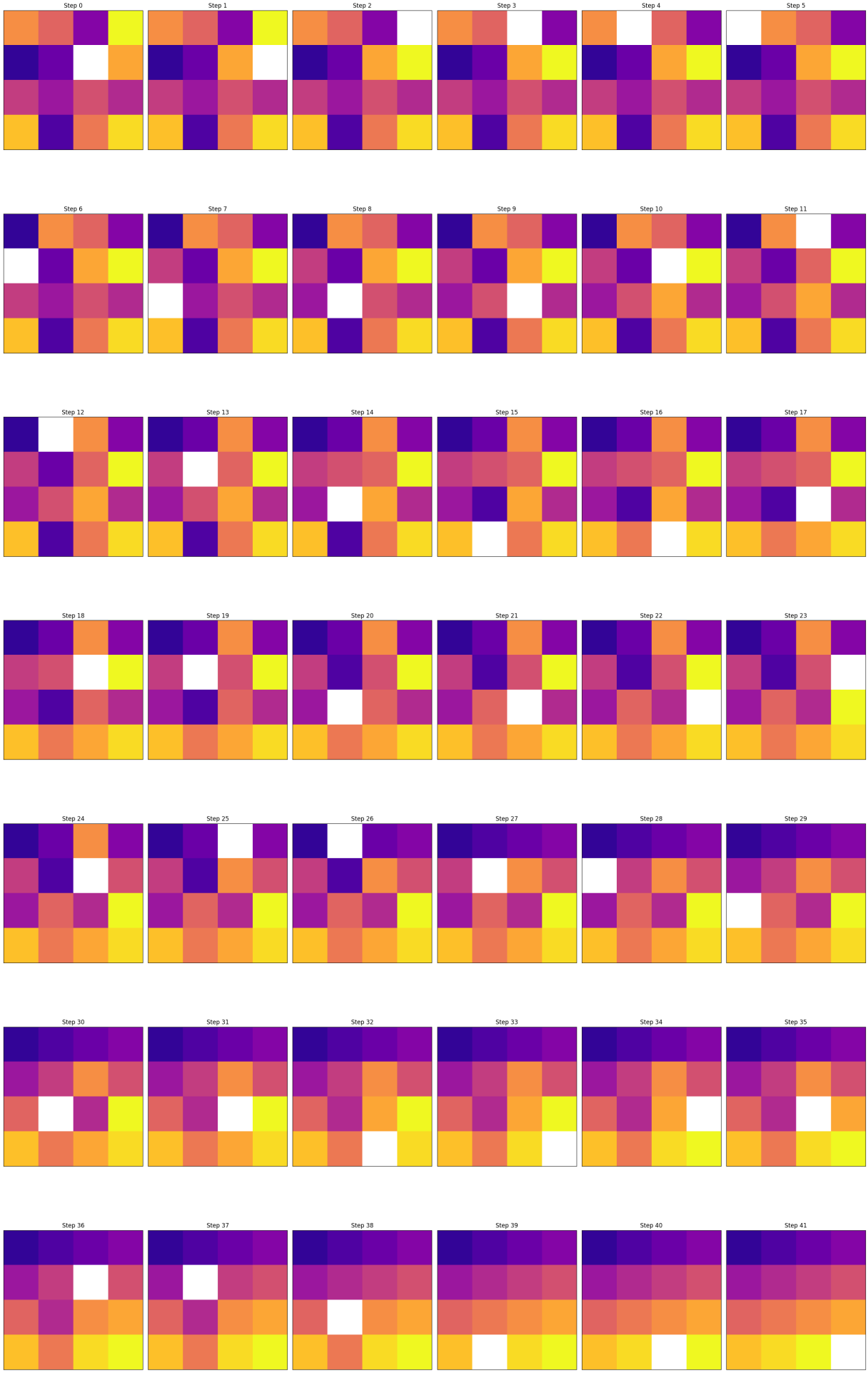

图 3-6 高难度实例解决方法

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. **汉明距离（Hamming Distance）**：这种方法计算当前状态中与目标状态不匹配的方块数量。对于15数码问题，就是计算除空格外，放置位置错误的方块数量。汉明距离简单易计算，但可能不够精确，因为它没有考虑到达目标状态所需的实际步数。

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。

在本次实验中，我们通过定义宏变量的方式来选择不同的启发函数，在程序编译的过程中会自动选择我们需要的启发函数，如下述代码所示：

```
// 通过宏定义确定启发函数
//#define HAMMING
//#define MANHATTAN
#define LINEAR

...

//计算结点h值
int computeHValue(PNode theNode) {
#ifdef HAMMING
    return hammingDistance(theNode);
#elifdef MANHATTAN
    return manhattanDistance(theNode);
#elifdef LINEAR
    return linearConflict(theNode);
#else
    cout << "错误：请先定义启发函数方式！" << endl;
#endif
}
```

3.3.2 实验方法

对于每种启发式函数，将使用A*搜索算法求解一系列不同难度级别的15数码问题实例。实验将记录每种情况下的求解时间和步数，以及算法的空间复杂度。通过比较这些不同启发式函数的性能，我们可以评估哪种方法在求解15数码问题时最有效。这种比较将有助于理解不同启发式函数对搜索算法效率的影响，从而为开发更高效的问题求解策略提供重要的洞见。如下面的代码所示：

```
// 记录开始时间
auto start = chrono::high_resolution_clock::now();

```

```
...// 程序代码

//计算结束时间
auto end = chrono::high_resolution_clock::now();

// 计算并输出求解时间
chrono::duration<double> elapsed = end - start;
cout << "求解时间: " << elapsed.count() << "秒" << endl;


// 输出状态数
cout << "中间状态数: " << numCount << endl;

// 输出步数
if (getGoal) {
    cout << "步数: " << tmpNode->g_value + 1  << endl;
}
```

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. **汉明距离（Hamming 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

通过运行程序，我们得到了下面的结果运行结果，表4-1展示了不同方法在相同实例下的性能差异。通过对比求解时间和步数，可以看出哪种方法在特定情况下更加高效。例如，在实例1中，曼哈顿距离表现出了更短的求解时间和更少的步数；而在实例2中，线性冲突求解时间更短、步数更少，显示出其在更复杂的布局中的优势。

表 4-1 不同实例在每种方法下的求解结果

实例名称	求解时间 (秒)	中间状态数	步数	启发函数
低难度实例	0.000327873	24	10	汉明距离
低难度实例	0.000260946	12	10	曼哈顿距离
低难度实例	0.000327643	12	10	线性冲突
高难度实例	TLE	TLE	TLE	汉明距离
高难度实例	22.2947	31056	41	曼哈顿距离
高难度实例	3.87911	13438	41	线性冲突

** TLE: Time Limit Exceed, 超出运行时长要求限制。*

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（汉明距离、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. **汉明距离**：
   - 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（时间超限TLE）。
   - 汉明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汉明距离。
   - 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汉明距离。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，虽然比汉明距离更准确，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 结果可视化

5.3.1 性能比较可视化

为了直观展示不同方法的性能差异，本实验使用条形图来表示求解时间和步数的比较。对于每种方法，使用一个条形图来表示不同实例下的求解时间，步数和中间状态数。这种可视化方法可以直观地显示出在不同复杂度的实例中，哪种方法更有效，以及它们在性能上的差异。

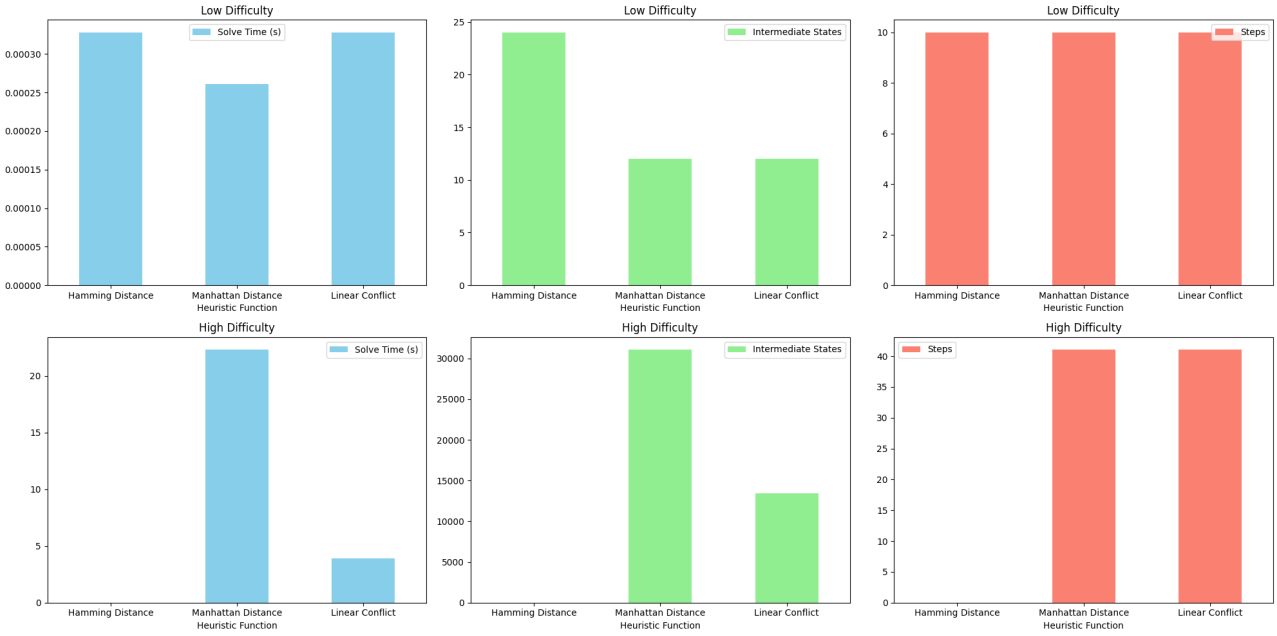

图 5-1 实验结果可视化

5.3.2 求解过程可视化

本实验的代码中通过宏定义DEBUG来打印中间过程的数据，便于展现求解过程中f值、g值和h值的变化，如下述代码所示：

```
// 是否打印中间状态
#define DEBUG

...

#ifdef DEBUG
        //打印中间状态
        printf("第%ld个状态是：", numCount++);
        outputStatus(tmpNode);
        printf("f值：%-2d; g值：%-2d; h值：%-2d\n\n",
               tmpNode->f_value, tmpNode->g_value, tmpNode->h_value);
#else
        numCount++;
#endif
```

输出如下：

```
5 1 3 4 2 0 7 8 10 6 11 12 9 13 14 15
第1个状态是：
5  1  3  4
2  0  7  8
10 6  11 12
9  13 14 15

f值：10; g值：0 ; h值：10

第2个状态是：
5  1  3  4
0  2  7  8
10 6  11 12
9  13 14 15
```

```
f值: 10; g值: 1 ; h值: 9

...
```

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，而汉明距离虽然能够解决问题，但效率稍低。在高难度实例中，汉明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，使其更精确地反映实际解决问题所需的步骤。

2. **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切换到线性冲突。

3. **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。

4. **内存和处理优化**：考虑到高难度实例中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。

6. **实际应用场景的适应性**：考虑将这些算法应用于实际问题时的适应性，如机器人路径规划等，可能需要对算法进行相应的调整和优化。

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。未来的工作应着重于进一步优化算法性能，同时探索新的算法和技术，以提高解决复杂问题的效率和准确性。

7 参考文献

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

[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] Manzini, G. (1995). BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75(2), 347-360.

[4] Felner, A., Korf, R. E., & Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22, 279-318.

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

8 附录

8.1 实验代码

8.1.1 a_star.cpp

求解十五数码问题的C++代码，仅使用C++的部分新特性，核心数据结构均为C语言实现。

```cpp
#include <iostream>
#include <cstdlib>
#include <chrono>

#define size 4
using namespace std;

// 通过宏定义确定启发函数
//#define HAMMING
//#define MANHATTAN
#define LINEAR

// 是否打印中间状态
#define DEBUG

//定义二维数组来存储数据表示某一个特定状态
typedef int status[size][size];
struct SpringLink;

//定义状态图中的结点数据结构
typedef struct Node {
    status data;                //结点所存储的状态
    struct Node *parent;        //指向结点的父亲结点
    struct SpringLink *child;   //指向结点的后继结点
    struct Node *next;          //指向open或者closed表中的后一个结点
    int f_value;                //结点的总的路径
    int g_value;                //结点的实际路径
    int h_value;                //结点的估计成本值
} NNode, *PNode;


//定义存储指向结点后继结点的指针的地址
typedef struct SpringLink {
    struct Node *pointData;     //指向结点的指针
    struct SpringLink *next;    //指向兄第结点
} SPLink, *PSPLink;


PNode open;
PNode closed;


//开始状态与目标状态
status start = {};
status target = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0};


```

```
//初始化一个空链表
void initLink(PNode &Head) {
    Head = (PNode) malloc(sizeof(NNode));
    Head->next = nullptr;
}


//判断链表是否为空
bool isEmpty(PNode Head) {
    if (Head->next == nullptr)
        return true;
    else
        return false;
}


//从链表中拿出一个数据
void popNode(PNode &Head, PNode &FNode) {
    if (isEmpty(Head)) {
        FNode = nullptr;
        return;
    }
    FNode = Head->next;
    Head->next = Head->next->next;
    FNode->next = nullptr;
}


//向结点的最终后继结点链表中添加新的子结点
void addSpringNode(PNode &Head, PNode newData) {
    auto newNode = (PSPLink) malloc(sizeof(SPLink));
    newNode->pointData = newData;
    newNode->next = Head->child;
    Head->child = newNode;
}


//释放状态图中存放结点后继结点地址的空间
void freeSpringLink(PSPLink &Head) {
    PSPLink tmm;

    while (Head != nullptr) {
        tmm = Head;
        Head = Head->next;
        free(tmm);
    }
}


//释放open表与closed表中的资源
void freeLink(PNode &Head) {
    PNode tmn;

    tmn = Head;
    Head = Head->next;
    free(tmn);
```

```
104
105     while (Head != nullptr) {
106         //首先释放存放结点后继结点地址的空间
107         freeSpringLink(Head->child);
108         tmn = Head;
109         Head = Head->next;
110         free(tmn);
111     }
112 }
113
114
115 //向普通链表中添加一个结点
116 void addNode(PNode &Head, PNode &newNode) {
117     newNode->next = Head->next;
118     Head->next = newNode;
119 }
120
121
122 //向非递减排列的链表中添加一个结点
123 void addAscNode(PNode &Head, PNode &newNode) {
124     PNode P;
125     PNode Q;
126
127     P = Head->next;
128     Q = Head;
129     while (P != nullptr && P->f_value < newNode->f_value) {
130         Q = P;
131         P = P->next;
132     }
133     //上面判断好位置之后，下面就是简单的插入了
134     newNode->next = Q->next;
135     Q->next = newNode;
136 }
137
138
139 // 汉明距离
140 int hammingDistance(PNode theNode) {
141     int num = 0;
142     for (int i = 0; i < 4; i++) {
143         for (int j = 0; j < 4; j++) {
144             if (theNode->data[i][j] != target[i][j])
145                 num++;
146         }
147     }
148     return num;
149 }
150
151 // 曼哈顿距离
152 int manhattanDistance(PNode theNode) {
153     int distance = 0;
154     for (int i = 0; i < 4; i++) {
155         for (int j = 0; j < 4; j++) {
156             int value = theNode->data[i][j];
157             if (value) {
158                 int targetX = (value - 1) / 4;
159                 int targetY = (value - 1) % 4;
```

```
                distance += abs(i - targetX) + abs(j - targetY);
            }
        }
    }
    return distance;
}

int linearConflict(PNode theNode) {
    int distance = manhattanDistance(theNode);
    // 在行方向检查
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            for (int k = j + 1; k < 4; ++k) {
                int currentValue = theNode->data[i][j];
                int compareValue = theNode->data[i][k];
                if (currentValue != 0 && compareValue != 0 &&
                    (currentValue - 1) / 4 == i &&
                    (compareValue - 1) / 4 == i &&
                    (currentValue - 1) % 4 > (compareValue - 1) % 4) {
                    distance += 2;
                }
            }
        }
    }
    // 在列方向检查
    for (int j = 0; j < 4; ++j) {
        for (int i = 0; i < 4; ++i) {
            for (int l = i + 1; l < 4; ++l) {
                int currentValue = theNode->data[i][j];
                int compareValue = theNode->data[l][j];
                if (currentValue != 0 && compareValue != 0 &&
                    (currentValue - 1) % 4 == j &&
                    (compareValue - 1) % 4 == j &&
                    (currentValue - 1) / 4 > (compareValue - 1) / 4) {
                    distance += 2;
                }
            }
        }
    }
    return distance;
}

//计算结点h值
int computeHValue(PNode theNode) {
#ifdef HAMMING
    return hammingDistance(theNode);
#elifdef MANHATTAN
    return manhattanDistance(theNode);
#elifdef LINEAR
    return linearConflict(theNode);
#else
    cout << "错误：请先定义启发函数方式！" << endl;
#endif
}

//计算结点的f，g，h值
```

```
void computeAllValue(PNode &theNode, PNode parentNode) {
    if (parentNode == nullptr)
        theNode->g_value = 0;
    else
        theNode->g_value = parentNode->g_value + 1;

    theNode->h_value = computeHValue(theNode);
    theNode->f_value = theNode->g_value + theNode->h_value;
}


//初始化函数，进行算法初始条件的设置
void initial() {
    //初始化open以及closed表
    initLink(open);
    initLink(closed);

    //初始化起始结点，令初始结点的父节点为空结点
    PNode nullptrNode = nullptr;
    auto Start = (PNode) malloc(sizeof(NNode));
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            Start->data[i][j] = start[i][j];
        }
    }
    Start->parent = nullptr;
    Start->child = nullptr;
    Start->next = nullptr;
    computeAllValue(Start, nullptrNode);

    //起始结点进入open表
    addAscNode(open, Start);
}


//将B节点的状态赋值给A结点
void statusAEB(PNode &ANode, PNode BNode) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            ANode->data[i][j] = BNode->data[i][j];
        }
    }
}


//两个结点是否有相同的状态
bool hasSameStatus(PNode ANode, PNode BNode) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (ANode->data[i][j] != BNode->data[i][j])
                return false;
        }
    }
    return true;
}

```

```
272
273 //结点与其祖先结点是否有相同的状态
274 bool hasAnceSameStatus(PNode OriginNode, PNode AncientNode) {
275     while (AncientNode != nullptr) {
276         if (hasSameStatus(OriginNode, AncientNode))
277             return true;
278         AncientNode = AncientNode->parent;
279     }
280     return false;
281 }
282
283
284 //取得方格中空的格子的位置
285 void getPosition(PNode theNode, int &row, int &col) {
286     for (int i = 0; i < 4; i++) {
287         for (int j = 0; j < 4; j++) {
288             if (theNode->data[i][j] == 0) {
289                 row = i;
290                 col = j;
291                 return;
292             }
293         }
294     }
295 }
296
297
298 //交换两个数字的值
299 void swap(int &A, int &B) {
300     int C; C = B; B = A; A = C;
301 }
302
303
304 //检查相应的状态是否在某一个链表中
305 bool inLink(PNode specificNode, PNode theLink, PNode &theNodeLink, PNode &preNode) {
306     preNode = theLink;
307     theLink = theLink->next;
308
309     while (theLink != nullptr) {
310         if (hasSameStatus(specificNode, theLink)) {
311             theNodeLink = theLink;
312             return true;
313         }
314         preNode = theLink;
315         theLink = theLink->next;
316     }
317     return false;
318 }
319
320
321 //产生结点的后继结点(与祖先状态不同)链表
322 void SpringLink(PNode theNode, PNode &spring) {
323     int row;
324     int col;
325
326     getPosition(theNode, row, col);
327
```

```
    //空的格子右边的格子向左移动
    if (col != 3) {
        auto rlNewNode = (PNode) malloc(sizeof(NNode));
        statusAEB(rlNewNode, theNode);
        swap(rlNewNode->data[row][col], rlNewNode->data[row][col + 1]);
        if (hasAnceSameStatus(rlNewNode, theNode->parent)) {
            free(rlNewNode);//与父辈相同，丢弃本结点
        } else {
            rlNewNode->parent = theNode;
            rlNewNode->child = nullptr;
            rlNewNode->next = nullptr;
            computeAllValue(rlNewNode, theNode);
            //将本结点加入后继结点链表
            addNode(spring, rlNewNode);
        }
    }
    //空的格子左边的格子向右移动
    if (col != 0) {
        auto lrNewNode = (PNode) malloc(sizeof(NNode));
        statusAEB(lrNewNode, theNode);
        swap(lrNewNode->data[row][col], lrNewNode->data[row][col - 1]);
        if (hasAnceSameStatus(lrNewNode, theNode->parent)) {
            free(lrNewNode);//与父辈相同，丢弃本结点
        } else {
            lrNewNode->parent = theNode;
            lrNewNode->child = nullptr;
            lrNewNode->next = nullptr;
            computeAllValue(lrNewNode, theNode);
            //将本结点加入后继结点链表
            addNode(spring, lrNewNode);
        }
    }
    //空的格子上边的格子向下移动
    if (row != 0) {
        auto udNewNode = (PNode) malloc(sizeof(NNode));
        statusAEB(udNewNode, theNode);
        swap(udNewNode->data[row][col], udNewNode->data[row - 1][col]);
        if (hasAnceSameStatus(udNewNode, theNode->parent)) {
            free(udNewNode);//与父辈相同，丢弃本结点
        } else {
            udNewNode->parent = theNode;
            udNewNode->child = nullptr;
            udNewNode->next = nullptr;
            computeAllValue(udNewNode, theNode);
            //将本结点加入后继结点链表
            addNode(spring, udNewNode);
        }
    }
    //空的格子下边的格子向上移动
    if (row != 3) {
        auto duNewNode = (PNode) malloc(sizeof(NNode));
        statusAEB(duNewNode, theNode);
        swap(duNewNode->data[row][col], duNewNode->data[row + 1][col]);
        if (hasAnceSameStatus(duNewNode, theNode->parent)) {
            free(duNewNode);//与父辈相同，丢弃本结点
        } else {
```

```
            duNewNode->parent = theNode;
            duNewNode->child = nullptr;
            duNewNode->next = nullptr;
            computeAllValue(duNewNode, theNode);
            //将本结点加入后继结点链表
            addNode(spring, duNewNode);
        }
    }
}


//输出给定结点的状态
void outputStatus(PNode stat) {
    putchar('\n');
    for (int i = 0; i < 4; i++, putchar('\n')) {
        for (int j = 0; j < 4; j++) {
            printf("%-2d ", stat->data[i][j]);
        }
    }
    putchar('\n');

/* // 这部分代码用于输出格式化的数据，便于使用Python进行数据可视化
 * putchar('\n');
 *     for (int i = 0; i < 4; i++) {
        printf("], [");
        for (int j = 0; j < 4; j++) {
            printf("%d, ", stat->data[i][j]);

        }
    }
*/

}


//输出最佳的路径
void outputBestRoad(PNode goal) {
    int depth = goal->g_value;

    if (goal->parent != nullptr) {
        outputBestRoad(goal->parent);
    }
    cout << "第" << depth-- << "步的状态:" << endl;
    outputStatus(goal);
}


void AStar() {
    PNode tmpNode;          //指向从open表中拿出并放到closed表中的结点的指针
    PNode spring;           //tmpNode的后继结点链
    PNode tmpLNode;         //tmpNode的某一个后继结点
    PNode tmpChartNode;
    PNode thePreNode;       //指向将要从closed表中移到open表中的结点的前一个结点的指针
    bool getGoal = false;   //标识是否达到目标状态
    long numCount = 1;      //记录从open表中拿出结点的序号

```

```
440     //记录开始时间
441     auto start = chrono::high_resolution_clock::now();
442 
443     initial();              //对函数进行初始化
444     initLink(spring);    //对后继链表的初始化
445     tmpChartNode = nullptr;
446 
447     while (!isEmpty(open)) {
448         //从open表中拿出f值最小的元素,并将拿出的元素放入closed表中
449         popNode(open, tmpNode);
450         addNode(closed, tmpNode);
451 #ifdef DEBUG
452         //打印中间状态
453         printf("第%ld个状态是: ", numCount++);
454         outputStatus(tmpNode);
455         printf("f值: %-2d; g值: %-2d; h值: %-2d\n\n",
456                tmpNode->f_value, tmpNode->g_value, tmpNode->h_value);
457 #else
458         numCount++;
459 #endif
460         //如果拿出的元素是目标状态则跳出循环
461         if (computeHValue(tmpNode) == 0) {
462             getGoal = true;
463             break;
464         }
465 
466         //产生当前检测结点的后继(与祖先不同)结点列表，产生的后继结点的parent属性指向当前检
    测的结点
467         SpringLink(tmpNode, spring);
468 
469         //遍历检测结点的后继结点链表
470         while (!isEmpty(spring)) {
471             popNode(spring, tmpLNode);
472             //状态在open表中已经存在，thePreNode参数在这里并不起作用
473             if (inLink(tmpLNode, open, tmpChartNode, thePreNode)) {
474                 addSpringNode(tmpNode, tmpChartNode);
475                 if (tmpLNode->g_value < tmpChartNode->g_value) {
476                     tmpChartNode->parent = tmpLNode->parent;
477                     tmpChartNode->g_value = tmpLNode->g_value;
478                     tmpChartNode->f_value = tmpLNode->f_value;
479                 }
480                 free(tmpLNode);
481             }
482                 //状态在closed表中已经存在
483             else if (inLink(tmpLNode, closed, tmpChartNode, thePreNode)) {
484                 addSpringNode(tmpNode, tmpChartNode);
485                 if (tmpLNode->g_value < tmpChartNode->g_value) {
486                     PNode tempNode;
487                     tmpChartNode->parent = tmpLNode->parent;
488                     tmpChartNode->g_value = tmpLNode->g_value;
489                     tmpChartNode->f_value = tmpLNode->f_value;
490                     freeSpringLink(tmpChartNode->child);
491                     tmpChartNode->child = nullptr;
492                     popNode(thePreNode, tempNode);
493                     addAscNode(open, tempNode);
494                 }
```

```
                free(tmpLNode);
            }
                //新的状态即此状态既不在open表中也不在closed表中
            else {
                addSpringNode(tmpNode, tmpLNode);
                addAscNode(open, tmpLNode);
            }
        }
    }

    //目标可达的话，输出最佳的路径
    if (getGoal) {
        cout << endl;
        cout << "路径长度为: " << tmpNode->g_value << endl;
        outputBestRoad(tmpNode);
    }

    //释放结点所占的内存
    freeLink(open);
    freeLink(closed);

    //计算结束时间
    auto end = chrono::high_resolution_clock::now();

    // 计算并输出求解时间
    chrono::duration<double> elapsed = end - start;
    cout << "求解时间: " << elapsed.count() << "秒" << endl;


    // 输出状态数
    cout << "中间状态数: " << numCount << endl;

    // 输出步数
    if (getGoal) {
        cout << "步数: " << tmpNode->g_value + 1  << endl;
    }

    getchar();


}


int main() {
    for (auto &i: start)
        for (int & j : i) {
            cin >> j;
        }
    AStar();
    return 0;
}
```

8.1.2 plot.ipynb

绘制可视化图像的Python代码。

```python
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
import pandas as pd
import matplotlib.colors as mcolors

print(matplotlib.matplotlib_fname())

def plot_15_puzzle(state, title):
    fig, ax = plt.subplots()
    matrix = np.array(state).reshape(4, 4)
    ax.matshow(np.where(matrix == 0, np.nan, matrix), cmap='plasma')

    for (i, j), val in np.ndenumerate(matrix):
        text = '' if val == 0 else f'{val}'.upper()
        ax.text(j, i, text, ha='center', va='center', color='white', fontsize=16)

    plt.xticks([])
    plt.yticks([])
    plt.title(title, pad=20, fontsize=14)
    plt.show()

# 定义不同难度级别的实例
low_difficulty = [5, 1, 3, 4,
                  2, 0, 7, 8,
                  10, 6, 11, 12,
                  9, 13, 14, 15]
high_difficulty = [11, 9, 4, 15,
                   1, 3, 0, 12,
                   7, 5, 8, 6,
                   13, 2, 10, 14]

target = [1, 2, 3, 4,
          5, 6, 7, 8,
          9, 10, 11, 12,
          13, 14, 15, 0]

# 绘制不同难度的15数码问题实例
plot_15_puzzle(low_difficulty, "Low Difficulty Instance")
plot_15_puzzle(high_difficulty, "High Difficulty Instance")
plot_15_puzzle(target, "Target State")

import matplotlib.pyplot as plt
import numpy as np


# 绘制低难度求解步骤
steps = [
    np.array([[5, 1, 3, 4], [2, 0, 7, 8], [10, 6, 11, 12], [9, 13, 14, 15]]),
    np.array([[5, 1, 3, 4], [0, 2, 7, 8], [10, 6, 11, 12], [9, 13, 14, 15]]),
    np.array([[0, 1, 3, 4], [5, 2, 7, 8], [10, 6, 11, 12], [9, 13, 14, 15]]),
```

```
    np.array([[1, 0, 3, 4], [5, 2, 7, 8], [10, 6, 11, 12], [9, 13, 14, 15]]),
    np.array([[1, 2, 3, 4], [5, 0, 7, 8], [10, 6, 11, 12], [9, 13, 14, 15]]),
    np.array([[1, 2, 3, 4], [5, 6, 7, 8], [10, 0, 11, 12], [9, 13, 14, 15]]),
    np.array([[1, 2, 3, 4], [5, 6, 7, 8], [0, 10, 11, 12], [9, 13, 14, 15]]),
    np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [0, 13, 14, 15]]),
    np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 0, 14, 15]]),
    np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 0, 15]]),
    np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 0]])
]


base_cmap = plt.cm.get_cmap('plasma')
colors = base_cmap(np.arange(base_cmap.N))
colors[0, :] = [1, 1, 1, 1]
custom_cmap = mcolors.ListedColormap(colors)

fig, axes = plt.subplots(nrows=2, ncols=6, figsize=(12, 6))

axes = axes.flatten()

for i, ax in enumerate(axes):
    if i < len(steps):
        ax.imshow(steps[i], cmap=custom_cmap, interpolation='nearest')
        ax.set_title(f"Step {i}")
        ax.set_xticks([])
        ax.set_yticks([])
    else:
        ax.axis('off')

plt.subplots_adjust(hspace=0.5, wspace=0.4)

plt.tight_layout()
plt.show()


# 绘制高难度求解步骤
steps = [
    np.array([[11, 9, 4, 15 ], [1, 3, 0, 12 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[11, 9, 4, 15 ], [1, 3, 12, 0 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[11, 9, 4, 0 ], [1, 3, 12, 15 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[11, 9, 0, 4 ], [1, 3, 12, 15 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[11, 0, 9, 4 ], [1, 3, 12, 15 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[0, 11, 9, 4 ], [1, 3, 12, 15 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 11, 9, 4 ], [0, 3, 12, 15 ], [7, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 11, 9, 4 ], [7, 3, 12, 15 ], [0, 5, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 11, 9, 4 ], [7, 3, 12, 15 ], [5, 0, 8, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 11, 9, 4 ], [7, 3, 12, 15 ], [5, 8, 0, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 11, 9, 4 ], [7, 3, 0, 15 ], [5, 8, 12, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 11, 0, 4 ], [7, 3, 9, 15 ], [5, 8, 12, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 0, 11, 4 ], [7, 3, 9, 15 ], [5, 8, 12, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 0, 9, 15 ], [5, 8, 12, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 8, 9, 15 ], [5, 0, 12, 6 ], [13, 2, 10, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 8, 9, 15 ], [5, 2, 12, 6 ], [13, 0, 10, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 8, 9, 15 ], [5, 2, 12, 6 ], [13, 10, 0, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 8, 9, 15 ], [5, 2, 0, 6 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 8, 0, 15 ], [5, 2, 9, 6 ], [13, 10, 12, 14]]),
```

```
    np.array([[1, 3, 11, 4 ], [7, 0, 8, 15 ], [5, 2, 9, 6 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 2, 8, 15 ], [5, 0, 9, 6 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 2, 8, 15 ], [5, 9, 0, 6 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 2, 8, 15 ], [5, 9, 6, 0 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 2, 8, 0 ], [5, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 11, 4 ], [7, 2, 0, 8 ], [5, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 3, 0, 4 ], [7, 2, 11, 8 ], [5, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 0, 3, 4 ], [7, 2, 11, 8 ], [5, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 2, 3, 4 ], [7, 0, 11, 8 ], [5, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 2, 3, 4 ], [0, 7, 11, 8 ], [5, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [0, 9, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [9, 0, 6, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [9, 6, 0, 15 ], [13, 10, 12, 14]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [9, 6, 12, 15 ], [13, 10, 0, 14]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [9, 6, 12, 15 ], [13, 10, 14, 0]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [9, 6, 12, 0 ], [13, 10, 14, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 11, 8 ], [9, 6, 0, 12 ], [13, 10, 14, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 7, 0, 8 ], [9, 6, 11, 12 ], [13, 10, 14, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 0, 7, 8 ], [9, 6, 11, 12 ], [13, 10, 14, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 6, 7, 8 ], [9, 0, 11, 12 ], [13, 10, 14, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 6, 7, 8 ], [9, 10, 11, 12 ], [13, 0, 14, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 6, 7, 8 ], [9, 10, 11, 12 ], [13, 14, 0, 15]]),
    np.array([[1, 2, 3, 4 ], [5, 6, 7, 8 ], [9, 10, 11, 12 ], [13, 14, 15, 0]])
]

base_cmap = plt.cm.get_cmap('plasma')
colors = base_cmap(np.arange(base_cmap.N))
colors[0, :] = [1, 1, 1, 1]  # Set the first row to white (RGBA)
custom_cmap = mcolors.ListedColormap(colors)

fig, axes = plt.subplots(nrows=7, ncols=6, figsize=(25, 42))

axes = axes.flatten()

for i, ax in enumerate(axes):
    if i < len(steps):
        ax.imshow(steps[i], cmap=custom_cmap, interpolation='nearest')
        ax.set_title(f"Step {i}")
        ax.set_xticks([])
        ax.set_yticks([])
    else:
        ax.axis('off')

plt.subplots_adjust(hspace=0.5, wspace=0.4)

plt.tight_layout()
plt.show()

# 可视化实验结果
import matplotlib.pyplot as plt
import pandas as pd

data = {
    "Instance": ["Low Difficulty", "Low Difficulty", "Low Difficulty", "High
Difficulty", "High Difficulty", "High Difficulty"],
```

```
    "Solve Time (s)": [0.000327873, 0.000260946, 0.000327643, float('nan'), 22.2947, 3.87911],  # TLE represented as NaN
    "Intermediate States": [24, 12, 12, float('nan'), 31056, 13438],
    "Steps": [10, 10, 10, float('nan'), 41, 41],
    "Heuristic Function": ["Hamming Distance", "Manhattan Distance", "Linear Conflict", "Hamming Distance", "Manhattan Distance", "Linear Conflict"]
}
df = pd.DataFrame(data)

df_low = df[df['Instance'] == 'Low Difficulty']
df_high = df[df['Instance'] == 'High Difficulty']

fig, axs = plt.subplots(2, 3, figsize=(20, 10))

# Plotting for Low Difficulty
df_low.plot.bar(x='Heuristic Function', y='Solve Time (s)', ax=axs[0, 0], color='skyblue', legend=True)
df_low.plot.bar(x='Heuristic Function', y='Intermediate States', ax=axs[0, 1], color='lightgreen', legend=True)
df_low.plot.bar(x='Heuristic Function', y='Steps', ax=axs[0, 2], color='salmon', legend=True)

# Plotting for High Difficulty
df_high.plot.bar(x='Heuristic Function', y='Solve Time (s)', ax=axs[1, 0], color='skyblue', legend=True)
df_high.plot.bar(x='Heuristic Function', y='Intermediate States', ax=axs[1, 1], color='lightgreen', legend=True)
df_high.plot.bar(x='Heuristic Function', y='Steps', ax=axs[1, 2], color='salmon', legend=True)

# Setting titles and labels
for i in range(3):
    axs[0, i].set_title("Low Difficulty")
    axs[1, i].set_title("High Difficulty")
    axs[0, i].set_xlabel("Heuristic Function")
    axs[1, i].set_xlabel("Heuristic Function")
    axs[0, i].set_xticklabels(axs[0, i].get_xticklabels(), rotation=0)
    axs[1, i].set_xticklabels(axs[1, i].get_xticklabels(), rotation=0)

plt.tight_layout()
plt.show()
```

8.2 实验数据表和图表

表 8-1 不同实例在每种方法下的求解结果

实例名称	求解时间 (秒)	中间状态数	步数	启发函数
低难度实例	0.000327873	24	10	汉明距离
低难度实例	0.000260946	12	10	曼哈顿距离
低难度实例	0.000327643	12	10	线性冲突
高难度实例	TLE	TLE	TLE	汉明距离
高难度实例	22.2947	31056	41	曼哈顿距离
高难度实例	3.87911	13438	41	线性冲突

** TLE: Time Limit Exceed, 超出运行时长要求限制。*

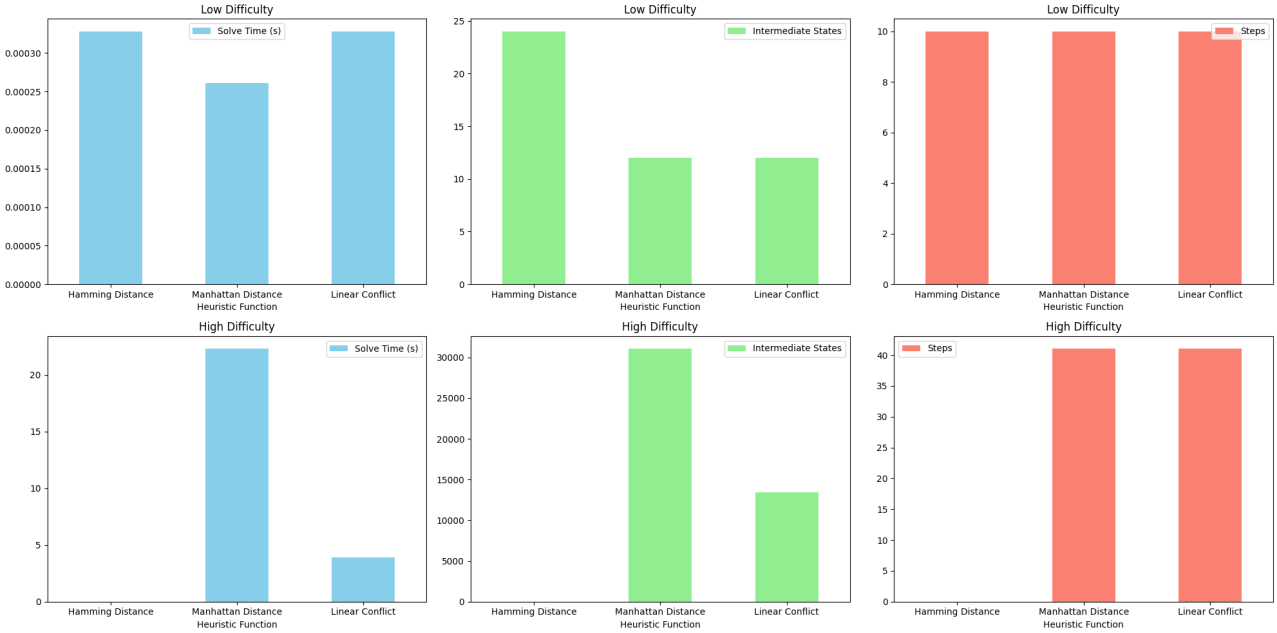

图 8-1 实验结果可视化