基于 A* 算法求解八数码问题

摘要: 本专题报告主要探讨八数码问题的求解方法及其研究现状。八数码问题是在一个 3x3 的方格中，通过移动数字块将初始状态转变为目标状态的经典问题。目前，A 算法作为一种启发式搜索算法被广泛应用，但在某些复杂情况下存在陷入局部最优解的问题。为解决这一问题，本报告分析了启发函数的实现和优化方法，并通过实验和文献综述得出不同启发函数对 A 算法效率的影响。在此基础上，提出了改进方案和建议，包括探索更复杂的八数码变种、研究并行化和分布式算法，以及应用机器学习等新兴技术来提高求解效率和准确性。实验和文献综述结果表明，合理选用启发式函数对提升算法效果至关重要。未来的研究方向可能包括进一步验证和改进改进方法，并探索优化方向和方法，以提高八数码问题求解的效果和效率。

关键词: A* 算法，启发函数

1. 引言

A* 算法属于人工智能领域中的搜索算法流派。搜索算法是人工智能中的一项核心技术，用于在问题的状态空间中寻找解决方案。在当前阶段，搜索算法已经成为人工智能领域中一个相对成熟和基础的研究方向。

当谈到基于 A* 算法求解八数码问题的背景时，我们可以提到八数码问题是一个经典的搜索问题，也被视为人工智能领域中的经典示例之一。它涉及到在一个 3x3 的方格中，通过移动数字块来将初始状态转变为目标状态。这个问题的解决对于算法设计和优化具有挑战性，并且在解决复杂问题时具有广泛的应用前景。

关于八数码问题，A 算法是一种常用的解决方法之一。它是一种启发式搜索算法，通过综合考虑启发函数和代价函数来进行有向图的搜索，以找到最优解。A 算法利用估计函数（启发函数）来评估每个搜索节点的代价，并通过优先级队列来选择具有最小估计代价的节点进行扩展。这种启发式的搜索策略能够在搜索空间中高效地找到最优解，避免无谓的搜索。

然而，在基于 A 算法求解八数码问题的研究中，仍然存在一些问题和挑战。首先，当八数码问题的状态空间非常大时，A 算法可能会面临指数级的搜索复杂性，导致计算资源和时间的开销很大。其次，八数码问题可能存在多个等效解，这些等效解在搜索过程中可能会使算法陷入局部最优解，从而无法找到全局最优解。此外，选择合适的启发函数也是一个关键的挑战，因为它直接影响搜索算法的效率和准确性。

为了解决这些问题，研究者们提出了许多改进的方法和技术。其中，IDA* 算法是一种采用迭代加深深度优先搜索的变种，它可以在有限的内存空间下有效地进行搜索。另外，双向搜索算法通过同时从初始状态和目标状态进行搜索，可以提高搜索效率。还有一些启发式函数的改进方法，例如曼哈顿距离、线性冲突等，可以提供更准确的估计代价，从而改善搜索效果。

业内对于八数码问题的分类主要集中在算法策略的不同方面。除了 A 算法和 IDA

算法之外，还有其他算法，如贪婪最佳优先搜索、遗传算法等，它们采用不同的搜索策略和优化方法来解决八数码问题。此外，八数码问题还可以扩展到更复杂的变种，如 15 数码问题、N 数码问题等。

然而，尽管已经取得了一些进展，仍然存在一些挑战和改进的空间。在某些复杂情况下，现有的算法仍然无法高效地找到最优解。此外，随着问题规模的增加，搜索空间的增长也会导致算法的效率下降。因此，未来的研究可以探索更高效的搜索策略和启发式函数，提出新的算法和技术来解决大规模八数码问题。此外，结合机器学习和深度学习的方法也可能为八数码问题的求解带来新的突破，例如使用强化学习来训练智能体解决八数码问题。

通过实验和文献综述，已经证明了一些改进方法的有效性，比如 IDA* 算法和双向搜索。这些方法在某些情况下显著提高了解决八数码问题的效率和准确性。此外，启发式函数的改进也取得了一定的成果，例如使用更精确的估计代价函数来指导搜索过程。

然而，需要注意的是，每种改进方法都有其适用的场景和局限性。在实际应用中，选择合适的方法取决于具体问题的特点和要求。因此，进一步的实验研究和比较分析是必要的，以便更好地评估不同方法的优劣和适用性。

未来，优化方向可以包括以下几个方面。首先，可以探索更高效的搜索策略，例如并行化和分布式算法，以充分利用多核处理器和分布式计算资源。其次，结合机器学习和深度学习的方法可以进一步改善八数码问题的求解效果，例如使用神经网络来学习启发式函数或优化搜索策略。此外，对于更复杂的八数码变种，如 15 数码问题和 N 数码问题，还可以研究针对这些问题特点的专门优化方法。此外，随着人工智能的快速发展，新的算法和方法不断涌现，对搜索算法的应用和改进也会产生新的影响和挑战。

本篇报告将着重分析启发函数的实现以及优化，并结合测试样例进行分析，实验中我采用不同的启发式函数，综合分析得出不同策略下的启发函数对于 A* 算法效率有较大的影响，认识到合理选用启发式函数的重要性。

2. 国内外发展现状分析

A 算法是一种启发式图搜索算法，其特点在于对估价函数的定义上。对于一般的启发式图搜索，总是选择估价函数 f 值最小的节点作为扩展节点。因此，f 是根据需要找到一条最小代价路径的观点来估算节点的，所以，可考虑每个节点 n 的估价函数值为两个分量：从起始节点到节点 n 的实际代价 g(n) 以及从节点 n 到达目标节点的估价代价 h(n)，且 h(n)≤h(n)，h(n) 为 n 节点到目的结点的最优路径的代价。八数码问题是人工智能中的一个典型问题，目前解决八数码问题的搜索求解策略主要有深度优先搜索、宽度优先搜索、启发式 A 算法。对这些算法进行研究，重点对 A* 算法进行适当改进，使用曼哈顿距离对估价函数进行优化

对使用这些算法解决八数码问题的效率进行比较，从步数、时间、结点数、外显率等各参数，通过具体的实验数据分析，进一步验证各算法的特性。在国内外的研究中，针对八数码问题，有很多团队提出了不

同的方法，由 W. J. Cook, A. M. H. Smith 和 W. H. Warren 团队提出了一种用于解决八数码问题的 A* 算法，该算法使用了一个启发式函数，估计当前状态到目标状态的距离; 由 S. H. Hong 和 S. K. Park 团队撰写的论文提出了一种新的用于解决八数码问题的 A* 算法，该算法使用了一个启发式函数，考虑了每个数字及其目标位置之间的曼哈顿距离; 由 K. Y. Lee 和 S. M. Kim 团队提出了一种用于解决八数码问题的并行 A* 算法，该算法使用分治法将搜索空间划分为更小的子问题，然后并行解决这些子问题。

3. 问题定义和建模

八数码问题也叫重排九宫问题，是人工智能中状态搜索中的经典问题。该问题描述为：在 3×3 的方格盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字，不同棋子上标的数字不相同。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。要求解决的问题是：给出一个初始状态和一个目标状态，找出一种从初始转变成目标状态的移动棋子步数最少的移动步骤。

所示的八数码问题的初始状态 S0 为问题的一个布局，需要找到一个数码移动序列使初始布局 S0 转变为目标布局 Sg。见图 3. 1 。

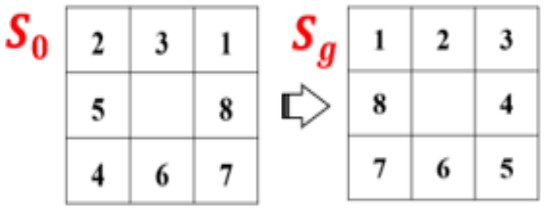

图 3. 1: 八数码问题的一种初始状态与目标状态

主要采用状态空间表示法来表示该问题。状态空间表示可形式化为四元组表示：(S,F,S0,G)

8 个数码的任意一种摆法就是一个状态，所有的摆法就是状态集合 S，它们构成了一个状态空间，大小为 9!；F 是操作算子的集合；S0 是初始状态；G 是目标状态。状态空间的解为从初始状态 S0 到目标状态 G 的操作算子序列。

对于操作算子设计，如果着眼在数码上，相应的操作算子就是数码的移动，其操作算子共有 4（方向）×8（数码）=32 个。如果着眼在空格上，即空格在方格盘上的每个可能位置的上下左右移动，其操作算子可简化为 4 个：① 将空格向上移 Up；② 将空格向左移 Left；③ 将空格向下移 Down；④ 将空格向右移 Right。

移动时要确保空格不会移出方格之外，因此并不是在任何状态下都能运用这 4 个操作算子。如空格在方格盘的右上角时，只能运用两个操作算子——向左移 Left 和向下移 Down。

4. 问题求解和实现算法

4.1. 算法原理

A* 算法是一种高效的启发式搜索算法，它通过估算节点的代价评估函数值并作为节点的综合优先级，当选择下一个需要遍历的节点时，再选取综合优先级最高的节点，逐步地找到最优路径。A* 算法中，若对所有的 x 存在 h(x)≤h*(x), 则称 h(x) 为 h*(x) 的下限，表示某种偏于保守的估计。采用的下限 h(x) 为启发函数的 A 算法，称为 A* 算法，其中限制 h(x)≤h*(x) 十分重要，它能保证 A* 算法找到最优解。在本问题中，g(x) 相对容易得到，就是从初始节点到当前节点的路径代价，即当前节点在搜索树中的深度。关键在于启发函数 h(x) 的选择，A* 算法的搜索效率很大程度上取决于估价函数 h(x)。一般而言，满足 h(x)≤h*(x) 前提下，h(x) 的值越大越好，说明其携带的启发性信息越多，A* 算法搜索时扩展的节点就越少，搜索效率就越高。

A* 算法的估价函数可表示为：

$$f'(n) = g'(n) + h'(n)$$

公式中 f'(n) 是估价函数，g'(n) 是起点到终点的最短路径值（也称为最小耗费或最小代价），h'(n) 是 n 到目标的最短路经的启发值。由于此处的 f'(n) 其实是无法预先知道的，所以实际上使用的是下面的估价函数：f(n) = g(n) + h(n)

其中 g(n) 是从初始结点到节点 n 的实际代价，h(n) 是从结点 n 到目标结点的最佳路径的估计代价。在这里主要是 h(n) 体现了搜索的启发信息，因为 g(n) 是已知的。用 f(n) 作为 f'(n) 的近似，也就是

用 g(n) 代替 g'(n)，h(n) 代替 h'(n)。这样必须满足两个条件：

(1)g(n)>=g'(n)（大多数情况下都是满足的，可以不用考虑），且 f 必须保持单调递增。

(2) h(n) 的单调限制：h 必须小于等于实际的从当前节点到达目标节点的最小耗费（h(n)<=h'(n)）。这一点特别的重要。可以证明，应用这样的估价函数是可以找到最短路径的。

在启发式搜索中，估计函数的定义是十分重要的。如定义不当，则 A* 算法不一定能找到问题的解，即使找到解，也不一定是最优的。

4.2. 算法步骤

具体步骤：从初始状态 S0 出发，分别采用不同的操作符作用于生成新的状态 x 并将其加入 open 表中（对应到状态空间图中便是根节点生成新的子节点 n）, 接着从 open 表中按照某种限制或策略选择一个状态 x 使操作符作用于 x 又生成了新的状态并加入 open 表中 (状态空间图中相应也产生了新的子节点)，如此不断重复直到生成目标状态。

下面主要讲述 A* 算法求解八数码问题的核心步骤：启发式函数的设计以及 open 表与 close 表的维护。

4.2.1 启发式函数设计

（1）启发式函数 h(n)1 h(n)= 不在位的元素个数。

（2）启发式函数 h(n)2 h(n)= 每个棋子与其目标位置之间的距离总和。类似于曼哈顿距离。

（3）启发式函数 h(n)3 按照广度优先搜索的策略，选取当前节点在搜索树中的深度作为 g(x)，但没有使用启发函数 h(n)（h(n)=0），其在找到目标状态之前盲目搜索，生成过多的节点，因此搜索效率相对较低。

4.2.2 open 表与 close 表的维护

open 表：先简单地认为是一个未搜索节点的表。

close 表：先简单地认为是一个已完成搜索的节点的表（即已经将下一个状态放入 open 表内）。

维护两表的主要规则如下：

规则一：对于新添加的节点 S（open 表和 close 表中均没有这个状态），S 直接添加到 open 表中；

规则二：对于已经添加的节点 S（open 表中并且 close 表中没有这个状态），若在 open 表中，与原来的状态 S0 的 f(n) 比较，取最小的一个；

规则三：下一个搜索节点的选择问题，选取 open 表中 f(n) 的值最小的状态作为下一个待搜索节点；

规则四：每次需要将带搜索的节点下一个所有的状态按照规则一、规则二更新 open 表、close 表，搜索完该节点后，移到 close 表中。

4.3. 算法过程

根据 A* 算法的实验原理，f(n) = g(n) + h(n)。在此估价函数公式中当 g(n) 已知的情况下，估价函数 f(n) 会或多或少地受到距离估计值 h(n) 的制约。节点距离目标点越近，h 值越小，f 值相对就越小。这种设计能够确保搜索算法朝着终点的方向进行，从而更有可能找到最短路径。因此，f 是根据需要找到一条最小代价路径的观点来估算节点的。

设计 A* 算法的流程图如图 4.1 所示。按照流程图编写伪代码，并进一步将其转化为完整的程序。其中 open 表用于保存已生成但尚未考察的节点，而 close 表则记录已经访问过的节点。在扩展节点的过程中，需要检查当前节点的后继节点是否已经存在于 open 表和 close 表中。具体编程思路可参考图 4. 1。

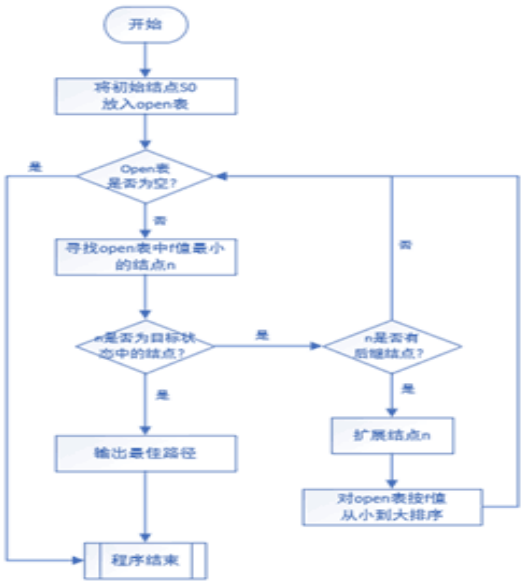

图 4. 1: A* 算法流程图

将起始点加入 open 表当 open 表不为空时：寻找 open 表中 f 值最小的点 current 它是终止点，则找到结果，程序结束。否则，Open 表移出 current，对 current 表中的每一个临近点若它不可走或在 close 表中，略过若它不在 open 表中，加入若它在 open 表中，计算 g 值，若 g 值更小，替换其父节点为 current，更新它的 g 值。若 open 表为空，则路径不存在。

对于某一个特定的八数码问题的初始状态与目标状态，求解问题的搜索树可以表现为

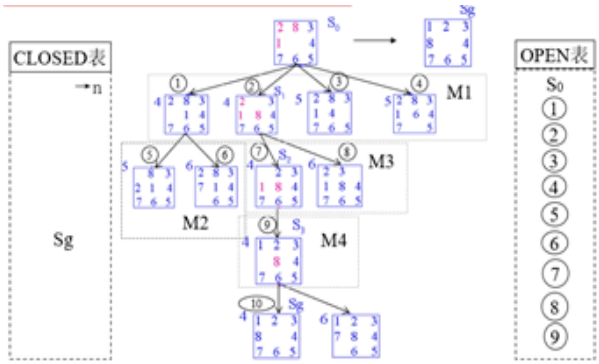

图 4. 2: A* 算法的搜索树及 open 表与 close 表的设置

5. 实现结果分析

这部分可首先叙述一下你的系统实现的软硬件环境；采用 C++ 实现求解八数码问题算法的主要过程。

5.1. 有关数据结构的定义

```
1  //定义二维数组来存储数据表示某一个特定状态
2  typedef int status [3][3];
3  struct SpringLink ;
4  //定义状态图中的结点数据结构
5  typedef struct Node{
6      status data;//结点所存储的状态，一个3*3矩阵
7      struct Node *parent;//指向结点的父亲结点
8      struct SpringLink *child;//指向结点的后继结点
9      struct Node *next;//指向open或者closed表中的后一个结点
10     int fvalue;//结点的总的路径
11     int gvalue;//结点的实际路径
12     int hvalue;//结点的到达目标的困难程度
13 }NNode,*PNode;
14
15 //定义存储指向结点后继结点的指针的地址
16 typedef struct SpringLink {
17     struct Node *pointData;//指向结点的指针
18     struct SpringLink *next;//指向兄第结点
19 }SPLink,*PSPLink;
20
21 PNode open;
22 PNode close
```

5.2. 有关函数的定义

为了使算法的时间统计更加准确与方便，同时减少输出的规模，在算法中省去了单步执行和绘画搜索树的过程。

（1）bool isable()

判断八数码问题是否有解的函数，初始状态与目标状态的逆序数之和的奇偶性相同则有解。

用 F(X) 表示棋盘中数字 X 前面比它小的数的个数，全部数字的 F(X) 之和为 Y=∑(F(X))。如果 Y 为奇数，则称原数字的排列是奇排列；如果 Y 为偶数，则称原数字的排列是偶排列。

（2）void initLink(PNode &Head)

初始化一个空的 PNode 类型链表。

（3）bool isEmpty(PNode Head)

判断链表是否为空。

（4）void popNode(PNode &Head, PNode &FNode)

从链表中拿出一个数据。

（5）void addSpringNode(PNode &Head, PNode newData)

向结点的最终后继结点链表中添加新的子结点。

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

（7）void freeLink(PNode &Head) 释放 open 表与 closed 表中的资源。

（8）void addNode(PNode &Head, PNode &newNode) 向普通链表中添加一个结点。

（9）void addAscNode(PNode &Head, PNode &newNode)

向非递减排列的链表中添加一个结点（已经按照权值进行排序）。

（10）void computeAllValue(PNode &theNode, PNode parentNode)

计算结点的 f，g，h 值。g 值等于父节点 g 值 +1，f=g+h。

（11）int computeHValue_1(PNode theNode)

第一种启发式函数，按照不在位的个数进行计算，其实现代码如下：

```
int computeHValue_1(PNode theNode){
    int num=0;
    for ( int i=0; i<3;++i )
        for ( int j=0; j<3;++j )
            if (theNode->data[i][ j ] !=
                target [ i ][ j ])
                num+=1;
    return num;
}
```

第二种启发式函数，按照棋子不在目标位置的距离总和进行计算，其实现代码如下：

```
int computeHValue_2(PNode theNode){
    int num=0;
    for ( int i=0; i<3;++i )
        for ( int j=0; j<3;++j )
            if (theNode->data[i][ j ]!=
                target [ i ][ j ] && theNode
                ->data[i][j ]!=0)
                for ( int x=0;x<3;++x)
                for ( int y=0;y<3;++y)
                if (theNode->data[i][ j ] ==
                    target [x ][ y ]) {
                num+=abs(x-i)+abs(y-j) ;
        break;
        }
    return num;
}
```

（13）void initial()

初始化函数，进行算法初始条件的设置。调用函数 initLink() 初始化 open 表及 close 表；初始化起始结点，令初始结点的父节点为空结点，进入 open 表。

（14）void statusAEB(PNode &ANode,PNode BNode)

将 B 节点的状态赋值给 A 结点。

（15）bool hasSameStatus(PNode ANode,PNode BNode)

判断两个结点是否有相同的状态。

（16）bool hasAnceSameStatus(PNode OriginNode,PNode AnceNode)

判断结点与其祖先结点是否有相同的状态。

（17）void getPosition(PNode theNode,int &row,int &col)

取得方格中空的格子的位置。

（18）bool inLink(PNode spciNode , PNode theLink , PNode &theNodeLink , PNode &preNode)

检查相应的状态是否在某一个链表中。

（19）void SpringLink(PNode theNode,PNode &spring,int op)

产生结点的后继结点 (与祖先状态不同) 链表。

（20）void outputStatus(PNode stat)

输出给定结点的状态。

（21）void outputBestRoad(PNode goal)

输出最佳的路径。

（22）void AStar(int op)

A* 算法的主体。

（23）main()

主函数，调用 A* 算法，并输出生成节点数目、扩展节点数目、运行时间等信息。

5.3. 系统测试

由于对 open 表与 close 表的动态变化过程输出的内容过多，导致 A* 算法的运行时间过长（比如第三种启发式函数需要几分钟才能运行完，如图 5-1），这远远超出了 A* 算法实际的搜索时间，明显不符合对运行时间计算的意义。故在测试中，对两表的遍历仅作部分展示。

图 5. 1: 运行时间记录

5.4. 随机生成八数码问题格局的测试

随机生成八数码问题的初始状态、目标状态，并判断是否有解。测试结果如 5. 2。

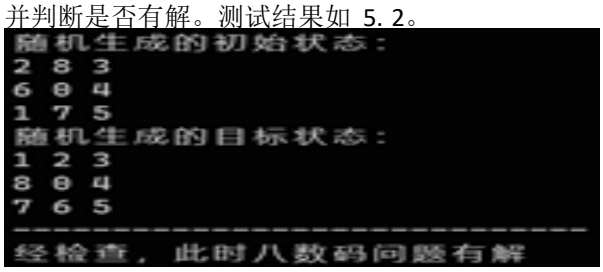

图 5. 2: 随机生成测试图

5.5. 选择预定义的启发式函数模块的测试

本模块根据输入 1 3，选择并应用对应的启发式函数 h(n)1 h(n)3。测试结果如 5. 3 。

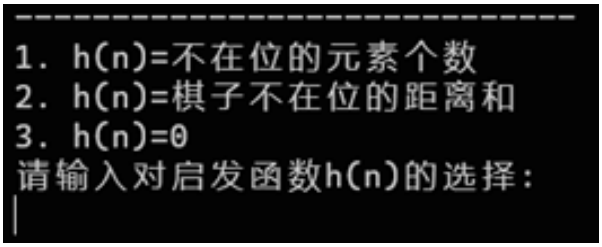

图 5. 3: 启发式函数选择模块测试图

5.6. OPEN 表和 CLOSED 表的动态变化过程展示的测试

可以根据选择展示两表的动态变化过程，两表的部分变化过程如图 5. 4 所示。选择模块如图 5. 5 所示。

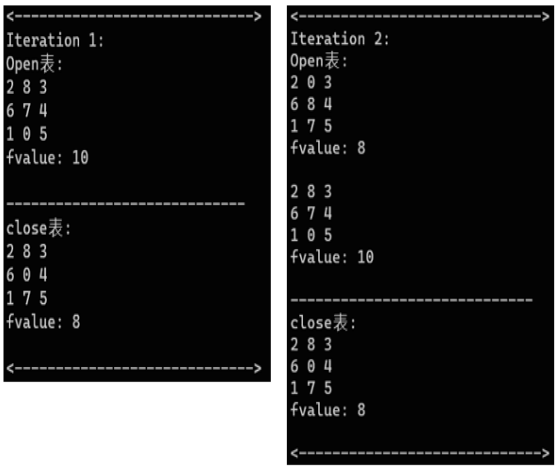

图 5. 4: OPEN 表和 CLOSED 表的动态变化过程展示测试图

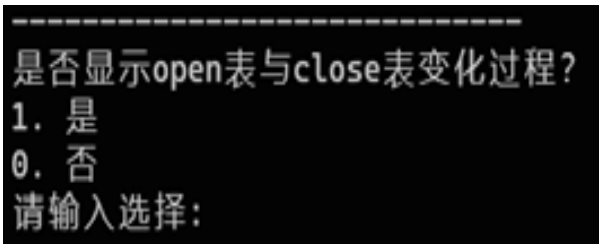

图 5. 5: 显示两表动态变化的选择模块测试图

5.7. 算法执行过程的测试

A* 算法求解执行过程中的状态转换如图 5. 6所示，最优路径输出与扩展节点数和算法执行时间的统计如图 5. 7所示。

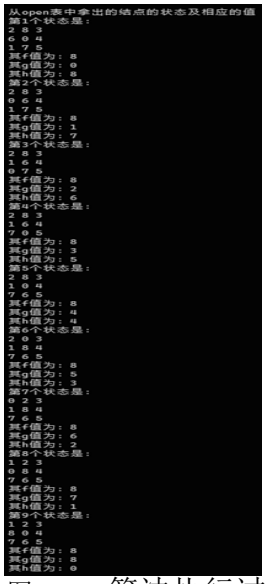

图 5. 6: 算法执行过程演示图

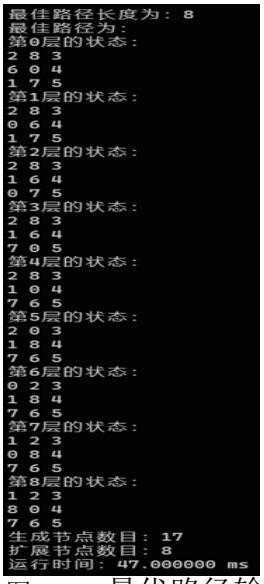

图 5. 7: 最优路径输出测试图

5.8. 算法执行性能的研究与比较

我们对比了这两种启发函数的效果，发现第二种启发函数相对于第一种启发函数表现更优。这表明通过合适选择启发函数，可以更有效地指导搜索算法朝着目标方向进行。进一步比较了采用启发性函数和不采用启发性函数的情况，结果显示采用启发性函数的搜索方法远远优于不采用启发性函数的方法。这验证了启发函数在提高搜索效率和找到更优解方面的重要性。

6. 总结展望和思考

6.1. 总结

在此次实验中，我巩固了人工智能导论课上所学的知识，并成功运用状态空间表示法来解决八数码问题。同时，我加深了对 A 算法的理解和掌握程度，并实践了将 A 算法应用于实际问题的优化过程。我还发现了选择不同的启发式函数对 A* 算法效率的影响，并认识到合理选择启发式函数的重要性。

然而，本次实验还存在一些不足之处。首先，算法执行过程的可视性不够强，输出结果不够简洁，无法进行单步执行。这可能导致对算法运行过程的理解和调试变得困难。其次，虽然部分不足可以归因于时间和问题规模限制，但其中也存在我个人学习不精和能力有限所致的问题。

在此次实验中，我巩固了人工智能导论课上所学的知识，并成功运用状态空间表示法来解决八数码问题。同时，我加深了对 A 算法的理解和掌握程

度，并实践了将 A 算法应用于实际问题的优化过程。我还发现了选择不同的启发式函数对 A* 算法效率的影响，并认识到合理选择启发式函数的重要性。

6.2. 展望

（1）探索更加适用的启发式函数，优化 A* 算法使其效率更高。

（2）调用外部库，使界面更加可视化。

参考文献

[1] 卜奎昊, 宋杰, 李国斌. 基于 A* 算法的八数码问题的优化实现 [J]. 计算机与现代化, 2008 (1): 29-31

[2] 姚雪梅. 人工智能中 A* 算法的程序实现——八数码问题的演示程序 [J]. 电脑与信息技术, 2002, 10(2): 1-3.

[3] 朱永红, 张燕平. 用 VC++ 实现基于 A* 算法的八数码问题 [J]. 计算机技术与发展, 2006, 16(9): 32-34.

[4] 詹志辉, 胡晓敏, 张军. 通过八数码问题比较搜索算法的性能 [J]. 计算机工程与设计, 2007, 28(11): 2505-2508.

[5] 龙振海, 林泓. 8 数码问题求解算法的改进与实现 [J]. 中国高新技术企业, 2010 (2): 19-21.