

N-Puzzle问题求解研究报告

**2023 年 秋 季学期**

姓名：周忠康 学号：21060021080

姓名：王颢凯 学号：21040021027

姓名：曹文汀 学号：21020041002

姓名：田晶怡 学号：21050005076

摘 要

Npuzzle问题是人工智能搜索中的一个经典问题，常用于测试搜索算法的效率和准确性。其中，判断Npuzzle问题是否有解是一个重要的问题，可以通过计算逆序数的个数来判断是否有解。

报告第一部分对问题进行了描述和可解性分析。之后于第二、三部分别介绍了A\*和IDA\*算法，涉及可采纳性证明并在第五部分介绍了它们的优缺点。第四部分详细介绍了实验使用的四种启发式函数，并于结合代码进行详细说明。第六部分展示了我们代码部分的实现。在第七部分设计了性能对比实验，用随机生成测试数据进行绘图对比分析A\*和IDA\*的性能以及各启发函数的性能。之后的第八部分介绍了我们SFML可视化的实现。

本次实验由四个阶段组成：对于第一阶段，在老师的框架上，我们构建了解决Npuzzle问题的程序，并在aFrontier中使用优先队列建立了open表，在该阶段，我们使用了“不在将位牌”和“曼哈顿距离”两种启发式函数，搜索算法为A\*。对于第二阶段，我们使用Zobrist来得到每种状态的哈希值，并使用IDA\*替代A\*，解决了可能出现的OutOfMemoryError问题。此外，考虑到前两种启发式函数求解效率并不理想，我们学习了线性冲突启发式，并用代码进行了实现。对于第三阶段，我们查阅相关论文，按照6-6-3的区块划分方法建立了不相交模式数据库，相比曼哈顿距离和线性冲突，大量减少了生成结点和扩展结点，成功降低搜索时间达到42秒。第四阶段，我们用SFML实现了求解路径的可视化，有较好的视觉效果。

**关键词：**N-Puzzle，A\*，IDA\*，线性冲突，不相交模式数据库，启发式搜索

目 录

[摘 要 2](#_Toc150337247)

[**1. N-Puzzle问题** 1](#_Toc150337248)

[**1.1 问题描述** 1](#_Toc150337249)

[**1.2 可解性分析** 1](#_Toc150337250)

[**2. 启发式A\*算法综述** 2](#_Toc150337251)

[**2.1 A\*算法** 2](#_Toc150337252)

[**2.1.1 算法介绍** 2](#_Toc150337253)

[**2.1.2 算法数据结构** 2](#_Toc150337254)

[**2.1.3 算法流程** 3](#_Toc150337255)

[**2.2 A\*算法可采纳性证明** 3](#_Toc150337256)

[**2.3 重复扩展节点问题** 4](#_Toc150337257)

[**3. 启发式搜索IDA\*算法** 4](#_Toc150337258)

[**3.1 IDA\*算法基本原理及其可采纳性** 4](#_Toc150337259)

[**3.2 IDA\*算法求解步骤** 5](#_Toc150337260)

[**3.3 IDA\*算法描述（伪代码）** 7](#_Toc150337261)

[**4. 启发函数** 8](#_Toc150337262)

[**4.1 启发函数的单调性** 8](#_Toc150337263)

[**4.2 不在将位牌** 8](#_Toc150337264)

[**4.3 曼哈顿距离** 9](#_Toc150337265)

[**4.4 线性冲突** 9](#_Toc150337266)

[**4.5 不相交模式数据库** 10](#_Toc150337267)

[**4.5.1 简述** 10](#_Toc150337268)

[**4.5.2 具体实现** 10](#_Toc150337269)

[**5. 两种算法的优缺点** 12](#_Toc150337270)

[**5.1 A\*算法** 12](#_Toc150337271)

[**5.1.1 A\*算法的优点** 12](#_Toc150337272)

[**5.1.2 A\*算法的缺点** 12](#_Toc150337273)

[**5.2 IDA\*算法** 13](#_Toc150337274)

[**5.2.1 IDA\*算法的优点** 13](#_Toc150337275)

[**5.2.2 IDA\*算法的缺点** 13](#_Toc150337276)

[**6. N-Puzzle求解问题的实现** 13](#_Toc150337277)

[**6.1 文件结构与前两阶段算法实现** 13](#_Toc150337278)

[**6.2 不相交模式数据库的划分方法与生成** 22](#_Toc150337279)

[**6.3 三个阶段的成果及实现** 30](#_Toc150337280)

[**7. 性能对比分析** 33](#_Toc150337281)

[**7.1 A\*与IDA\*性能比较** 33](#_Toc150337282)

[**7.2 各类启发式函数的性能分析** 37](#_Toc150337283)

[**8. 解路径的可视化** 38](#_Toc150337284)

[**9. 总结及展望** 42](#_Toc150337285)

[参考文献 45](#_Toc150337286)

1. **N-Puzzle问题**
   1. **问题描述**

N-Puzzle问题最早出现在1879年，由Noyes Palmer Chapman创建。最初，这个谜题是作为一种玩具拼图出现的，用来娱乐和锻炼大脑。最著名的版本是"8-Puzzle"，其中玩家必须移动数字块，将它们重新排列成正确的顺序。这个问题引起了人们的兴趣，被广泛研究，并成为计算机科学和人工智能中的经典问题。

N-Puzzle是一个方格拼图，通常是一个 的正方形棋盘。其中一个格子是空白的，其余的格子上有编号为到的数字块。初始状态和目标状态都是在同一个的棋盘上，只是数字块的排列顺序不同。目标状态通常是数字按顺序排列，形成一个从左上角到右下角的递增序列，而初始状态则是一个随机或混乱的排列。玩家的任务是通过将空白格子与相邻的数字块进行交换，重新排列数字块，以最终达到目标状态。

解决N-Puzzle问题需要使用不同的搜索算法，如深度优先搜索、广度优先搜索、A\*搜索，改进的A\*搜索---IDA\*算法等，以找到最短路径或最优解。

我们采用的是A\*搜索，IDA\*算法，然后用线性冲突以及数据库作为启发函数，进一步优化搜索过程。

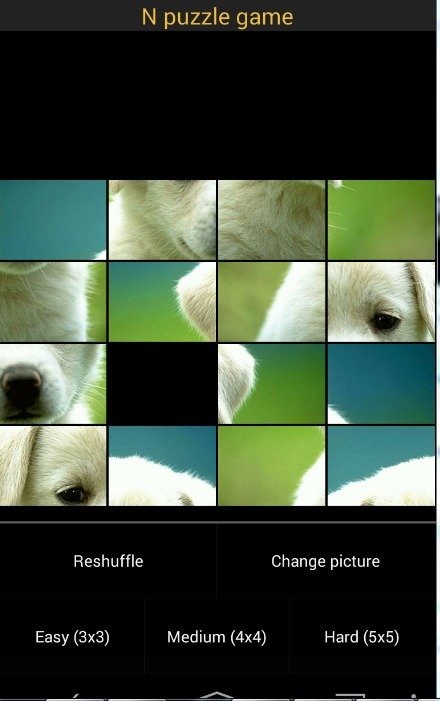


图 N-Puzzle问题

* 1. **可解性分析**

**判断有解：**

一个的棋盘，当时，有且仅有个有解的情况；当时，有且仅有个有解的情况；否则，有且仅有 个有解的情况[1]。

**证明：**

当时，将棋盘上的所有数字格按从左到右、从上到下的顺序填进一个序列 A 中。我们定义一个序列的逆序对数为。我们假设计算逆序的方法为，从一个数向后看，统计比它小的数有多少，然后再求和。记 A 数组的逆序数为。

首先证明几个引理：

**引理1.2.1：若为奇数，则棋盘有解的必要条件为。**

空白格的行为只有四种：上、下、左、右。向左或向右都不会改变 A 数组，因此我们讨论向上和向下。这两个操作是对称的，我们不妨考虑向上。

向上的操作会使 A 数组中某一个格子向右跳 格（在 A 数组中，空白格的上面的格子和空白格右边的格子之间有个格子）。记这个格子为，设跳过的这个格子为，设中有个比小的格子。

现在我们考虑跳完之后整个序列逆序数的变化。首先，不会对除了和 之外的格子在计算的时候产生影响。其次，右移后，原本在中比它小的现在不能算了，因此 要减去。最后，中的格子在计算逆序数的时候，由于多了的存在，要加上，因为有个比小，就有个比大）。因此，的变化为 。

同理，向下造成总逆序数的变化为。因为是奇数，所以和都是偶数，即是偶数。我们知道，最终情况的逆序数为 0 ，是偶数，因此对于所有有解的棋盘都是偶数，证毕。

**引理1.2.2：若是偶数，设空白格在第行，则棋盘有解的必要条件为。**

设从当前状态到最终状态的过程中，空白格向上移动了次，向下移动了次，则。因为是偶数，所以向上或向下移动一次造成的为奇数。设空白格向上移动造成的为，向下移动造成的为，则且，则 ，证毕。

由此，我们可以判断N-Puzzle问题是否有解：设 A 序列的逆序数为，空白格所在的行数为，则棋盘有解当且仅当。

1. **启发式A\*算法综述** 
   1. **A\*算法**
      1. **算法介绍**

A\*（A-Star)算法是一种静态路网中求解最短路最有效的方法。公式表示为：，其中是节点从初始点到目标点的估价函数，是在状态空间中从初始节点到节点的实际代价，是从到目标节点最佳路径的估计代价。

这两部分里，项保持了搜索的宽度优先成分，有利于搜索的完备性，但会影响搜索的效率，容易构造；体现了搜索的启发信息，有利于搜索的效率，但影响搜索的完备性，难以构造。

* + 1. **算法数据结构**

算法需要不断维护两个表：open表和closed表，open表负责将搜索的孩子结点放入并进行排序，所以是一个优先队列。 本项目中使用frontier存储 并使用优先队列的方法完成对这个容器的存取排序，该容器总是将f值最小的结点放到首位。closed表负责记录经过的结点，并在找到目标结点后回溯到目标。

Open表定义：protected final Frontier frontier

Close表定义：protected final Set<State> explored;

图2-1 OPEN表和CLOSED表定义

* + 1. **算法流程**

**步1** 把初始节点放入OPEN表。

**步2** 若OPEN表为空, 则搜索失败, 退出。

**步3** 移出OPEN表中第一个节点放入CLOSED表中, 并冠以顺序编号

**步4** 若目标节点, 则搜索成功, 结束。

**步5** 若不可扩展, 则转步2。

**步6** 扩展,生成一组子节点,并利用评价函数为子节点估值，对这组子节点做如

下处理：

(1) 处理重复状态

(2) 对其余子节点配上指向的返回指针后放入OPEN表中, 并对OPEN表按

值以升序排序, 转步2。

* 1. **A\*算法可采纳性证明**

**A\*算法的性质：**，其中：

从起始节点到节点的路径耗散

可纳启发函数

**假设** 设是任意两个节点，有：，其中为大于0的常数



**几个等式：**其中是初始节点，是目标节点，是到的最佳路径上的节点。

**结论1：**A\*是完备、最优的（即使不丢弃重复访问的节点，以上结论也是成立的）

**算法可纳：**对任一状态空间图，当从初始结点到目标结点有路径存在时，如搜索算法总能在有限步骤内找到一条从初始结点到目标结点的最佳路径，并在此路径上结束，则称该搜索算法是可采纳的。

**证明如下：**

**A\*算法的性质-定理1.1**

**对有限图，如果从初始节点到目标节点有路径存在， 则算法A\*一定成功结束。**

证明：[反证法]设A\*搜索失败，即算法因OPEN表空而退出。

由于有解，不妨令()表示某一解， 从逆向逐个向前检查，直到找到出现在CLOSED表中的 一个结点，如在OPEN表中，则在OPEN表中。OPEN表空之前，必被扩展。如是目标结点，则搜索成功；否则，在OPEN表空之前被加入CLOSED表。这两种情况都与假设矛盾，故对有限图若有解必成功，证毕。

**A\*算法的性质-引理1.1** 

**对无限图，若有从初始节点到目标节点的路径，则A\*不结束时，在OPEN表中即使最小的一个f值也将增到任意大，或有。**

**A\*算法的性质-引理1.2** 

**A\*结束前，OPEN表中必存在(是最佳路径上的节点)**

证明：设一条最佳路径为（）。初始化时，在OPEN表中；A\*算法结束前，该最佳路径从开始连续的一段或直 到出现在搜索图中。若这部分最后的结点为（），则在OPEN表中，且到的最佳路径已找到，即（）已找到。这时，。 由于，故。

**A\*算法的性质-定理1.2/推论1.1**

**定理1.2：对无限图，若从初始结点到目标结点有路 径存在，则A\*也一定成功结束。**

证明：[反证法]假设A\*不结束，由引理1.1有，或OPEN表中最小的一个值也变成无界，这与引理1.2（存在，）的结论矛盾，故A\*会成功结束。



**推论1.1：OPEN表上任一具有的结点，最 终都将被A\*选作为扩展的结点，它最终被扩展。**

证明：A\*一定成功结束，所以在搜索过程中，必定会扩展出，即必在OPEN表中，有；出现在OEPN表中，则算法最后会选择，成功退出，在此之前，所有的结点一定会被扩展。

**A\*算法的性质-定理1.3** 

**若存在初始结点到目标结点的路径，则A\*必能找到最优解结束**。

证明：① 由定理1.1和1.2，A\*一定会找到一个目标结点结束。

②【反证法】假设找到一个目标结点结束了，但到不是最佳路径，即： 根据推论1.1知，OPEN表中任一满足的结点， 最终都将被A\*选为扩展结点。而最佳路径上的任意结点，都满足，所以A\*必扩展完最佳路径上的所有结点后，才选并结束，而这样就找到了最佳路径。此与假设不符，故定理得证。

* 1. **重复扩展节点问题**

在搜索的过程中有可能在不同路径下多次访问同一个节点，因此有可能重复将它加入open表中，而如果启发函数满足单调限制条件，第一次到达的路径就是最优的。扩展某结点时先看该结点是否已经扩展过，如果扩展过则略过。实现的可以线形遍历closed表，但效率不高时间复杂度为，考虑每个结点可以用一个整数标识，用二叉平衡查找树可以得到更好的时间复杂度，使用hash方法的时间复杂度为。

1. **启发式搜索IDA\*算法**
   1. **IDA\*算法基本原理及其可采纳性**

在介绍IDA\*之前，先要了解一下迭代深化深度优先搜索。在计算机科学中，迭代深化搜索（iterative deepening search）或者更确切地说迭代深化深度优先搜索 (iterative deepening depth-first search (IDS or IDDFS)) 是一个状态空间（状态图）搜索策略。在这个搜索策略中，一个具有深度限制的深度优先搜索算法会不断重复地运行，并且同时放宽对于搜索深度的限制，直到找到目标状态。IDDFS 与广度优先算法是等价的，但对内存的使用会少很多；在每一步迭代中，它会按深度优先算法中的顺序，遍历搜索树中的节点，但第一次访问节点的累积顺序实际上是广度优先的。

迭代深化 A\*（IDA\*）是一种图遍历和路径查找方法，它能确定加权图中定义的起点节点与一组目标节点中任意一个节点之间的最短路径。它是一种迭代深化的深度优先搜索，采用了 A\* 搜索算法的思想，即使用启发式函数来评估到达目标的剩余成本。

A\* 的内存限制版本称为 IDA\*。它能执行 A\* 的所有操作，并具有定位最短路径的最佳功能，但占用内存较少。

迭代深化 A Star 使用启发式来选择探索哪些节点以及在哪个深度停止探索，而迭代深化 DFS 则利用简单的深度来决定何时结束当前迭代并继续探索更深的节点。

**要点：**

* **图遍历算法。**
* **使用路径搜索算法，在加权图中找到起点和目标节点之间的最短路径。**
* **迭代深度优先搜索算法的替代算法。**
* **使用启发式函数，这是一种源自 A\* 算法的技术。**
* **它是一种深度优先搜索算法，因此比 A\* 算法占用更少的内存。**
* **IDA\* 是一种可接受的启发式算法，因为它从不高估达到目标的成本。**
* **它专注于搜索最有希望的节点，因此不会在所有地方都达到相同的深度**。
  1. **IDA\*算法求解步骤**

**公式：**

**具体步骤：**

步骤 1：初始化

设置根节点为当前节点，并找出其。

步骤 2：设置阈值

将成本限制设为节点的阈值，即该节点允许进一步探索的最大。

步骤 3：节点扩展

将当前节点扩展到其子节点，并算出其。

步骤 4：剪枝

如果任何节点的大于阈值，则修剪该节点。

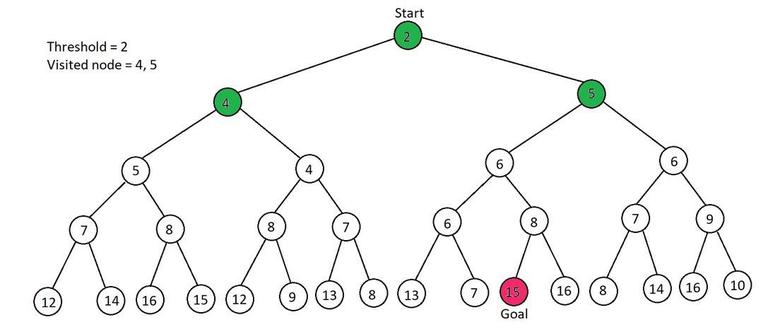
步骤 5：返回路径

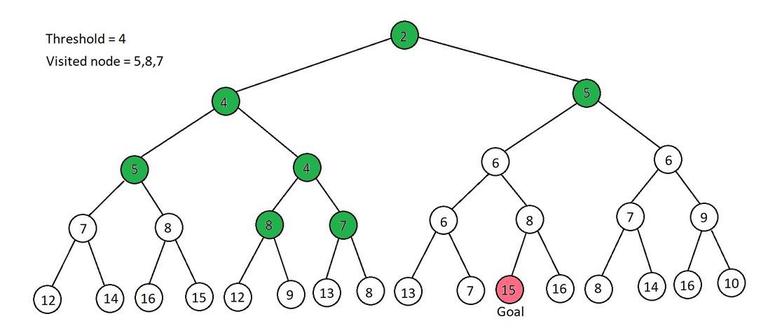
如果找到目标节点，则返回从起始节点到目标节点的路径。

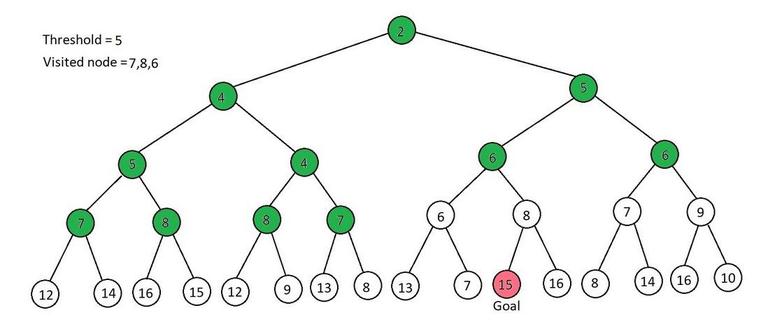
步骤 6：更新阈值

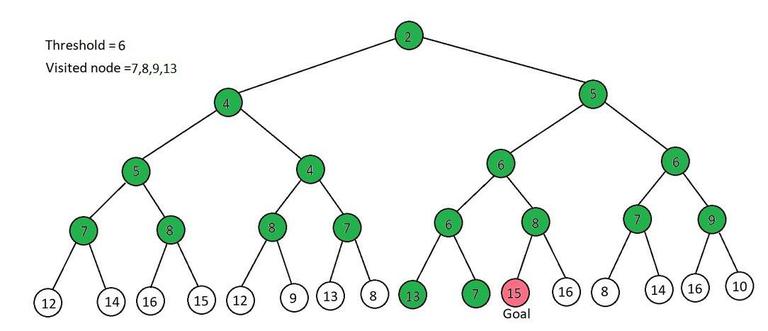
如果未找到目标节点，则重复步骤 2，用已访问节点列表中的最小剪枝值更改阈值。继续进行，直到找到目标节点。

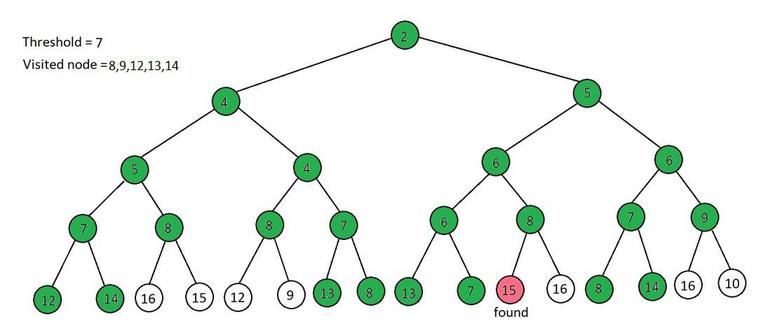
**举例：**











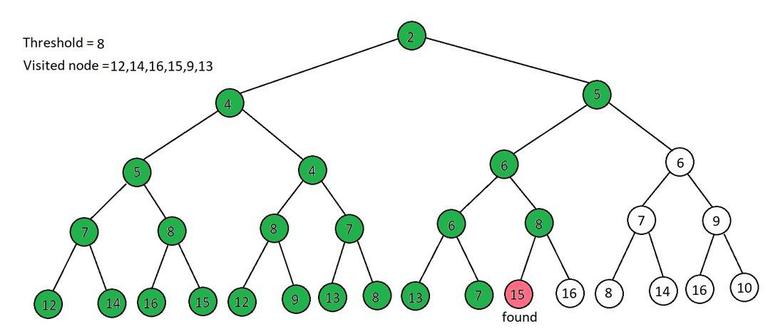


图3-1 IDA\*举例

从图中可以看出，使用IDA\*算法搜索得到的目标路径为 2→5→6→8→15。

* 1. **IDA\*算法描述（伪代码）**

Procedure IDA\_STAR(StartState)

Begin

PathLimit := H(StartState) - 1;

Succes := False;

Repeat

inc(PathLimit);

StartState.g = 0;

Push(OpenStack, StartState);

Repeat

CurrentState := Pop(OpenStack);

If Solution(CurrentState) then

Success = True

Elseif PathLimit >= CurrentState.g + H(CurrentState) then

For each Child(CurrentState) do

Push(OpenStack, Child(CurrentState));

until Success or empty(OpenStack);

until Success or ResourceLimtsReached;

end;

1. **启发函数**
   1. **启发函数的单调性**

单调性是指启发函数具有以下特性：如果从节点A到节点B的实际代价（通常是路径的实际代价）不小于从节点A到节点C的实际代价，并且从节点B到目标节点的启发函数值不小于从节点C到目标节点的启发函数值，那么从节点A经过节点B到目标节点的总代价不小于从节点A经过节点C到目标节点的总代价。这意味着，如果启发函数是单调的，那么它不会低估到目标节点的代价，从而确保搜索算法朝着最优解的方向前进。

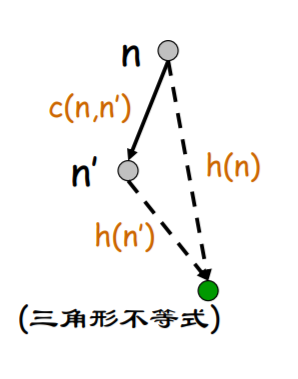


图4-1 三角形不等式

启发式函数h(n)告诉A\*从任意结点n到目标结点的最小代价评估值。选择一个好的启发式函数是重要的。对于一种极端情况，如果h(n)是0，则只有g(n)起作用，此时A演变成Dijkstra算法，这保证能找到最短路径。如果h(n)经常都比从n移动到目标的实际代价小（或者相等），则A保证能找到一条最短路径。h(n)越小，A扩展的结点越多，运行就得越慢。如果h(n)精确地等于从n移动到目标的代价，则A star将会仅仅寻找最佳路径而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，你仍可以在一些特殊情况下让它们精确地相等。只要提供完美的信息，A star会运行得很完美。如果h(n)有时比从n移动到目标的实际代价高，则A star不能保证找到一条最短路径，但它运行得更快。另一种极端情况，如果h(n)比g(n)大很多，则只有h(n)起作用，A star演变成BFS算法。

* 1. **不在将位牌**

"不在将位" 启发式函数的计算方式是数出当前状态中有多少数字块没有在其正确的位置上。换句话说，它计算当前状态与目标状态之间的汉明距离，也就是它们之间的差异之处。这个数字越小，表示离目标状态越近。

这个启发式函数之所以有用，是因为它提供了一个快速的估计，让搜索算法能够优先考虑将数字块移到正确位置的移动。虽然 "不在将位" 启发式函数相对简单，但在解决N-数码问题时通常能够提供不错的性能，特别是对于小规模问题。

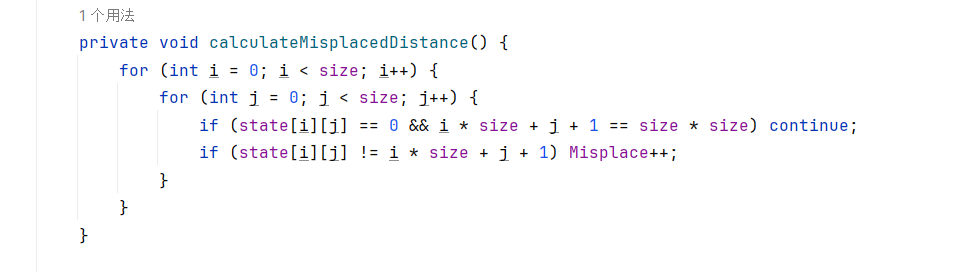


图4-2不在将位启发函数

* 1. **曼哈顿距离**

假设有两个点A和B，它们的坐标分别为和。那么点A到点B的曼哈顿距离是：

其中 "|" 表示绝对值。这个公式计算了点A到点B的横向和纵向位移的绝对值之和。

曼哈顿距离常用于计算在网格上移动的问题，如拼图游戏、路径规划、城市布局等。在拼图游戏中，它通常用于估计一个状态与目标状态之间的距离，作为启发式函数，帮助搜索算法（如A\*搜索）指导搜索方向，以寻找最优解。

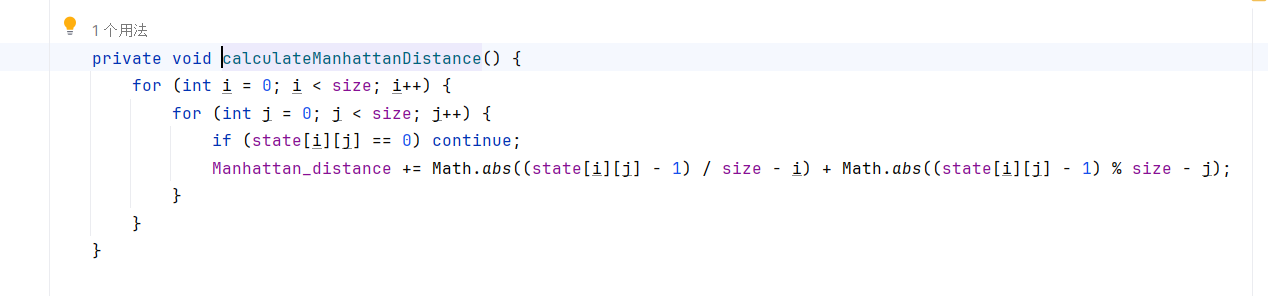


图4-3 曼哈顿距离启发函数

* 1. **线性冲突**

线性冲突发生在两个拼图块A和B在同一行或同一列上，并且它们都不在正确的顺序中。这意味着拼图块A在拼图块B的前面，但它们实际上在相反的顺序中。为了解决线性冲突，至少一个拼图块必须移动到正确的位置。每次拼图块的移动都会增加2的冲突数，因为它会解决两个拼图块的冲突。线性冲突启发式函数估计的总冲突数是所有线性冲突的数量的总和。

曼哈顿距离算的是每个滑块自己本身移动的花费，线性冲突计算的是每行和每列两个滑块的冲突数，因此可以叠加。因此启发函数应该是曼哈顿距离+总冲突数。

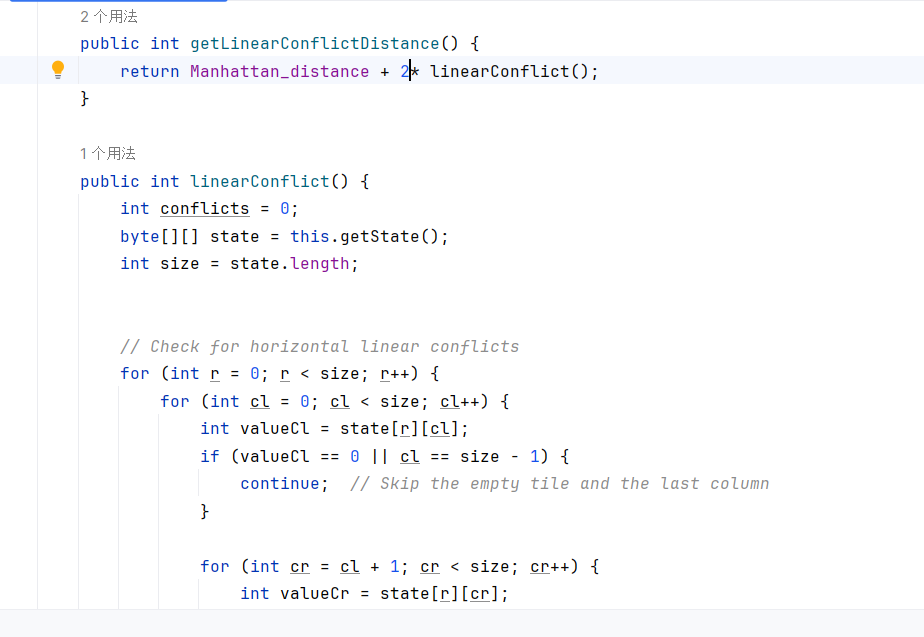


图4-4 线性冲突启发函数

* 1. **不相交模式数据库**
     1. **简述**

为了构建滑动拼图的不相交模式数据库[2,3,4]，首先我们将滑块划分为不相交的组，每一个滑块都属于一个唯一的组中。然后，对于每个组，我们预先计算出将组中所有滑块移至目标位置所需最小移动次数的表。每个组都有一个这样的表，即一个不相交模式数据库。接着，给定搜索中的特定状态，对于每组滑块，我们使用这些滑块的位置来计算对应表的索引，通过检索对应表，找到该组中滑块所需的移动次数，然后将每个组的值相加，以计算给定状态的整体启发式。该值将至少和曼哈顿距离一样大，并且通常更大，因为它考虑了同一组中的滑块之间的交互。

上述不相交模式数据库和非加法数据库[5]之间的主要区别在于，非加法数据库包括解决模式滑块所需的所有移动，包括不在模式集合中的瓦片的移动。给定两个这样的数据库，即使它们的滑块之间没有重叠，我们也只能将两个值中的最大值作为一个可纳的启发式，因为在一个数据库中已计数的移动可能会移动另一个数据库中的滑块，因此这些移动将被计算两次。在不相交模式数据库中，我们只计算组中滑块的移动。第二个区别是不相交模式数据库不考虑空白位置，从而减小了它们的大小。对于所有可能的空白位置，不相交模式数据库包含解决一组瓷砖所需的最少移动次数。

曼哈顿距离是不相交模式数据库的一个简单的特殊情况。对于N-Puzzle问题，如果将N个滑块分为N个组，其中每组仅包含一个滑块，此时从当前状态移动到下一个状态的花费恰好为曼哈顿距离。如果将N各滑块都划分到一个组里，则对于每个状态，从当前状态移动到下一个状态的花费为真实的花费，即每次搜索都能找到正确的路径，但是这样的总状态数会很大。对于15-Puzzle问题，在该划分方法下的总状态数为，数量级为 ，总状态数过大。不难发现，如果我们将N个滑块分成多组，分组数越多，总状态数会越少，但是状态转移的花费也会相应减少，导致算法中启发函数越小，从而使得扩展结点增多，搜索路径增大，最后导致搜索时间成本增加。因此可以考虑将15-Puzzle问题分成两组7-8模式，或分成三组6-6-3模式或5-5-5模式问题。由于程序运行有内存限制，两组分区所需的空间开销是三组分区的100倍以上，超出了个人笔记本电脑运行内存，所以我们在实验中选择了6-6-3模式划分，该模式中6区块的状态数为，数量级为。

* + 1. **具体实现**

**滑块划分模式：**

考虑到个人笔记本电脑的运行内存限制，我们将15-Puzzle问题划分为三组6-6-3模式，即将15个滑块划分为6-6-3三个区块，划分方法如下图所示。

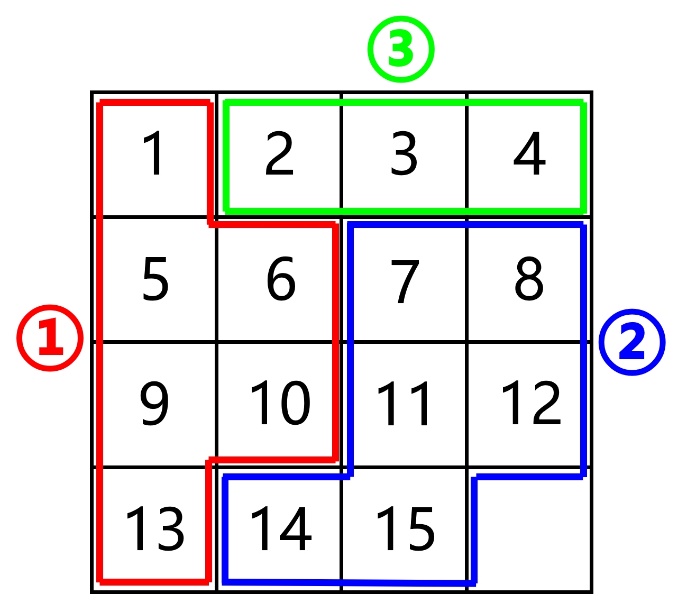


图4-5 6-6-3模式划分示意图

**对单个区块进行搜索：**

当只考虑一个区块时，由于区块之外的滑块都不影响区块内滑块的移动，因此区块以外的滑块除了空格之外都视为-1。下图所示为区块1的模式图。

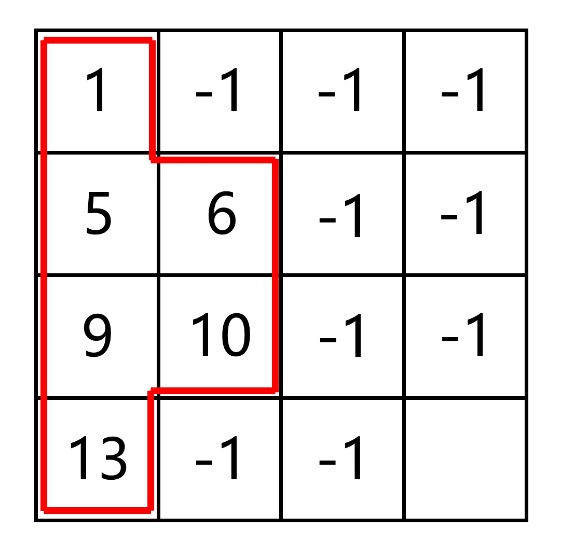


图4-6 区块1模式图

当只考虑一个区块时，对空格进行Breath First Search，当空格与-1交换时，花费不会增加，而与区块内的滑块交换时，花费增加1。下图所示为区块1的搜索图。

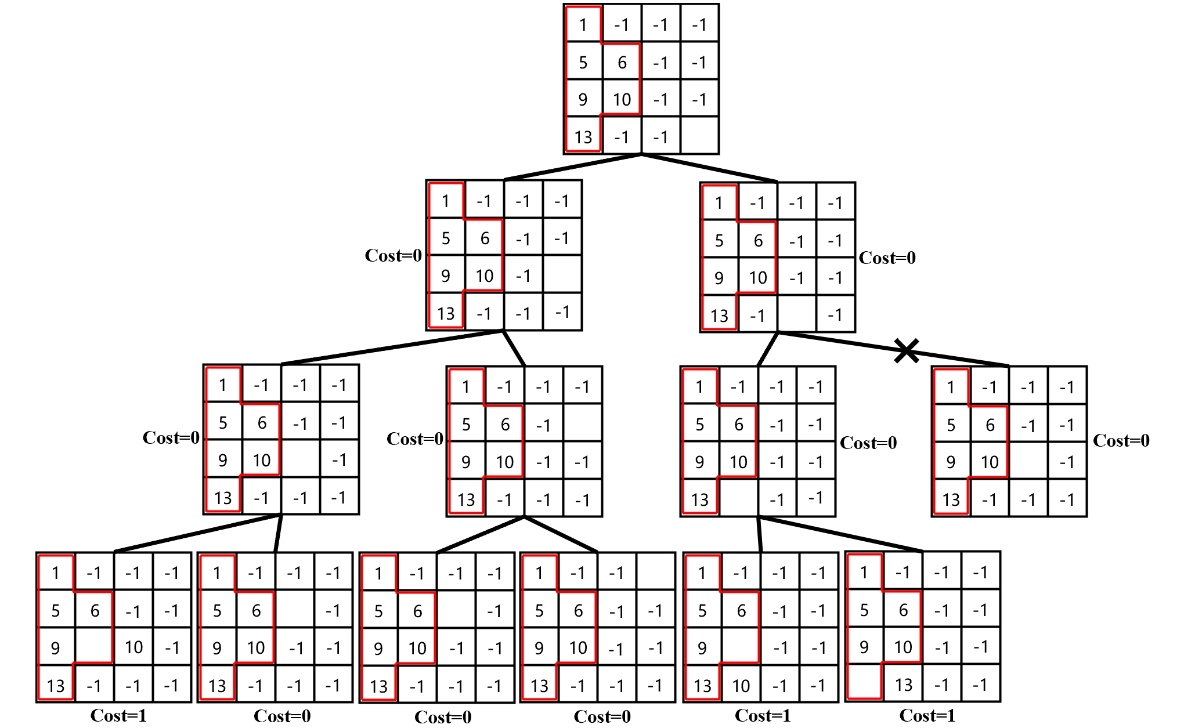


图4-7 区块1搜索图

1. **两种算法的优缺点**
   1. **A\*算法**
      1. **A\*算法的优点**

* 最佳解决方案：A\*算法保证找到最优解决方案，即具有最低代价的路径，这在很多问题中非常重要。
* 可自定义启发式函数：A\*算法的性能高度依赖于启发式函数的选择，您可以根据问题的特点设计不同的启发式函数，以提高算法的效率。
* 剪枝：A\*算法使用启发式函数来引导搜索，因此能够有效地剪枝掉不太可能通往最优解的分支，从而提高搜索效率。
* 可优化空间复杂度：使用适当的数据结构，如二进制堆，可以有效控制A\*算法的内存使用，因此可以用于解决大规模问题。
  + 1. **A\*算法的缺点**
* 启发式函数设计难度：寻找一个有效的启发式函数可能需要问题特定的领域知识，这可能会很困难，尤其是对于复杂的问题。
* 可能陷入局部最优解：如果启发式函数不足够准确，A\*算法有时可能会陷入局部最优解，而不是找到全局最优解。
* 时间复杂度高：A\*算法的时间复杂度与问题的复杂性成正比，对于某些问题，特别是NP难问题，可能需要大量时间来找到解决方案。
* 空间复杂度高：A\*算法可能需要大量内存来存储开放列表，特别是在处理大规模问题时。
* 不适用于不可行路径规划：如果没有可行的路径到达目标状态，A\*算法将搜索整个状态空间，可能会导致无穷循环。
  1. **IDA\*算法**
     1. **IDA\*算法的优点**
* IDA\* 保证找到最佳解决方案（如果存在）。
* IDA\* 避免了传统深度优先搜索的指数级时间复杂度。通过使用“迭代深化”方法，搜索深度逐渐增加。
* 与 A\* 算法相比，IDA\* 使用的内存量有限，因为它使用深度优先搜索。
* IDA\*是一种可接受的启发式方法，它从不高估实现目标的成本。
* 它可以有效地处理大量状态和大型分支因子。
  + 1. **IDA\*算法的缺点**
* 一次又一次地浏览访问过的节点。它不会跟踪访问的节点。
* IDA\* 可能比其他搜索算法（如 A\* 或广度优先搜索）获得解决方案的速度更慢，因为它会一次又一次地探索和重复探索节点。
* 它比 A\* 算法需要更多的时间和功率。

1. **N-Puzzle求解问题的实现**
   1. **文件结构与前两阶段算法实现**
      1. **文件结构**

主要在stu文件夹中完成工作，文件目录如下。

├─algs4

│ └─util

│ StdRandom.java

│ Stopwatch.java

│ StopwatchCPU.java

│

├─core

│ ├─problem

│ │ Action.java

│ │ Problem.java

│ │ ProblemType.java

│ │ State.java

│ │

│ ├─runner

│ │ EngineFeeder.java

│ │ SearchTester.java

│ │

│ └─solver

│ ├─algorithm

│ │ │ Searcher.java

│ │ │

│ │ ├─heuristic

│ │ │ HeuristicType.java

│ │ │ Predictor.java

│ │ │

│ │ └─searcher

│ │ AbstractSearcher.java

│ │ BestFirstSearcher.java

│ │

│ └─queue

│ EvaluationType.java

│ Frontier.java

│ Node.java

│

└─stud

└─g01

├─DB 不相交模式数据库实现

│ Block.java

│ PatternDataBase.java 生成不相交模式数据库

│ State.java

│

├─problem

│ └─npuzzle

│ Direction.java

│ Move.java

│ NPuzzleProblem.java Npuzzle问题，继承Problem

│ NPuzzleState.java Npuzzle问题状态，继承State

│ PuzzleBoard.java

│ Zobrist.java 实现Zobrist哈希

│

├─queue

│ aFrontier.java Npuzzle问题队列

│ PqFrontier.java

│

├─runner

│ PuzzleFeeder.java 读入问题构造Npuzzle问题

│

└─solver

IDAStar.java 实现IDA\*

* + 1. **NpuzzleState.java**

**类对象变量：**

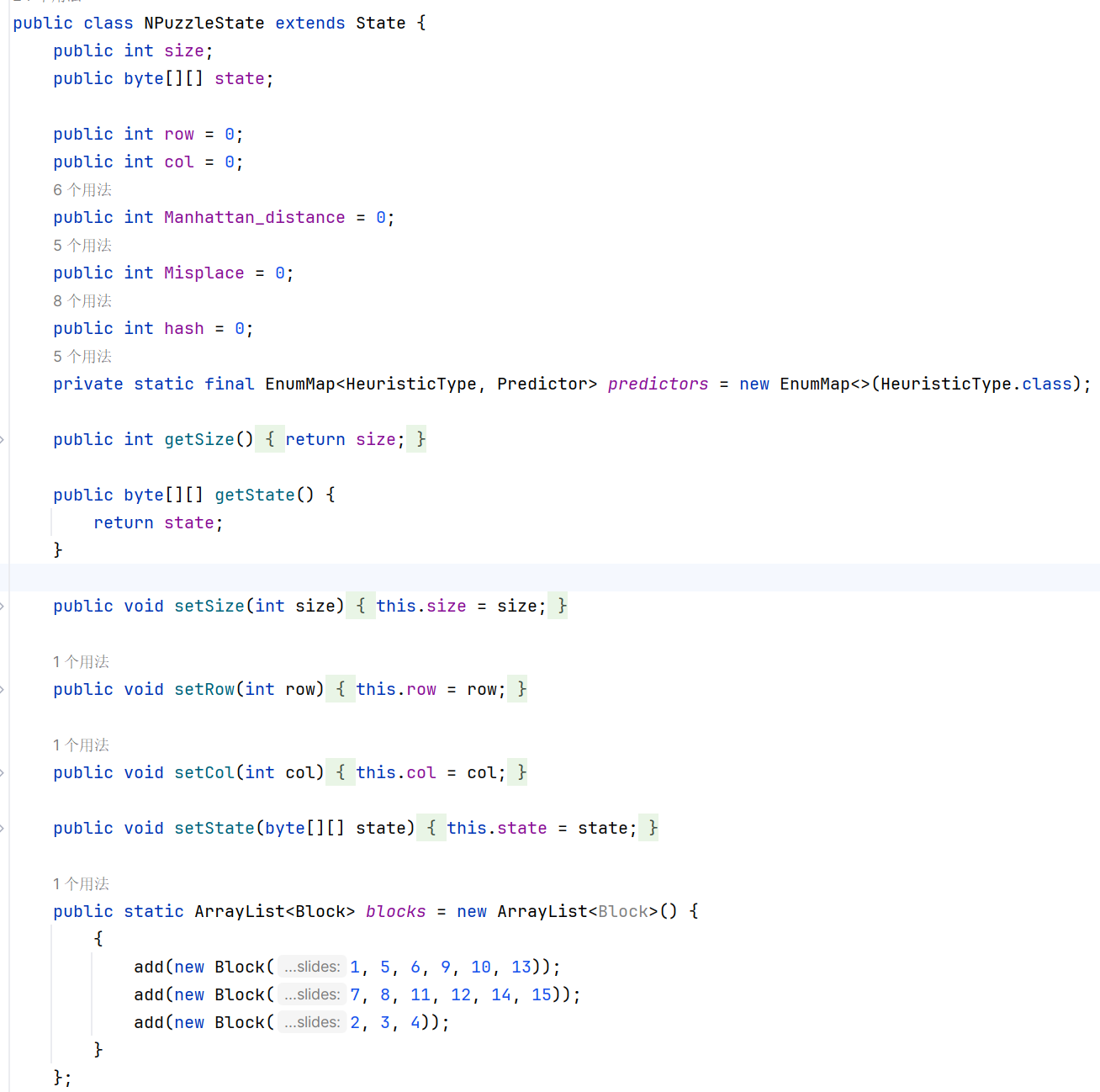


图6-1类对象变量

**构造函数：**

以格局构建NpuzzleState对象，fillStateBoard函数用于把形参传入成员变量变为一种状态，flag==True时，代表此状态为初始状态，要将初始状态初始化，调用initialDistancesAndHash实现。

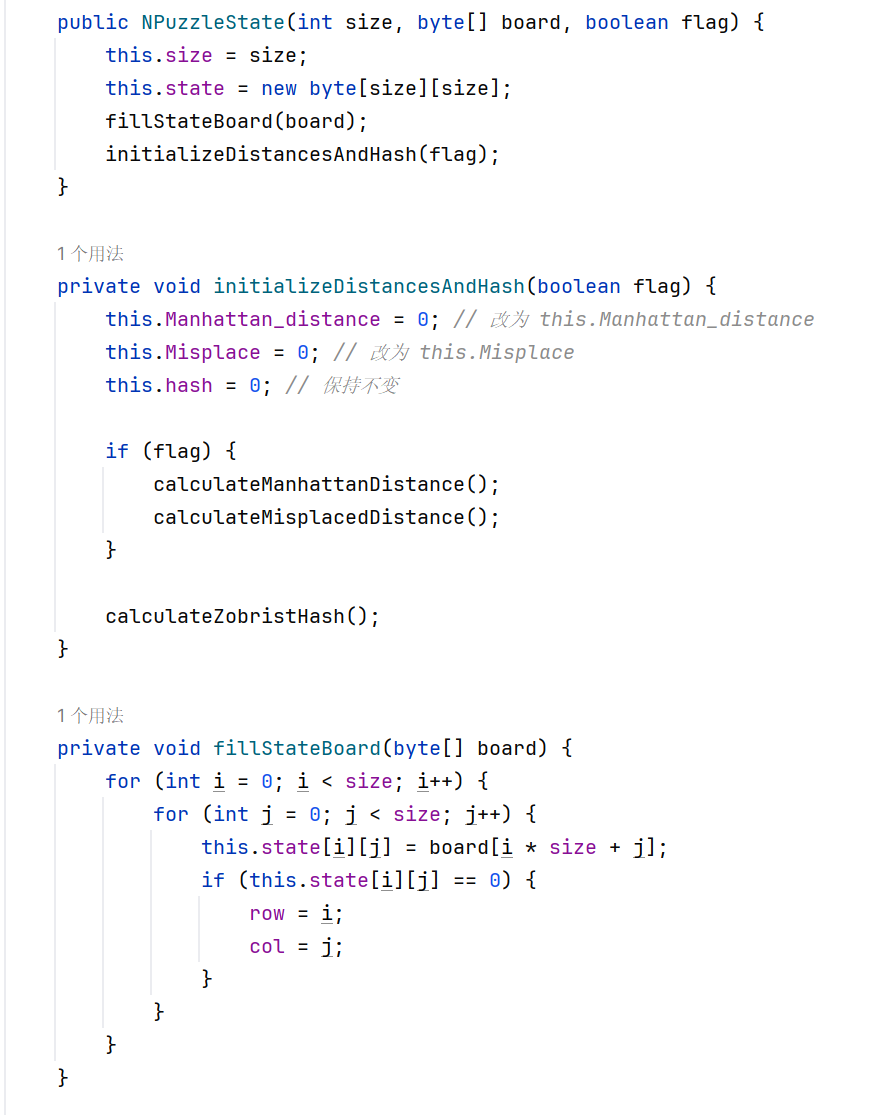


图6-2构造函数

**启发函数：**

存入静态代码块中，不需要创建对象实例就可以调用。

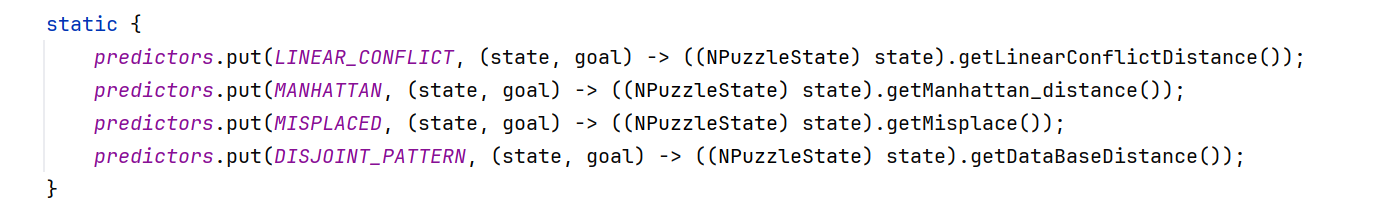


图6-3 四种启发式函数

**计算曼哈顿距离：**

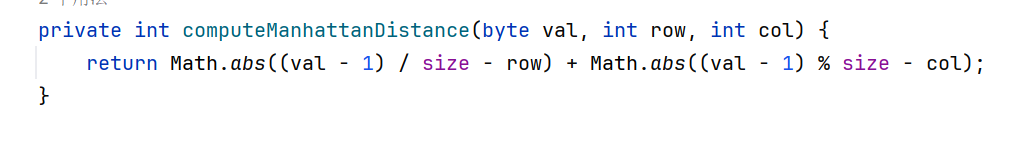


图6-4 曼哈顿距离

**计算错位将牌数：**

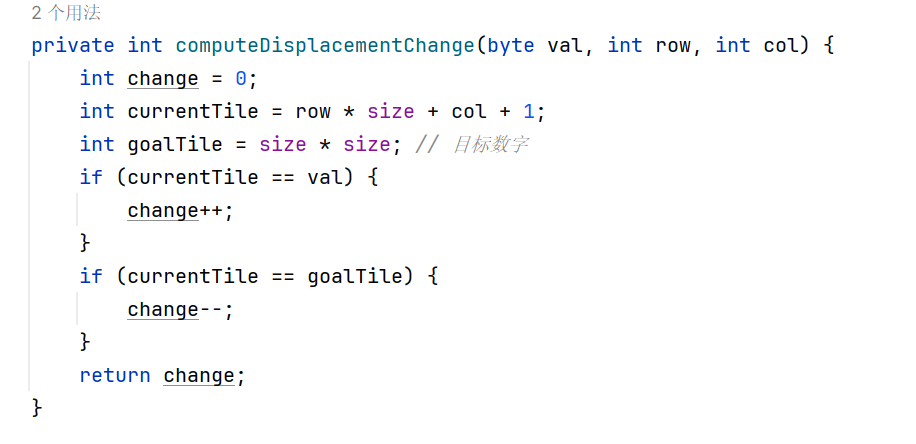
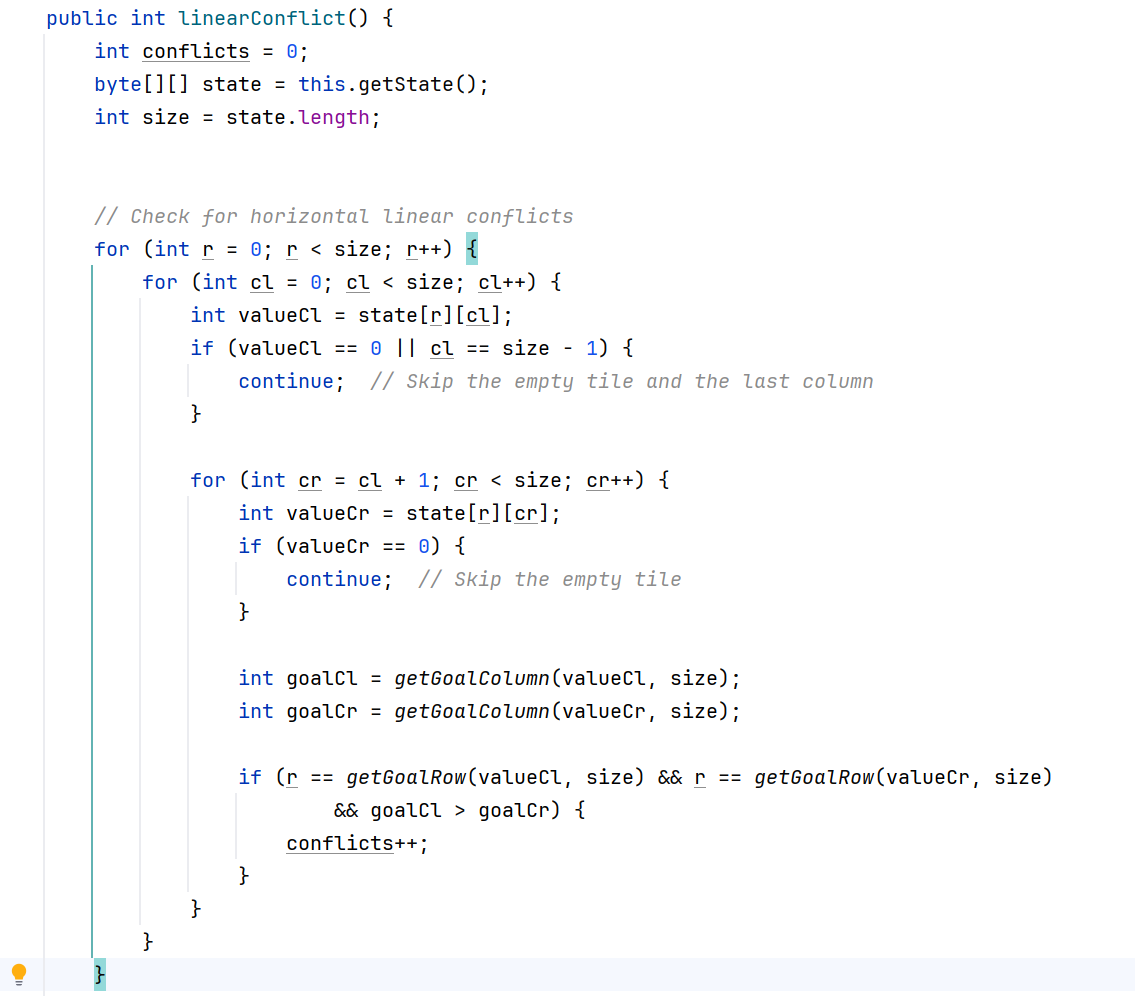


图6-5 错位将牌数

**计算线性冲突：**



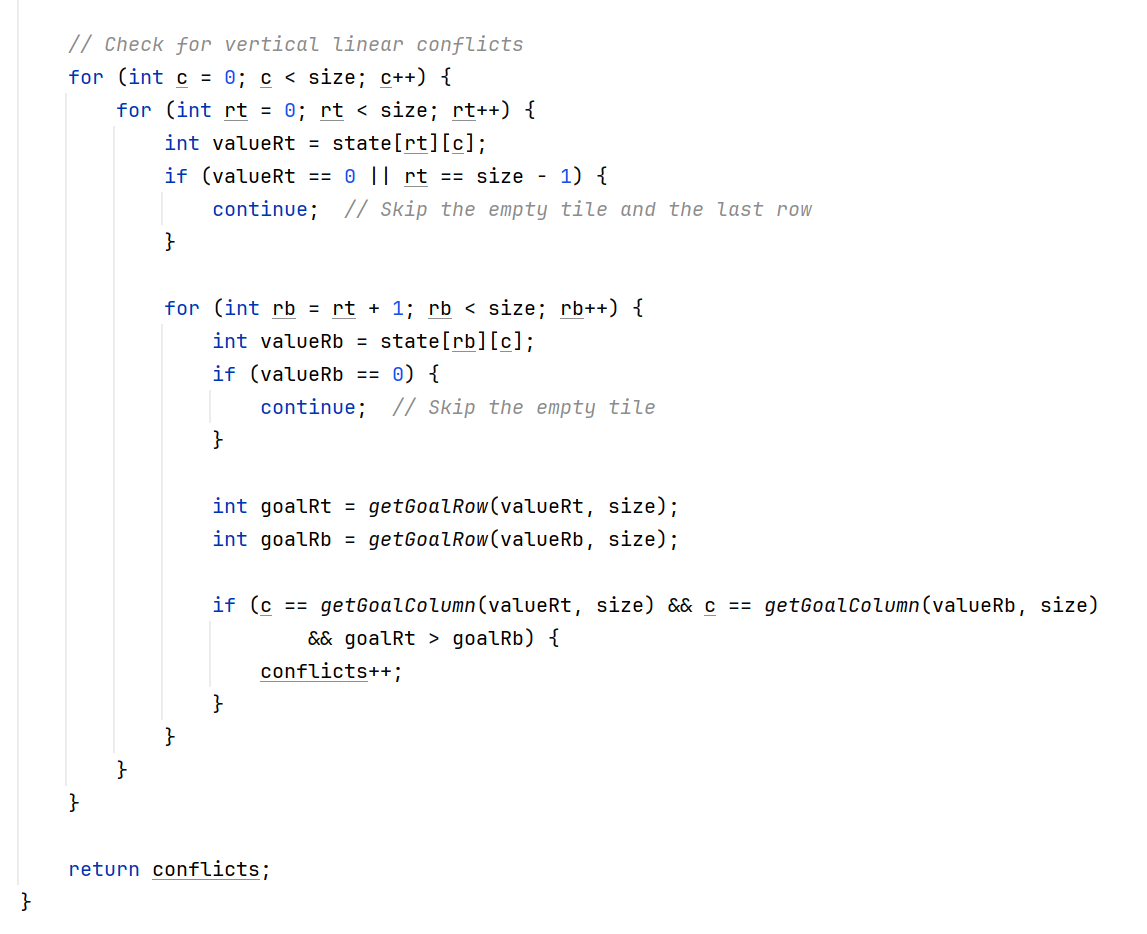


图6-6 线性冲突

**计算下一步状态：**



图6-7计算下一步状态

* + 1. **NpuzzleProblem.java**

重写sovable和applicable，cost每次都为1。判断可解性时，要计算每个状态的逆序数，通过getReverseCount实现。



图6-8判断可解性

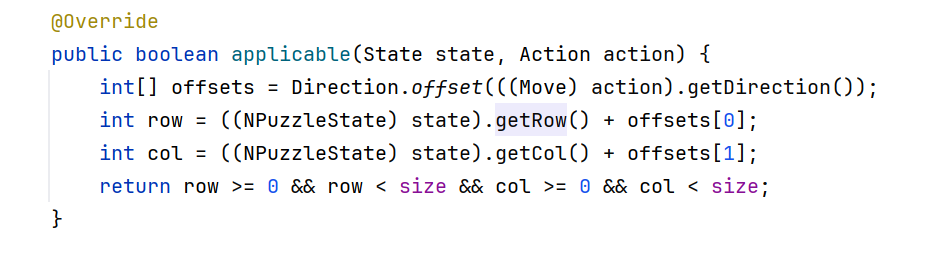


图6-9判断是否可走

* + 1. **aFrontier.java**

open表为PriorityQueue，closed表为HashMap。



图6-10 open表



图6-11 closed表

* + 1. **Zobrist.java**

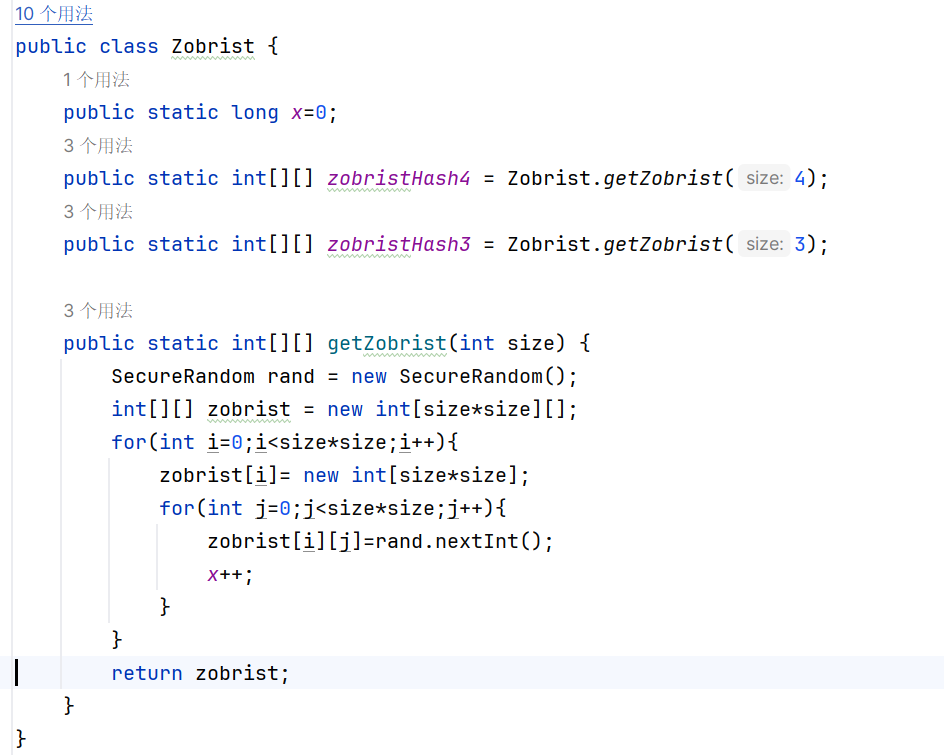


图6-12 Zobrist哈希

* + 1. **PuzzleFeeder.java**

构建N-puzzle问题。

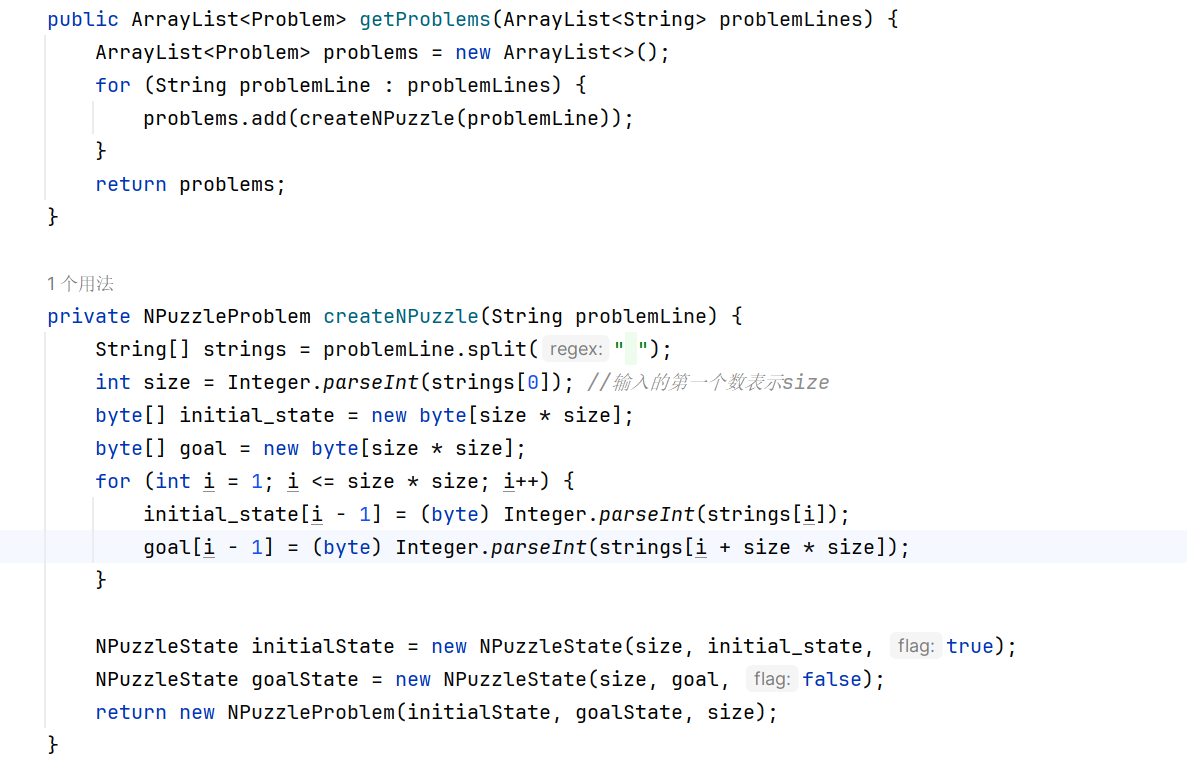


图6-13 PuzzleFeeder

* + 1. **IDAStar.java**



图6-14 IDA\*

* + 1. **Direction.java**

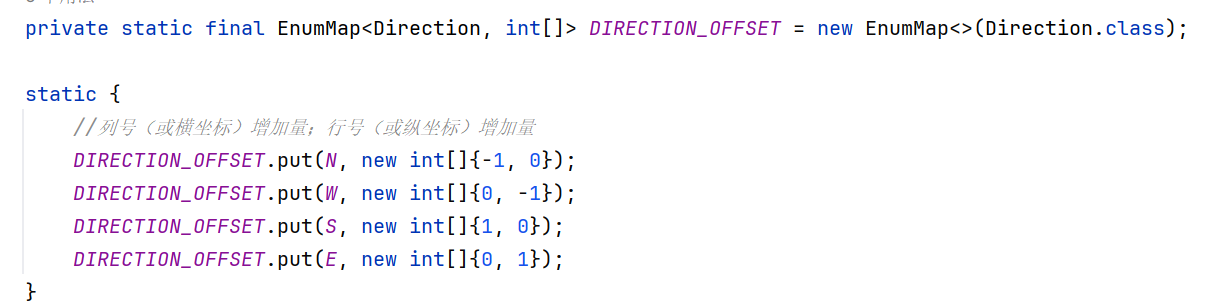


图6-15 4个方向

* 1. **不相交模式数据库的划分方法与生成**
     1. **Block类：划分区块**

首先实现Block类对区块进行封装，Block类包含以下成员变量，各变量的具体用途如下图所示。

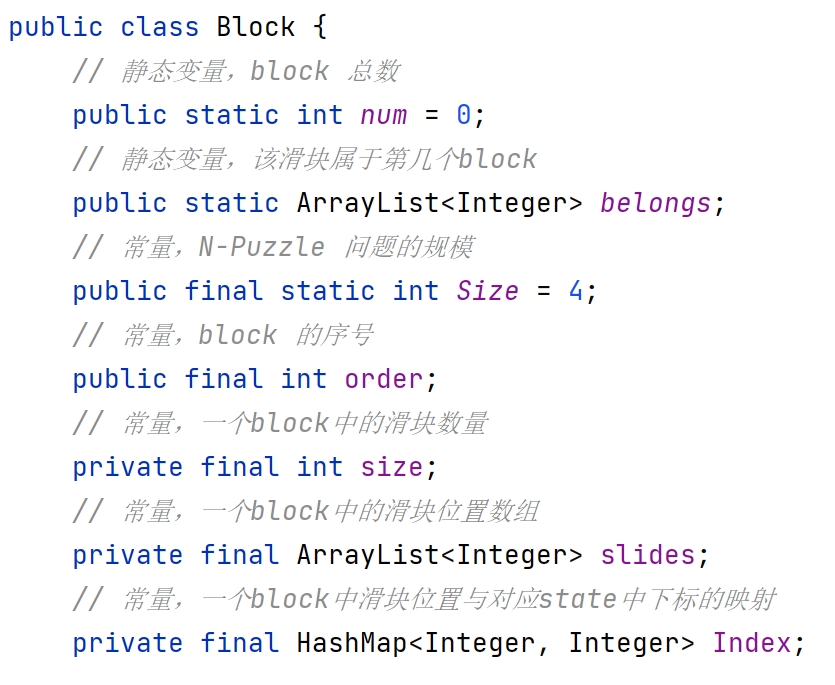


图6-16 Block类包含的成员变量

Block类的构造函数实现了通过输入的区块划分模式即区块中滑块的位置确定一个区块，输入是一组可变长度的int型数字，slides用于记录区块中的滑块位置，Index用于建立滑块位置与state中下标之间的映射，belongs用于建立滑块位置与区块之间的映射，方便后续通过滑块位置查找区块。



图6-17 Block类构造函数的具体实现

我们在NPuzzleState类中对Block类进行了构造，输入6-6-3区块划分模式，对15-Puzzle问题进行划分。

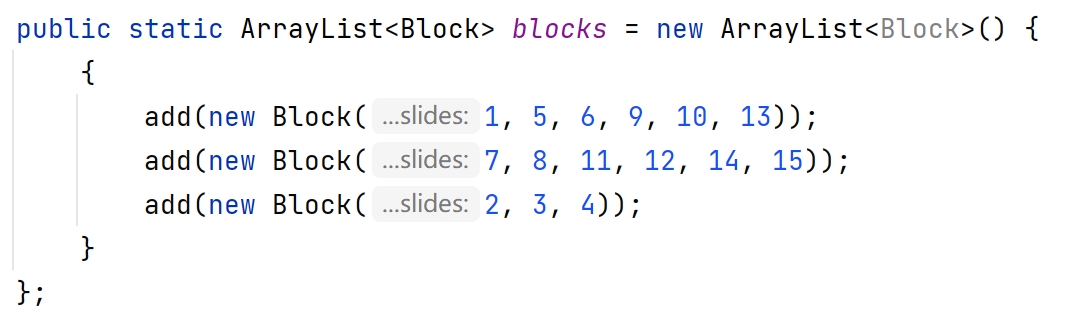


图6-18 构造6-6-3区块划分模式

* + 1. **State类：状态设置、编码和译码**

由于每个滑块的位置可以使用从1到16的编号表示，所以只需要4比特即可表示一个滑块的位置。在搜索的同时还需要记录空格的位置，所以表示一个包含6个滑块的区块的一个状态只需要4×(6+1)=28比特，表示一个包含3个滑块的区块的一个状态只需要4×(3+1)=16比特，因此一个区块的一个状态只需要一个int型变量即可存储。

因此，我们定义了State类封装区块的状态，一个state代表一个区块当前的一个状态。该类的成员变量定义和作用如下。

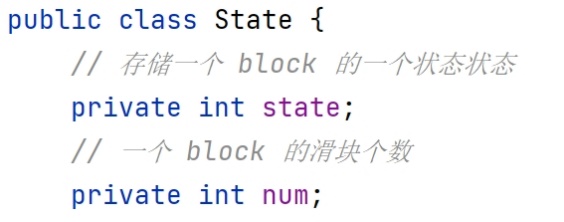


图6-19 State类包含的成员变量

我们在State类中定义了压缩存储状态的方法。slidesToState方法用于将一个block的一个状态即一个滑块位置序列slides进行编码，将其压缩为一个int型变量进行存储。stateToSlide方法用于获取一个state中index位置上的滑块的位置信息。setStateSlide方法用于修改一个state中index位置上的滑块的位置信息。

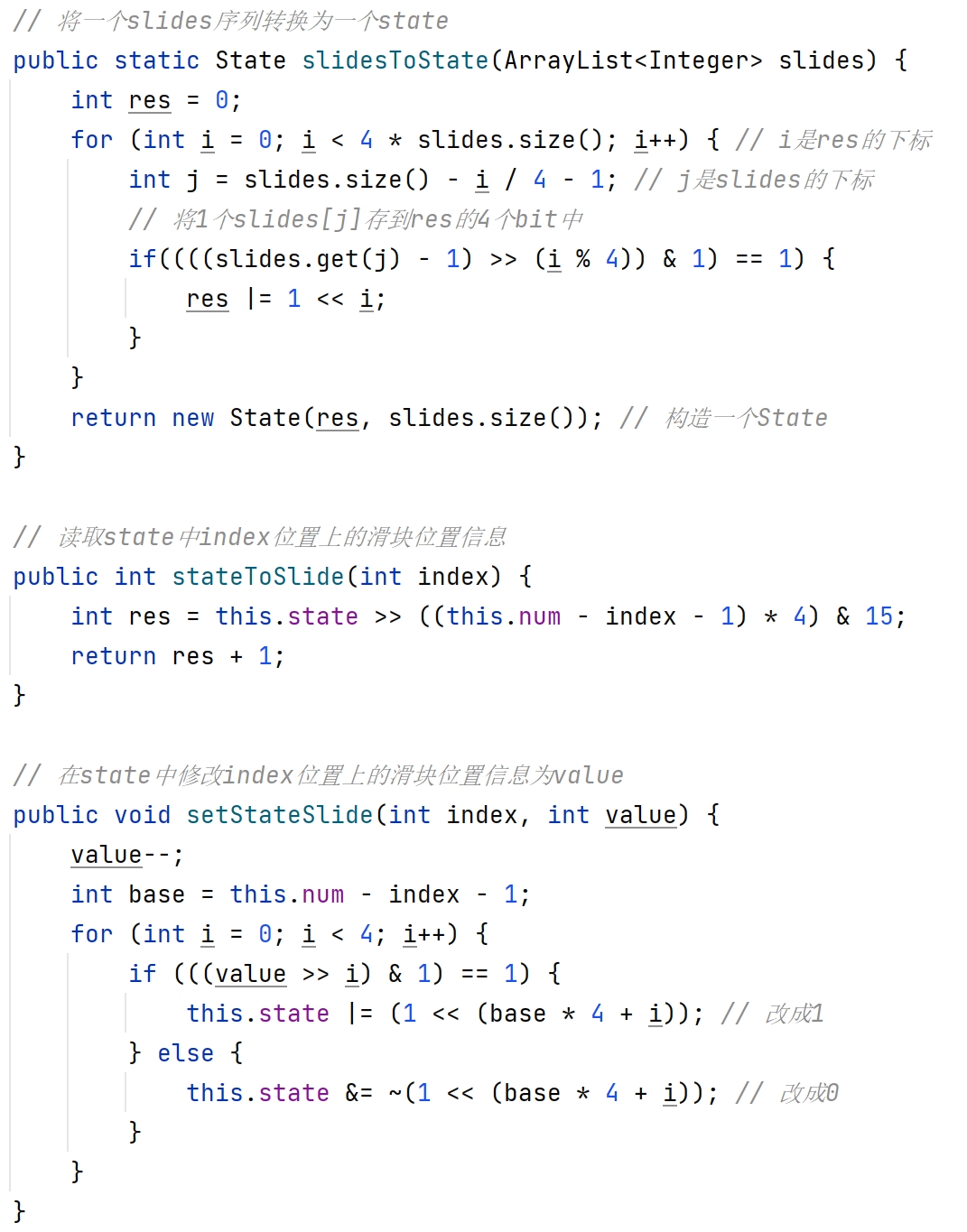
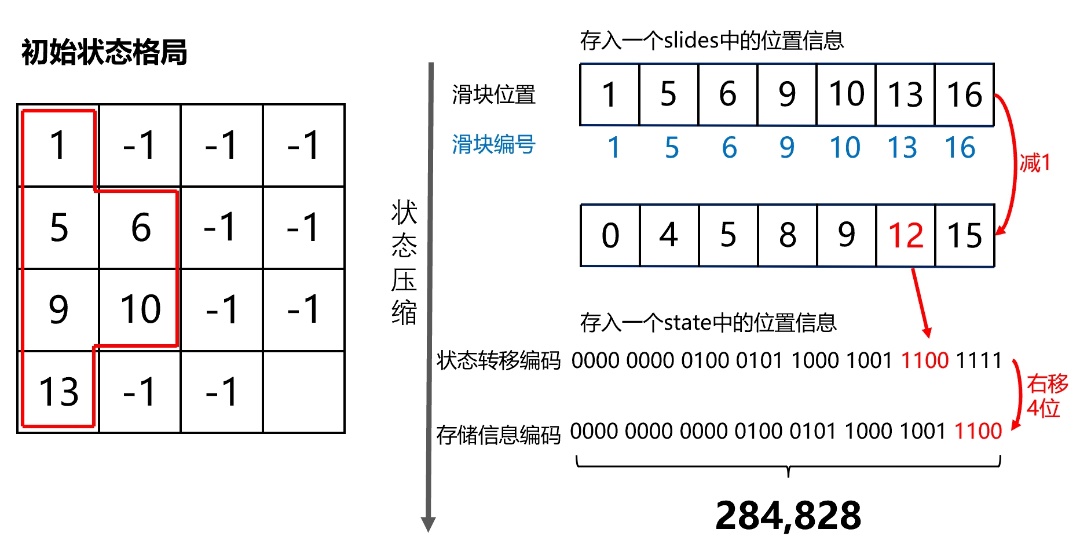


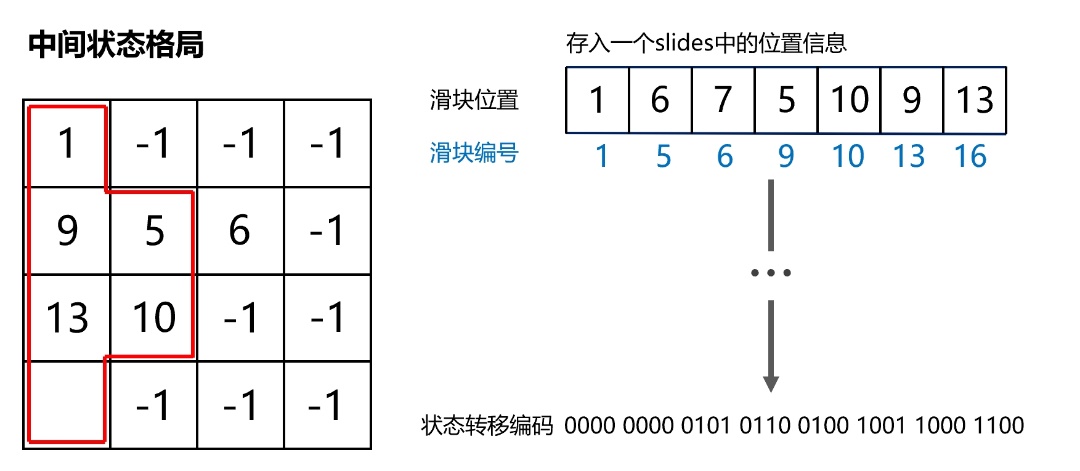
图6-20 State类中压缩存储状态的方法

需要注意的是，我们在搜索时记录的state是需要添加空格位置信息的，而存入数据库中的是不含空格位置信息的state。

当只考虑一个区块时，我们对一个滑块位置信息序列slides进行状态压缩，将其存储为一个state，其过程如下图所示。状态转移编码是搜索时使用的编码，是包括空格位置信息的；而存储信息编码是存入数据库中的编码，是不包括空格位置信息的。



(a) 初始状态格局



(b) 中间状态格局

图6-21 状态压缩的过程

* + 1. **PatternDataBase类：模式数据库的创建和使用**

模式数据库使用PatternDataBase类进行封装，其中包括以下成员变量。stateCost使用TreeMap红黑树的数据结构，记录一个状态的距离花费。Table用于记录不相交模式数据库中的数据。

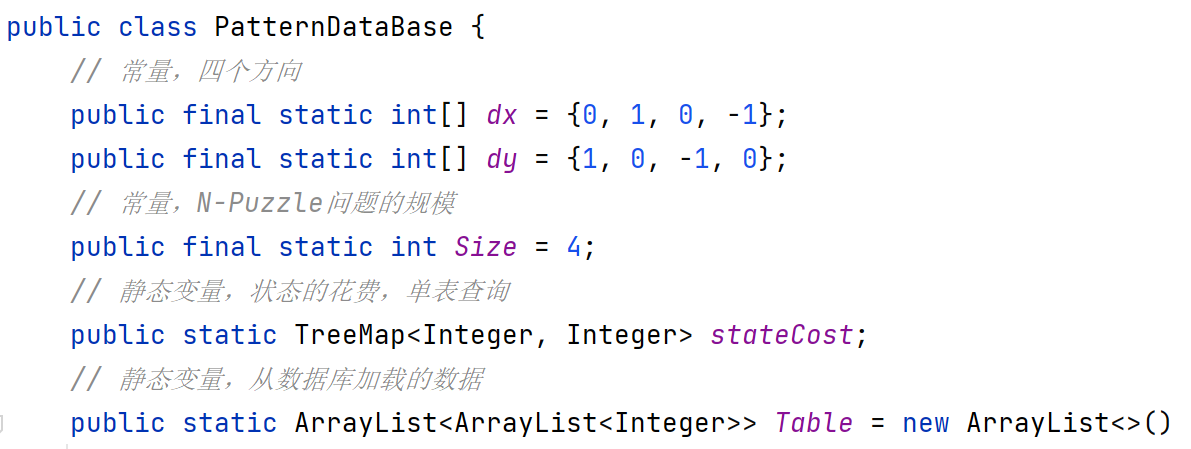


图6-22 PatternDataBase类包含的成员变量

要创建模式数据库，我们需要对每一个区块进行Breath First Search，从而获取每个区块中不同状态的距离花费，可将其存入文件中，便于后续读取。

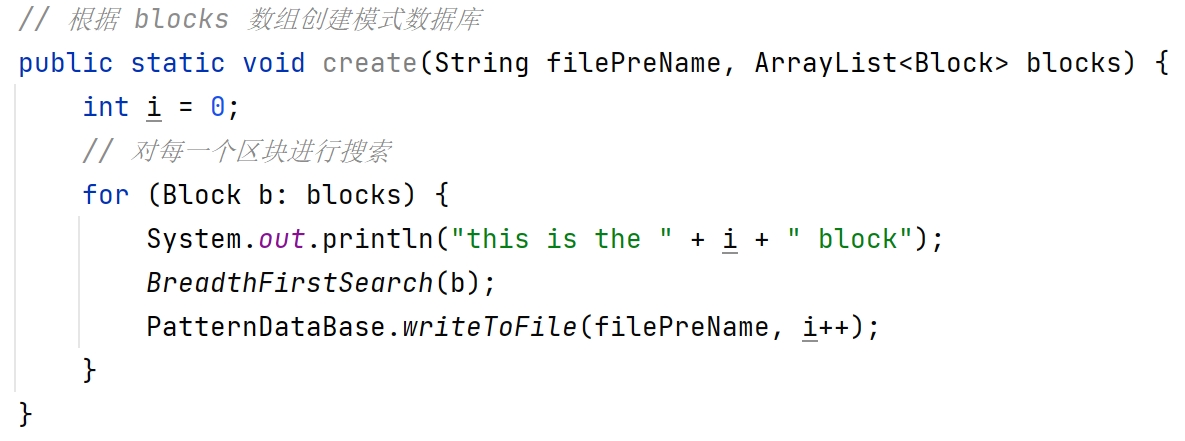
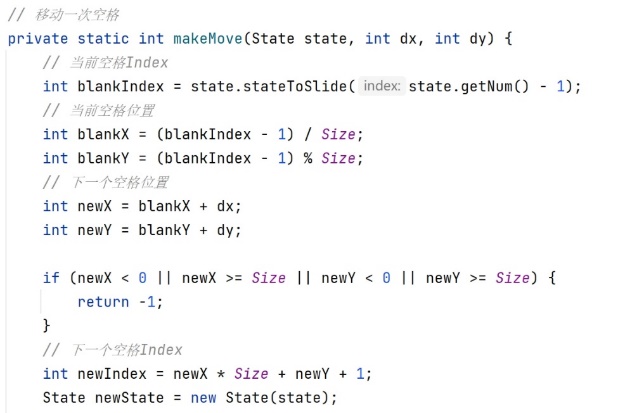


图6-23 创建模式数据库

makeMove方法用于移动一次空格，即将空格与其上下左右四个方向的滑块进行交换，将当前状态更新为下一个新状态，计算新的距离花费，并进行存储，同时需要避免记录重复状态。



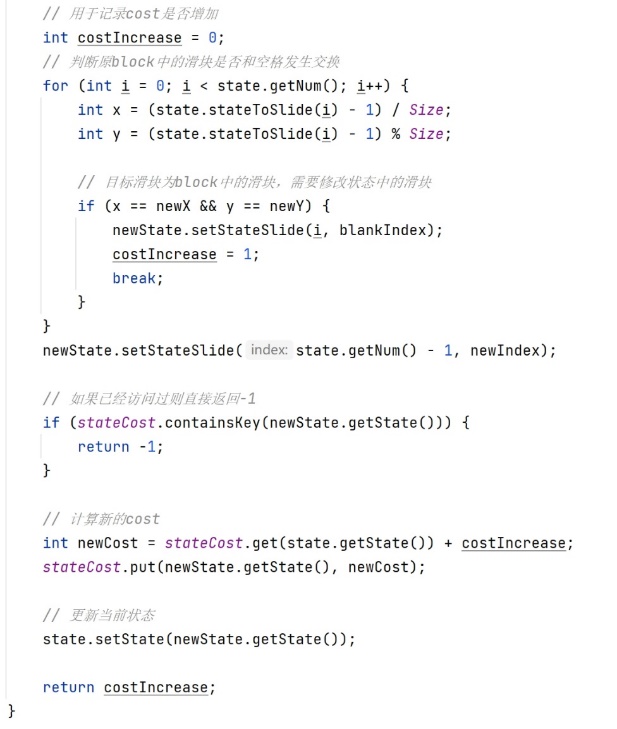


图6-24 移动空格

BreadthFirstSearch方法是对每一个区块进行Breath First Search，计算从目标状态到当前状态的估价函数。在搜索过程中，空格和区块相关滑块进行交换时花费增1，和区块不相关滑块进行交换时花费不增。因此使用双端队列Deque来进行存储，花费为0时插入队首，花费为1时插入队尾，保证队列的有序性。每次从队列中取队首元素，然后将空格进行四个方向的移动，由于是Breath First Search，所以每个状态第一次搜索到时的花费一定是最小的，再次搜索到时的花费一定不会减少，所以使用stateCost记录每个状态第一次搜索时的花费。如果该状态已被访问过，则停止继续搜索。

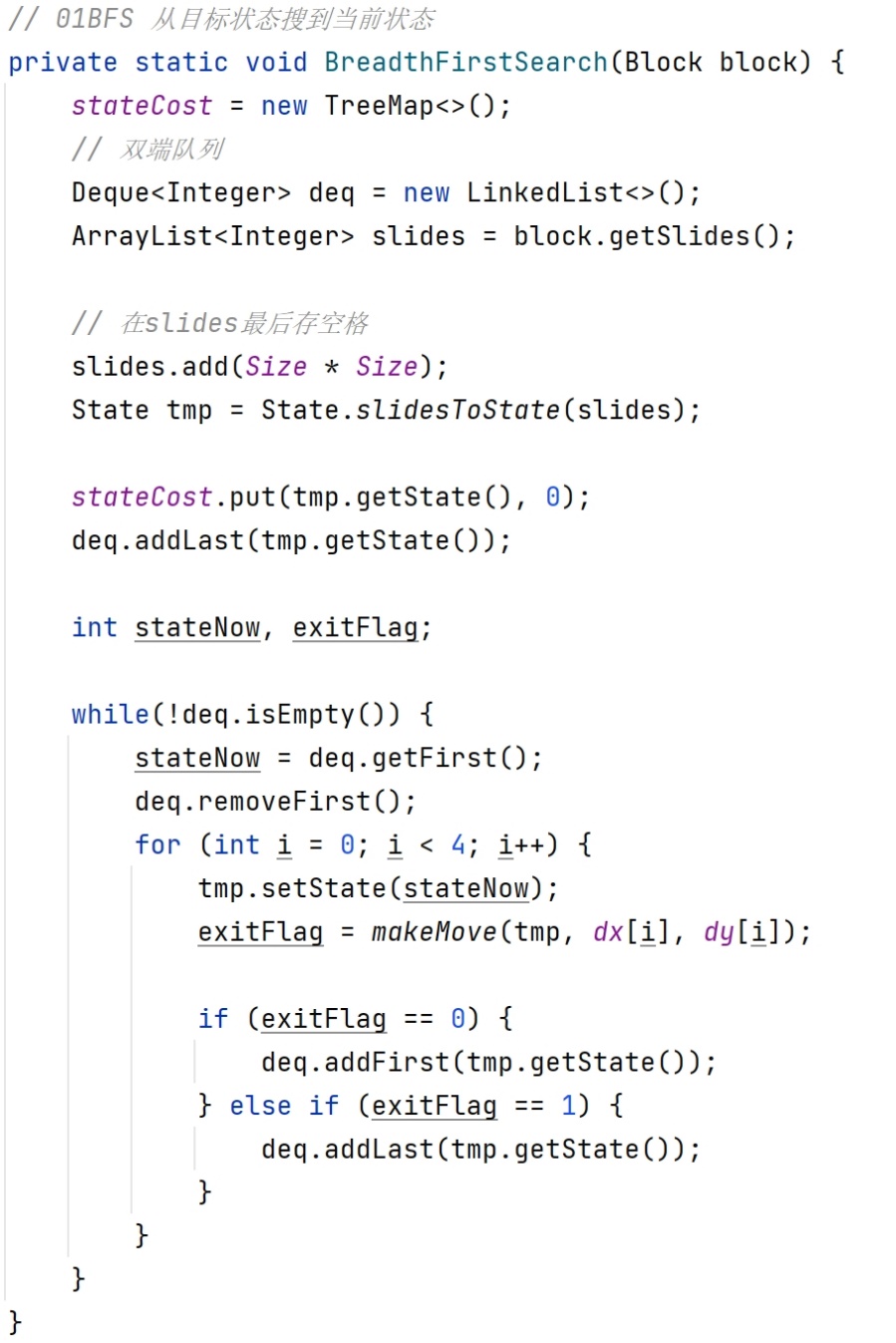
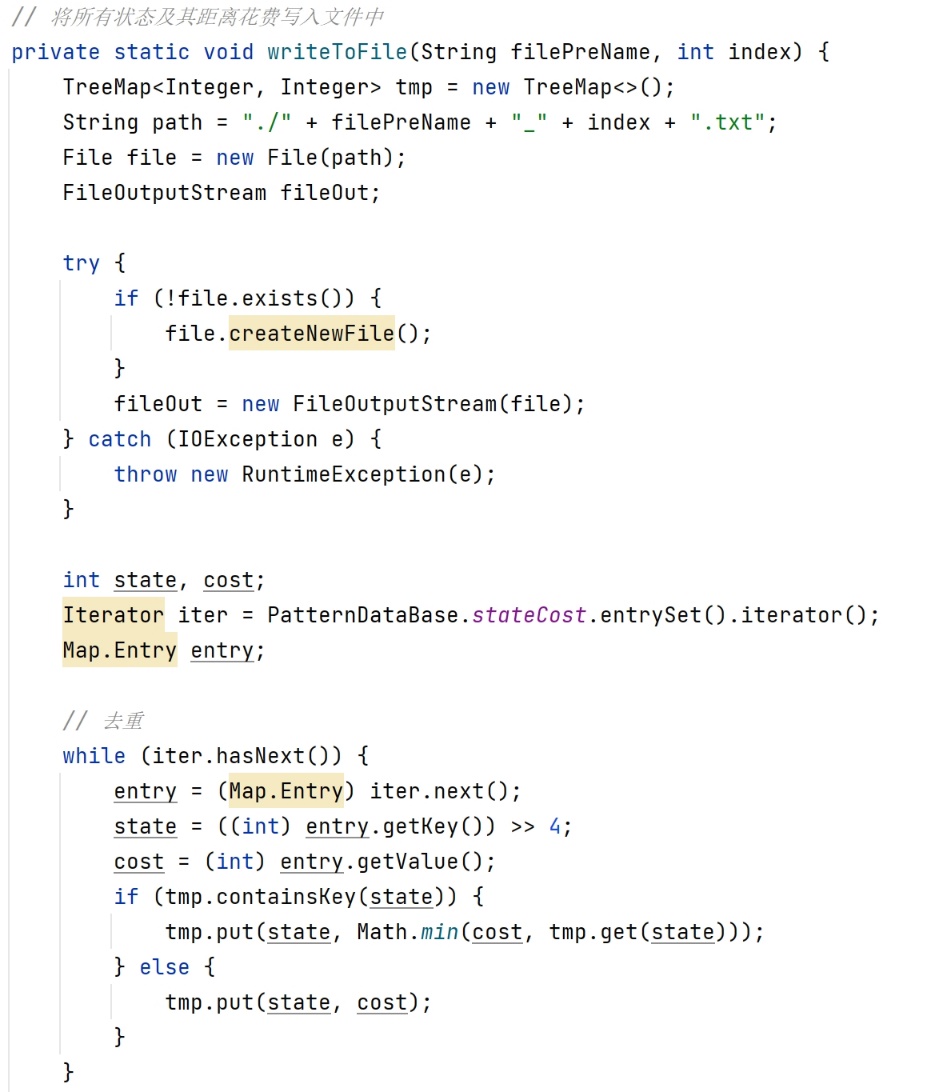


图6-25 Breath First Search

writeToFile方法用于在搜索完成后将所有状态及其距离花费写入文件中，需要先将搜索时的state右移4位，去掉空格位置信息，当状态相同时取最小值作为该状态的花费，将state和cost一起存入文件中。



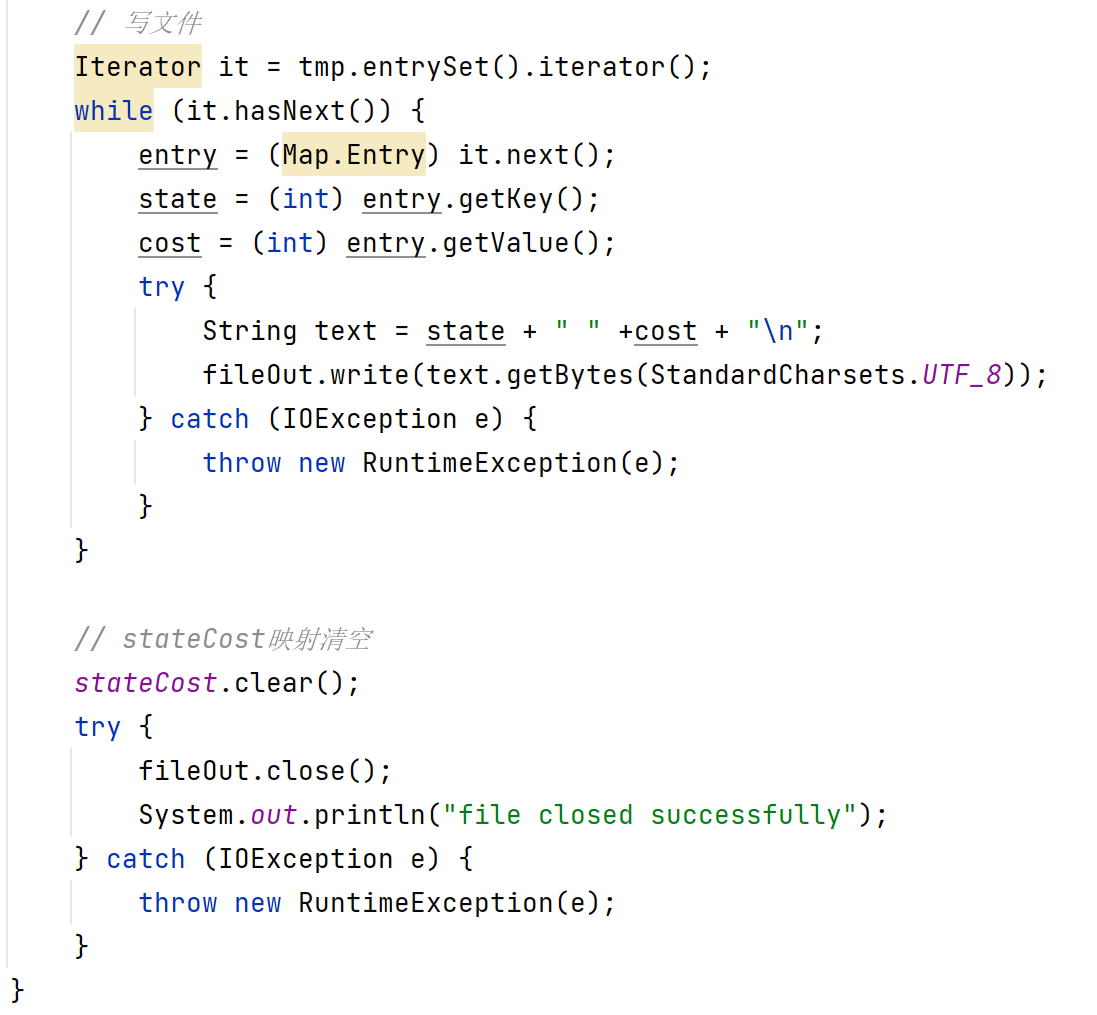


图6-26 将state和cost写入文件中

readFromFile方法用于从数据库中读取state和cost，使用动态可变数组进行存储，同时使用state作为下标实现了查询cost。



图6-27 从数据库中读取state和cost

getDistance方法通过当前格局和区块划分模式计算每一个区块的距离花费，最后将这些距离花费相加，返回总的距离花费，作为不相交模式数据库的启发函数。

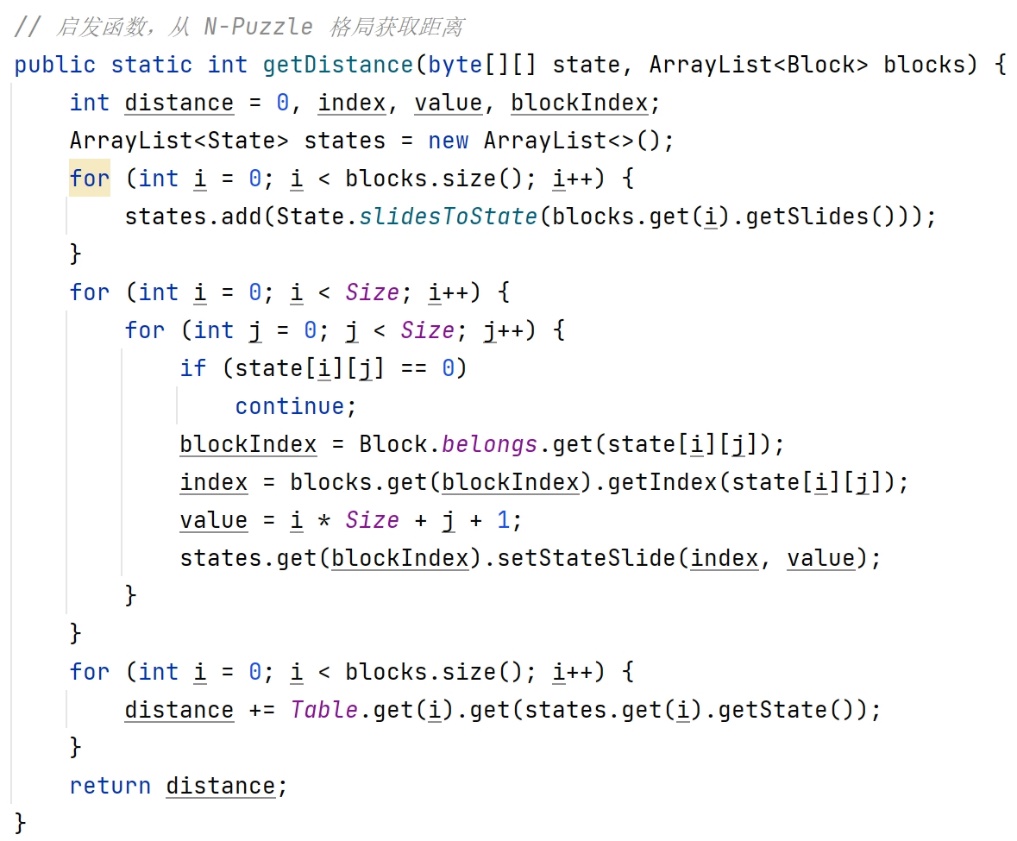


图6-28 从数据库中读取state和cost

* 1. **三个阶段的成果及实现**
     1. **第一阶段**

**要求**：能够在1秒中之内解决以下问题实例。使用AStar + 优先队列

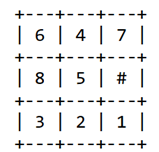
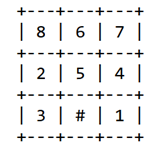


图6-29 第一阶段要求

Input:

|  |  |
| --- | --- |
| 数据1 | 3 5 0 8 4 2 1 7 3 6 1 2 3 4 5 6 7 8 0 |
| 数据2 | 3 6 4 7 8 5 0 3 2 1 1 2 3 4 5 6 7 8 0 |
| 数据3 | 3 8 6 7 2 5 4 3 0 1 1 2 3 4 5 6 7 8 0 |
| 数据4 | 4 15 1 9 8 13 5 2 10 6 14 3 4 12 0 7 11 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据编号 | 启发式函数 | 执行时间 | 生成节点数 | 拓展节点数 |
| 1 | MANHATTAN | **0.0s** | 1455 | 931 |
| MISPLACED | 0.01094s | 9232 | 6040 |
|  | | | | |
| 2 | MANHATTAN | **0.02188s** | 29610 | 20904 |
| MISPLACED | 0.0875s | 153980 | 137686 |
|  | | | | |
| 3 | MANHATTAN | **0.01094s** | 28152 | 20607 |
| MISPLACED | 0.09844s | 150874 | 137858 |
|  | | | | |
| 4 | MANHATTAN | **0.54688s** | 1287823 | 691421 |
| MISPLACED | OutOfMemoryError | OutOfMemoryError | OutOfMemoryError |

Output:

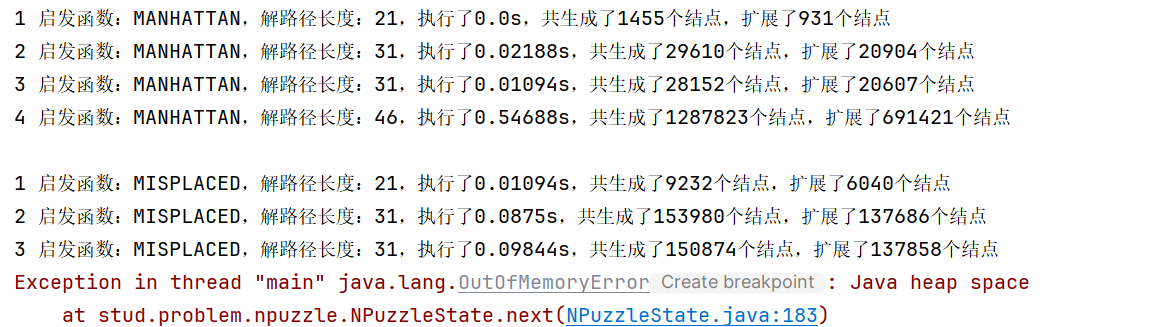


图6-30 第一阶段输出

* + 1. **第二阶段**

**要求：**对于4阶的（15-Puzzle），使用IDA\*，能够在1分钟之内解出下列实例。建议使用Zobrist哈希。

Input:

|  |  |
| --- | --- |
| 数据1 | 4 8 13 0 6 1 15 9 14 3 4 5 11 7 2 10 12 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据2 | 4 2 9 5 11 8 3 4 14 7 10 1 12 0 15 6 13 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据3 | 4 4 7 0 9 12 10 11 8 14 6 15 1 2 5 3 13 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据4 | 4 12 10 3 2 0 7 14 9 1 15 5 6 8 4 13 11 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据5 | 4 12 1 5 6 2 11 7 9 14 10 0 4 15 3 13 8 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据6 | 4 4 6 15 13 12 9 10 2 8 0 7 3 14 5 1 11 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |

Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据编号 | 启发式函数 | 执行时间 | 生成节点数 | 拓展节点数 |
| 1 | MANHATTAN | **0.1875s** | 153907 | 81351 |
| LINEAR\_CONFLICT | 0.296875s | 92800 | 48777 |
|  | | | | |
| 2 | MANHATTAN | 4.921875s | 2541216 | 1375299 |
| LINEAR\_CONFLICT | **0.6875s** | 358504 | 195725 |
|  | | | | |
| 3 | MANHATTAN | 49.46875s | 17488759 | 9928998 |
| LINEAR\_CONFLICT | **4.90625s** | 2134335 | 1182740 |
|  | | | | |
| 4 | MANHATTAN | 163.390625s | 34079618 | 18994894 |
| LINEAR\_CONFLICT | **18.25s** | 4733263 | 2618215 |
|  | | | | |
| 5 | MANHATTAN | 12.53125s | 4758161 | 2605114 |
| LINEAR\_CONFLICT | **1.984375s** | 646990 | 350824 |
|  | | | | |
| 6 | MANHATTAN | OutOfMemoryError | OutOfMemoryError | OutOfMemoryError |
| LINEAR\_CONFLICT | **178.140625s** | 26720263 | 15135539 |

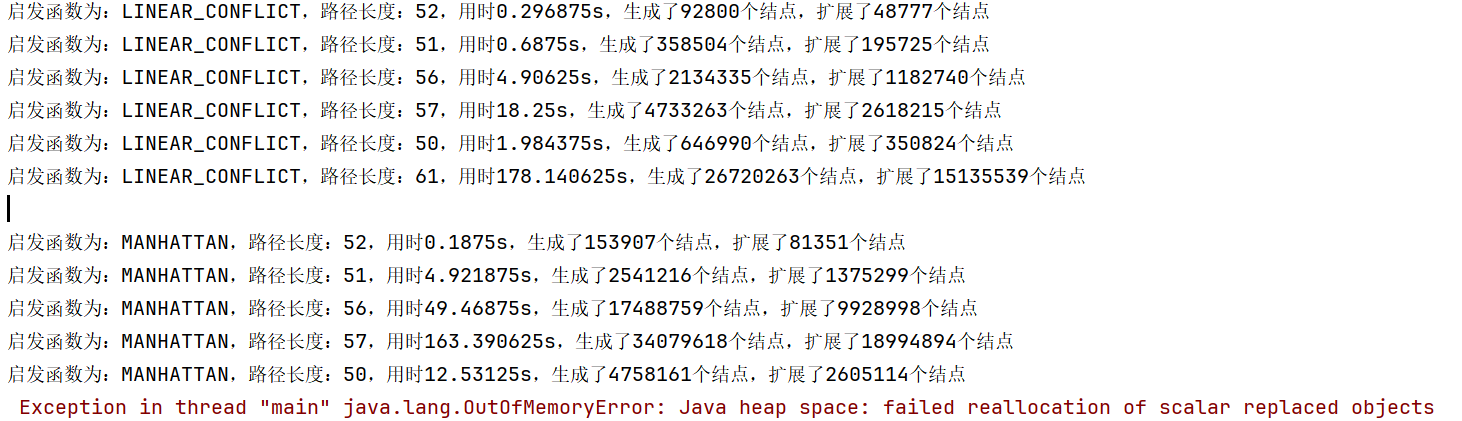


图6-31 第二阶段输出

* + 1. **第三阶段**

要求：使用disjoint pattern和模式数据库，能够在1分钟之内，解出以下实例。

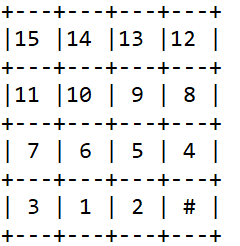
****

图6-32 第三阶段要求

Input:

|  |  |
| --- | --- |
| 数据1 | 4 8 13 0 6 1 15 9 14 3 4 5 11 7 2 10 12 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据2 | 4 2 9 5 11 8 3 4 14 7 10 1 12 0 15 6 13 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据3 | 4 4 7 0 9 12 10 11 8 14 6 15 1 2 5 3 13 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据4 | 4 12 10 3 2 0 7 14 9 1 15 5 6 8 4 13 11 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据5 | 4 12 1 5 6 2 11 7 9 14 10 0 4 15 3 13 8 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据6 | 4 4 6 15 13 12 9 10 2 8 0 7 3 14 5 1 11 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |
| 数据7 | 4 15 14 13 12 11 10 9 8 7 6 5 4 3 1 2 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 |

Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据编号 | 路径长度 | 执行时间 | 生成节点数 | 拓展节点数 |
| 1 | 52 | 0.05625s | 20313 | 10462 |
| 2 | 51 | 0.11250s | 86591 | 44225 |
| 3 | 56 | 0.13125s | 123690 | 64265 |
| 4 | 57 | 0.41250s | 306757 | 158473 |
| 5 | 50 | 0.03125s | 28024 | 158473 |
| 6 | 61 | 3.27500s | 1388293 | 731547 |
| 7 | 70 | 42.79375s | 22187123 | 12233956 |

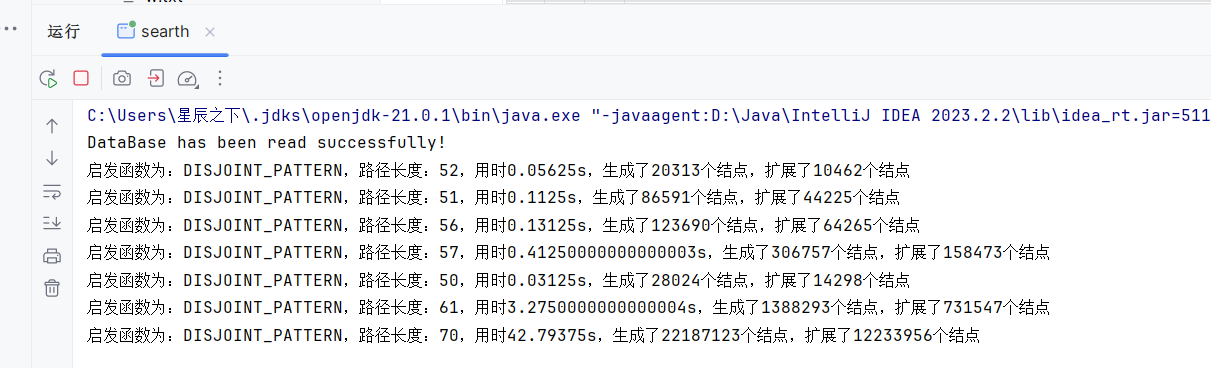


图 6-33 第三阶段输出

1. **性能对比分析**
   1. **A\*与IDA\*性能比较**

首先通过shuffle函数，随机生成100个3\*3数据集和20个4\*4数据集。

|  |  |  |
| --- | --- | --- |
|  | 大小 | 规模 |
| 数据集1 | 3\*3 | 100 |
| 数据集2 | 4\*4 | 20 |

3\*3数据集选取启发函数为：

Misplaced错位将牌数。

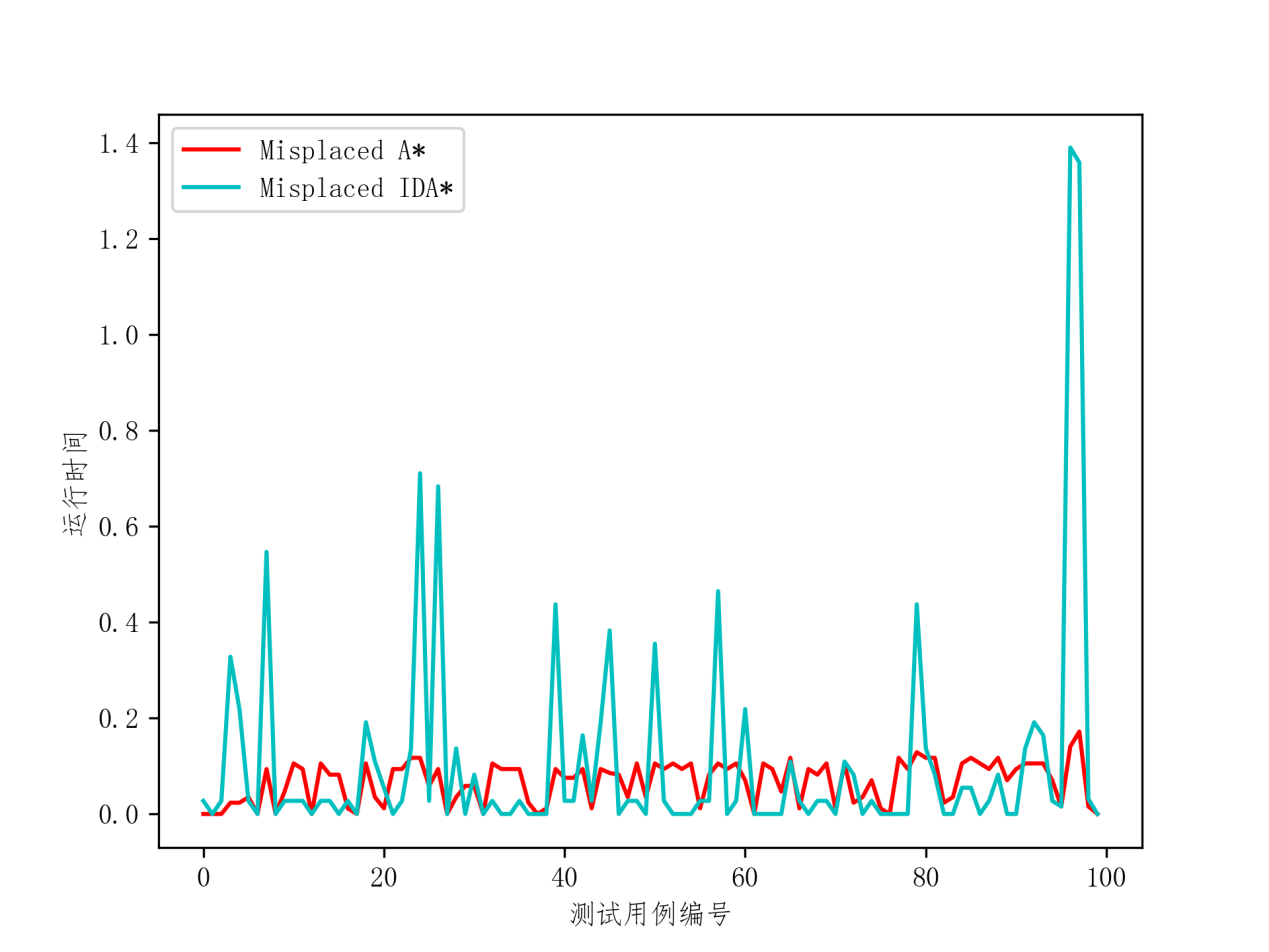


图7-1 A\*和IDA\*错位将牌运行时间对比

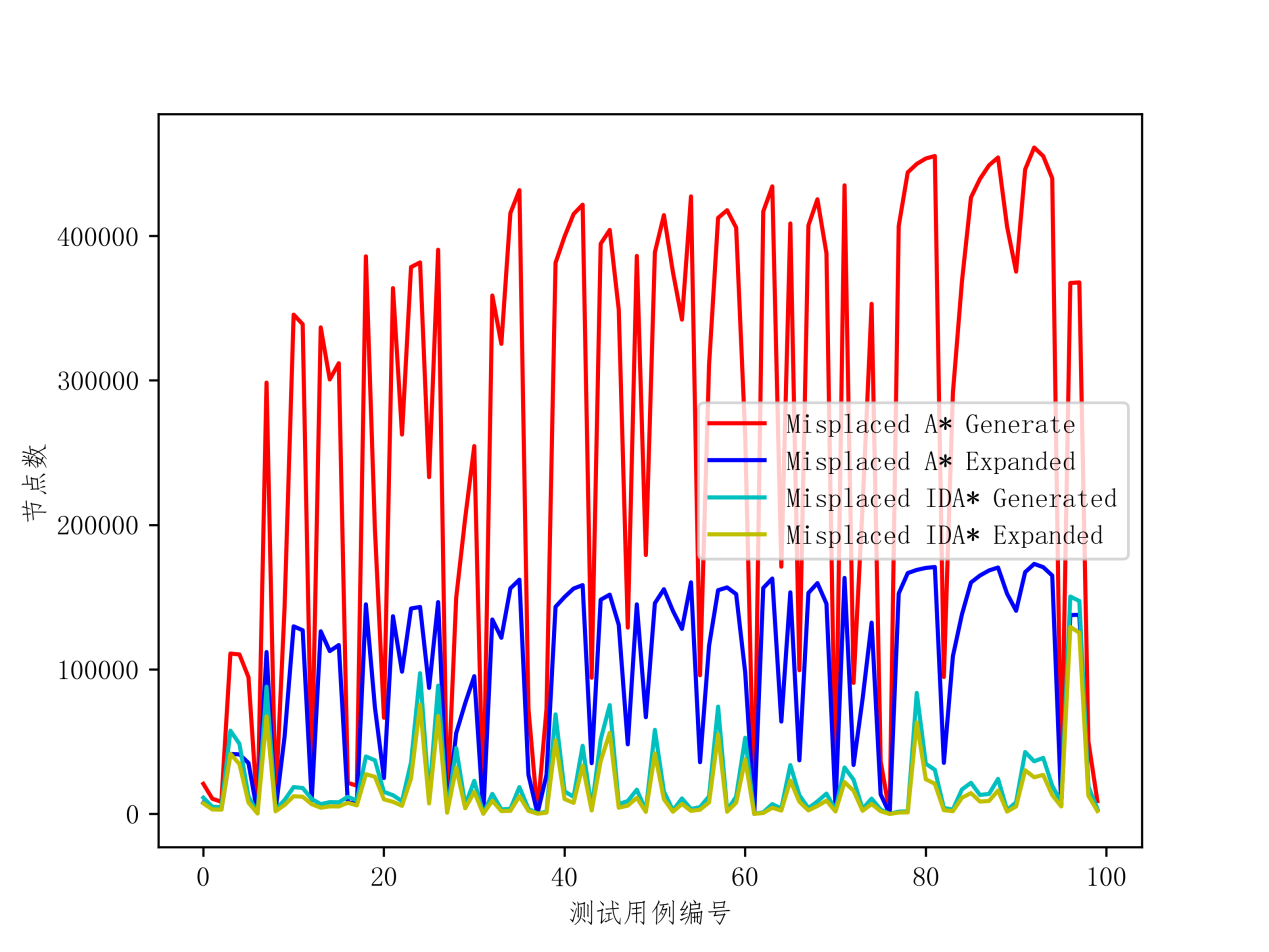


图7-2 A\*和IDA\*错位将牌生成与拓展节点数对比

4\*4数据集选取启发函数为：

LINEAR\_CONFLICT线性冲突和DISJOINT\_PATTERN不相交模式数据库。

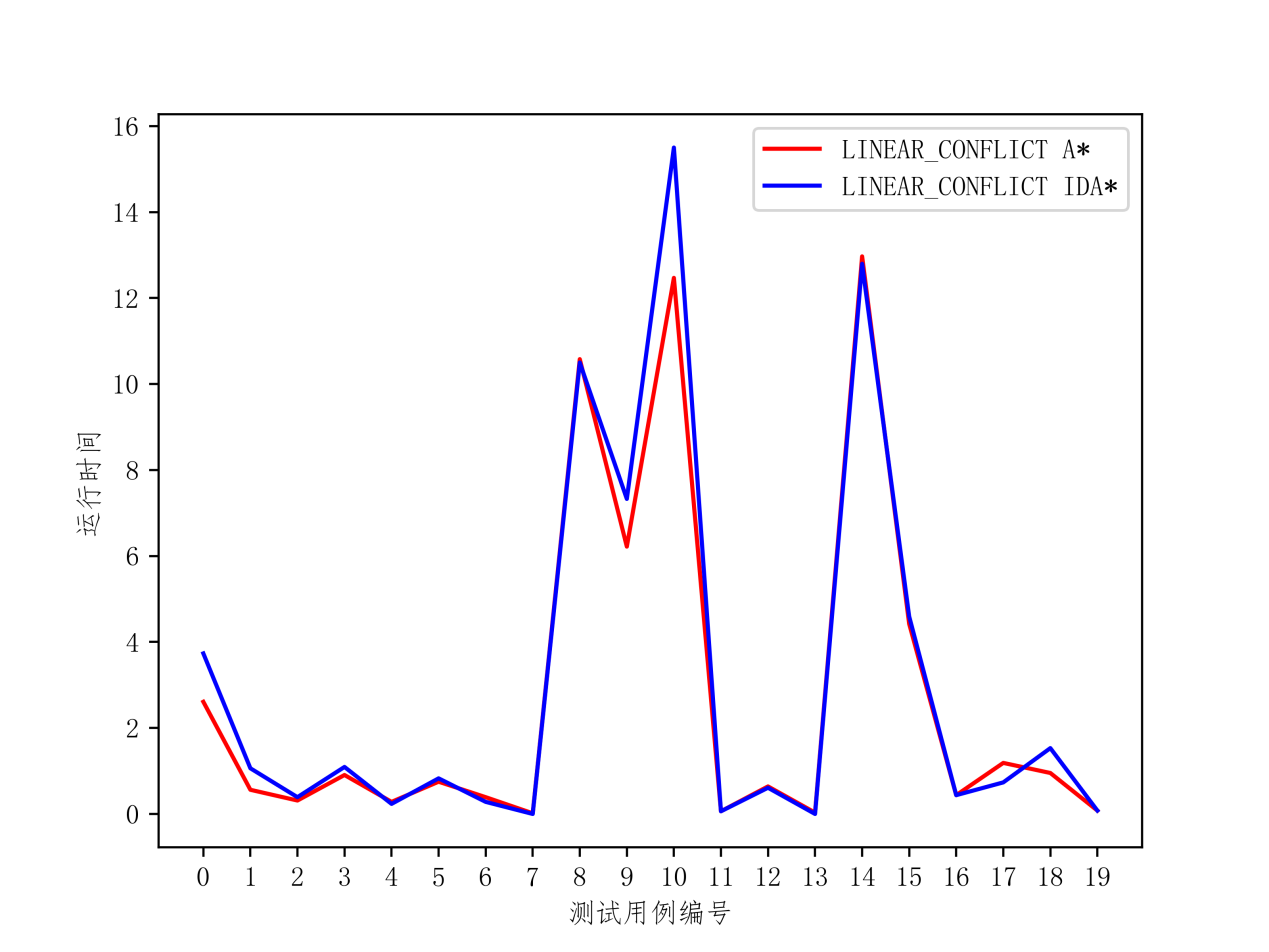


图7-3 A\*和IDA\*线性冲突运行时间对比

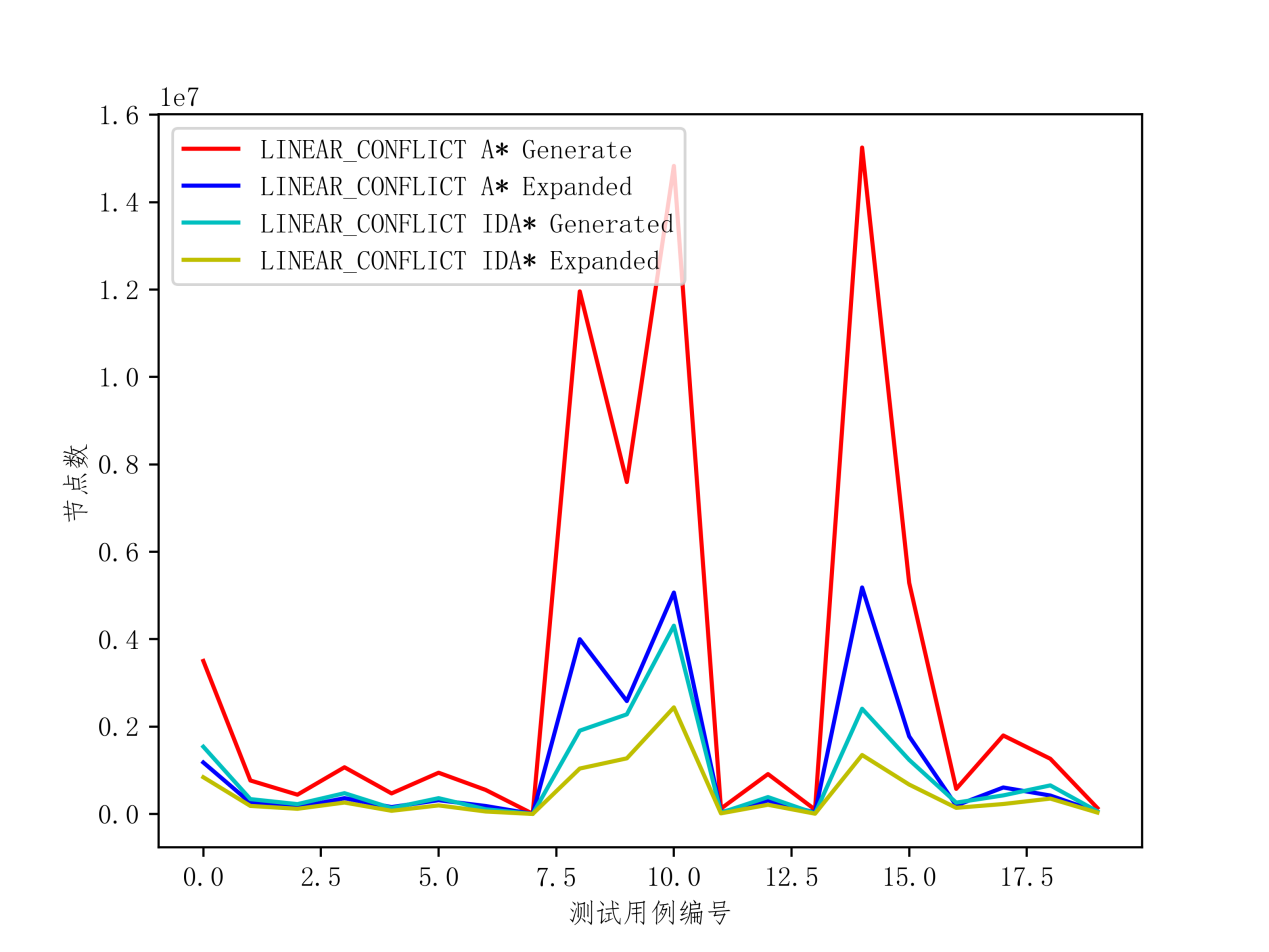


图7-4 A\*和IDA\*线性冲突生成与拓展节点数对比

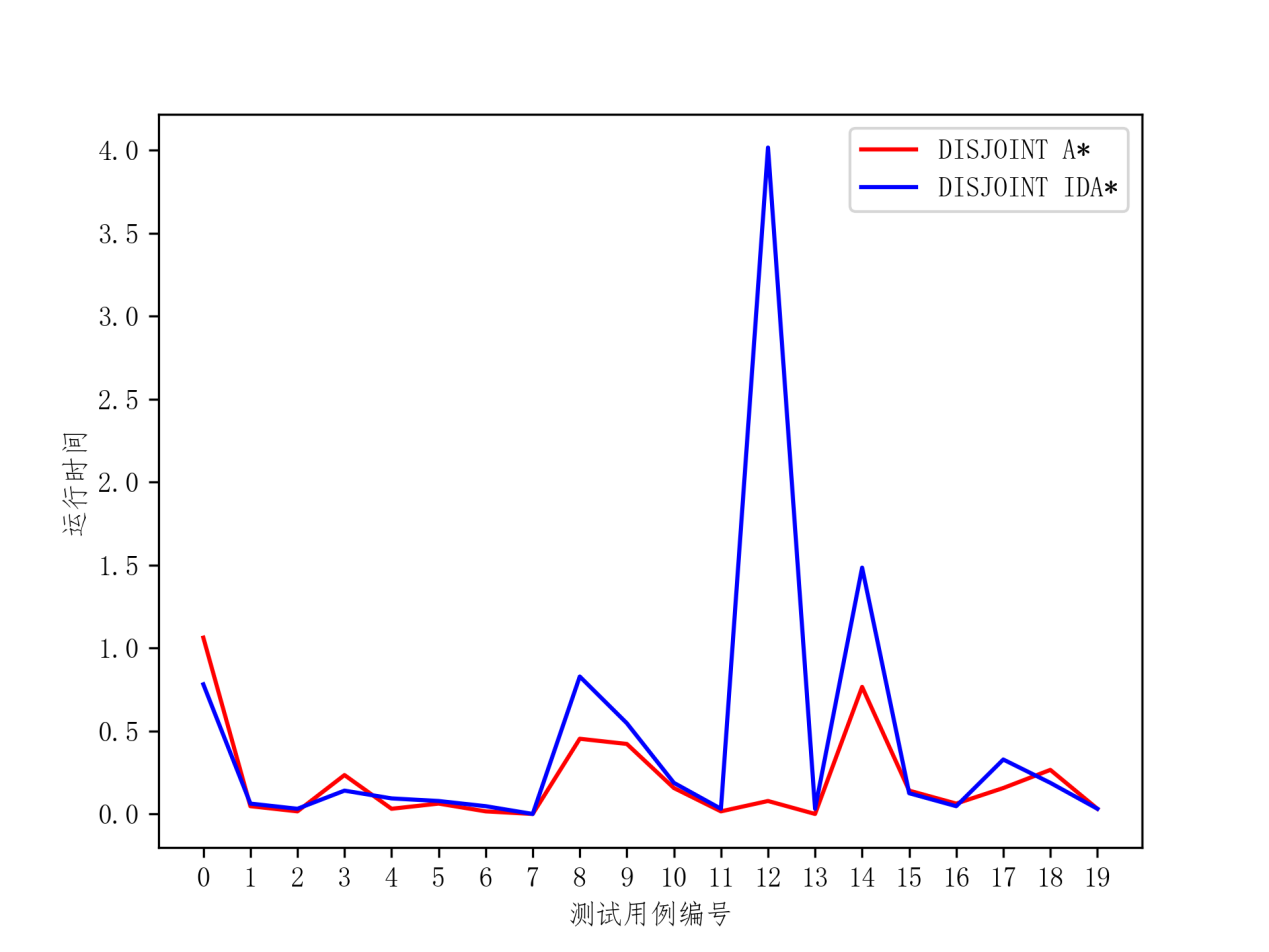


图7-5 A\*和IDA\*不相交数据库运行时间对比

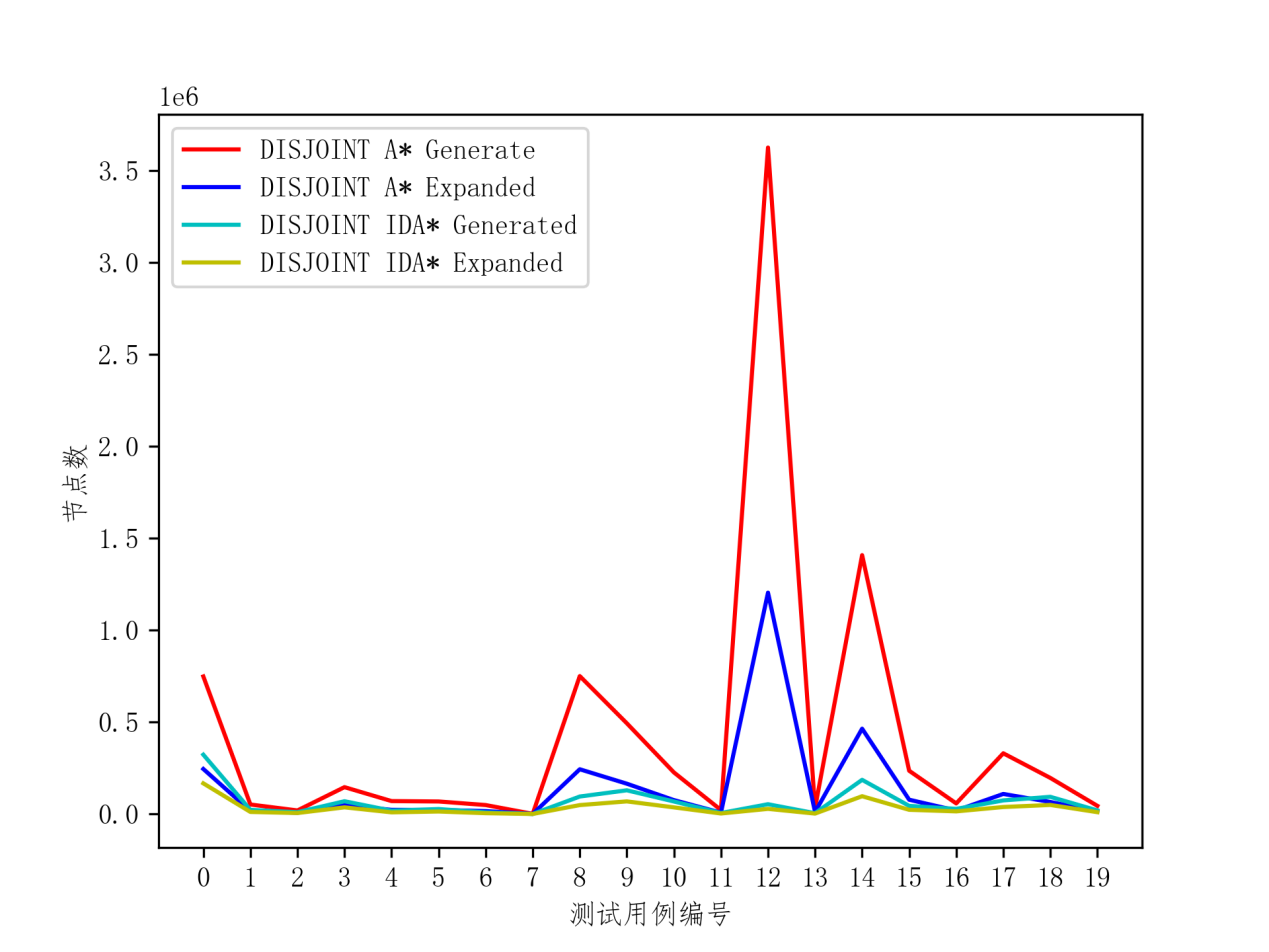


图7-6 A\*和IDA\*不相交数据库生成与拓展节点数对比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据集 | 启发函数 | A\*平均时间 | IDA\*均时间 | A\*平均节点 | IDA\*均节点 |
| 3\*3数据集 | Misplaced | **0.0674421875** | 0.106171875 | 268934.49 | **16342.88** |
| 4\*4数据集 | LINEAR\_CONFLICT | **2.79375** | 3.08984375 | 1143355.0 | **474684.5** |
| 4\*4数据集 | DISJOINT\_PATTERN | **0.20078125** | 0.45390625 | 141556.8 | **33049.55** |

由上述图例和表格可以看出，A\*和IDA\*在不同启发函数下性能的差异。

**3阶：**在启发函数为Misplaced的情况下，通过图6-1-1和图6-1-2可以看出，在大部分测试用例中，IDA\*的运行时间大约为A\*运行时间的1.5倍-3倍，但是A\*拓展节点数和生成节点数要远大于IDA\*。由此可以看出，在问题不是那么复杂的情况下，A\*的性能要优于IDA\*。如果问题情况复杂，使用A\*时，可能造成OutOfMemoryError，IDA\*就不会。

**4阶：**在启发函数为LINEAR\_CONFLICT和DISJOINT\_PATTERN的情况下，通过图6-1-3到图6-1-6可以看出，A\*和IDA\*的运行时间差异不是很大，但A\*拓展节点和生成节点数远大于IDA\*，这样就造成了在4阶问题时A\*会OutOfMemoryError。

**总结：**由于A\*实现类似于BFS，所以要存储多个节点状态，这样就会造成占用很多的内存问题。而IDA\*思想基于DFS，不会广度把节点都加进来，所以IDA\*生成和拓展的节点数远小于A\*，也不会产生内存问题，但是IDA\*实现有递归过程，运行时间方面有时就不如A\*运行时间短。

* 1. **各类启发式函数的性能分析**

此节采用4\*4数据集，由于启发函数Misplaced错位将牌数在此数据集经过测试会超时，所以采用其他3种启发函数。

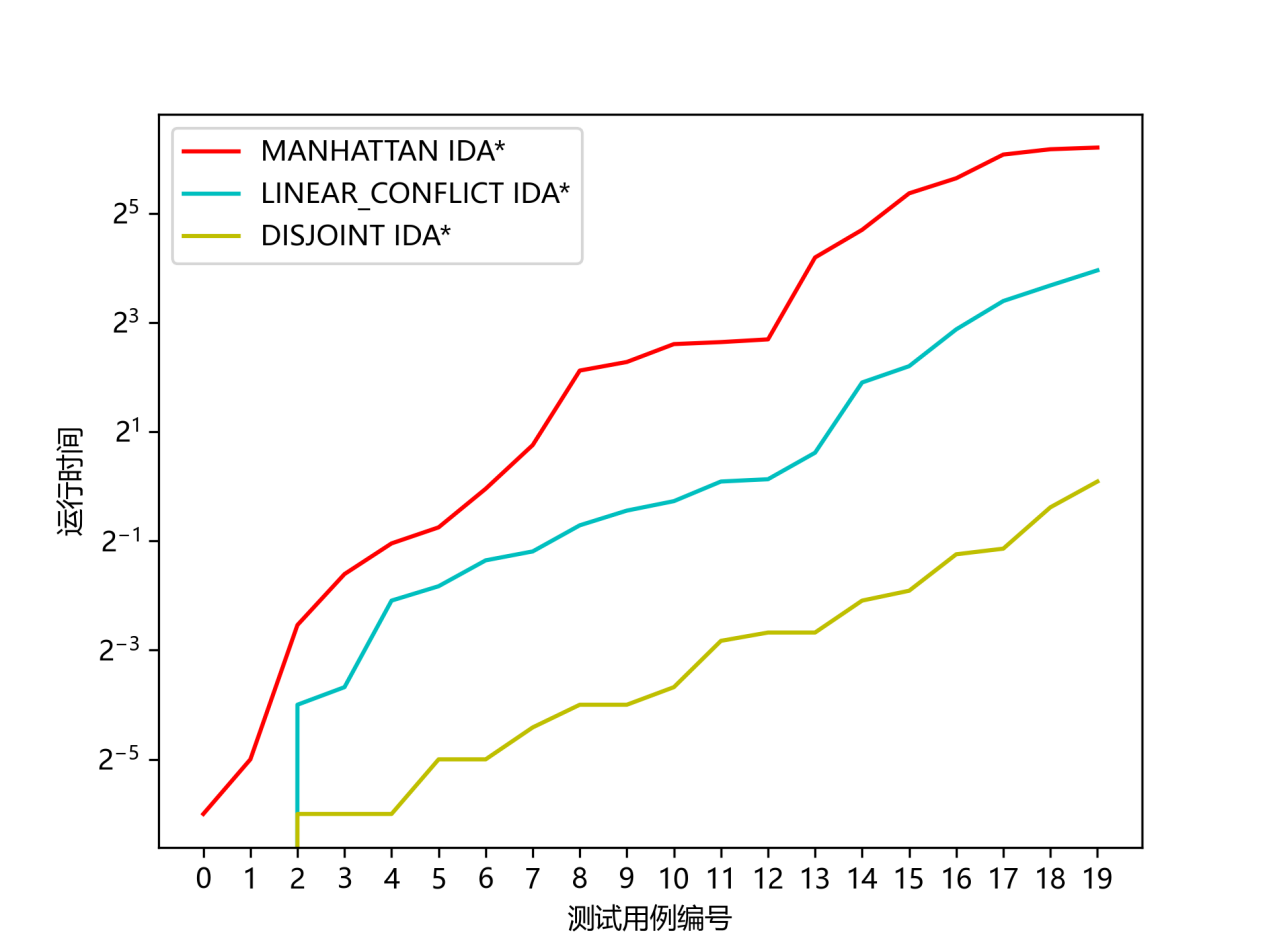


图7-7 曼哈顿、线性冲突和不相交数据库运行时间对比

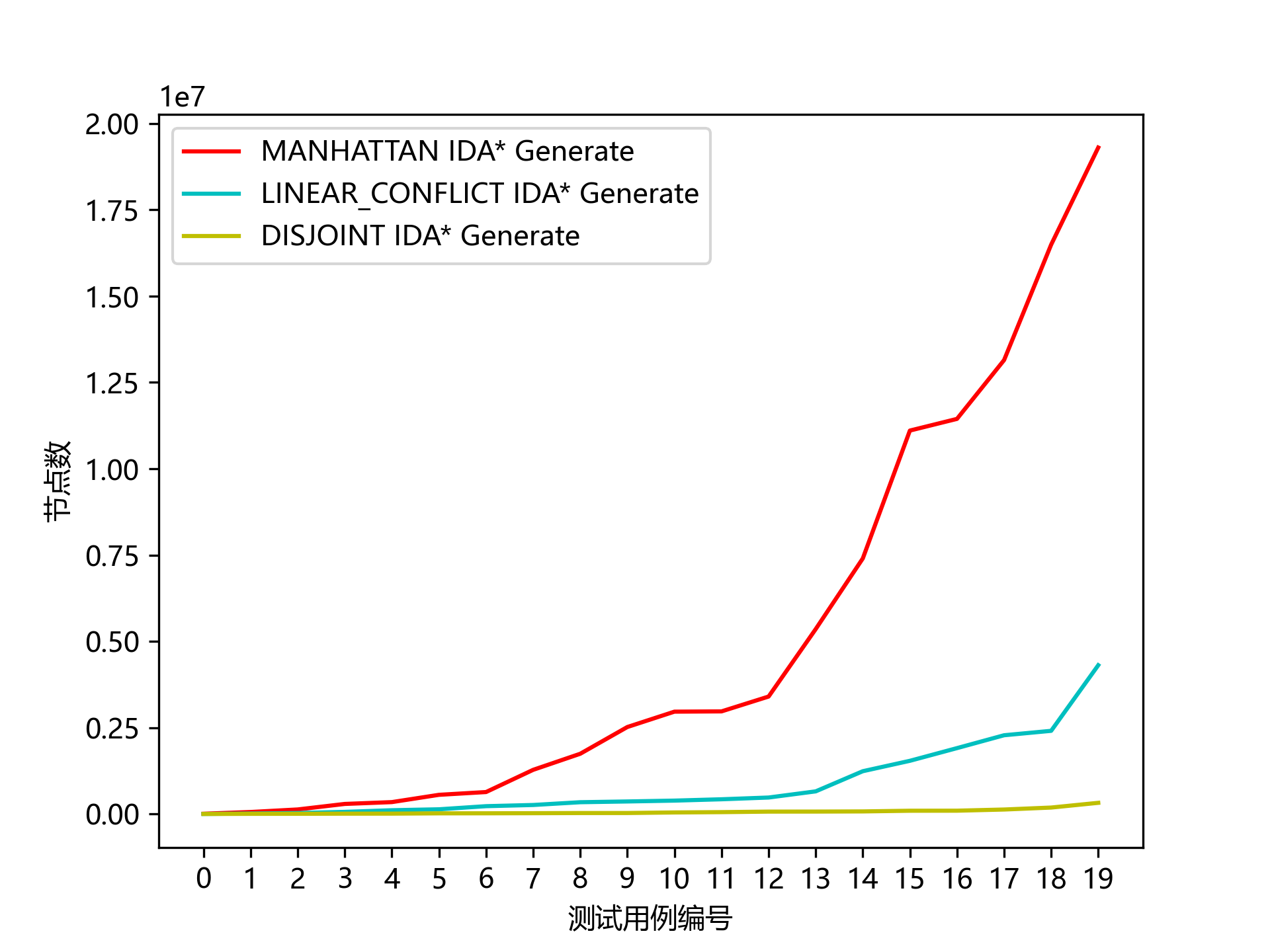


图7-8 曼哈顿、线性冲突和不相交数据库生成节点数对比

|  |  |  |  |
| --- | --- | --- | --- |
| 启发式函数 | 平均时间 | 生成+拓展节点 | 节点/秒 |
| MANHATTAN | 19.02890625 | 7878098.1 | **414,006** |
| LINEAR\_CONFLICT | 3.08984375 | 1232613.7 | 398,924 |
| DISJOINT\_PATTERN | **0.20078125** | **63806.05** | 317,788 |

**总结：**由图和表格可以看出，曼哈顿距离的运行时间是最长的，生成节点数也是最多的，其次是线性冲突，最佳启发函数是不相交模式数据库。

1. **解路径的可视化**
   1. **SFML简介**

我们使用SFML来实现解路径可视化，本方法通过c++实现，根据搜索程序给出的多个问题的解路径文本文件进行可视化展示。

SFML (Simple and Fast Multimedia Library) 是一个开源的、跨平台的C++多媒体库，它提供了一系列简单易用的接口和工具，可以方便地创建各种图形、音频、视频等应用程序。SFML 支持 Windows, Linux, macOS 和 Android 四种操作系统。SFML 提供了以下功能：

* 窗口管理：创建窗口，处理输入事件（键盘，鼠标），显示图像；
* 图形绘制：支持 2D图形绘制，包括基本图形（点，线，矩形，圆等）、渲染纹理、精灵动画等；
* 音频处理：支持 PCM 音频流播放、录制，以及音量控制、特效等；
* 网络通信：支持 TCP 和 UDP 协议的网络通信；
* 多线程处理：支持多线程并发处理，可以在主线程上更新窗口和处理输入事件。

本项目中的滑块主要通过Sprite和Texture实现。

一个可视化的2D游戏对象，比如一个角色，在SFML中被称为Sprite(精灵)。Sprite通常会使用一个或多个Texture(纹理)作为它的皮肤贴图。在SFML中，Texture对应于具体的图像。所有的Sprite都有自己的原点。如果我们将一个Sprite(精灵)绘制到屏幕上的一个特定位置，实质上就是将Sprite的原点(0,0)图中，在(1920, 1080)的屏幕中，小人儿(sprite)的中心坐标为(960, 540)。定位到这个位置。

texture是一个图像。但我们称之为“ texture”，因为它有一个非常特殊的角色：映射到2D实体。sprite只不过是一个纹理矩形。



图8-1 Sprite and Texture

SFML的官方网站：<https://www.sfml-dev.org/index.php>。

* 1. **代码解析**

初始化，设置窗口及图案大小，这里根据图案png文件的大小设置窗口大小为256\*256（8数码设置为192\*192），每一个滑块的大小设置为64\*64，每个按钮图案纹理加载到t[20]上。数组grid用来记录每次布局。

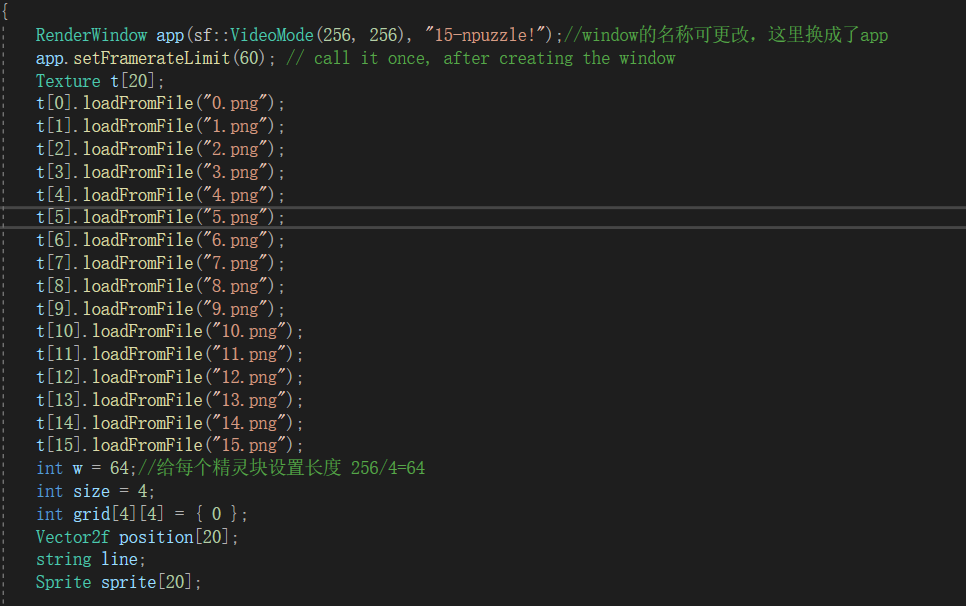


图8-2 数字推盘构建

用istringstream文件流一行一行读取文本文档里的内容，根据grid读取到的滑块编码对精灵sprite进行纹理映射。例如，读入初始布局15，14，13，12，11，10，9，8，7，6，5，4，3，1，2，0，根据滑块编号选择t[n]中的滑块图案通过setTexture生成16个精灵布局。每次读取新的路径再更新一次布局，实现解路径可视化，并输出完成的步数。

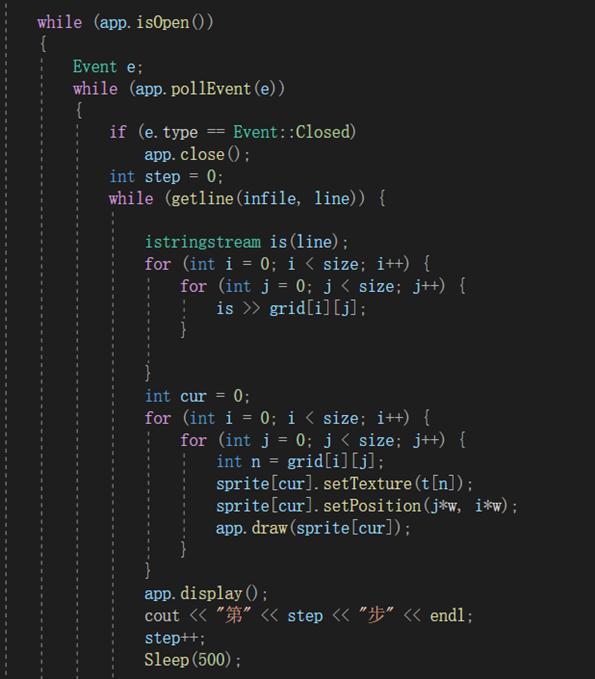


图8-3 将路径映射到精灵棋盘

值得一提的是精灵中的move函数：

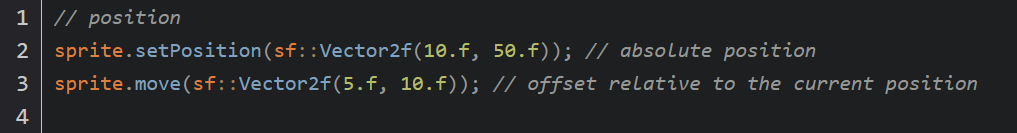


图8-4 SFML官方声明

它实现的是精灵相对于当前位置的偏移，但因为我们这里将空白的位置也定义成了一个精灵，在每次滑块移动的时候如果考虑通过move实现每一次路径的展现，那必然要考虑空白滑块周围的数字滑块的移动，会将简单的问题复杂化，所有我直接选择根据新布局重新绘制以实现路径可视化。

* 1. **项目接口**

本项目可视化实现主要是通过读取生成的解路径文本文档实现。

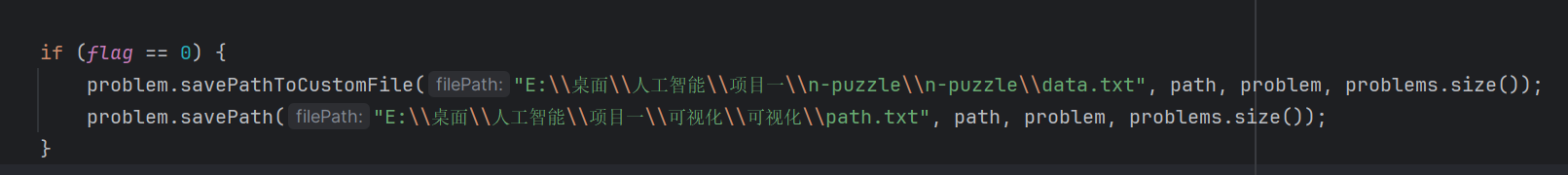


图8-5 java文件中存储路径

在java项目中将每次搜索完成后的棋盘局面存储在一个txt文档中，位于可视化c++项目文件夹下。

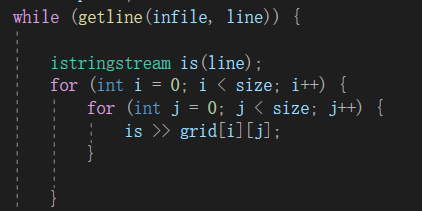


图8-6 c++文件中读取文本文档

接着SFML项目中通过istringstream文件流按行读取棋盘布局到二维数组grid中，进行可视化。

存储生成的文本文档如下：

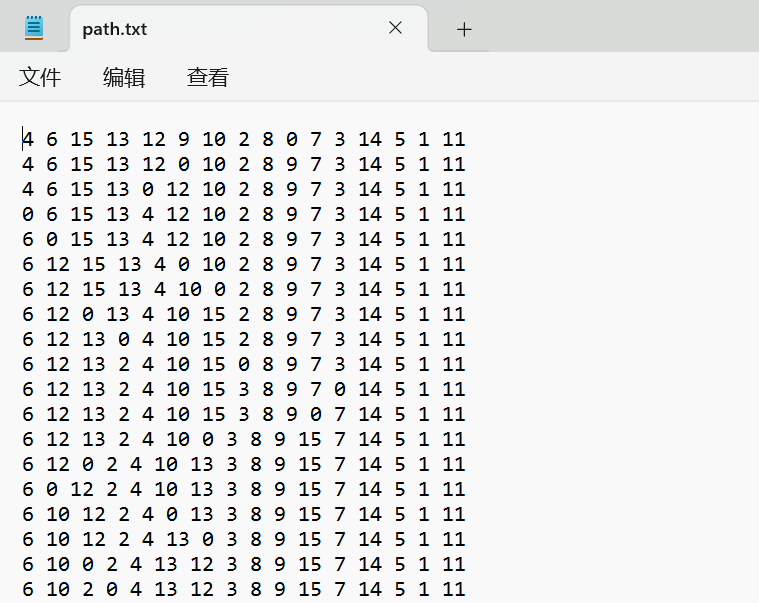


图8-7 文本文档

* 1. **成果展示**
     1. **8数码**

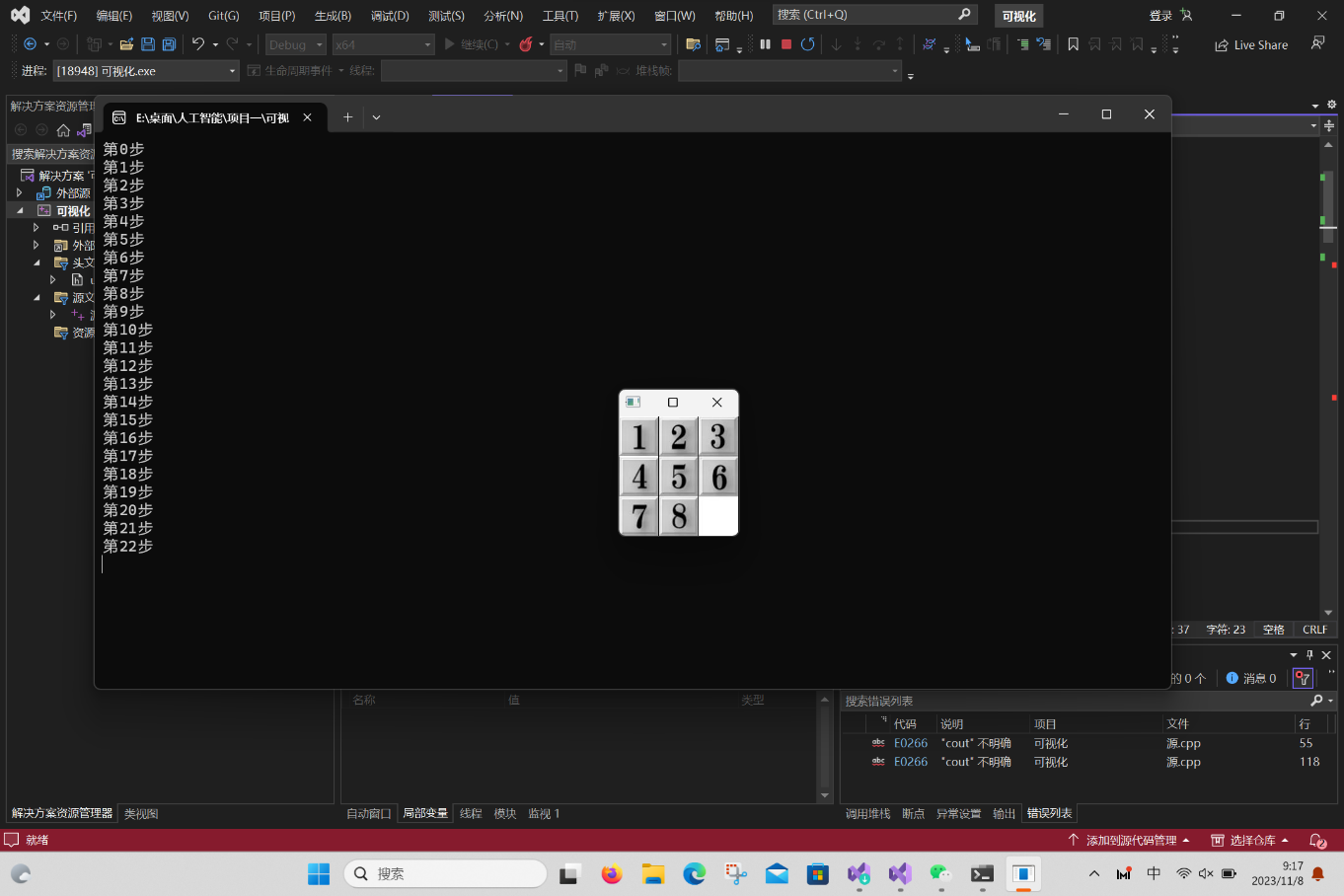


图8-8 文本文档

****

* + 1. **15数码**

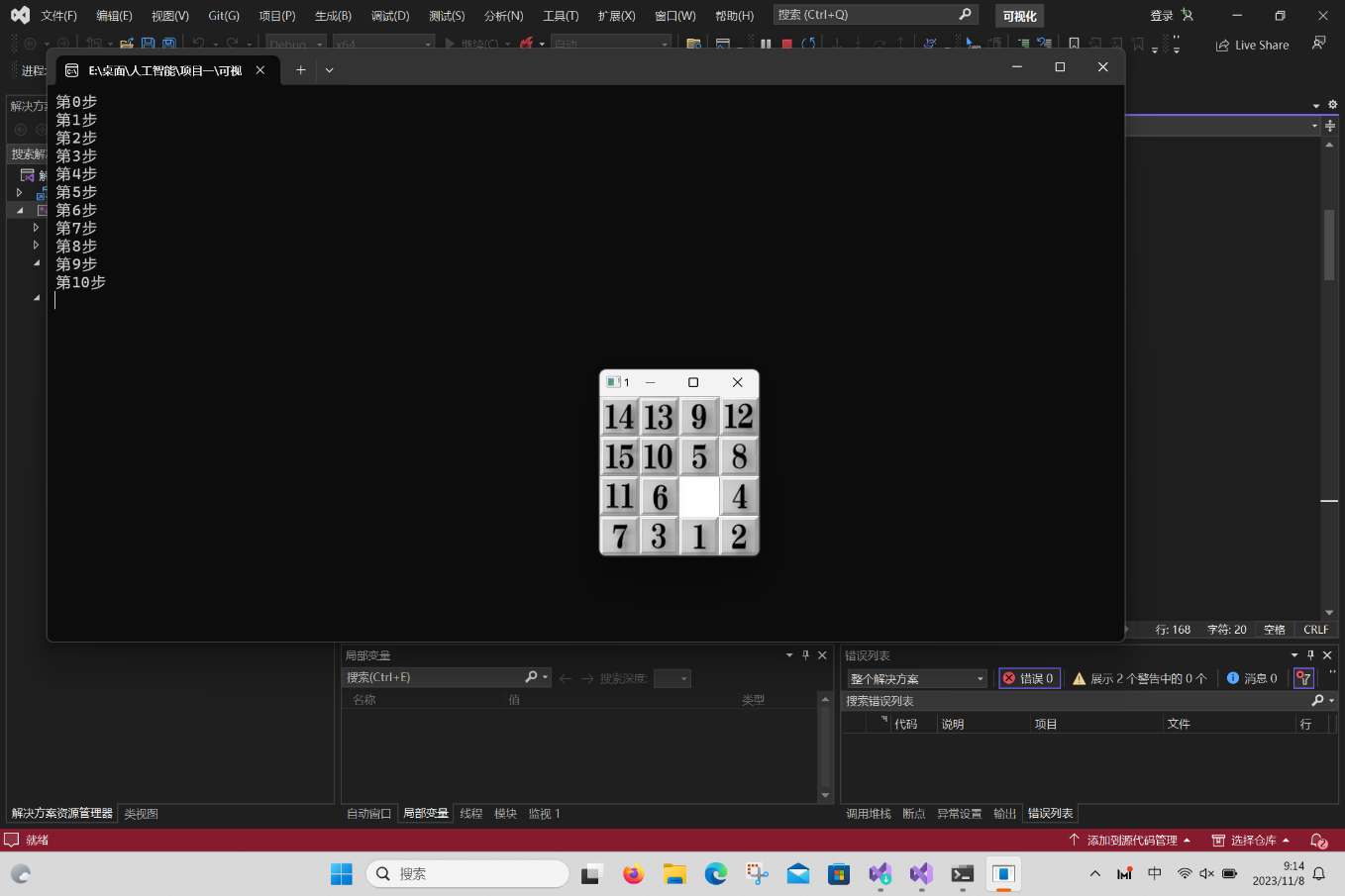


图8-9 文本文档

****

1. **总结及展望**
   1. **周忠康**

在本次项目中，我主要负责一二部分的A\*算法和IDA\*算法的构建，并撰写了对应部分的研究报告和PPT。在项目初期，我首先对A\*和IDA\*进行了再一次的深入学习，并理解了老师给定的算法框架。对于这个问题来说，一个好的启发函数和列表的数据结构可以让运行速度有一个很大的提升。我们采用了ZobristHash的方式去索引状态，这样将复杂度从O(VectorSize）变为O（1），启发函数越接近实际的H\*，算法需要搜索的结点越少，效率越高。在优化算法的过程中，我发现对于最复杂的情况即15 14 13...这样的完全逆序排列，如果是通过数据库的方式，时间开销也只能在60秒左右，但如果将线性冲突计算的值乘以一个倍数，这样可能导致估价函数大于实际距离，这个问题可以在秒级别解决。当然经过讨论我们认为，这个论文中给出的线性冲突计算公式是严格小于实际值的，而我们的有可能大于实际值。对于N-Puzzle问题，一个复杂问题可能在搜索过程中有上百万个结点，而实际解路径只有几十结点，因此估价函数大于实际距离大部分都会发生在非最短路径上的结点，这样导致大大减少了在非最短路径上结点的搜索。在实际情况中，为了减少时间成本获得一个较优的解，可以这样尝试。这是我对于A\*和IDA\*算法的一点想法。

对于三四问，我在学习了不相交数据库之后，经过沟通确定了调用端口的使用以及融合入框架，同时输出路径信息方便可视化。这次项目锻炼了我对Java的学习和掌握，也让我对算法有了更深的认识，同时也提高了项目合作与分工的能力。

* 1. **王颢凯**

在本次项目中，我主要负责完成一二阶段的实现，并编写了相应部分的研究报告和PPT。在项目初期，我首先将老师给的框架整篇阅读，理清类与类、接口之间的关系。同时进行A\*与IDA\*算法的学习。值得一提的是，我们启发函数增添了线性冲突，这目前是我发现性能比较优秀的启发函数。

我首先编写了用于搜索的open表和close表，open表通过PriorityQueue实现，close表通过HashMap实现。在队列和哈希表中我们采用存取每个状态的ZobristHash值，这样会减轻很大的存取负担与麻烦。

后续我编写了IDA\*算法，通过这次代码实现，让我对理论的理解更进一步，只有实践才能让所学所想提升。在第三第四阶段代码实现后，我将各部分代码进行了整合，并且对各处的错误进行详细排查与debug，最终得到终版。

最后我对各个启发函数、A\*、IDA\*和不相交数据库的性能进行了详细的分析。我通过shuffle函数生成100个3\*3的数据集和20个4\*4的数据集，将跑出的运行时间、节点生成与拓展数，最终在Python上绘制对比图片可视化结果，得到了较为客观的评价。

总体来说，通过此次实验，让我对启发式搜索有了更深的理解与认识，对算法的代码动手实践了一次。同时，实现项目也提升了我的代码编写和debug能力，提升了我对java语言的动手能力，最后也锻炼了我数据分析与可视化分析的技能。

* 1. **曹文汀**

在本次项目中，我主要负责第三部分不相交模式数据库的实现，并负责该部分报告撰写和PPT制作讲解。在项目初期，我学习了A\*算法和IDA\*算法，加深了对启发式搜索算法的理解。

对于第三部分，我先查阅了许多英文文献和资料，提高了阅读文献的能力。通过阅读文献，我了解了不相交模式数据库的基本原理和正确性。在解决较复杂的15-Puzzle问题时，不相交模式数据库中提前存储了简化问题的估价距离，在解决困难问题时直接使用预先存储的距离作为估价函数，这样就能降低搜索时花费的时间。搜索解路径时空间与时间总有一方会难以接受，而为了更快的解决问题，我们可以通过预先处理简化问题并存储下来，较难的问题在解决时再利用预先产生的数据，这样便能大大降低解决问题的时间，对于实际应用问题来说这种方法非常有效。同时，在数据库存储时，我使用了一种简单的状态压缩算法，将区块状态编码成一个int类型数据，从而降低了存储空间。

在本次项目里，我深入接触了Java面向对象编程，提升了我的多语言编程能力，数据库部分的实现，对我来说是一个很好的锻炼，增强了我编写程序和调试程序的能力，加深了我对算法的理解。生成数据库和官方数据进行比对后，验证了结果的正确性。

* 1. **田晶怡**

在本次项目中，我主要负责第四部分可视化的实现，并负责该部分报告撰写、PPT制作讲解以及项目实验报告的整合处理。在此项目之前我有过Qt可视化实现的经历，完成过图像处理器的可视化实现，但在本次实验中我选择了老师推荐的一种新的可视化实现方式——SFML。网络上对于SFML的教程少之又少，全英文的官方网站是我学习这种可视化实现的主要渠道。在学习中我发现，这是种开源且可以在多个平台上实现的C++多媒体库，一些常见的小游戏（如俄罗斯方块、麻将、贪吃蛇等）都可以通过SFML轻松实现，尤其是交互部分，它的窗口管理功能中的Event对象可以直接调用函数实现键盘、鼠标交互，反而实现本项目中的自动实现滑块移动成了难事。最终的实现代码非常简单，但在想到它之前我走了不少弯路，比如读取txt中空白滑块的位置再调用move函数实现相对移动、构建移动方向通过滑块交换实现……始终无法实现自动的、丝滑的移动，最后还是选择了根据每次更新的局面重新绘制棋盘，设置了sleep函数控制滑块移动的速度。

在本次项目中，除了掌握了一种新的可视化实现方式，我还学习到了哈希技术和不相交模式数据库相关知识。因为对第三部分很感兴趣，我阅读了Additive pattern database heuristics、Disjoint pattern database heuristics等文章，使我阅读文献的能力得到了提升。虽然最后我的代码因为有缺陷而未被采纳，但学习到了一种很好提高搜索效率的方式，即将可能情况存入表中构建索引然后进行快速查找的方式。同时积极与组内其他成员沟通也加深了我对A\*算法、IDA\*算法以及对线性冲突的理解，对java代码的阅读理解能力提升。

## 参考文献

[1] [n-puzzle问题\_npuzzle问题介绍-CSDN博客](https://blog.csdn.net/qq_43549984/article/details/99706583)

[2] Culberson J C, Schaeffer J. Pattern databases[J]. Computational Intelligence, 1998, 14(3): 318-334.

[3] Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. Artificial Intelligence 134:9-22.

[4] W.W. Johnson, W.E. Storey, Notes on the 15 puzzle, Amer. J. Math. 2(1879)397-404

[5] Felner,A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. JAIR 22:279-318.