# 人工智能-第一次课程作业报告

授课教师：张宇 作者：蒋雨初-58121102

## 1 问题描述

### 1.1题目介绍

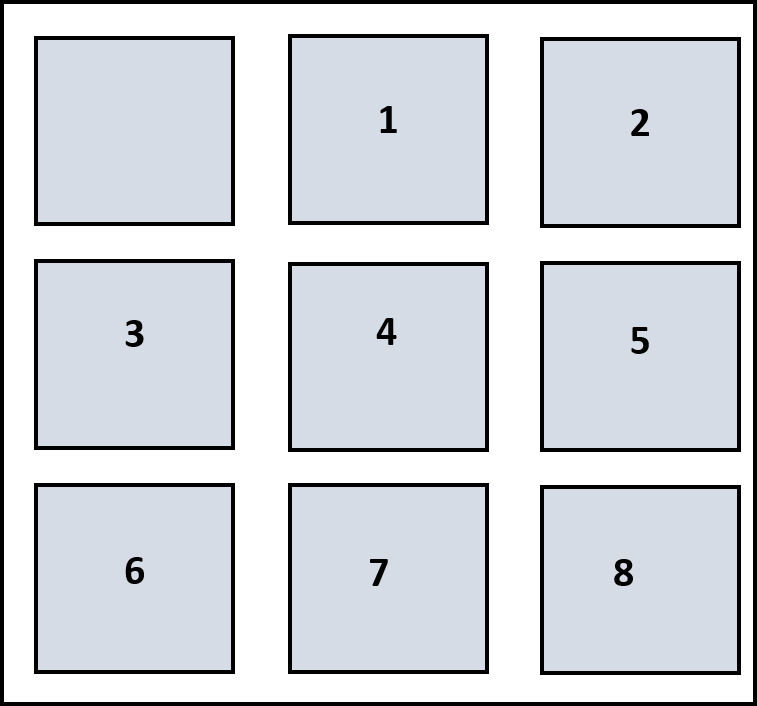
在九宫格里放在1到8共8个数字还有一个是空格，与空格相邻的数字可以移动到空格的位置，问给定的状态最少需要几步能到达目标状态（用0表示空格），目标状态如图1所示。

图1 目标状态

### 1.2任务说明

本次实验主要实现八数码问题的四种算法，分别为广度优先算法、深度有限算法、基于曼哈顿距离的A\*搜索算法和基于不正确数码位置的A\*搜索算法。

在框架EightFigurePuzzlesFrameWork中实现四个函数，在输入的回退步数和实验方法设置下能够正确输出数码移动过程、结果是否正确和移动步数。

### 1.3实验环境

#### 设备规格

设备名称 LAPTOP-TFMBQKQ8

处理器 AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

机带 RAM 16.0 GB (15.4 GB 可用)

设备 ID 6CC35513-8821-49C9-A60B-C44F31F302A7

产品 ID 00342-35891-56086-AAOEM

系统类型 64 位操作系统, 基于 x64 的处理器

笔和触控 没有可用于此显示器的笔或触控输入

#### 系统规格

版本 Windows 11 家庭中文版

版本 22H2

安装日期 ‎2022/‎9/‎28

操作系统版本 22621.521

体验 Windows Feature Experience Pack 1000.22634.1000.0

#### 开发环境

Microsoft Visual Studio Professional 2022

Version 17.2.4

VisualStudio.17.Release/17.2.4+32602.215

Microsoft .NET Framework

Version 4.8.09032

Installed Version: Professional

Visual C++ 2022 00483-00000-00004-AA929

Microsoft Visual C++ 2022

#### 语言标准

C++20 or latest.

### 1.4评价标准

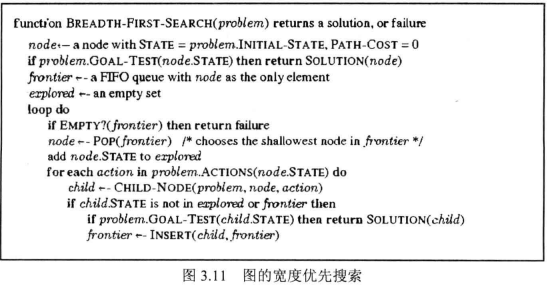
本实验主要考察解决方案的步数、运行时间和空间使用（通过分配的节点数体现）。可以通过比较相同步数的解决方案来得到不同搜索算法的效率。

## 2 实验方案

### 2.1 宽度优先搜索

宽度优先搜索( breadth- first search)是简单搜索策略，先扩展根结点，接着扩展根结点的所有后继，然后再扩展它们的后继，依此类推。一般地，在下一层的任何结点扩展之前，搜索树上本层深度的所有结点都应该已经扩展过。宽度优先搜索是一般图搜索算法（图3.7）的一个实例，每次总是扩展深度最浅的结点。这可以通过将边缘组织成FIFO队列来实现。就是说，新结点（结点比其父结点深）加入到队列尾，这意味着浅层的老结点会在深层结点之前被扩展。对一般图搜索算法做简单修改，目标的测试是在结点被生成的时候，而不是结点被选择扩展的时候。我们会在讨论时间复杂度的时候解释这一点。要注意的是，算法具有一般的图搜索框架，忽视所有到边缘结点或已扩展结点的新路径；可以容易地看出，这样的路径至少和已经找到的一样深。所以，宽度优先搜索总是有到每一个边缘结点的最浅路径。

图3.11给出了伪代码。



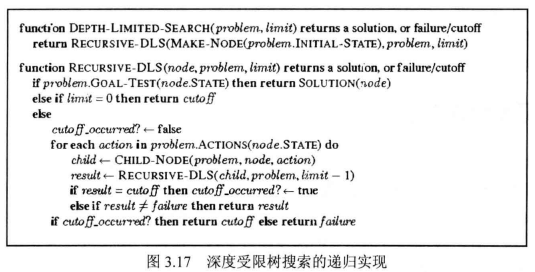
### 2.2 深度受限搜索

深度受限搜索实际上是深度优先搜索的改进版，所以先从深度优先搜索开始介绍。

**深度优先搜索**( depth- first search)总是扩展搜索树的当前边缘结点集中最深的结点。搜索过程如图3.16所示。搜索很快推进到搜索树的最深层，那里的结点没有后继。当那些结点扩展完之后，就从边缘结点集中去掉，然后搜索算法回溯到下一个还有未扩展后继的深度稍浅的结点。深度优先搜索算法是图3.7的图搜索算法的实例；宽度优先搜索使用FIFO队列，而深度优先搜索使用LIFO)队列。LIFO队列指的是最新生成的结点最早被选择扩展。这一定是最深的未被扩展结点，因为它比它的父结点深——上一次扩展的则是这个父结点因为当时它最深。作为一个可行的TREE- SEARCH实现，通常使用调用自己的递归函数来实现深度优先搜索算法，可以依次对当前结点的子结点调用该算法。

深度优先搜索算法的效率严重依赖于使用的是图搜索还是树搜索。避免重复状态和冗余路径的图搜索，在有限状态空间是完备的，因为它至多扩展所有结点。而树搜索，则不完备，算法有可能会陷入的死循环。深度优先搜索可以改成无需额外内存耗费，它只检查从根结点到当前结点的新结点；这避免了有限状态空间的死循环，但无法避免冗余路径。在无限状态空间中，如果遭遇了无限的又无法到达目标结点的路径，无论是图搜索还是树搜索都会失败。

在无限状态空间深度优先搜索会令人尴尬地失败，而这个问题可以通过对深度优先搜索设置界限l来避免。就是说，深度为1的结点被当作没有后继对待。这种方法称为深度受限搜索( depth- limited search)。深度界限解决了无穷路径的问题。图3.17展示了该算法的伪代码。

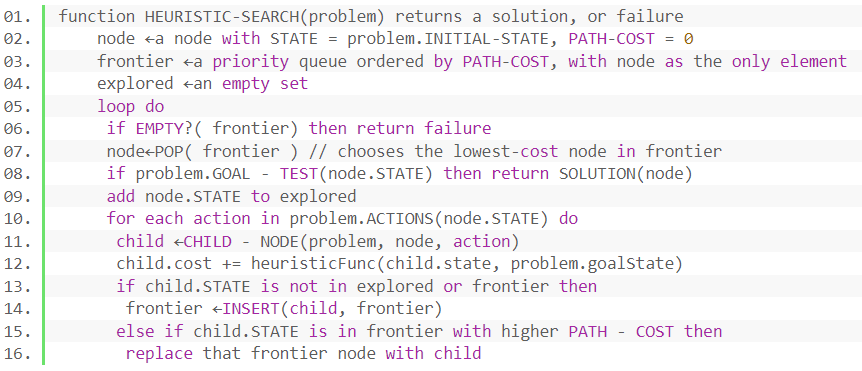


### 2.3 A\*算法

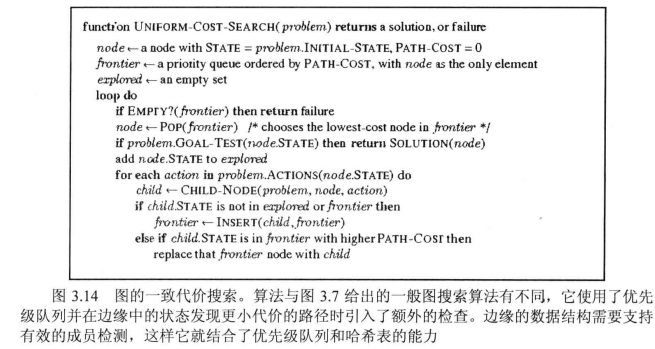
最佳优先搜索的最广为人知的形式称为A\*搜索（可以该为“A星搜索”）。它对结点的评估结合了g(n),即到达此结点已经花费的代价，和h(n),从该结点到目标结点所花代价：

由于g(n)是从开始结点到结点n的路径代价，而h(n)是从结点n到目标结点的最小代价路径的估计值，因此

这样，如果我们想要找到最小代价的解，首先扩展g(n)+h(n)值最小的结点是合理的。可以发现这个策略不仅仅合理：假设启发式函数h(n)满足特定的条件，A\*搜索既是完备的也是最优的。算法与一致代价搜索类似，除了A\*使用g+h而不是g。下图展示了A\*算法的伪代码。



对比一致代价搜索（伪代码如下图），可以发现A\*与之几乎相同，除了多了第12行。

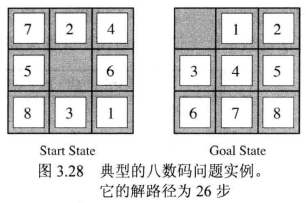


那么接下来，该算法的求解就只依赖于启发式函数的选择了。

一个随机产生的八数码问题的平均解步数是22步。分支因子约为3（当空格在棋盘正中间的时候，有四种可能的移动；而当它在四个角上的时候只有两种可能；当在四条边上的时候有三种可能)。这意味着到达深度为22的穷举搜索树将考虑大约个状态。图搜索可以把这个数目削减大约170000倍，因为只有个可达到的不同状态。这是一个容易管理的数目，但是考虑15数码问题，这个数目是大约103，因此我们需要找到好的启发函数。如果想用A\*算法找到最短解路径，我们需要一个绝不会高估到达目标的步数的启发式函数。15数码问题的启发式函数研究有很长的历史；这里有两个常用的：

#### 2.3.1 以misplace为启发函数

h1=不在位的棋子数。图3.28中所有的8个棋子都不在正确的位置，因此起始状态的h1=8。h1是一个可采纳的启发式函数，因为要把不在位的棋子都移动到正确位置上，每个错位的棋子至少要移动一次。



#### 2.3.2 以Manhattan距离为启发函数

h2=所有棋子到其目标位置的距离和。因为棋子不能斜着移动，计算距离指的是水平和竖直的距离和。这有时被称为市街区距离或曼哈顿距离。h2也是可采纳的，因为任何移动能做的最多是把棋子向目标移近一步。图3.28中起始状态的棋子1~8得到的曼哈顿距离为

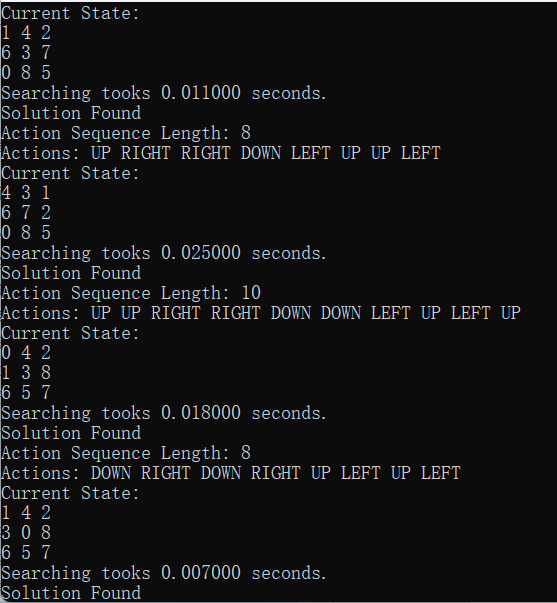
可以看到正如我们所希望的，这两个启发式函数都没有超过实际的解代价26。

## 3实验结果

通过进行反复迭代（50次，打乱30步）来得到实验数据。

### 3.1 宽度优先搜索

篇幅限制此处仅展示四组结果



对于50次迭代，总共耗时12.569s，并且全部都找到了解，以下是以上四组迭代的解的大小、时间使用和空间使用。

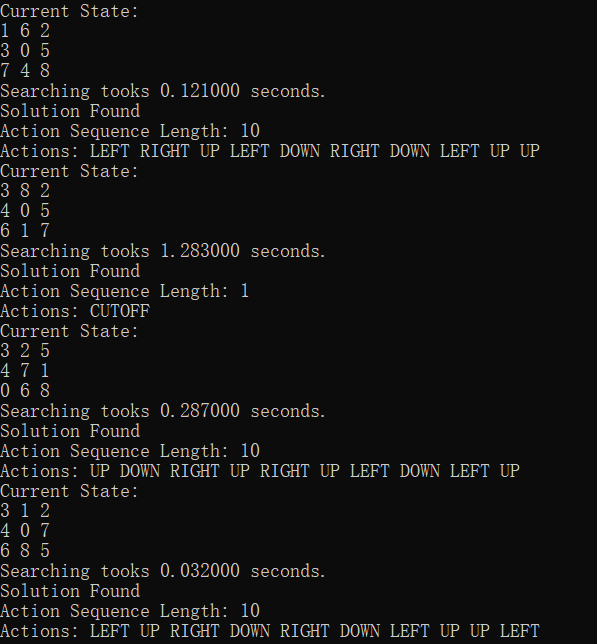


下面图是全部数据的汇总情况：



### 3.2 深度限制搜索

以下为部分迭代数据，可以看到深度限制搜索并不能完全找到解。



对于50次迭代，总共耗时19.791s，有10次未成功找到解，成功率为80%。其中，第四组为cutoff，



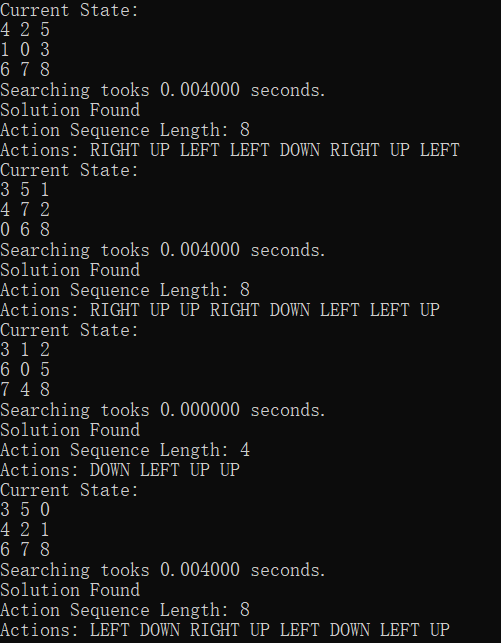
下面图是全部数据的汇总情况：

其中，solution size为1的皆为cutoff。



### 3.3 A\*算法

#### 3.3.1 以misplace为启发函数



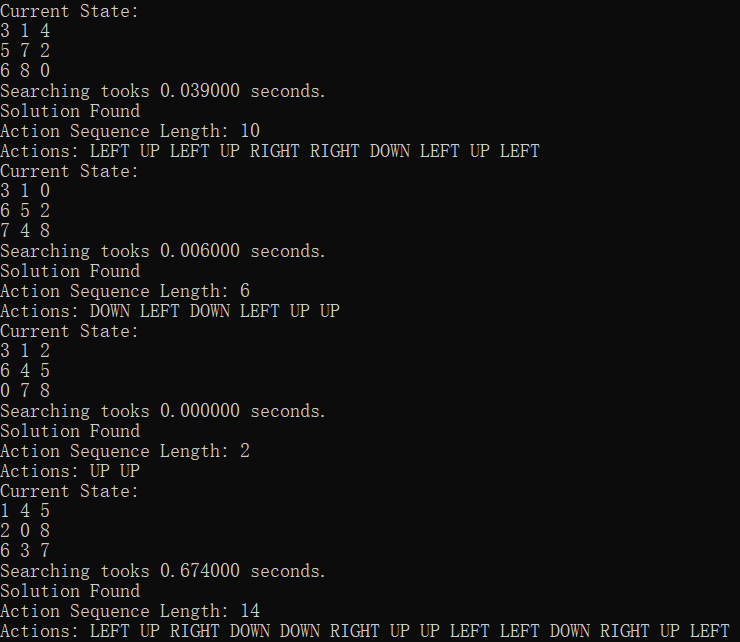
对于50次迭代，总共耗时7.867s，并且全部都找到了解，以下是以上四组迭代的解的大小、时间使用和空间使用。



下面图是全部数据的汇总情况：



#### 3.3.2 以Manhattan距离为启发函数



对于30次迭代，总共耗时10.31s，并且全部都找到了解，以下是以上四组迭代的解的大小、时间使用和空间使用。



下面图是全部数据的汇总情况：



## 4实验分析

### 4.1 宽度优先搜索

#### 完备性

很容易知道宽度优先搜索是完备的——如果最浅的目标结点处于一个有限深度d,宽度优先搜索在扩展完比它浅的所有结点（假设分支因子b是有限的）之后最终一定能找到该目标结点。目标结点一经生成，我们就知道它一定是最浅的目标结点，原因是所有比它的浅的结点在此之前已经生成并且肯定未能通过目标测试。最浅的目标结点不一定就是最优的目标结点；从技术上看，如果路径代价是基于结点深度的非递减函数，宽度优先搜索是最优的。最常见的情况就是当所有的行动要花费相同的代价。

#### 性能

到目前为止我们讨论的宽度优先搜索的性能都是好的方面。但是它在时间和空间耗费上却不好。假设搜索一致树( uniform tree)的状态空间中每个状态都有b个后继。搜索树的根结点生成第一层的b个子结点，每个子结点又生成b个子结点，第二层则有个结点。这些结点的每一个再生成b个子结点，在第三层则得到个结点，依此类推。现在假设解的深度为d。在最坏的情况下，解是那一层最后生成的结点。这时的结点总数为：

(如果算法是在选择要扩展的结点时而不是在结点生成时进行目标检测，那么在目标被检测到之前深度d上的其他结点已经被扩展，这时时间复杂度应为。)

空间复杂度：对任何类型的图搜索，每个已扩展的结点都保存在探索集中，空间复杂度总是在时间复杂度的b分之一内。特别对于宽度优先图搜索，每个生成的结点都在内存中。那么将有个结点在探索集中，个结点在边缘结点集中。所以空间复杂度为,即它由边缘结点集的大小所决定。即使转换为树的搜索问题也节省不了多大的存储空间，如果状态空间有重复路径的话，这种转换会耗费大量时间。

### 4.2 深度限制搜索

#### 完备性

如果我们选择了l<d,即是说，最浅的目标结点的深度超过了深度限制，那么这种搜索算法是不完备的。如果选择的l>d,深度受限搜索同样也不是最优的。它的时间复杂度是，,空间复杂度是。深度优先搜索可以看作是特殊的深度受限搜索，其深度。

#### 性能

深度优先搜索的时间复杂度受限于状态空间的规模（当然，也可能是无限的）。另一方面，深度优先的树搜索，可能在搜索树上生成所有个结点，其中m指的是任一结点的最大深度；这可能比状态空间大很多。要注意的是m可能比d(最浅解的深度)大很多，并且如果树是无界限的，m可能是无限的。

而空间复杂度。对图搜索而言，深度优先搜索只需要存储一条从根结点到叶结点的路径，以及该路径上每个结点的所有未被扩展的兄弟结点即可。一旦一个结点被扩展，当它的所有后代都被探索过后该结点就从内存中删除。考虑状态空间分支因子为b最大深度为m,深度优先搜索只需要存储个结点，大大节约了空间。

### 4.3 A\*算法

#### 完备性

在每一步的代价都大于等于某个小的正值常数的前提下，完备。

#### 性能

A\* 的时间复杂度取决于启发式函数。

在无界搜索空间的最坏情况下，数扩展的节点数在解的深度（最短路径）中呈指数增长 d: ，其中 b 是分支因子（每个状态的后继的平均数量）。这假设目标状态完全存在，并且可以从起始状态到达；如果不是，并且状态空间是无限的，算法将不会终止。

启发式函数对 A\* 搜索的实际性能有重大影响，因为一个好的启发式允许A\* 修剪掉许多不知情的搜索会扩展的 节点。它的质量可以表示为有效分支因子 b\* 的项，可以通过以下方式为问题实例凭经验确定测量展开的节点数 N 和解的深度，然后求解

好的启发式是那些具有低有效分支因子（最佳为 ）的启发式。

当搜索空间是一棵树时，时间复杂度是多项式的，有一个目标状态，启发式函数 h 满足以下条件：

其中 是最优启发式，即从 x 到目标的确切成本。换句话说，h的误差不会比返回从 x 到目标的真实距离的“完美启发式” 的对数增长得更快。

## 5 结论

宽度优先搜索可以找到最浅的节点，尽管最浅的目标结点不一定就是最优的目标结点。从技术上看，如果路径代价是基于结点深度的非递减函数，宽度优先搜索是最优的。最常见的情况就是当所有的行动要花费相同的代价。它的优点是对于解决最短或最少问题特别有效，而且寻找深度小每个结点只访问一遍，结点总是以最短路径被访问，所以第二次路径确定不会比第一次短。缺点是内存耗费量大（需要开大量的数组单元用来存储状态）。

深度优先搜索的优点是能找出所有解决方案而且由于它优先搜索一棵子树，然后是另一棵，所以和广搜对比内存需要相对较少。缺点是要多次遍历，搜索所有可能路径，标识做了之后还要取消，所以在深度很大的情况下效率不高。有时，甚至可能出现死循环。而用于改进的深度受限搜索则偶尔可能不完备。

A\* 算法优点在于对环境反应迅速，搜索路径直接，是一种直接的搜索算法，因此被广泛应用于路径规划问题。 其缺点是实时性差，每一节点计算量大、运算时间长，而且随着节点数的增多，算法搜索效率降低，而且A\* 算法并没有完全遍历所有可行解，所得到的结果不一定是最优解。

## 6 完整代码

BFS是已完成的，此处不给出。

DLS关键代码：

std::vector<Action> recursiveDLS(Node& node, Problem& problem, int limit)

{

if (problem.isGoal(node.state)) return getSolution(node);

else if (limit == 0) return { Action::CUTOFF };

else {

bool cutoff\_occurred = false;

for (auto action : problem.getValidActions(node.state)) {

auto\* child = new Node(childNode(problem, node, action, 1));

problem.nodePtrs.insert(child);

auto result = recursiveDLS(\*child, problem, limit - 1);

if (result[0] == Action::CUTOFF)

cutoff\_occurred = true;

else if (result[0] != Action::FAILURE)

return result;

std::cout << problem.nodePtrs.size() << "\n";

problem.freeMemory(child);

}

if (cutoff\_occurred)

return { Action::CUTOFF };

else

return { Action::FAILURE };

}

}

DLS关键代码：其中名字空间rng是std::ranges

std::vector<Action> aStar(Problem& problem, heuristicFunc heuristicFunc)

{

static auto get\_container = []<class Adapter>(Adapter & a)->typename Adapter::container\_type& {

struct hack : Adapter {

static typename Adapter::container\_type& get(Adapter& a) { return a.\* & hack::c; }

};

return hack::get(a);

};

Node\* initNode = new Node(NULL, Action::FAILURE, 0, problem.initState);

problem.nodePtrs.insert(initNode);

std::priority\_queue<Node\*, std::vector<Node\*>, Node::cmp> frontier;

auto& underlying = get\_container(frontier);

std::map<std::vector<int>, Node\*> stateNodePtrMap;

frontier.push(initNode);

stateNodePtrMap[initNode->state] = initNode;

std::set<std::vector<int>> explored;

while (true)

{

if (frontier.empty()) return { Action::FAILURE };

Node\* node = frontier.top();

frontier.pop();

stateNodePtrMap.erase(node->state);

if (problem.isGoal(node->state)) return getSolution(\*node);

explored.insert(node->state);

for (Action action : problem.getValidActions(node->state))

{

Node\* child = new Node(childNode(problem, \*node, action, 1));

problem.nodePtrs.insert(child);

child->cost += heuristicFunc(child->state, problem.goalState);

auto res = stateNodePtrMap.find(child->state);

if (res == stateNodePtrMap.end() && !inSet(explored, child->state)) {

frontier.push(child);

stateNodePtrMap[child->state] = child;

}

else if (res != stateNodePtrMap.end() && child->cost > res->second->cost) {

\*res->second = \*child;

stateNodePtrMap.erase(res);

stateNodePtrMap[child->state] = child;

rng::make\_heap(underlying, Node::cmp());

}

}

}

return { Action::FAILURE };

}

Misplace启发函数：

int heuristic::misplace(std::vector<int>& state, std::vector<int>& goalState)

{

int h1 = 0;

for (int i = 0; i < state.size(); ++i)

if (state[i] && state[i] != goalState[i])

++h1;

return h1;

}

Mahattan启发函数：

int heuristic::manhattan(std::vector<int>& state, std::vector<int>& goalState)

{ int h2 = 0;

for (int i = 0; i < state.size(); ++i)

if (state[i]) {

auto [x, y] = getPosition(state.size(), state[i]);

h2 += x + y;

}

return h2;

}