武汉大学计算机学院

本科生实验报告

基于 A*算法求解八数码问题的方案

专业名称：计算机科学与技术

课程名称：人工智能引论

学生学号：2018301040082

学生姓名：陈启明

二〇一九 年 11 月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：　　日期：

摘要

A^* 算法是一种在静态路网中给定目标求解最短路径的有效搜索方法，它是一种较常见的启发式搜索算法，其启发式函数表示为$f(n) = h(n) + g(n)$，常被用于游戏中 NPC 与 BOT 的移动计算。

为了加深对 A^* 算法的理解，发挥 A^* 算法在实际问题中的优势，在本实验中选取八数码问题为背景，将 A^* 搜索算法应用于寻求八数码问题的最优路径，体验 A^* 算法的优势与不足，对该算法有更直观与清晰的认识。

在本次实验中，为了深度理解 A^* 算法背后的详细算法流程，体会 Open 表与 Closed 的作用，将选取 C++作为编程语言，选取 Visual Studio 2017 作为集成开发环境，借助 C++中的 STL 库，对 A^* 算法求解八数码问题进行模拟。

关键词： A^* 算法，八数码问题，C++

目 录

1 实验目的

1.1 实验目的

A^* 算法是一种在静态路网中给定目标求解最短路径的有效搜索方法，它是一种较常见的启发式搜索算法，常被用于游戏中 NPC 与 BOT 的移动计算。

为了加深对 A^* 算法的理解，发挥 A^* 算法在实际问题中的优势，在本实验中选取八数码问题为背景，将 A^* 搜索算法应用于寻求八数码问题的最优路径，培养运用该算法解决实际问题的能力，体验 A^* 算法的优势与不足，从而对 A^* 算法拥有更深，更直观的理解。

2 实验设计

2.1 概述

2.1.1 或图搜索策略

根据图的实际背景可分为或图和与/或图两种， A^* 算法是或图搜索算法中的一种常见算法。

图算法只记录状态空间那些被搜索过的状态，它们组成一个搜索图叫 G。G 由两张表内的结点组成：

Open 表：用于存放已经生成，且已用启发式函数作过估计或评价，但尚未产生它们的后继结点的那些结点，也称未考察结点。

Closed 表：用于存放已经生成，且已考察过的结点。

结构 Tree,它的结点为 G 的一个子集。Tree 用来存放当前已生成的搜索树，该树由 G 的反向边组成。

或图的通用搜索策略如下：

设 S_0：初态 S_g：目标状态

1.产生一个仅由 S_0 组成的 open 表；

2.产生一空 closed 表；

3.如果 open 为空，失败退出；

4.在 open 表上按某一原则选出第一个优先结点，称为 n，将 n 放到 closed 表中，并从 open 表中去掉 n ；

5.若 n∈ S_g，则成功退出；解为在 Tree 中从 n 到 S_0 的路径，或 n 本身。

6.产生 n 的所有后继，将后继中不是 n 的前驱点的点构成集合 M，将其装入 G 作为 n 的后继,

7.对 M 中的元素 P 分别作两类处理：

7.1 若 P∉G，即 P 不在 open 表中也不在 closed 表中，则 P 加入 open 表，同时加入搜索图 G 中，对 P 进行估计放入 Tree 中。

7.2 若 P∈G，则决定是否更改 Tree 中 P 到 n 的指针。

8.转 3。

2.1.2 *A**算法

在或图通用搜索算法中，从 Open 表中按照某种原则选出第一个优先结点。如果从 Open 表中选取结点的策略不同，那么对应的搜索算法也不同。

在 A^* 算法中，Open 表将根据评估函数$f(n)$的值选取最佳者，这里的评估函数$f(n)$采用以下形式：

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

$g(n)$为从初始结点到当前结点的n的路径代价；

$h(n)$为当前结点n到目标结点的最小代价的估计值。如果该估计值不大于n结点到目标结点的实际代价，则称该算法为 A^* 算法。

算法选择具有最小 f 值的结点进行扩展。

需要指出的是，该算法对 h 函数的选取要求较高。如果$h(n)$估计过低，那么会扩展过多的结点，造成大量浪费；如果$h(n)$估计过高，则可能错过目标，无法找到最优解。$h(n)$越接近 n 到目标结点的实际代价，算法效率越高。

2.2 算法描述

2.2.1 状态的表示

在八数码问题中，状态可以被定义为八个数码的位置信息，每个整数的取值从 1 到 8，而数字 0 表示空白方块，每一个八数码的盘面代表每一个状态结点。为了适应 A^* 算法的需要，每个状态必须拥有启发式函数的信息，即$f(n)$，$g(n)$，$h(n)$的值。为了方便数码的移动，对于每一个状态结点，可额外增加空白块 0 的位置信息（比如以 zeroRow，zeroCol 表示空白块所在的行号和列号）。此外，为了输出路径的需求，需要记录最佳路径中每个结点的父结点，所以对于每个状态，其数据成员还应该加上一个指向父结点的指针域。

根据以上的分析，可以创建一个状态结点类，其数据成员如图 2-1 所示。与状态密切相关的数组与启发函数值被声明为私有成员，零行、零列、父结点等辅助信息被声明为公有成员。

```
#define ROW 3
#define COL 3
class State
{
private:
    int num[ROW][COL];
    int f;  //目标函数值
    int g;  //g函数，初始结点到当前结点的代价
    int h;  //当前节点到目标结点的估计值
public:
    State *parent;  //前驱节点，用于记录路径
    int zeroRow;
    int zeroCol;
```

图 2-1 状态类的数据成员

2.2.2 启发式函数的选择

启发函数$f(n) = g(n) + h(n)$。

$g(n)$为从初始结点到当前结点的n的路径代价，由于在八数码问题中，每一步的路径权值均为 1，故$g(n)$为结点 n 的深度。

$h(n)$为当前结点n到目标结点的最小代价的估计值。在这里，我们采用状态 n 到目标结点的曼哈顿距离，即所有数字当前位置以最短路径走到正确位置的步数之和（实现如图 2-2 所示）。

```
void State::setH(const State & goal)
{
    unordered_map<int, pair<int, int>> theMap;  //key为元素值，value为该数码的位置
    unordered_map<int, pair<int, int>> goalMap;
    for (int i = 0; i < ROW; i++)
        for (int j = 0; j < COL; j++)
        {
            theMap[this->num[i][j]] = pair<int, int>(i, j);
            goalMap[goal.num[i][j]] = pair<int, int>(i, j);
        }
    for (int i = 1; i < 9; i++)
    {
        int dx = abs(theMap[i].first - goalMap[i].first);
        int dy = abs(theMap[i].second - goalMap[i].second);
        h += (dx + dy);
    }
}
```

图 2-2 计算曼哈顿距离的实现

2.2.3 算法流程的伪代码

创建两个表，OPEN 表保存所有已生成而未考察的节点（且按照每个结点的 f 值从小到达排序，保证 f 值小的结点位于队首），CLOSED 表中记录已访问过的节点。算起点的估价值，将起点放入 OPEN 表。

```
while(OPEN!=NULL)
    {
        从 OPEN 表中取启发函数值 f 最小的节点 n;
        if(n 节点==目标节点)
            成功退出;
        for(当前节点 n 的每个子节点 X)
        {
            计算 X 的代价 g;
            if (X in OPEN)
            {
                if( X 的代价 g 小于 OPEN 表中结点的代价 g )
                    {
                        把 n 设置为 X 的父亲;
                        更新 OPEN 表中的代价值 g 以及启发函数值 f;
                    }
            }
            if (X in CLOSED)
            {
                if( X 的代价 g 小于 CLOSED 表中结点的代价 g )
                    {
                        把 n 设置为 X 的父亲;
                        更新 CLOSED 表中的代价值 g 以及启发函数值 f;
                        把 X 节点从 CLOSED 中删去放入 OPEN
                    }
            }
```

```
            if (X not in both)
            {
                把 n 设置为 X 的父亲;
                求 X 的启发函数值 f;
                并将 X 插入 OPEN 表中;
             }
        }//end for
        将 n 节点插入 CLOSED 表中;
        按照启发函数值 f 从小到大将 OPEN 表中的节点排序;
    }//end-while

    利用最终的目标结点的指向父亲的指针，输出路径
```

2.3 实验演示

2.3.1 实验平台

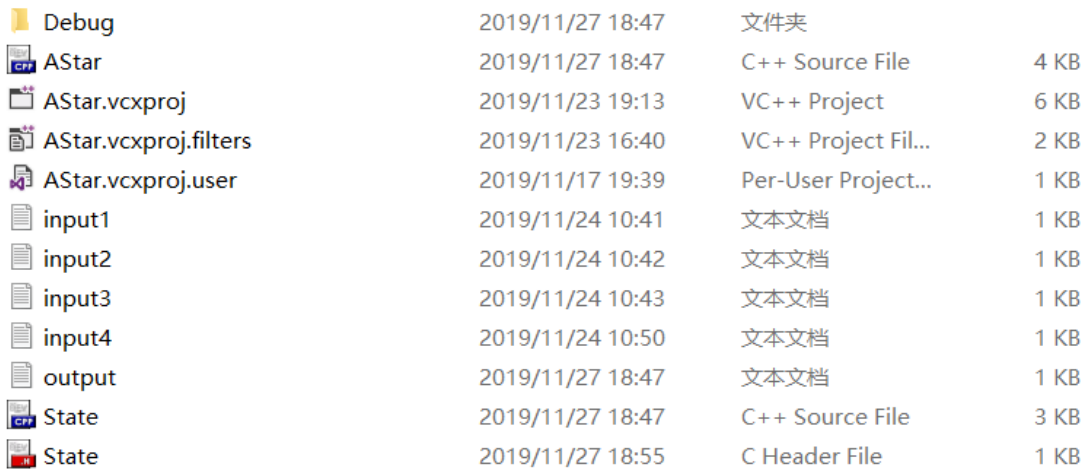
本实验采用的编程语言为 C++，实验平台为 Visual Studio 2017。

在模拟八数码问题的解决过程中，均采用 txt 文件输入输出。程序从文件中读取初始状态与目标状态，并且将解决路径输出到相应文件中。

2.3.2 项目结构

项目目录下，代码共分为三个文件。State.h 为状态节点类的定义文件，State.cpp 为类的实现文件，AStar.cpp 为主文件，负责算法的实现。其中 txt 文件负责算法的 I/O，算法从 input.txt 文件中读取输入，将输出结果置于 output.txt 中。

项目目录中的内容如图 2-3 所示，State 类的定义如图 2-4 所示。



Debug	2019/11/27 18:47	文件夹	
AStar	2019/11/27 18:47	C++ Source File	4 KB
AStar.vcxproj	2019/11/23 19:13	VC++ Project	6 KB
AStar.vcxproj.filters	2019/11/23 16:40	VC++ Project Fil...	2 KB
AStar.vcxproj.user	2019/11/17 19:39	Per-User Project...	1 KB
input1	2019/11/24 10:41	文本文档	1 KB
input2	2019/11/24 10:42	文本文档	1 KB
input3	2019/11/24 10:43	文本文档	1 KB
input4	2019/11/24 10:50	文本文档	1 KB
output	2019/11/27 18:47	文本文档	1 KB
State	2019/11/27 18:47	C++ Source File	3 KB
State	2019/11/27 18:55	C Header File	1 KB

图 2-3 项目目录下的文件

```
class State
{
private:
    int num[ROW][COL];
    int f;  //目标函数值
    int g;  //g函数，初始结点到当前结点的代价
    int h;  //当前节点到目标结点的估计值
public:
    State *parent;  //前驱节点，用于记录路径
    int zeroRow;
    int zeroCol;
    State(int a[ROW][COL]);
    //State();
    ~State();
    State(const State &state);  //复制构造函数
    void setH(const State &goal);   //计算当前状态的h，采用曼哈顿距离
    void setG(int depth);   //设置g，即深度
    void setF();        //更新启发函数值f
    bool operator==(const State &state) const;  //重载等于号，状态完全相同时则相等
    std::vector<State> getSuccessor();  //产生当前结点的后继
    void showState();
    int getF();
    int getG();
    int getH();
};
```

图 2-4 State 类的定义

2.3.3 运行结果

八数码的初始状态和目标状态存放在文本文档 input.txt 中，执行程序后，输出结果路径被存放到了 output.txt 中。

第一次测试的输入和输出如图 2-5 所示；第二次测试的输入和输出如图 2-6 所示。

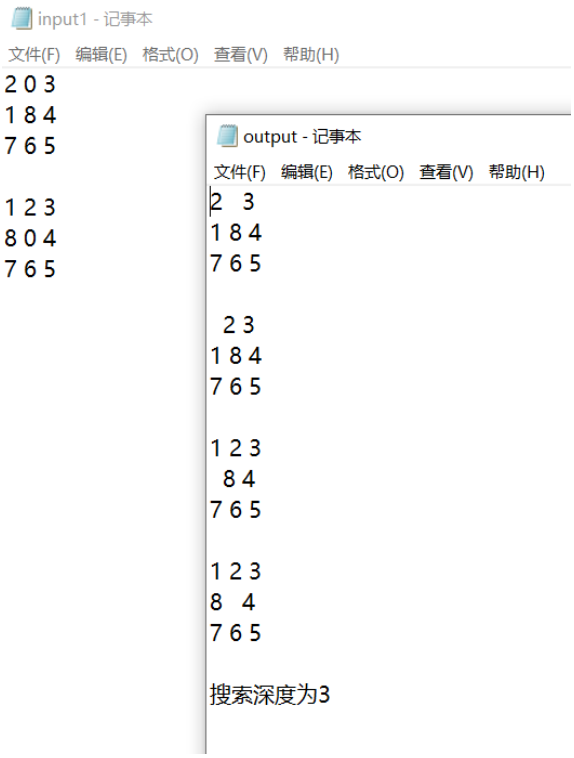


图 2-5 第一次测试的输入与输出

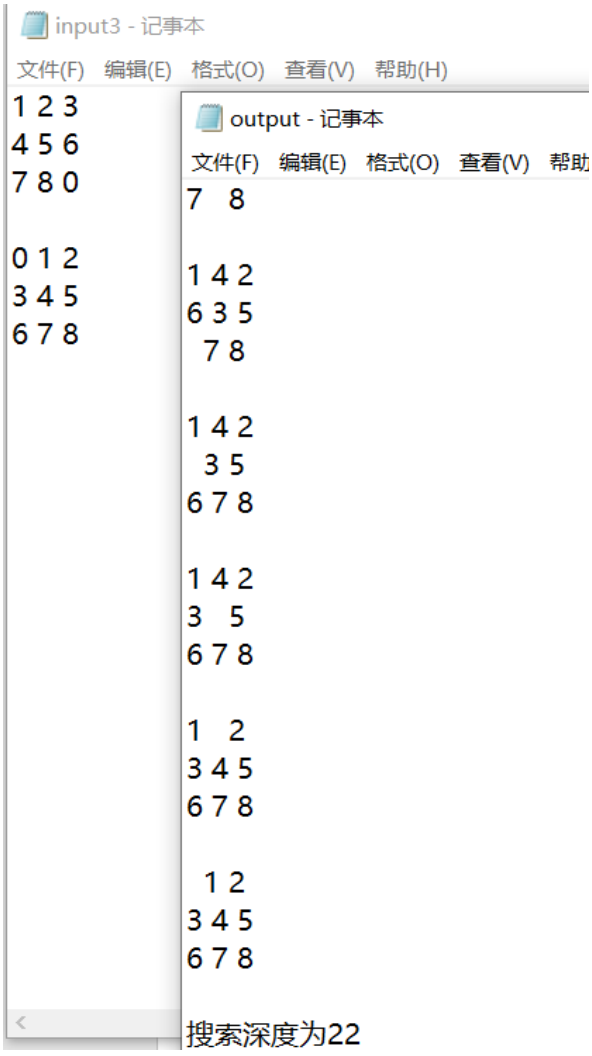


图 2-6 第二次测试的输入与部分输出（全部输出路径见录屏）

3 总结

3.1 实验中遇到的问题及解决方法

(1)关于 Open 表的实现，采用 STL 库中优先队列这一数据结构。由于优先队列只能从一端移除数据，所以如果要从 Open 表中删除或修改指定元素十分困难。这里采取的解决方案是创建一个和 Open 优先队列同步进行操作的 OpenTable，OpenTable 基于 vector 实现。

（2）为了方便从 Open 表和 Closed 表中找到指定状态的结点或删除指定状态的结点，需要额外设置相关函数（如图 2-7 所示）。值得注意的是，这里的等号运算符的意义已经被重载，表示两个状态数组完全相同。

```
void deleteFromTable(State &state, vector<State> &table)
{
    for (int i = 0; i < table.size(); i++)
    {
        if (state == table[i])
        {
            table.erase(table.begin() + i);
            return;
        }
    }
}

int findElementFromTable(State &state, vector<State> &table)    //从表中寻找指定元素，找到则返回索引值，没找到返回-1
{
    for (int i = 0; i < table.size(); i++)
    {
        if (state == table[i])
        {
            return i;
        }
    }
    return -1;
}
```

图 2-7 deleteFromTable：从表中删除指定元素；findElementFromTable：从表中寻找指定元素

（3）优先队列中的结点需要按照 f 值从小到大进行排列，为此需要 cmp 仿函数作为优先队列定义时的第三个参数。cmp 的具体实现如图 2-8 所示。

```
//仿函数，定义优先队列中的大小关系，即以f为标准进行比较
struct cmp {
    bool operator()(State a, State b) {
        return a.getF() > b.getF();
    }
};
```

图 2-8 cmp 的实现，采取大根堆

3.2 结论

A^* 算法一定能保证找到最优解，但效率和启发函数 h 密切相关。在本实验中，选取曼哈顿距离作为启发式函数 h 相对提高了搜索效率。

试验结果表明，程序的输出与预期相符。在该实验中，A^* 算法成功求解了八数码问题，找到了最优解，并用 C++进行模拟实现，以文件 I/O 的方式展示了结果。综上所述，实验达到预期目的，加深了对 A^* 算法的理解，熟悉了算法背后的整个流程，并成功解决了实际问题，培养了动手实践能力。

教师评语评分

评语：

评分：

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）