華中科技大學

人工智能导论
课程设计报告

选题名称：基于 A*算法求解八数码问题

课程名称：人工智能导论

专业班级：计算机科学与技术 2208 班

学　　号：U202215642

姓　　名：田清林

指导教师：冯琪

报告日期：2024.1.1

计算机科学与技术学院

摘要

N-数码问题一直是人工智能领域的一个基础问题。在众多的 n-数码问题中，最受欢迎的版本之一是 8-数码问题。它包括一个被分割成 3x3 网格的区域，其中包含 8 个（编号为 1-8）的方块和一个空的格子（编号为 0）。我们被给定一个初始状态，需要达到指定的目标状态。本文首先介绍了 8 数码问题及其问题表示，接下来分析了 A*算法的原理和启发函数的设计，简述了算法步骤和过程，之后讲述了程序设计，并进行测试。在这个项目中，本人使用了 A*算法并采用了各种启发函数，如错位的方块数、曼哈顿距离、线性冲突数目，并对三种启发函数进行了性能分析。编码实现过程中，本人用 Qt 编写了图形化界面，可以单步、多步、连续求解并使用 Graphviz 画出搜索树。最后写下总结与展望。

目录

1 问题描述与知识表示

1.1 问题描述

最常见的例子是 8-数码问题，它是一个 3x3 的棋盘，编号 1 到 8 的数字方块以及一个空白方块。玩家通过交换空白方块和相邻的数字方块来达到目标状态，通常是按照升序排列的顺序。本文中选择的目标状态为（1 2 3 4 5 6 7 8 0）

要求：

（1）至少定义 3 种不同的启发式函数，编程实现求解八数码问题的 A*算法；

（2）要求用可视化界面演示算法执行过程，应能选择预定义的启发式函数，能随机初始化初始状态，能单步执行，也能连续执行，能画出搜索树，同时标出估价函数在每个节点的各项函数值，能展示 OPEN 表和 CLOSED 表的动态变化过程；

（3）能统计出扩展节点数和算法执行时间，以便对采用不同启发式函数的 A*算法的性能做对比研究。

1.2 知识表示

使用状态空间表示八数码问题的状态空间。

9 个方块（8 个数码与 1 个空）的任意一种摆法就是一个状态，所有摆法就是状态集合 *S*，构成一个状态空间，大小为 9!。

要求实现的某个目标状态 $\boldsymbol{S_g}$：这里为 9 个方块按照行优先顺序排列为 1，2，3，4，5，6，7，8，0。

操作符 F：移动空白方块，共有 4 个，对应上下左右四个方向。为保证空格不会移动到方格盘以外，并非任何状态下均可使用这 4 个操作符。

该问题转化为求出某个合适棋子的走步序列，将初始棋局$\boldsymbol{S_0}$变换为目标棋局$\boldsymbol{S_g}$

2．算法设计与分析（A*算法）

2.1 算法的原理

2.1.1 A*算法介绍

图(树)搜索中，若搜索的每一步都利用估价函数：f(n)=g(n)+h(n)，对 OPEN 表中的节点排序，则该搜索算法为“A*算法”。

具有 f(n)=g(n)+h(n)策略的启发式算法，可成为 A*算法的充分条件为：

① 搜索树上存在从起点到终点的最优路径；

② 问题域是有限的；

③ 从任何一个节点转移到其子节点，代价>0；

④ h(n)≤h*(n)；

⑤ h 函数是相容(单调)的。

五个条件均满足，一个具有 f(n)=g(n)+h(n)策略的启发式算法即能成为 A*算法，并一定能找到最优解。

2.1.1 A*算法估价函数介绍

估价函数“f(n)=g(n)+h(n)” 可简单理解为：从初始节点 S0 到目标节点 Sg 所需耗费的总代价。

这个总代价可分成两个部分：

① 从初始节点 S0 到中间节点 n(搜索的中间状态)，已经消耗的实际代价 g(n)；（在这个问题为从初始状态到中间状态的操作步数）

② 对从中间节点 n 到目标节点 Sg 的代价（在这个问题为从中间状态到目标状态的操作步数），由于实际代价在求解出答案前很难求出，所以用预测代价---h(n)代替。

可以看出，估计值\预测值的准确性会直接影响求解效率及能否求得合理的解。

可以证明，在 g(n)相对固定情况下，若任意估计值均小于等于真实值，也就是 h(n)≤h*(n) ，即可认定最终解就是问题的最优解。

八数码问题可用 A*算法求解，需要注意以下几点：

① 每一步搜索均运用估价函数 f(n)=g(n)+h(n)，对 OPEN 表中的节点排序；

② 采用的搜索策略为“A 算法”的“全局择优/局部择优”；

③ 满足前述五个条件；（A 算法对 A* 算法的估计\预测）

④ 无论是选择“不在位的数码个数”、“离目标节点的剩余距离”…，都只是对启发信息(估计值\预测值)“h”选择的不同，但均是 A*算法的应用。

2.1.2 启发函数

本人选择了三个启发函数，分别为

1) 错位数码的个数。

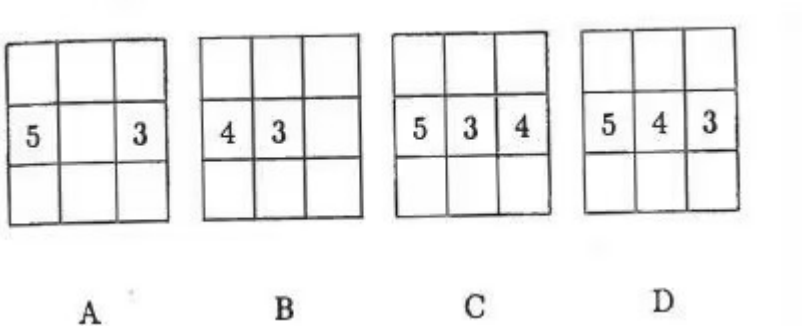
2) 各数码到目标状态对应数码的曼哈顿距离

3) 各数码到目标状态对应数码的曼哈顿距离+线性冲突个数

其中线性冲突的定义如下：

如果两块数码版都位于各自的目标行或目标列中，而且在曼哈顿启发式的相同设置中，它们必须互相越过对方才能到达最终目标位置，那么这两块牌就是线性冲突的。

如下图所示，A,B,C,D 都存在线性冲突，其中 A 图中 5 和 3 存在线性冲突，C 图中的 5 和 3、5 和 4 都存在线性冲突。



我们知道，实际上，数码版不可能真正滑过对方，需要额外的步骤才能消解线性冲突。曼哈顿距离并没有考虑这一问题，所以我们为曼哈顿距离加上了消解线性冲突所必要的步骤。例如 A 图中，我们加上额外的两步作为消解线性冲突的代价。

当某一行列中出现不止一个线性冲突时，我们会从参与线性冲突最多的一项开始消解线性冲突，直到不存在线性冲突。例如，在 C 图中，由于 5 同时与 3、4 发生冲突，只调整 5 的位置即可同时消解两个线性冲突，于是加上 2 步。

2.1.2 算法步骤及过程

1) 首先，将初始状态$\boldsymbol{S_0}$，根据选择启发函数的不同算出新状态的 g(n)与 h(n)值，放入 OPENLIST。
2) 每次取出 OPENLIST 中估价函数 f(n)=g(n)+h(n)最小的，并对其执行适当的操作符，产生新状态$\boldsymbol{S_n}$，并根据选择启发函数的不同算出新状态的 g(n)与 h(n)值放入 OPENLIST，已经执行过操作的状态放入 CLOSEDLIST。
3) 继续，直到产生目标状态$\boldsymbol{S_g}$为止。

注： 由于启发函数 h(n)的选择满足要求，所求出的必然为最优解，即操作数最小。

3. 系统详细设计

3.1 有关数据结构的定义

3.1.1 程序设计中使用的数据结构：

表3-1 数据结构定义表

数据类型	该类型包含的数据	简述作用

Node	int dis;//当前步骤数g(n)
	int f;//启发函数值h(n)
	int now;//空格当前所处的位置
	int state;//当前状态的康托值
		存储状态结点

PATH	int dis;//此时步骤数g(n)
	int f;//启发函数值h(n)
	int k;//下一个编号，编号为0则为终点
	int op；//操作数，1234分别对应l,r,u,d
		存储求解路径

3.2 主要算法设计

3.2.1 可解状态的查找

分辨 8 数码板的可解性可以求出 1-8 数码板（不包括空白板）的逆序对数量，若逆序对数量为偶数则可解，否则不能解。

编程实现为 check(int contor)函数，并在随机生成 0-8 排列后检查是否可解，若不可解进行调整，直到其可解。

3.2.2 康托展开

对于数码版各状态，利用康托展开可以将其映射到 1 到 362880 的正整数，便于存储。还定义了 incontor 函数，可以将 contor 值再转换回 9 个元素的数组。

3.2.3 Visit 存储数码板状态

程序执行过程中，用 visit 数组存储状态康托值为 i 的数码板状态，分别为未访问 UNVISIT、在 Open list 内 OPEN、在 Closed list 内 CLOSED、最终答案 ANS 状态，便于算法实现。在每次执行算法时重置。

4. 系统测试

4.1 系统实现

1.硬件环境：

处理器：AMD Ryzen 7 6800H with Radeon Graphics　3.20 GHz

机带 RAM：16.0 GB (15.2 GB 可用)

系统类型：64 位操作系统，基于 x64 的处理器

2：软件环境：

环境：QT Creator 4.11.1，编码：UTF-8

4.1.1 头文件的定义

```
#ifndef HEAD_H
#define HEAD_H

#define UNVISITED 0 // 标记结点状态
#define OPEN 1      // 在 OPENLIST
#define CLOSED 2    // 在 CLOSEDLIST
#define ANS 3       // 是最终答案路径上的结点
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <iostream>
#include <queue>
#include <vector>
#include <windows.h>

using namespace std;
extern const int tar; // 目标康托值
extern int open_num, closed_num;
extern int path_length;
extern int op;
extern unsigned char visit[362881]; // 标记各状态的结点的状态
extern char p[362881];
struct node {
  int dis;   // 步骤数
  int f;     // 启发函数值
```

```
  int now;   // 空格当前所处的位置
  int state; // 用康托值表示当前状态
  bool operator<(const struct node b) const { return dis + f > b.dis +
b.f; }
};
struct PATH {
  int dis; // 此时
  int f;   // 启发函数值
  int k;   // 下一个编号，编号为 0 则为终点
  int op;  // 操作类型
};
extern struct PATH Path[362881];      // 记录路径
extern priority_queue<struct node> q; // 优先队列
bool check(char *s, int now);         // 检查是否有答案
int randomize(char *s);               //
void solve(int op);
int Contor(char *a);
void incontor(char *res, int x);
void swap(int &a, int &b);
int linear_conflict(int contor);//返回线性冲突应加的步数
int Mdis(int x, int y, int i);
void reset();
void printTree();
#endif
```

4.1.2 主要函数及函数功能的说明：

算法的主体为 solve 函数，在实际 qt 程序中对 solve 进行了修改以便进行与窗口的交互。

执行过程中，数码板状态结点放入优先队列 q，每次取出 f(n)= g(n)+ h(n)最小的值，进行执行各操作符，若操作后的结点未放入 CLOSEDLIST，则将新结点入队，否则忽略该结点。当操作后的结点康托值等于目标康托值时说明找到了解，此时循环停止。可以输出结果。

```
void solve(int op) {
  struct node temp;
  int now;
  int c; // contor
  int cc;
  int x, y;
  int f;
```

```
int d[4] = {-1, 1, -3, 3};
char di[4] = {'l', 'r', 'u', 'd'};
int dir[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
const int tar = 46234; // 目标康托值
while (!q.empty()) {
  temp = q.top();
  q.pop();
  if (visit[temp.state] !=
      CLOSED) { // 队列中同时存在 OPEN，将 open list 内结点移入 CLOSED list
    visit[temp.state] = CLOSED;
    closed_num++;
    open_num--;
  } else
    continue;
  char state[9];
  incontor(state, temp.state);
  cc = temp.state;
  now = temp.now;
  for (int i = 0; i < 4; i++) // 四个方向左，右
  {
    x = now % 3 + dir[i][0];
    y = now / 3 + dir[i][1];
    if (x < 0 || x > 2 || y < 0 || y > 2)
      continue;                          // 越界
    swap(state[now + d[i]], state[now]); // 交换位置
    c = Contor(state);                   // 记录康托值
    swap(state[now + d[i]], state[now]); // 换回来
    if (visit[c] == CLOSED)              // 已经进入 CLOSED 则不再入队
      continue;
    switch (op) { // 根据选用启发函数不同,更新启发函数的值
    case 1:       // 不在正确位置的数量
      if (state[now + d[i]] == now + d[i] + 1)
        f = temp.f + 1;
      else if (state[now + d[i]] == now + 1)
        f = temp.f - 1;
      else
        f = temp.f;
      break;
    case 2: // 曼哈顿距离
      f = temp.f - Mdis(x, y, state[now + d[i]] - 1) +
          Mdis(now % 3, now / 3, state[now + d[i]] - 1);
```

```
        break;
      case 3: // linear_conflict
        f = temp.f - Mdis(x, y, state[now + d[i]] - 1) +
            Mdis(now % 3, now / 3, state[now + d[i]] - 1) -
            linear_conflict(cc) + linear_conflict(c); // dis 更新
        break;
      default:
        printf("error!\n");
        return;
      }
      if (!visit[c]) // 没有访问过
      {
        Path[c].dis = temp.dis + 1;
        Path[c].f = f;
        Path[c].k = cc; // 上一个
      }
      if (c == tar) {
        int j = 0;
        for (int i = c; i; i = Path[i].k) {
          visit[i] = ANS;
          j++;
        }
        path_length = j - 1;
        return;
      }
      q.push({temp.dis + 1, f, now + d[i], c});
      if (!visit[c]) {
        visit[c] = OPEN;
        open_num++;
      }
    }
  }
  printf("error");
  return;
}
```

4.2 程序测试

打开可执行文件 eight.exe，首先弹出选择启发函数的对话框。

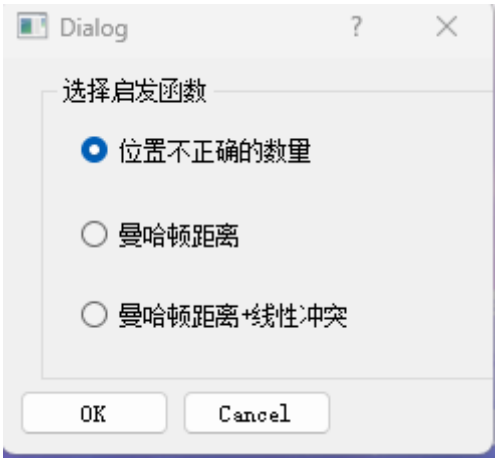


图 4-1 选择启发函数的对话框

这里先选择第一个启发函数，即错位方块数量，进入主界面。

图 4-2 主界面

按下随机初始化按钮，会随机初始化出一个可解的 8 数码问题。

图 4-3 随机初始化

单步执行求解程序，右边 OPEN LIST NUM、CLOSED LIST NUM 等更新。

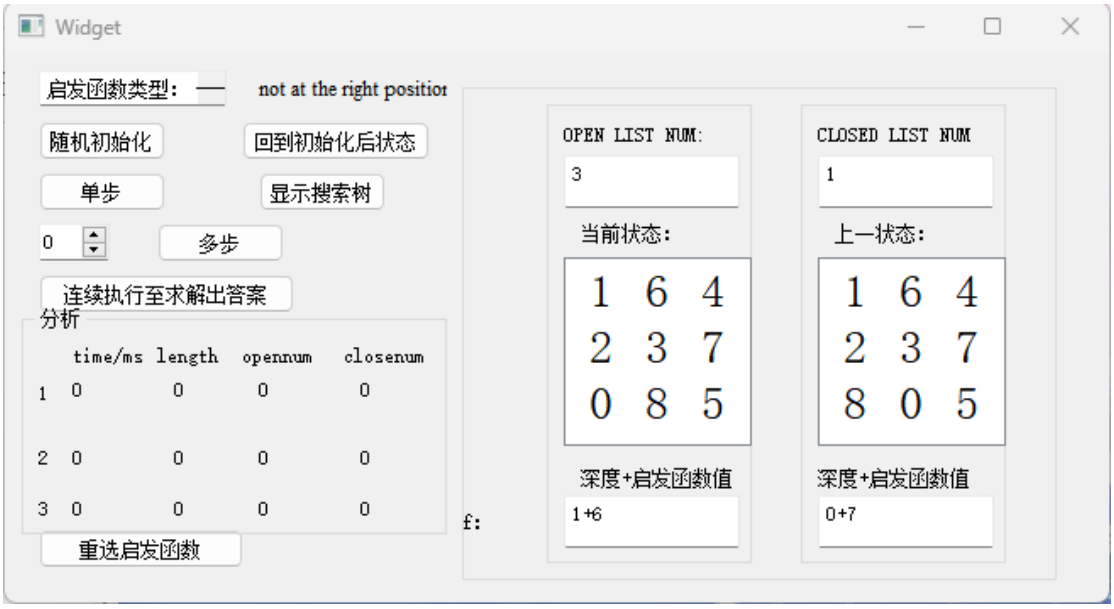


图 4-4 单步执行

左边选择多步执行的步数，点击多步执行按钮，这里选择连续执行三步。

图 4-5 多步执行

点击连续执行至求解出答案，弹出对话框。

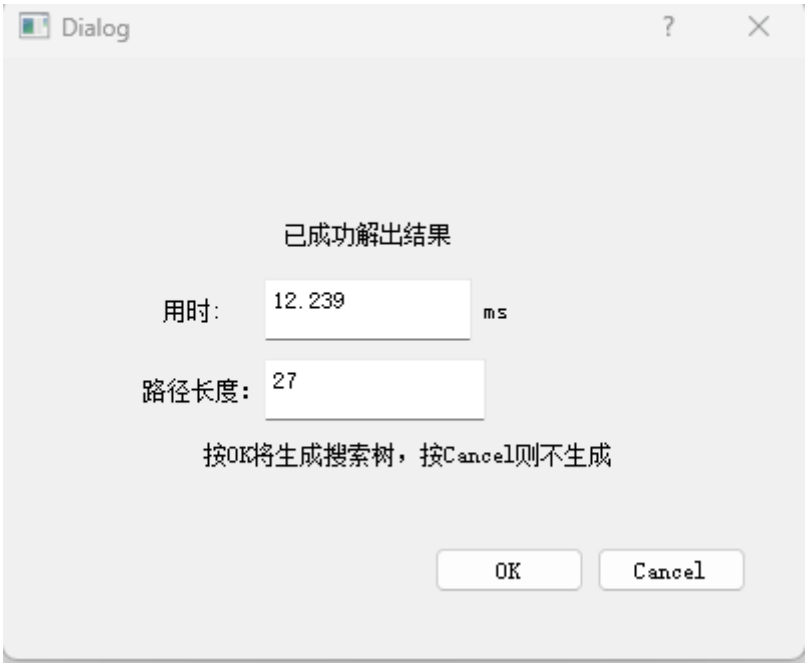


图 4-6 连续执行至求解出答案

之后重新选择启发函数，这里选择启发函数 2（曼哈顿距离）。

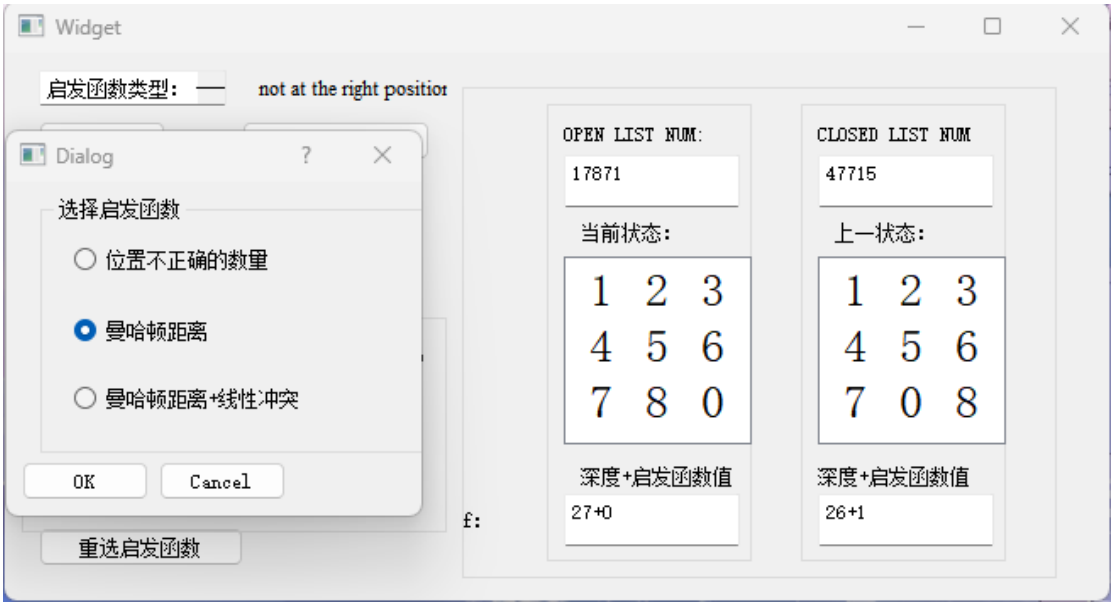


图 4-7 选择启发函数 2

选择启发函数后会自动回到随机化初始后的位置，点击连续执行至求解出答案，弹出带有结果信息的对话框。

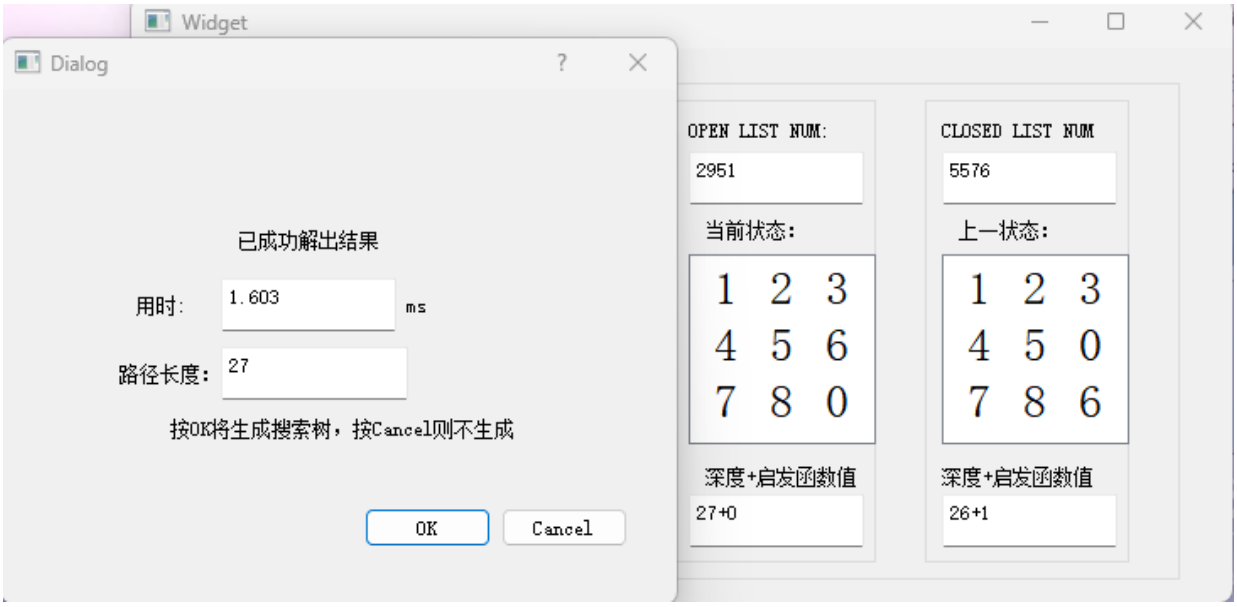


图 4-8 连续执行至求解出答案（2）

点击确定生成搜索树，并点击查看搜索树。部分搜索树如下。

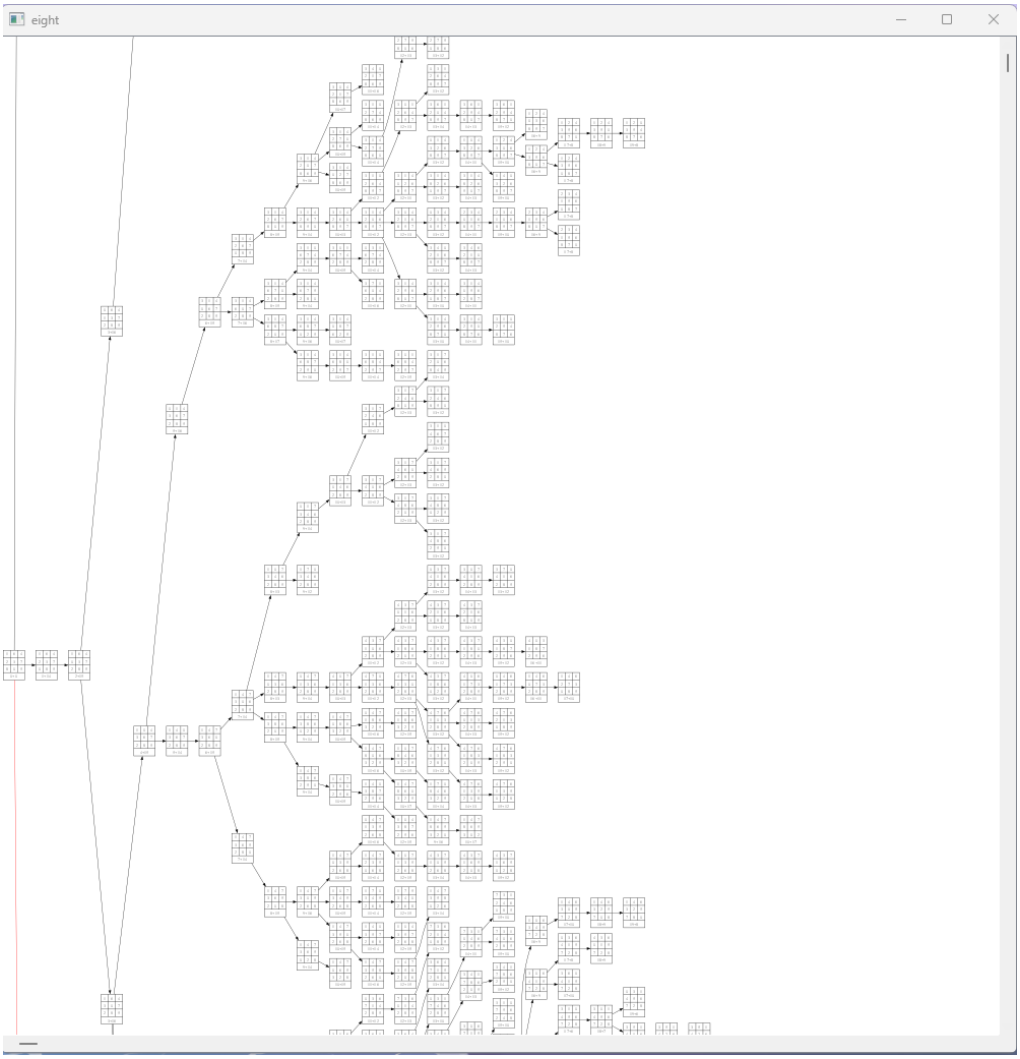


图 4-9 查看搜索树（粗略）

点开可执行文件同文件夹下的 Tree.png 文件，可以看到完整的搜索树，一下为搜索树部分截图，其中答案路径用红色标出。

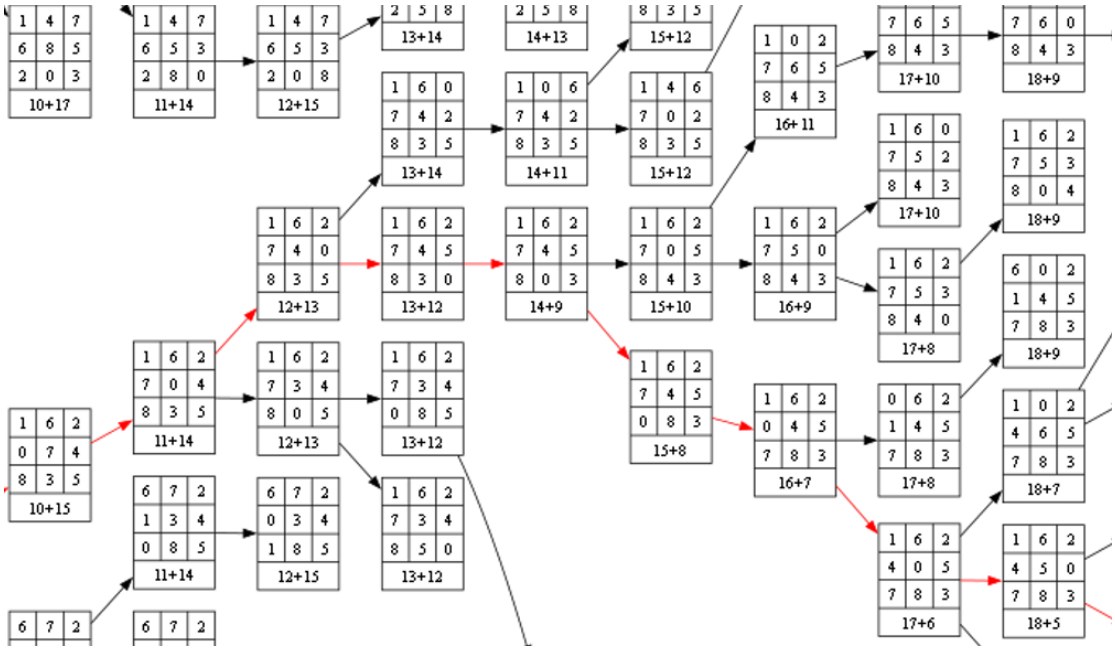


图 4-10 搜索树部分截图

再次换成启发函数 3，点击连续执行至求解出答案。

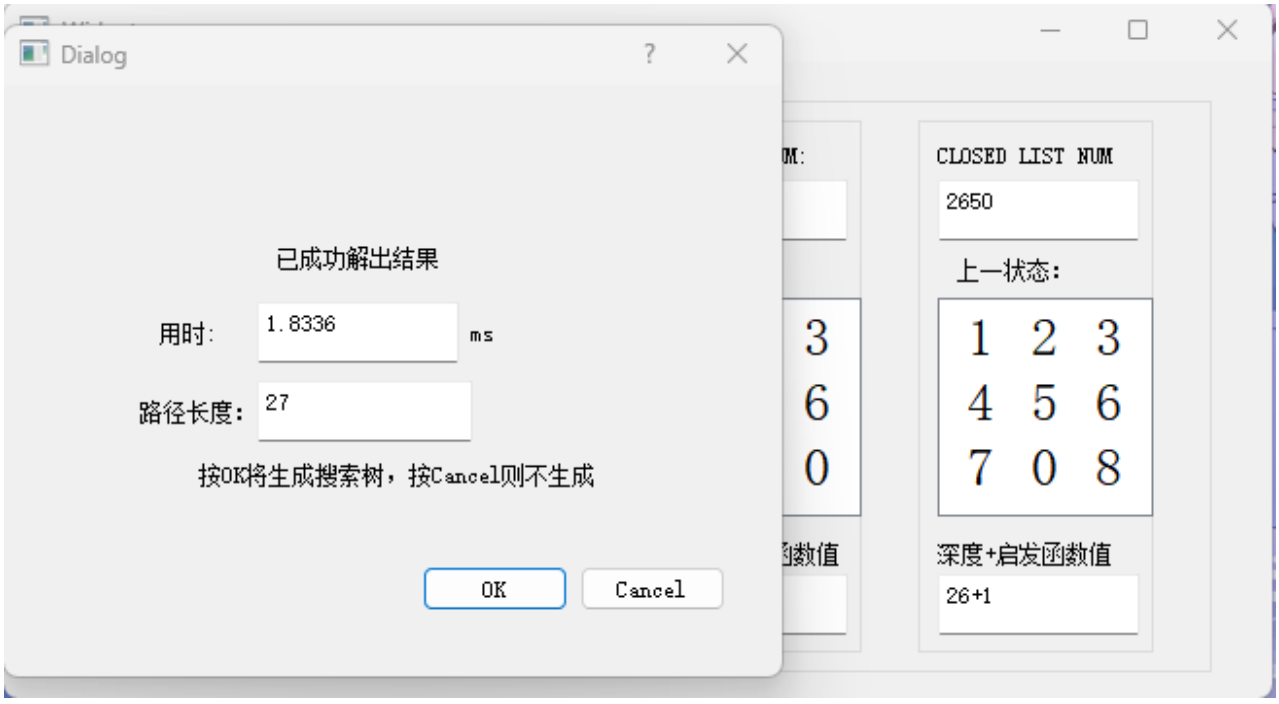


图 4-11 连续执行至求解出答案（3）

可以看到窗口左下存储了三个启发函数的执行信息。

图 4-12 三个启发函数的性能分析

点击“回到初始化状态”按钮，可以看到右边回到了初始化后的状态，随时可以再次求解。

图 4-13 回到初始化后状态

4.2.1 3 种启发函数的性能对比

测试数据：随机生成的可解的八数码问题。

表 4-1 3 种启发函数性能测试

Misplaced tiles				Manhattan distance				Linear conflict			
Time/ms	Path length	Open list	Closed list	Time/ms	Path length	Open list	Closed list	Time/ms	Path length	Open list	Closed list
0.1317	15	214	333	0.0356	15	58	80	0.766	15	46	69
0.3652	17	613	959	0.0698	17	110	170	0.0923	17	87	121
1.4141	21	2630	4599	0.1442	21	232	368	0.1388	21	121	205
2.1025	22	708	1239	0.4648	22	708	1239	0.6804	22	407	680
6.6044	25	11497	24866	0.647	25	1253	2328	0.7729	25	452	834
7.2011	26	12891	27589	0.3836	26	659	1110	0.4989	26	364	560
9.4265	26	15382	37028	1.0987	26	1819	3334	1.1579	26	873	1530
12.4681	27	18743	49090	0.4087	27	772	1318	0.6284	27	442	700

根据测试结果可以发现：

1.使用错位数码版个数的启发函数性能明显差于另外两种启发函数(用时更长，扩展结点数更多，OPENLIST 和 CLOSEDLIST 中结点更多)。

2.曼哈顿距离启发函数扩展结点数较线性冲突启发函数更多，OPENLIST 和 CLOSEDLIST 中结点更多，但一般用时更短。

原因分析：

1.首先，可以看出三种启发函数的大小关系有：Misplaced tiles < Manhattan distance < Linear conflict，而且三者都小于实际距离，可以保证求得最优解。一般来说启发函数越值越大，启发效果越好，扩展结点数更少。

2.启发函数的更新成本上：Misplaced tiles 需要 O(1)，Manhattan distance 需要 O(1)（这里每次操作后只改变移动的数码版的曼哈顿距离），Linear conflict 需要 O(n^3)，Linear conflict 远大于前两者，所以即使 Linear conflict 启发函数的扩展结点数更少，更多情况下它的用时仍大于 Manhattan distance。

部分测试运行截图。

图 4-14 性能测试图 1

图 4-15 性能测试图 2

图 4-16 性能测试图 3

图 4-17 性能测试图 4

5. 总结与展望

5.1 全文总结

对自己的工作做个总结，主要工作如下：

（1）独立完成了课程设计所有的代码编写，完成了课程设计作业。这个版本已经数不清是第几个版本了，一次次的迭代是为了更好的性能和体验。

（2）学习了 Qt 并运用 Qt 绘制了图形化界面。Qt 是本人一直想要学习而苦于没有时间，导致很多课设都只能用黑色的运行界面来解决。第一次学习并使用 Qt，虽然界面较为简陋，但也基本满足了需求。

（3）学习 Graphviz 的使用和 dot 语言的编写。画出一个如此庞大的搜索树确实让我苦恼了一阵子，但是后来了解到了 Graphviz 并学习如何使用它，解决了我的问题，画出的搜索树也很精美。

5.2 工作展望

1) 对 A*算法类似的启发式算法的了解与学习。不同于普通算法，启发式算法是一个很有趣的领域，早在做课程设计求解 sat 问题时我就学习并实现了 CDCL 算法。在学习 n 数码问题的研究过程中，我通过各种渠道了解到了更多求解的知识与方法，例如使用“pattern database heuristics”求解 n 数码问题，这激起了我浓厚的兴趣，也正是这份兴趣让我选择了这个选题。以后我会多了解学习这些富有趣味的内容。
2) 加强对人工智能知识的学习，学习机器学习、深度学习等内容。2023 年人工智能在世界掀起了波澜，chatgpt 发布后，各科技公司争相进入了 AI 大战，国内和国际上生成式 AI 大规模爆发。我们作为计算机专业的学生，应当积累必要的人工智能知识，并进行充分实践，才可能抓住时代机遇，在新时代发挥自己的力量。