**实 验 报 告**

**课程名称：** 人工智能实验

**实验项目：** **八数码的搜索策略**

**实验仪器：** PC机

**学 院： 计算机学院**

**专 业： 大数据**

**班级姓名： 大数据1701班 张丹颖**

**学 号： 2017011760**

**日 期： 2019年6月14日**

**指导教师： 蒋玉茹**

**同组成员：**

**成 绩：**

**一 实验问题描述** 本实验是为了让学生对于人工智能算法有进一步的认识，了解盲目搜索策略和启发式搜索策略的基本原理，真正理解机器智能的原理与方法，掌握相关的人工智能实现的技术路线和工具方法。本实验学生将实现一个八数码问题。**二 实验目的** 在解决八数码问题中，深入体会盲目搜索策略和启发式搜索策略，能够运用编程语言实现典型的搜索算法。**三 实验要求** 1. 能够利用交互手段动态设置实验相关参数 2. 能够利用可视化手段展示搜索策略的动态过程 3. 实现以下搜索策略 a) 宽度优先搜索 b) 深度优先搜索 c) 有界深度优先搜索 d) 代价树的宽度优先搜索和深度优先搜索（假设节点之间转移代 价为目标节点的深度） e) 局部最佳优先搜索和全局最佳优先搜索（可利用教材中的估价函数）**四 实验报告**

**注：以下所有程序编码方式**GBK!

1.描述系统实现所需的技术 技术？暂时还不会什么高深的技术。我用到的编程语言是C++，IDE是Clion, 会一些基本的dfs,bfs 算法。

2.描述各个算法和对应的程序 分析：八数码实际上是一个图的最短路问题（图是有向的状态空间图）。节点就是九个格子的状态编号。

①宽度优先搜索 具体代码见dfs中main.cpp

**思路**：State类型的大数组st 存取所有的八数码状态，为此数组设置头指针front和尾指针rear;

bfs搜索过程中，先找到0的位置，分别将0上下左右移动，若移动合法，拓展新节点，将新节点加入set集合，这里用到一个技巧，set集合可以判定重复状态（函数try\_to\_insert（int）），将八数码的九个数转换为一个10进制数存到set中。维护父亲数组fa。移动头尾指针。

根据fa数组，从下到上打印节点。

**运行结果：样例1：**

**2 8 3 1 0 4 7 6 5**

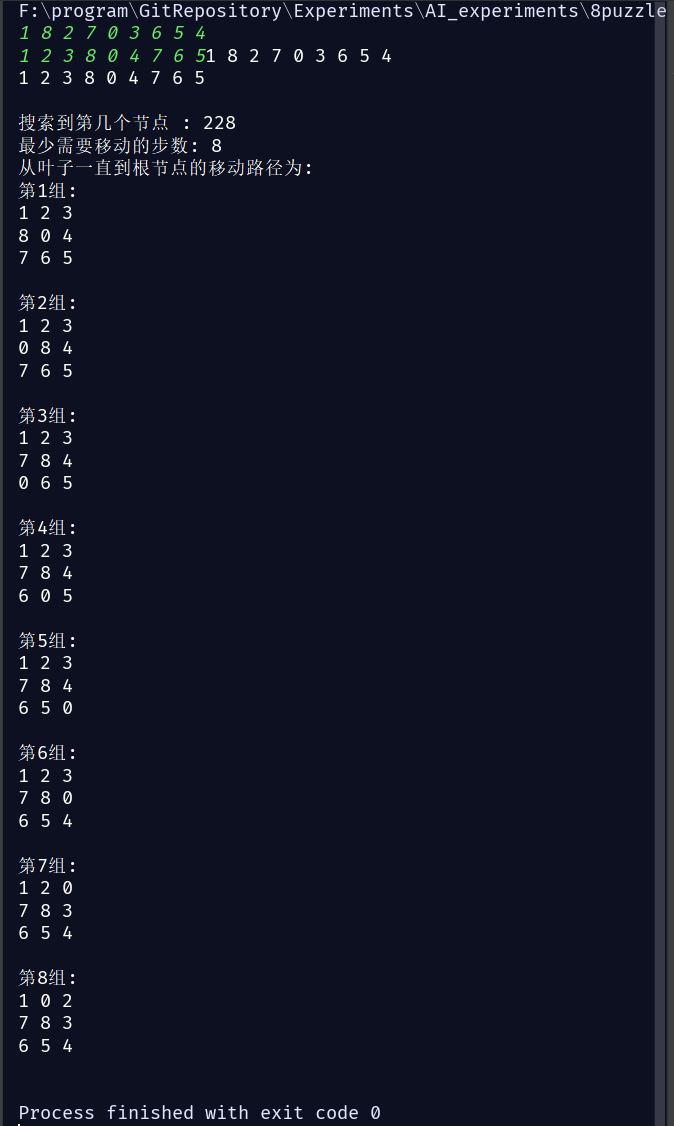
**1 2 3 8 0 4 7 6 5**



**样例2：**

**1 8 2 7 0 3 6 5 4**

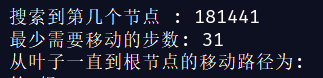
**1 2 3 8 0 4 7 6 5**



**样例3：**

**1 8 2 7 0 3 6 5 4**

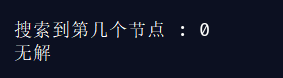
**1 2 3 8 0 4 7 6 5**



**样例4：（无解）**

**1 2 3 4 5 6 7 8 0**

**1 2 3 4 5 6 8 7 0**



**小总结：**

**遇到困难和解决如下：**

* typedef int State[9] 自定义数据类型typedef的使用。

开始想用一个二维数组存取八数码，但是，还要存取每个数码的状态，已经上升到三维数组。受到刘汝佳的启发，进行数组降维，利用newz = newx \* 3 + newy逆运算的技巧：x = z/3, y = z%3，将把数码的二维数组降为一维，减少了一个维度，同时结合State自定义类型，代码简洁明了不少。

* bfs判重，避免同一个节点访问多次

不可能声明一个vis(s[1] s[2] s[3] s[4] s[5] s[6]……)九维数组去判重，九维素组里面每一个都都还包含着九个元素占据很大空间且浪费内存，开始的想法是竞赛中最常用的最高效的解码编码方式，但是能力有限，没写出来。转而换为STL封装好的set集合判重，将九个数字转换为10进制数，解决问题。

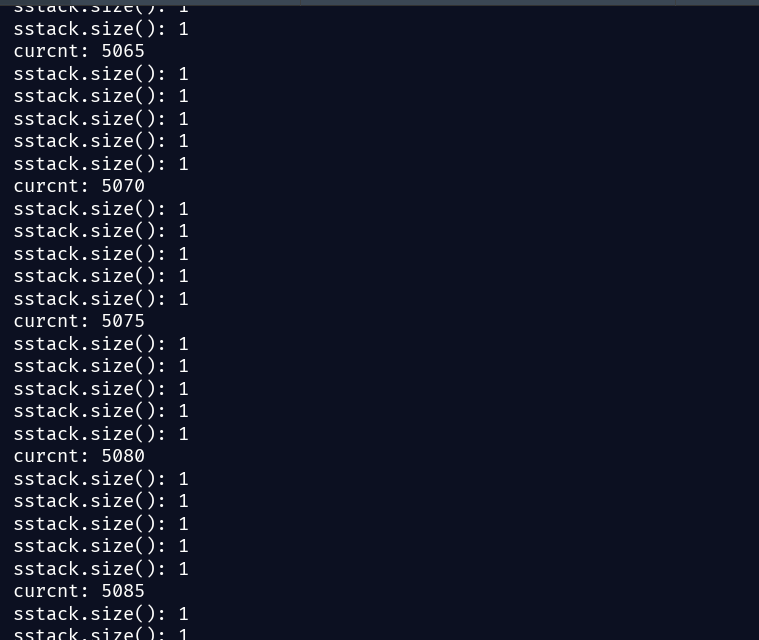
* tips ：使用ctring 的memcmp 和memcpy 完成整块内存的比较和复制，比循环比较和循环复制快，但要注意参数和括号返回值，之前掉了一个括号，bfs死循环，调试了好久……

②深度优先搜索  **思路：**大题思路和前面相同，只不过取节点时运用到了栈，每次都拓展当前节点，当找到目标，或者当前节点上下左右移动都不合适，就跳出递归。无界深度优先可能一直深入递归得不到结果.

**测试样例** local.txt

2 8 3 1 0 4 7 6 5

1 2 3 8 0 4 7 6 5

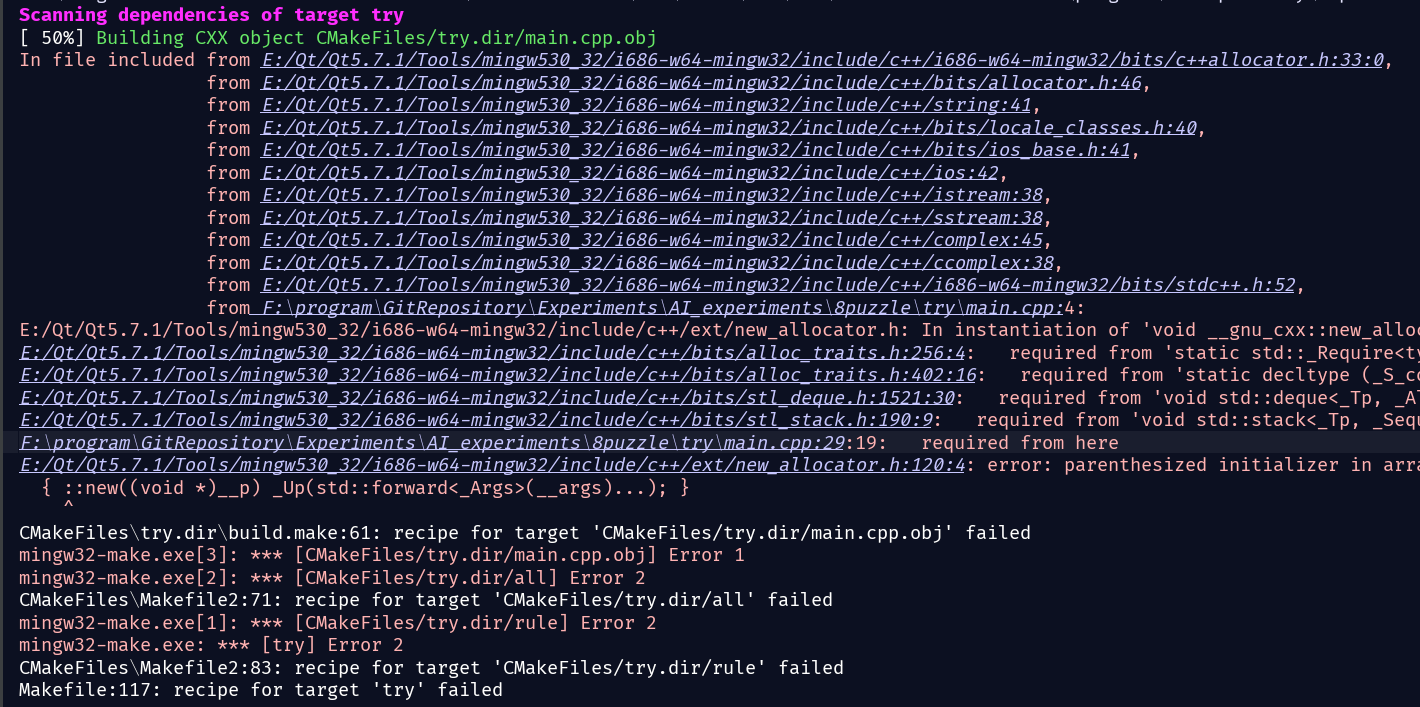
中间打印结果

一直往深了递归，出不来结果。

结论：不应该用没有代价的深度优先，极有可能一直循环直到爆栈。

遇到的困难和解决：

* 前面是采用一维数组，front 和rear 模拟的队列进出，但是本题中，我并没有模拟栈，而是直接采用自己定义的State 类型数组：typedef int State[9], State 去定义一个八数码方块，但是，编译器不允许在stack 内放入这样一个类型，会报错。



只好采用 结构体。

* dfs结束后并不停止递归回调问题

深度优先找到目标节点后，并不会终止回调递归，即找到节点后并没有结束整个程序。这里我加入一个flag 标记，只有在没找到节点的情况下，才继续做dfs

* 从文件读取数据遇到问题（对IDE不熟悉）

多次测试频繁输入输出数据比较麻烦，我准备设置从文件读取。开始没有想到，会在文件读取处有任何困难。结果不正确，以为是我代码的问题，自己检查代码，调试了好久，但是没发现问题，后来打印输入输出结果时，发现输进去的数据全是0，根本没读进去数据。查资料发现是IDE的工作区并不在当前.cpp 文件的工作区，导致找不到数据。

（我很生气的是，找不到数据，IDE也不报错T\_T啊啊啊啊）

③有界深度优先搜索

思路：用一个大的State 类型数组path存储深度沿途经过的每个节点，深度优先这里不使用递归，而是手动栈sstack;

栈不为空时，取出来的节点放入到path 里面。

若取出来的当前节点s是目标节点，就根据path 内目标节点的father节点位置，在path内跳跃性往上找，打印沿途节点必定是路径。

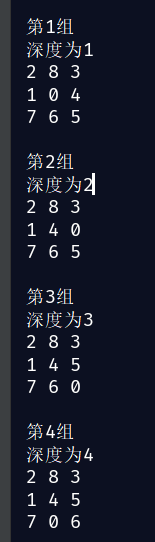
若取出来的不是目标节点，拓展当前节点的子节点，将没有重复的节点入栈。记录父亲节点在path中的位置。

**测试数据1：**

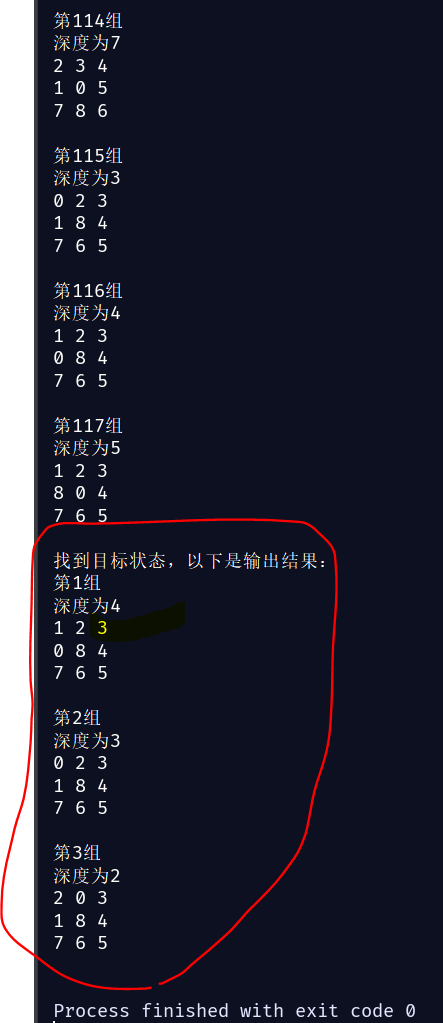
**2 8 3 1 0 4 7 6 5**

**1 2 3 8 0 4 7 6 5**

**运行结果：**



此处省略110组

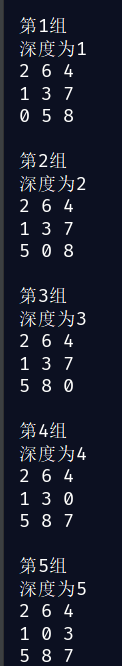
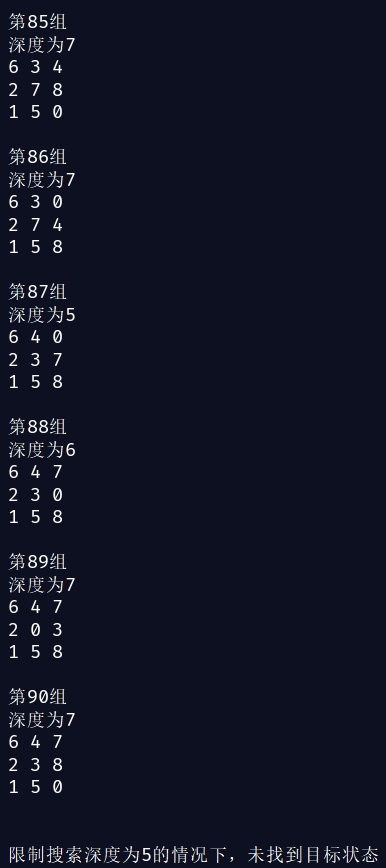


**测试数据2：（搜索深度为5的情况下找不到目标节点）**

**2 6 4 1 3 7 0 5 8**

**8 1 5 7 3 6 4 0 2**

运行结果：

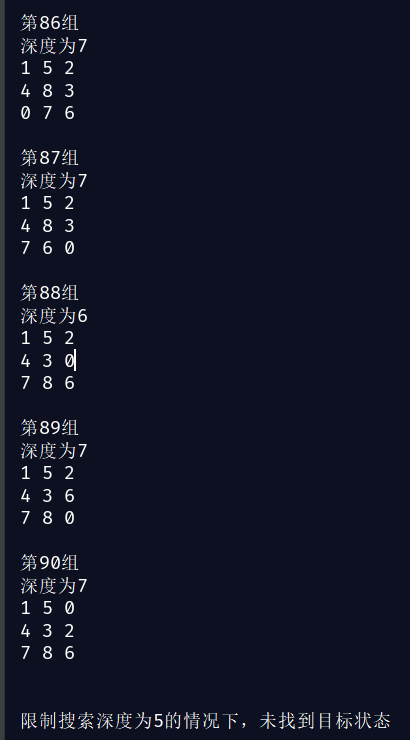
 ….. 

**测试数据3：（无解的情况）**

**1 2 3 4 5 6 7 8 0**

**1 2 3 4 5 6 8 7 0**

**运行结果：**



遇到的困难和解决：

* **深度设置不合适，开始设置深度s.deep > 20000，而我电脑经过测试，最大的栈只能支持到17647的深度，最终到爆栈也找不到节点。**

**将栈的深度改小之后，实现了有效回溯，不会陷入很深，最终能较快找到结果。这里说明深度优先搜索必须要有界。**

**还有一个小问题，判断若deep > 临界深度，停止往深了走，应该是continue,而不是break;**

* **注意初始化节点和将节点入栈的顺序。比如根节点，应该先初始化根节点，比如让根节点的父亲节点为-1，再入栈。**
* **注意初始策略的选择。仔细比较我的有界深度优先和无界深度优先发现差别很大。原数组打印路径是 将深度遍历过程中所有的节点放入vector，发现节点时再打印出来，但是这样做其实有问题。深度遍历到子节点都不可拓展的情况，就得回溯，而回溯过程中的节点是无效节点，不能放入路径**
* **注意手动实现栈和递归的关系。递归也是使用栈，不过栈是系统栈。而像我所写的手动实现了栈，就不要再递归dfs了，这里纯属不懂两者的关系，导致我手动实现栈又递归调用系统栈。**
* **效率优化问题。原本拓展节点的顺序是**

**上下左右。**

**const int dx[] = {-1,1,0,0};**

**const int dy[] = {0,0,-1,1};**

**现在是 上右下左**

**const int dx[] = {-1,0,1,0};**

**const int dy[] = {0,1,0,-1};**

**上下左右的顺序，比如一直往上递归，直到撞墙，这个时候回溯，不应该回到 往下走，而应该是往左或往右。若往下走，移动0的位置，又回到了过来的位置，要调用函数try\_to\_insert(t)多判断一次。这样的“从来的地方在回去”的顺序，拓展节点多了之后影响效率。**

④代价树的宽度优先搜索

分析：代价树宽度优先：经过分析，发现代价树以深度为代价的宽度优先和没有代价的宽度优先本质一样。有两种实现方法，一种是直接延续我没有代价的bfs，维护一个deep的一维数组就行了。（虽然按照深度的队列实现，每次最小的肯定在数组最前面，但是为了普适性，万一以后的代价不是深度了怎么办？这里的**深度deep 是泛指“代价”**。所以还要写一个函数找最小的深度（代价）的过程。）

第二种大换血，换成结构体，内部维护八数码方块和深度。

这里采用第二种。

开始是多写一个函数，找最小代价

State find\_min\_cost(){

State mi = v[0];

for(int i = 1; i < v.size(); i++){

if(mi.deep > v[i].deep){

mi = v[i];

}

}

return mi;

}

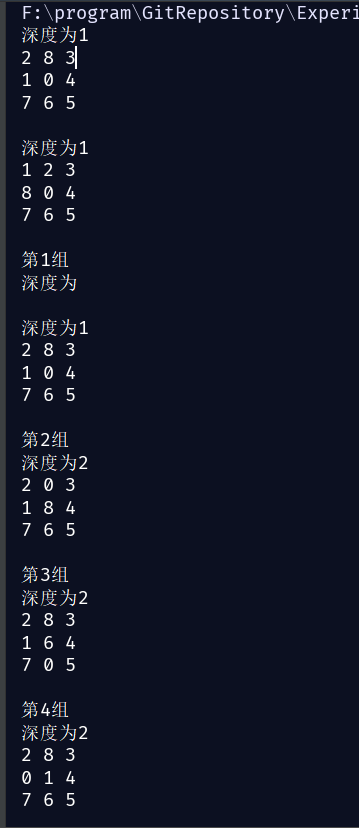
后来改换成数据结构 优先队列priority\_queue，每次取第一个元素就行了，注意要重载比较的小于号，重载成比较深度。

测试样例1：

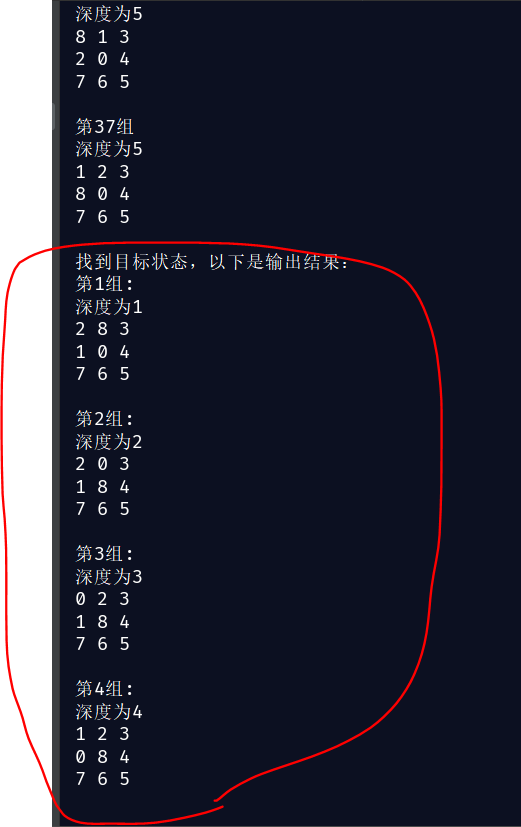
**2 8 3 1 0 4 7 6 5**

**1 2 3 8 0 4 7 6 5**

**运行结果：**



**……**



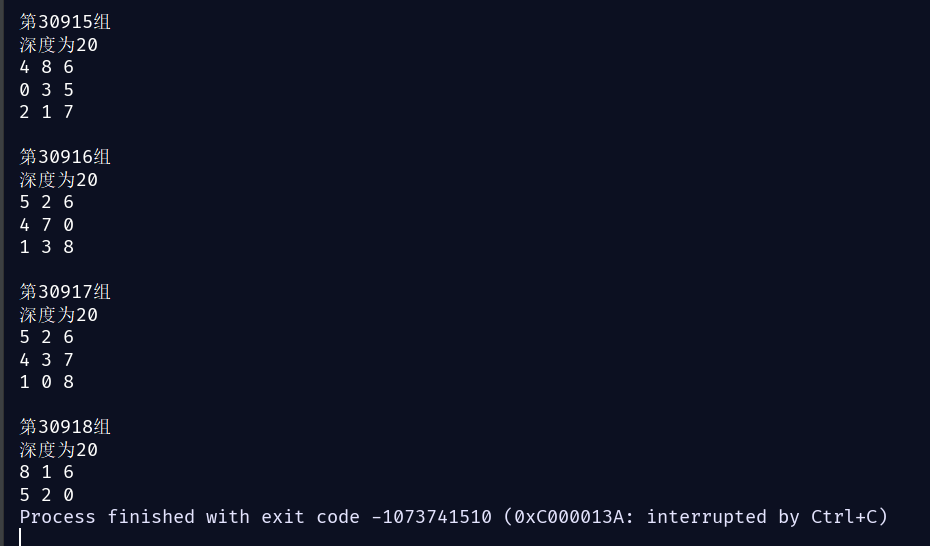
**测试数据2（有解但求解时间长）**

**2 6 4 1 3 7 0 5 8**

**8 1 5 7 3 6 4 0 2**

**运行结果**

**由于时间问题，我手动暂停在了第30918个节点，第20层，目前为止没有找到目标节点，可能还要往深了走**

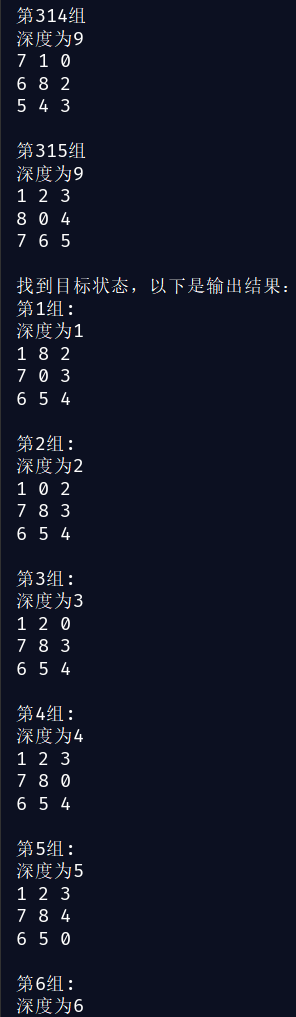
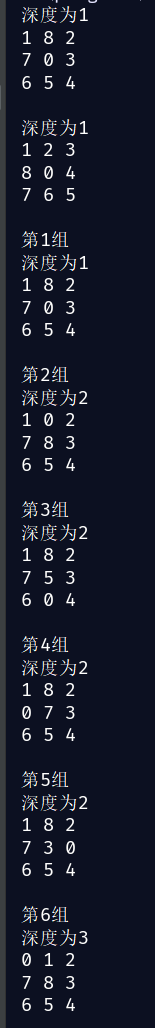


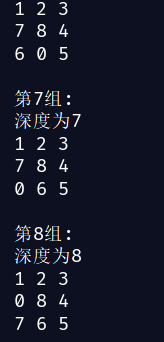
**测试数据3**

**1 8 2 7 0 3 6 5 4**

**1 2 3 8 0 4 7 6 5**

**运行结果**





**测试数据4 （无解）**

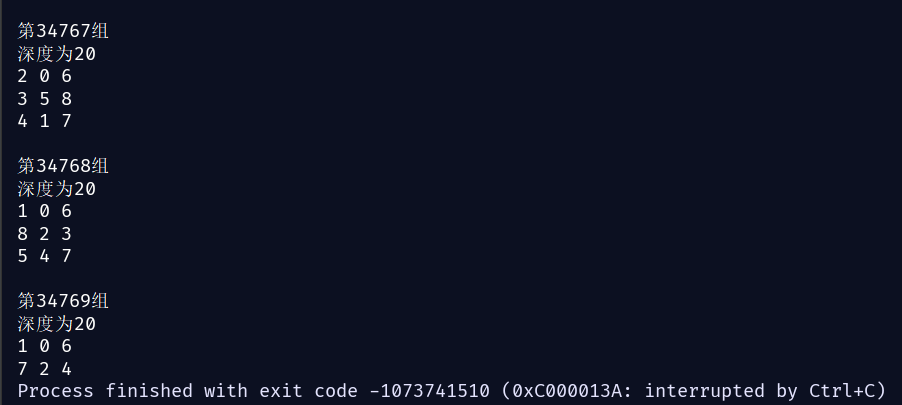
**1 8 2 7 0 3 6 5 4**

**1 2 3 8 0 4 7 6 5**

**运行结果**

**无解的情况不行限制深度的话会一直跑**

**这里我掐断在第34769组**



遇到的困难和解决：

* **优先队列priority\_queue的用法不熟悉。**

**优先队列默认是大根堆，而依据程序要求是每次取最小代价的节点，矛盾造成程序出错。所以在用一些数据结构时，若不是特别笃定输出结果是不是想要的，千万要进行测试。实际上，知道结果也要测试，这样才能知其然然后知其所以然。**

**解决方法是重载小于号处做一个调整。**

* **对于不熟悉的函数调用，不熟悉就不要写。**

**比如复制数组memcpy(t, s, sizeof(s))，上一个程序中没有出错是因为复制的是state 类型变量，而这里复制的是state 类型指针，由于不清楚，复制数组取内存值，到底是取指针本身的值，还是取数组指向的值。容易出错，就老老实实写复制循环**

**for(int i = 0; i < 9 ;i++){**

**t->a[i] = s->a[i];**

**}**

**t->deep = s->deep;**

* **注意此题中记录下已经找到三个子拓展节点的root节点，**

**便于打印路径和释放内存时，能找到根，从出发进行递归释放内存等一系列操作。**

**if(!is\_root){ //将完整的root 存起来，便于从根节点出发打印路径**

**root = s;**

**is\_root = true;**

**}**

**delete\_tree(root);**

* **搞清楚为节点new 空间和delete 释放空间的重要性和时机。**

**每一个new 空间、为之分配空间的节点，都应该释放掉内存。即使是临时变量，也要new出空间，否则，若临时变量是指针，一旦出了定义域，指针失效。**

**此处重新学习了pop的具体机制是delete，即pop掉某个节点后，该节点就不存在了，所以指向它的指针无效。这里写了一个测试函数测试一下pop 包不包含delete**

#include <bits/stdc++.h>

#include<stack>

using namespace std;

int main(){

stack<int> s;

for(int i = 0; i < 5; i++){

s.push(i\*i);

}

int \*p1 = &s.top();

cout << \*p1 << endl;

s.pop();

int \*p2 = &s.top();

s.pop();

int \*p3 = &s.top();

s.pop();

int \*p4 = &s.top();

s.pop();

int \*p5 = &s.top();

s.pop();

for(int i = 100; i < 105; i++){

s.push(i\*i);

}

int a = 100;

// int b = 3009;

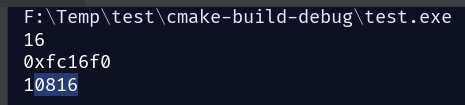
cout << p1 << endl;

cout << \*p1 << endl;

return 0;

}

运行结果：



pop 掉之后，p1便有原来的指向16变为指向10816

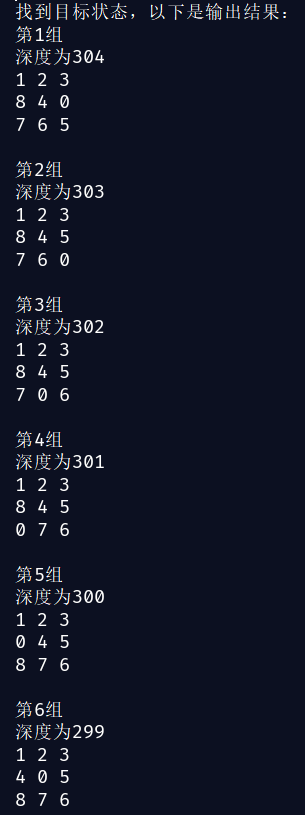
⑤代价树的深度优先：如果说代价树的宽度优先是选取全局的代价最小的节点（此处即深度最小），那么代价树的深度优先就是选取刚刚拓展的几个结点（最多四个）里面代价最小的，体现了局部最优的思

经过分析，发现代价树的深度优先和无界深度优先本质相同，每次从栈中取出的节点一定是刚刚拓展的节点之一，且由于代价是深度，刚刚拓展的节点的代价一样。同理，代价树的深度优先也不是完备的，因为代价树的深度优先搜索法可能进入无穷分支路径而搜索到不同问题的解。

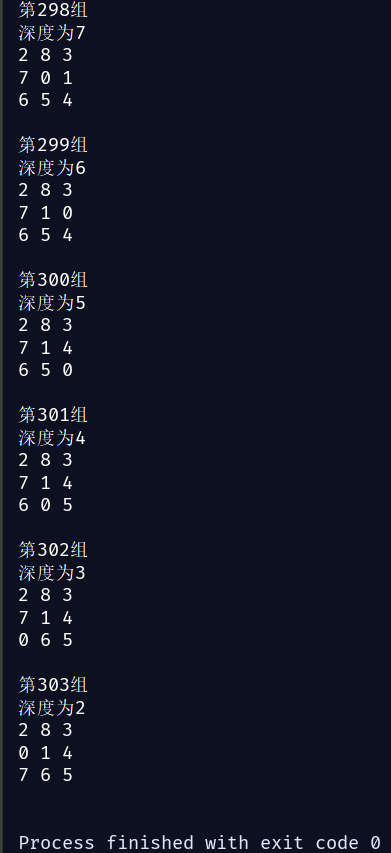
测试数据1 （有解但是很深）

**2 8 3 1 0 4 7 6 5**

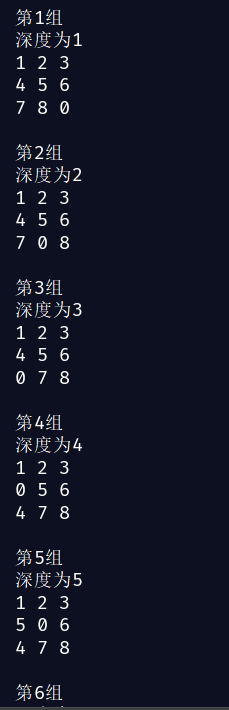
**1 2 3 8 0 4 7 6 5**



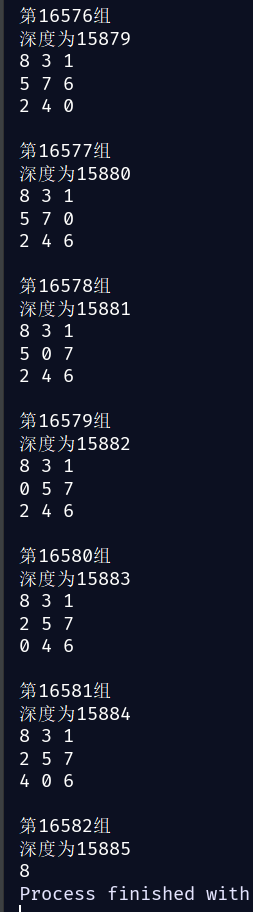
……此处省略292组



测试数据2 无解一直深处递归



……此处省略16570 组



⑥全局最佳优先搜索（可利用教材中的估价函数）算法思想：教材中的估价函数是当前节点的深度和对应位置不同点的个数加和，专门写个函数：int culculatePrice(State \*t) 计算每个节点的代价，放到节点的结构体中。 注意此时这里的深度deep，是泛指，指的是代价。

变动的地方：

//计算代价，t 和目标节点相差的个数和此时t节点的深度

int calculatePrice(State \*t){

int pprice = 0;

for(int i = 0; i < 9; ++i){

if(t->a[i] != goal->a[i]){

pprice++;

}

}

return pprice + t->deep;

}

拓展新节点时加上代价计算

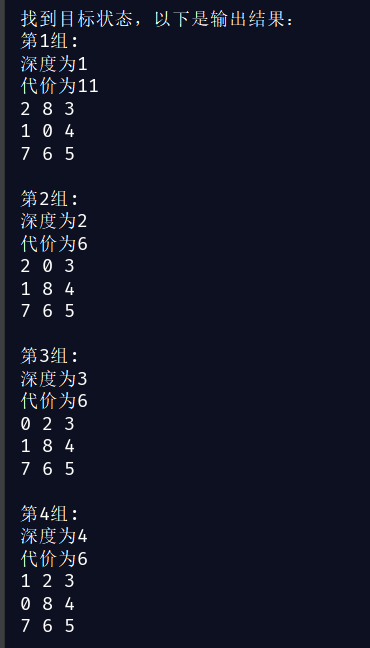
t->price = calculatePrice(t);

测试数据1 （有解）

**2 8 3 1 0 4 7 6 5**

**1 2 3 8 0 4 7 6 5**

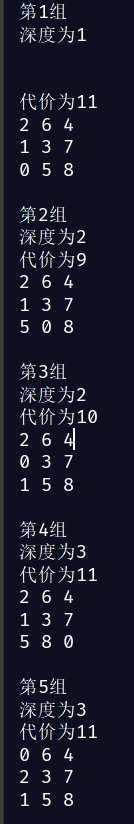
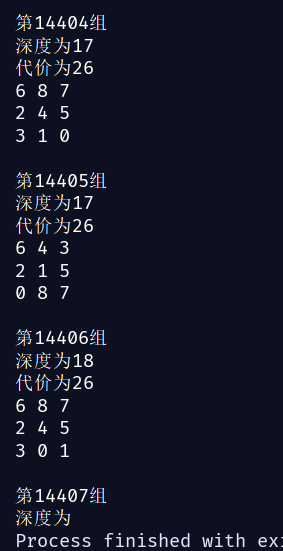
下列是运行结果（注意编码方式GBK,否则乱码）



测试数据2 （有解但是暂时未找到，手动终止）

**2 6 4 1 3 7 0 5 8**

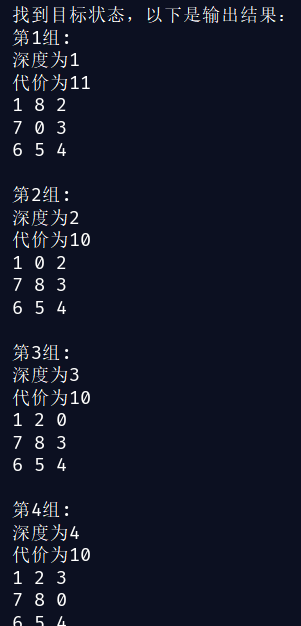
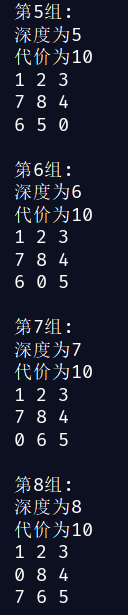
**8 1 5 7 3 6 4 0 2**

测试数据3 （有解，路径比测试数据1长）

**2 6 4 1 3 7 0 5 8**

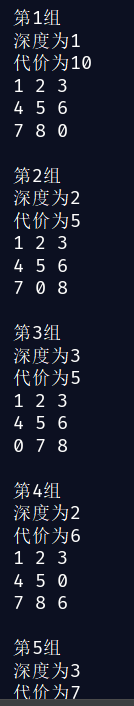
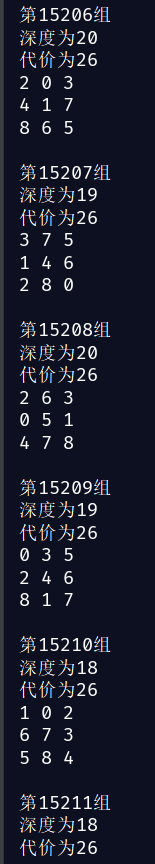
**8 1 5 7 3 6 4 0 2**

测试数据4 （无解，手动终止）

**1 2 3 4 5 6 7 8 0**

**1 2 3 4 5 6 8 7 0**

 ……

⑦局部最佳优先搜索

局部和全局的区别是，局部是从当前拓展节点选取代价最小的，全局是所有节点代价最小的，其中若把深度当做估价函数，或者把代价当做估价函数，深度优先搜索和代价树的深度优先搜索都可以看做局部优先搜索的特例。而全局优先搜索还则是选取节点评判标准不同的宽度有限，大致上来讲，深度越深，不同节点的个数越多。

这里就和代价树的深度优先有区别了，代价树的深度优先我做了一个简化，因为特殊之处在于代价是深度，栈顶元素肯定是刚拓展的节点，且刚拓展的节点代价（深度）相同，为了简化就取栈顶。

但是这里的局部优先就不一样了，刚拓展的节点们代价不一定相同，所以不能就取栈顶。

由于每次只在后继节点的范围内选取下一个要考察的节点，类似于稍微“扩大范围”的深度优先。

我的思路是 将优先队列缩小范围，只存刚拓展的节点，定时清空，这样能保证每次取到局部最优。

变动的地方：

// 取出最小代价的节点，这里的最小是在刚刚拓展的几个节点里面最小的。

//清空优先队列

while(!Q.empty()){

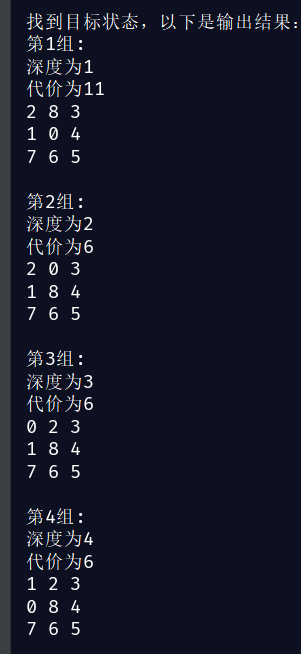
Q.pop();

}

**测试数据1 有解**

**2 8 3 1 0 4 7 6 5**

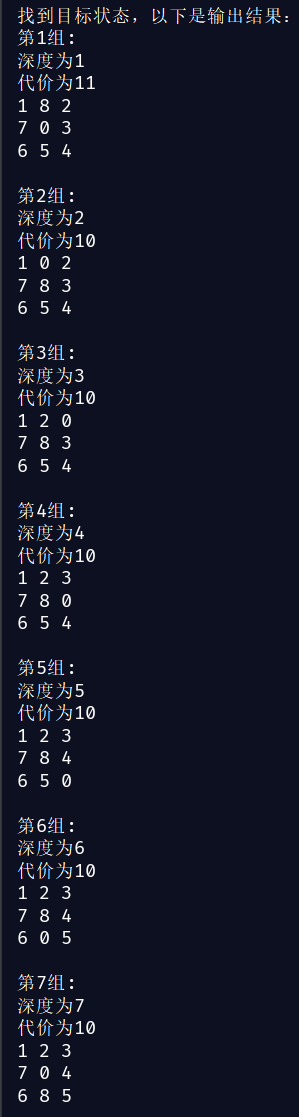
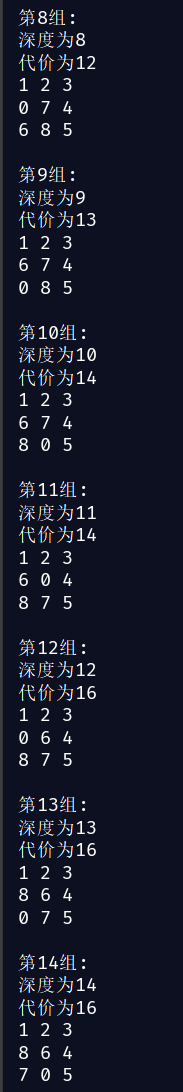
**1 2 3 8 0 4 7 6 5**



**测试数据2 有解**

**1 8 2 7 0 3 6 5 4**

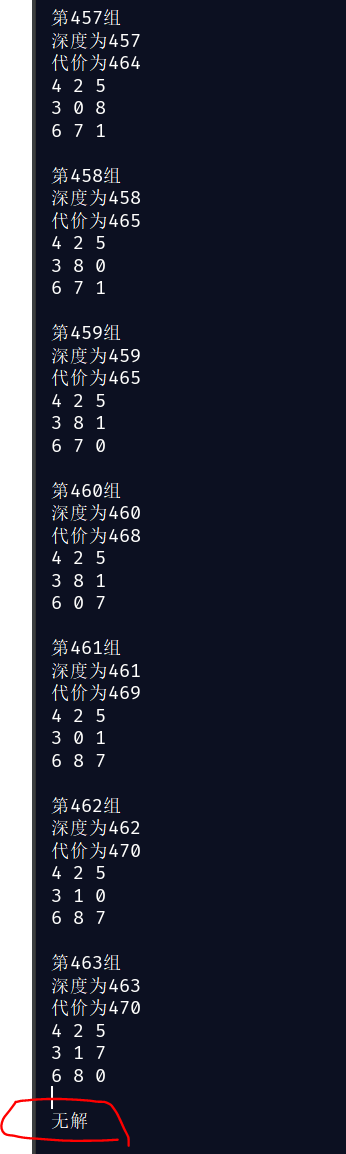
**1 2 3 8 0 4 7 6 5**

**测试数据3 无解**

**1 8 2 7 0 3 6 5 4**

**1 2 3 8 0 4 7 6 5**



3.指出参考资料，如书籍或者网站 ①《人工智能教程》（第二版）张仰森

用来了解基本概念和手动模拟搜索和剪枝的过程

②《算法入门经典》（第二版）刘汝佳

P198页八数码问题，参考部分bfs的代码

③<https://github.com/duochirou/EightPuzzle_Python/blob/master/main.py#L174>

理顺搜索的思路

4.代码与报告打包上传到课堂派

**大总结**本次实验具体编程过程中遇到的问题和解决办法已经在前文叙述，接下来从整体层面谈一谈这次实验的收获：

①第一点要认真审题，不要将问题复杂化。前面的实验要求说“可视化”，我以为是要将八数码过程展示成动态交互界面，像个计算器那样，便从头开始学Qt ，学到一半才发现，直接打印出来不就好了？ 其中学习Qt的过程导致我实验迟迟没开始写，老是在担心可视化的问题，可见我对题目和实验初衷的误解。

②万事开头难，当我写下第一个bfs时，差点流下激动的泪水，太不容易了，要注意的事项太多，稍微不注意就可能会出错。第一个代码写成后，后面的大体思想都差不多，只是一些搜索限制条件，优先选择策略不同，还有结束搜索的条件不同，稍加注意，就能实现。

③充分利用中途打印结果。看到打印出来的结果时，不要只关注结果是对是错，还要关注结果为什么会是这样。比如我在有界深度优先搜索遇到的问题。本来深度优先搜索是撞到墙就回溯，但是我打印中间结果时多看了一眼，发现路径并不是撞墙就回溯，而是提前就回溯了。刚开始以为代码出现问题，后来仔细分析发现，将父节点的三个（最多三个）

儿子打入栈，最先拓展的节点最后取出来，反而是刚打入的节点很快被取出来，导致没撞墙就回溯，但是程序没有问题。这就催生出了拓展节点的顺序问题。是“上下左右”还是“上右下左”。

④总结这七种搜索的差别和适用范围。其实只要代码大致框架出来了，具体实现都不会差别很大，这里主要是用到的数据结构差异。此外有一些搜索策略比较经典，值得把套路记下来。比如启发式搜索策略的最佳优先搜搜，避免像深度优先和宽度优先那样盲目搜索，盲目生成所有节点的后继节点，确定了在搜索树中的剪枝和修剪的节点。

同时要注意估价函数的选取，其中启发信息有的能减少搜索的工作量，却不能保证搜索得到最佳路径。因此选取的估价函数要能使问题求解路径代价和求此路径花费的搜索代价最小，例如书中给出的棋子不同个数和深度值之和就是一个比较好的估价函数。

另外，介绍了这么多搜索策略，我决定增加一个内容：

* + 对每个搜索策略进行横向比较，对于同一对数据，比较他们搜寻的深度和搜寻速度，加上所走步数，综合比较效率。

答：效率一看便知，对于同一组测试数据，有的策略陷入死胡同（无界深度优先等），而有的几步就出来了（宽度优先等）。

以上，是我对本次实验的一点小总结，希望老师批评指正。