**由八数码问题探究BFS的三次优化**

王凌

（同济大学,软件学院,上海 201800）

**摘要：**搜索算法是利用计算机的高性能来有目的地穷举一个问题解空间的部分或所有的可能情况，从而求出问题的解的一种方法。它是得到所需结果与解决实际问题的利器，是很多高深算法的思想基础，是人工智能算法的起点。正是因为搜索算法的重要，为降低其时间复杂度的优化方法已然成为一门学问。我从解决经典的八数码问题入手，调研并用C++代码实现了由BFS依次到双向宽搜、A\*搜索、IDA\*搜索的三次优化算法。最后通过查找资料分析归纳了这些算法的特性。

**关键词：**搜索算法，宽度优先，双向宽搜，A\*，IDA\*，搜索优化

**Explore the Three Optimizations of BFS from Eight Puzzle Problem**

*Wang Ling*

( Tongji University, ShangHai 201800, China)

**Abstract:** The search algorithm is a method that uses the high performance of the computer to purposefully enumerate part or all of the possible situations in the solution space of a problem to find the solution of the problem. It is a weapon to obtain desired results and solve practical problems, the ideological foundation of many advanced algorithms and outset of artificial intelligence algorithms. Because of the importance of search algorithms, optimization methods to reduce their time complexity have become a science. I started researching by solving the classic Eight Puzzle Problem and implemented three optimization algorithms from BFS to bidirectional wide search, A\* search, and IDA\* search with C++ code. Finally, the characteristics of these algorithms are summarized by collecting and analyzing data.

**Key words:** search algorithm, breadth first, bidirectional wide search, A\*, IDA\*, search optimization

**中图分类号：**TP311 **文献标志码：**A

本学期我们学习了图的深度优先搜索和广度优先搜索两种主要的搜索算法。

在《挑战程序设计竞赛》这本书中，第一节讲的就是搜索，这足以见得搜索算法之于整个算法体系的地位。事实上，在ACM竞赛或是平时解决实际问题的时候，搜索算法不仅仅局限于图的搜索。

正是因为搜索算法的重要，为降低其时间复杂度的优化方法已然成为一门学问。

在《算法竞赛入门经典》（第二版）的第七章暴力求解法之中，详细介绍了由枚举到深搜回溯到剪枝、由宽搜到双向宽搜的优化思路。这本书还介绍了结合了深搜与宽搜两者优点的迭代加深IDA\*算法和人工智能算法的起点——启发性搜索A\*算法。这些搜索算法的优化，都使得暴力和笨拙的搜索算法变得灵巧轻盈。

在选题的时候，我考虑了搜索算法的两个经典问题——八皇后问题和八数码问题。前者的主要解决方法是深搜，后者的主要解决方法是宽搜。最终选择八数码的主要原因有，首先我所能想到的解决八数码问题的搜索的优化方式更多更全面，其次我也参考了网上现有的资料，其中看了之后收获最大的是一篇叫《八数码的八境界》**[1]**的博文，也从中得到了很多启发，最终决定结合前人的基础和自己的思考，写一篇关于搜索优化的文章。

# 1 八数码问题

“八数码”问题又叫“九宫格”问题。在3×3的棋盘上，摆有八个棋子，每个棋子上标有1至8的某一数字。棋盘中留有一个空格，空格用0来表示。空格周围的棋子可以移到空格中。要求解的问题是见图1：给出一种初始布局（初始状态）和目标布局（为了使题目简单,设目标状态为123804765），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。**[2]**

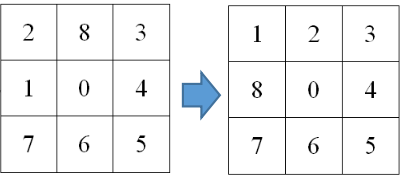


图1 八数码题目描述

这是我高中参加NOIP遇到的一道很有启发意义的问题，这个问题和我小时候玩过的“华容道”游戏很像——属于滑块类游戏，即在一定范围内，按照一定条件移动一些“块”，最后满足一定的要求。而追其本质，是探究如何从初始状态到目标状态的问题，而这种问题我们一般会考虑搜索算法。

# 2 八数码状态的确定

搜索算法的第一个重要问题是如何定义和描述状态，在我们这学期的数据结构中，学过的DFS和BFS基本是用来遍历树结构和图的。而在所学的数据结构之中，每一个结点类就是一种“状态”，树结点的状态由结点的值、左孩子和友孩子的位置来描述，图结点的状态由结点的值、入边和出边来描述。状态的定义和描述影响了程序状态变化时的空间与时间复杂度，所以会出现利用位运算描述状态的状态压缩的搜索与动态规划算法。而这个问题的状态可以用一个3\*3的二维数组来描述，也可以用题目中所给的一个long long类型的9位数字来描述，还可以用一个string类型来保存。与此同时，我们考虑状态的变换。发现每一次状态的变换最多有四种情况，如果0在中间，那么可能从上下左右四个方向移动“块”；如果在边上，那么有三个方向；如果在角上，那么有两个方向。我们发现用数组记录的状态转换的实现比较容易，而用一个9位数字记录之前的所有状态则更节约空间。故最终我们选择用一个long long类型来记录状态，但处理数据的时候，将数字转化为数组进行处理。

# 3 BFS的实现

选定状态之后，我们考虑搜索的方式。由于求的是最少移动步数，所以我们用BFS，记录每一步都能到达哪些状态。BFS步骤如下：

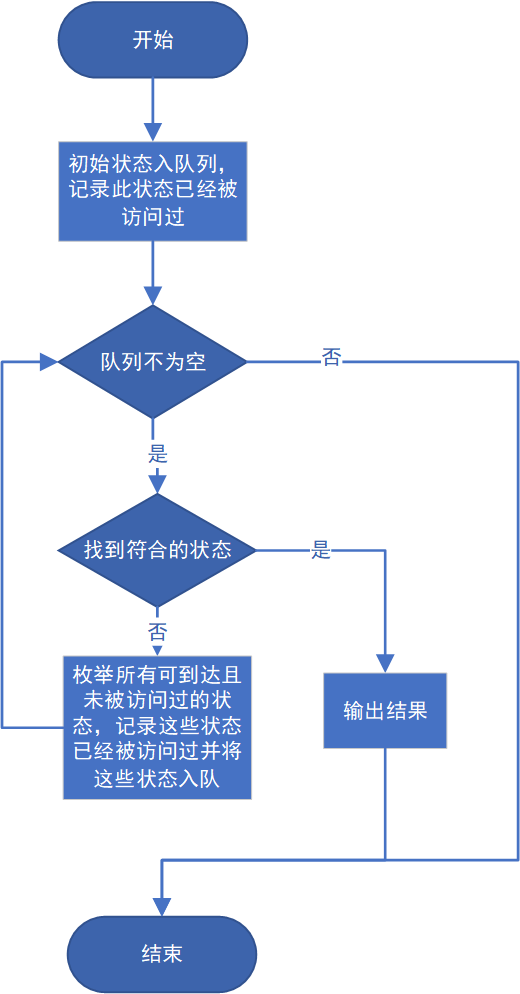


图2 BFS流程图

代码如下：

#include<bits/stdc++.h>

using namespace std;

long long goal=123804765,start;

class Node

{

public:

    long long chess;

    int step;

};

int dx[5]={1,-1,0,0};

int dy[5]={0,0,1,-1};

int s[4][4];

map<int,int> vis;//记录有没有访问过

queue<Node> q;

void BFS()

{

    Node head;

    head.chess=start;

    head.step=0;

    q.push(head);

    vis[start]=1;

    while(!q.empty())

    {

        Node cur=q.front();

        q.pop();

        if(cur.chess==goal)//找到了结果

        {

            cout<<cur.step<<endl;

            break;

        }

        int sx,sy;//记录的0位置的坐标

        long long temp=cur.chess;

        for(int i=3;i>=1;i--)

        //将用longlong记录状态转化为一个3\*3的二维数组棋盘再进行处理

        {

            for(int j=3;j>=1;j--)

            {

                s[i][j]=temp%10;

                temp/=10;

                if(!s[i][j]) sx=i,sy=j;

            }

        }

        for(int i=0;i<4;i++)

        {

            int xx=sx+dx[i],yy=sy+dy[i];

            //出界，放弃此方向，转到下一个方向

            if(xx<1||xx>3||yy<1||yy>3) continue;

            swap(s[sx][sy],s[xx][yy]);

            long long next=0;

            for(int j=1;j<=3;j++)

       //将3\*3的数组棋盘转化为一串longlong的状态进行记录

            {

            for(int k=1;k<=3;k++) next=next\*10+s[j][k];

            }

            if(vis[next])

            //已经被访问过

            {

                swap(s[sx][sy],s[xx][yy]);//消除影响

                continue;

            }

            vis[next]=1;

            Node nextnode;

            nextnode.step=cur.step+1;

            nextnode.chess=next;

            q.push(nextnode);//进队列

            swap(s[sx][sy],s[xx][yy]);

    //消除影响，返回cur状态下的数组，之后还有其他方向需要搜索

        }

    }

}

int main()

{

    cin>>start;

    if(start==goal)

    {

        cout<<"0\n";

        return 0;

    }

    BFS();

    return 0;

}

# 4 双向宽搜

## 4.1双向宽搜的思路与步骤

众所周知，我们对深搜的一般优化是剪枝，那么宽搜呢？数据结构这一学其我们也接触过宽搜的一种优化，但是是打星号的内容，没有细讲。它就是双向宽搜。

双向宽搜的算法思想为分别以目标状态和初始状态为起点，分别扩展节点，当扩展到相同的节点时，就找到了目标，最短路径为起始状态和目标状态到搜索到的点的路径之和。

而这道问题有明确的起始状态和目标状态，要求是搜索一条从起始状态道目标状态的最短路径，且移动（即状态变化）的规则具有对称性，符合选择双向宽搜算法的使用要求。

双向宽搜实现的流程图如下：

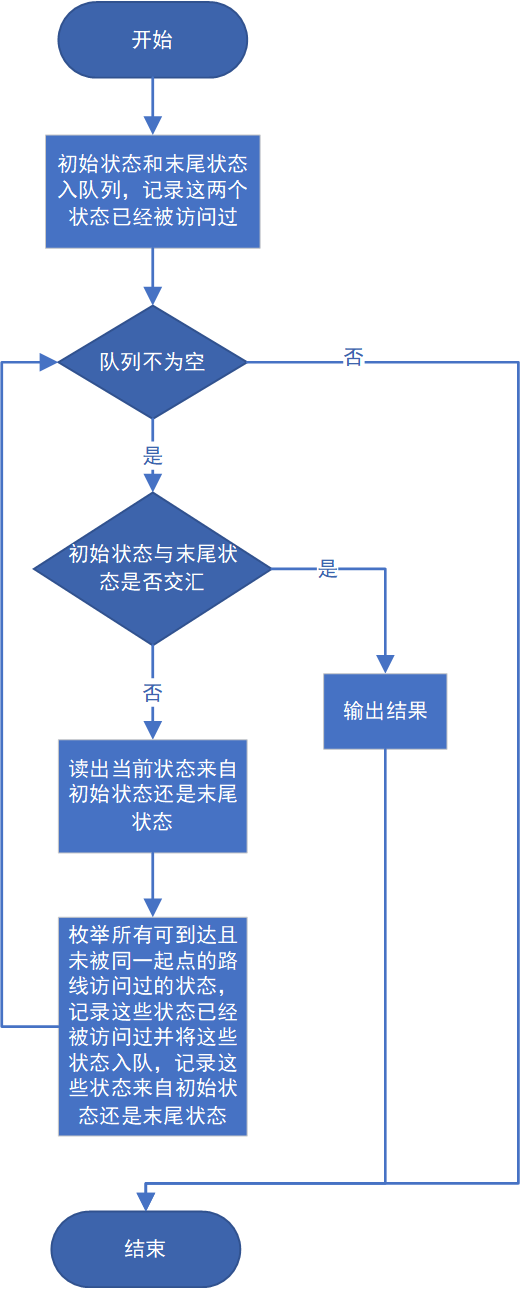


图3 双向宽搜流程图

## 4.2双向宽搜的时间复杂度与空间复杂度

原本BFS的时间复杂度如下方左图，是上下两个函数所围绕的面积转化为右侧的四个所围绕的面积（面积表示的是搜索的状态个数），减少了两个划线的状态的搜索时间。

而双向宽搜的空间复杂度同理。

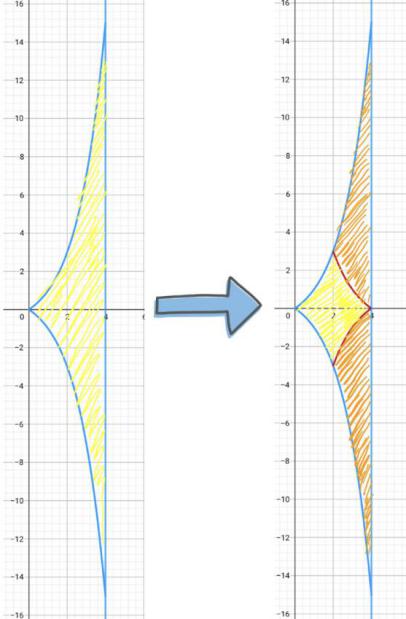


图4 双向宽搜复杂度

## 4.3双向宽搜的实现

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

ll start,goal=123804765;

ll s[4][4];

map<int,int> dis;//记录路径长度，即状态变化的次数

map<int,int> vis;//记录有没有访问过

int dx[5]={1,-1,0,0};

int dy[5]={0,0,1,-1};

void double\_BFS()

{

    queue<ll>q;

    if(start==goal)//如果起点等于终点，则不用走就直接输出

    {

        cout<<"0\n";

        exit(0);

    }

    vis[start]=1;//从起点搜索访问过的点，标记为1

    vis[goal]=2;//从终点搜索访问过的点，标记为2

    dis[start]=0;dis[goal]=0;//初始化

    q.push(start);//从起点状态开始变化

    q.push(goal);//从目标状态往回转移

    while(!q.empty())

    {

        ll cur=q.front();//cur记录队首元素

        q.pop();//出队列

        int sx,sy;

        ll temp=cur;

        for(int i=3;i>=1;i--)

        //将用longlong记录状态转化为一个3\*3的二维数组棋盘再进行处理

        {

            for(int j=3;j>=1;j--)

            {

                s[i][j]=temp%10;

                temp/=10;

                if(!s[i][j]) sx=i,sy=j;

            }

        }

        for(int i=0;i<4;i++)

        {

            int xx=sx+dx[i],yy=sy+dy[i];

            //出界，放弃此方向，转到下一个方向

            if(xx<1||xx>3||yy<1||yy>3) continue;

            swap(s[sx][sy],s[xx][yy]);

            ll next=0;

            for(int j=1;j<=3;j++)

        //将3\*3的数组棋盘转化为一串longlong的状态进行记录

            {

                for(int k=1;k<=3;k++)

                {

                    next=next\*10+s[j][k];

                }

            }

            if(vis[cur]==vis[next])

            //如果这个状态已经被同起点出发第二次访问

            //则跳过循环，不继续做下去

            {

                swap(s[sx][sy],s[xx][yy]);//消除影响

                continue;

            }

            if(vis[next]+vis[cur]==3)

//vis和为3的情况下，vis[next]和vis[cur]其中一个为1一个为2

//从cur状态转化到next状态只需要一步，说明已经找到了八数码的最短路径

            {

                cout<<dis[cur]+dis[next]+1<<endl;

             //cur状态转化到next状态需要一步，所以还要加一

                exit(0);//退出程序

            }

            dis[next]=dis[cur]+1;//状态变化次数+1

            vis[next]=vis[cur];

     //继承属性，用来分辨这个点是从起点来的还是从终点变化来的

            q.push(next);//进队列

            swap(s[sx][sy],s[xx][yy]);

    //消除影响，返回cur状态下的数组，之后还有其他方向需要搜索

        }

    }

}

int main()

{

    cin>>start;

    double\_BFS();

    return 0;

}

# 5 启发式搜索

## 5.1启发式搜索的思路及步骤

而宽搜本身还可以继续优化，这就有两个思路，一个是启发式搜索A\*算法，另一个是迭代加深IDA∗算法。首先要实现的是优化BFS的A\*算法。A\*算法的重点是估价函数：。

其中G\*等于从起点状态出发，沿着产生的路径，移动到当前状态的移动耗费。在此程序中就是（当前结点对象.step+1）的值。H\*等于从当前状态移动到终点状态的预估移动耗费。这通常被称为“启发式”的，所以A\*算法又被称作启发式搜索。H\*的求法很多，常见的方法一般用曼哈顿距离进行估值。但是我们这里到终点状态的“距离”是抽象的，H\*=除了0位置之外现在状态和目标状态差几个棋子。

所有的状态存进优先队列（小根堆）中，按照F\*的大小由小到大排序。每次取估价最小的状态进行搜索与状态转移，先考虑最有可能最短到达的状态，并且转移之后再次更新优先队列，这样能更早找到最短路径。

综上，A∗其实是利用优先队列priority\_queue的BFS再加上一个估价函数，而普通的BFS就是每个点到终点的估价均相同的A\*算法的一种特殊的最差的情况。而A\*算法，也开启了人工智能算法的篇章。

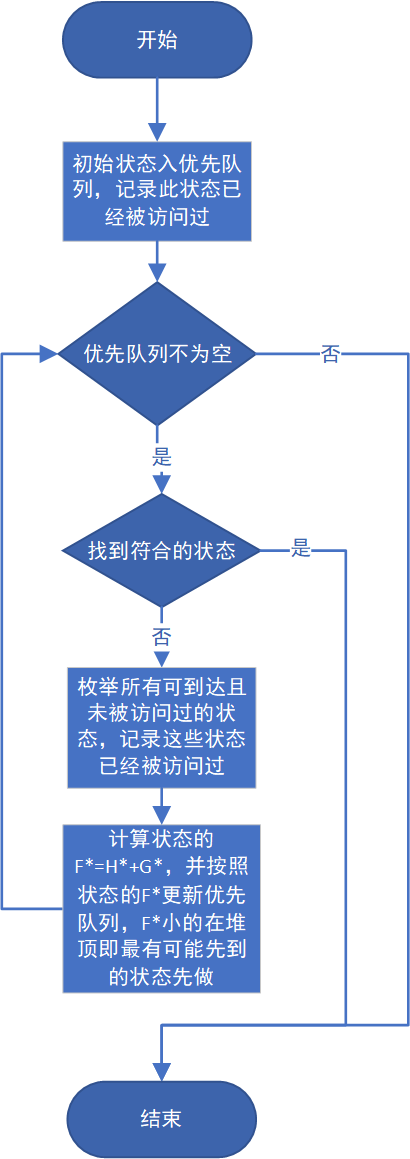


图5 启发式搜索流程图

## 5.3启发式搜索的实现

在本程序之中，估价函数返回的是除了0位置之外现在状态和目标状态差几个棋子。这和八数码问题到达最终结果的路径长度是不完全吻合的，甚至可以找到估价函数很小，但是实际的路径长度很大的反例。但是由于在一个非最优解的路径中，随着深度增大，在搜到目标之前，一定会在某一层出现都比最优解估价大的情况，所以最终先搜到目标的结果，就是最优解。同时我们也可以知道，A\*搜索算法的关键在于估价函数H\*的建立，如果H\*函数估价越正确，那么搜索的时间复杂度将会越低。

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

char s[15];

string goal="123804765";

int h\_star(string cur)//估价函数

{

    int difference=0;

    for(int i=0;i<9;i++)

        if(goal[i]!=cur[i] && goal[i]!=0) difference++;

        //除了0位置之外现在状态和目标状态差几个棋子

    return difference;

}

class Node

{

public:

    int f\_star,step;

    string current;

    // 由于定义小根堆用了定义在Node上priority\_queue,vector,greater上的库

    // Node类需要排序，所以需要重载>与<号

    bool operator<(const Node &x) const

    {

        return f\_star<x.f\_star;

    }

    bool operator>(const Node &x) const

    {

        return f\_star>x.f\_star;

    }

};

priority\_queue<Node,vector<Node>,greater<Node> > q;//定义小根堆，每次取f\_star最小的结点

map<string,bool> mapp;

map<string,int> dis;

int dx[4]={0,1,-1,0};

int dy[4]={1,0,0,-1};

void A\_STAR()

{

    while(!q.empty())

    {

        Node t=q.top();

        q.pop();

        string cur=t.current;

        if(cur=="123804765")

        {

            printf("%d\n",t.step);

            exit(0);

        }

        int sx,sy;

        for(int i=0;i<9;i++)

            if(cur[i]-'0'==0) sx=i/3+1,sy=i%3+1;

//找到当前状态0的坐标

        int tmapp1=(sx-1)\*3+sy-1;

//0放到数组之中的位置

        for(int i=0;i<4;i++)

        {

            int xx=dx[i]+sx,yy=dy[i]+sy;

            if(xx<1 || xx>3 || yy<1 || yy>3) continue;//越界

            int tmapp2=(xx-1)\*3+yy-1;

//坐标为（xx,yy）的点放到数组之中的位置

            swap(cur[tmapp1],cur[tmapp2]);

//移动，cur字符串发生变化

            if(mapp[cur]==0 || (mapp[cur]==1&&(t.step+1)<dis[cur]) )

            //当移动后的cur这种状态没有访问过 或者 虽然访问过，但是这种状态变换步数更少

            {

                dis[cur]=t.step+1;//更新步数

                q.push((Node){h\_star(cur)+t.step+1,t.step+1,cur});//进队列

                mapp[cur]=1;//更新访问状态

            }

            swap(cur[tmapp1],cur[tmapp2]);

//消除影响，返回原状态

        }

    }

}

int main()

{

    cin>>s;

    if(h\_star(s)==0)

    //如果估值函数判定当前状态和目标状态相差为0，即一样则输出0步

    {

        cout<<0<<endl;

        return 0;

    }

    q.push((Node){h\_star(s),0,s});

//将起点状态进入队列

    mapp[s]=1;

    dis[s]=0;

    A\_STAR();

    return 0;

}

# 6 迭代加深搜索

## 6.1迭代加深搜索的思路及步骤

个人理解，迭代加深IDA\*搜索算法的核心是——以BFS的思想写DFS。

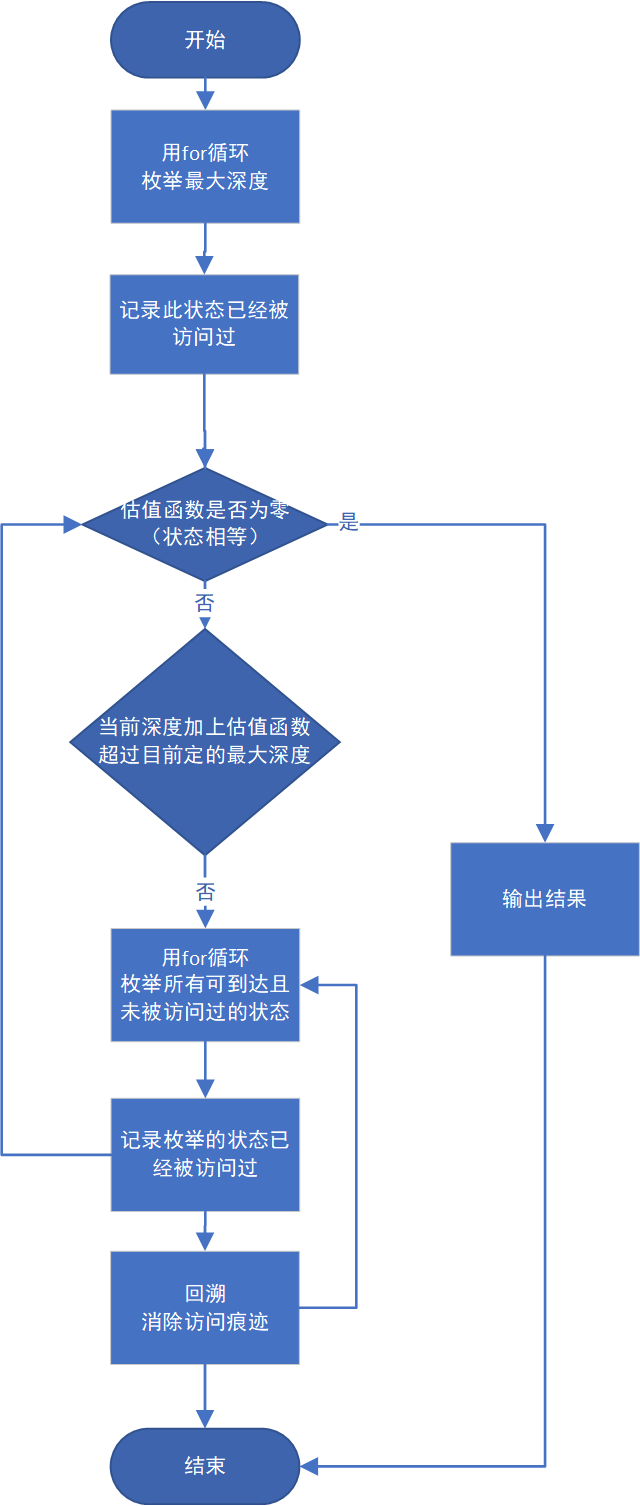


图6 迭代加深搜索流程图

具体来说就是，首先深度优先搜索i层，若没有找到可行解，再深度优先搜索i+1层，直到找到可行解为止。由于深度是从小到大逐渐增大的，所以当搜索到结果时可以保证搜索深度是最小的。这也是迭代加深搜索在一部分情况下可以代替广度优先搜索的原因。

## 6.2 迭代加深搜索算法的实现

#include<bits/stdc++.h>

using namespace std;

char s[15];

char goal[]={'1','2','3','8','0','4','7','6','5'};

int maxdepth;

int h(char \*cur)//估价函数

{

    int difference=0;

    for(int i=0;i<9;i++)

      if(goal[i]!=cur[i] && cur[i]!=0) difference++;

        //除了0位置之外现在状态和目标状态差几个棋子

    return difference;

}

int dx[]={0,1,-1,0};

//dx[0]与dx[3]方向相反，dx[1]与dx[2]方向相反

//即下标相加等于3的方向相反，后有剪枝

int dy[]={1,0,0,-1};

bool ida\_star(int depth,char \*a,int pre)

{

    if(h(a)==0) return true;

    //估值函数h(a)==0，说明当前状态和最终状态相同

    if(depth+h(a)-1>maxdepth) return false;//剪枝，如果当前深度加上估值步数超过当前定义的最大深度那么就退出

    int sx,sy;

    for(int i=0;i<9;i++)

        if(a[i]=='0') sx=i/3+1,sy=i%3+1;

    //找到0的坐标(sx,sy)

    for(int i=0;i<4;i++)//枚举可达状态

    {

        int xx=dx[i]+sx,yy=dy[i]+sy;

        if(xx<1 || xx>3 || yy<1 || yy>3 || pre+i==3) continue;

        //前面四个是判断有没有越界，如果越界退出

       //pre+i==3是一个小的剪枝，判断有无返回原来的状态

        swap(a[(xx-1)\*3+yy-1],a[(sx-1)\*3+sy-1]);

        if(ida\_star(depth+1,a,i)) return true;//深搜

        swap(a[(xx-1)\*3+yy-1],a[(sx-1)\*3+sy-1]);

//回溯

    }

    return false;

}

int main()

{

    cin>>s;

    if(h(s)==0)

    //如果一开始估价函数为0，说明不需要移动

    {

        cout<<"0"<<endl;

        system("pause");

        return 0;

    }

    for(maxdepth=1;;maxdepth++)

  //从1开始枚举maxdepth的大小，以达到一层一层深搜的目的

    {

        if(ida\_star(0,s,-1))

        {

            cout<<maxdepth<<endl;

            //maxdepth就是最多走几步

            system("pause");

            return 0;

        }

    }

    return 0;

}

# 7 搜索算法的比较

首先，在常用的两大搜索算法宽搜和深搜中，BFS常用于找最优解，缺点是需要存下所有状态，需要较大的存储空间；而DFS所需存储空间小，但由于对状态漫无目的地穷举有可能会浪费很多时间。

这也正说明了在各种情况下，我们总是用空间来换时间或是用时间来换空间来解决问题。

对于BFS，我们有三种优化方式。

首先，如果有明确的起始状态和目标状态，要求是搜索一条从起始状态道目标状态的最短路径，那么可以使用双向宽搜，这样节约了空间与时间。

其次，我们可以定义一个估价函数，并将存储BFS状态的队列改为优先队列，优先队列的队首存的是最有可能最先到达目标状态的状态。这就是A\*算法。

最后，我们还可以用IDA\*搜索算法——即用BFS的思想写DFS，这样，我们既可以利用DFS的小存储的优点，也可以利用BFS的特性来找到最优解。A\*和IDA\*的时间与空间复杂度都取决于估价函数的准确性，在此不再赘述。

**参考文献：**

[1]liugoodness.《八数码的八境界》. https://www.cnblogs.com/goodness/archive/2010/05/04/1727141.html

[2]yeszy.LuoguP1379.https://www.luogu.com.cn/problem/P1379