# 启发式搜索

https://blog.csdn.net/nc\_illy/article/details/65449769

启发式搜索

启发式搜索就是在状态空间中的搜索对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。这样可以省略大量无谓的搜索路径，提高了效率。在启发式搜索中，对位置的估价是十分重要的。采用了不同的估价可以有不同的效果。

在启发式搜索中，我们每次找到当前“最有希望是最短路径”的状态进行扩展。对于每个状态的我们用函数F来估计它是否有希望。F包含两个部分：

F = G + H

G：就是普通宽度优先搜索中的从起始状态到当前状态的代价，

H：是一个估计的值，表示从当前状态到目标状态估计的代价。

H是由我们自己设计的，H函数设计的好坏决定了启发式算法的效率。H值越大，算法运行越快。

但是在设计评估函数时，需要注意一个很重要的性质：****评估函数的值一定要小于等于实际当前状态到目标状态的代价****。

否则虽然你的程序运行速度加快，但是可能在搜索过程中漏掉了最优解。相对的，只要评估函数的值小于等于实际当前状态到目标状态的代价，就一定能找到最优解

F：评估值和状态值的总和。

同时在启发式搜索中将原来的一个队列变成了两个队列：openlist和closelist。

在openlist中的状态，其F值还可能发生变化。而在closelist中的状态，其F值一定不会再发生改变。

整个搜索解的流程变为：

1. 计算初始状态的F值，并将其加入openlist
2. 从openlist中取出F值最小的状态u，并将u加入closelist。若u为目标状态，结束搜索；
3. 对u进行扩展，假设其扩展的状态为v：若v未出现过，计算v的f值并加入openlist；若v在openlist中，更新v的F值，取较小的一个；若v在closelist中，抛弃该状态。
4. 若openlist为空，结束搜索。否则回到2。

利用这个方法可以避免搜索一些明显会远离目标状态的状态，从而缩小搜索空间，早一步搜索到目标结果。

在启发式搜索中，最重要的是评估函数的选取，一个好的评估函数能够更快的趋近于目标状态。

启发式搜索在某些情况下并不一定好用，一方面取决于评估函数的选取，另一个方面由于在选取状态时也会有额外的开销。而快速趋近目标结果所减少的时间，能否弥补这一部分开销也是非常关键的。

所以根据题目选取合适的搜索方法才是最重要的。

问题：八数码求解的步数

伪代码：

search(status):

start.status = status

start.g = 0 // 实际步数

start.h = evaluate(start.status)

start.f = start.g + start.h

openlist.insert(start)

While (!openlist.isEmpty())

u = openlist.getMinFStatus()

closelist.insert(u)

For v is u.neighborStatus

If (v in openlist) Then

// 更新v的f值

If (v.f > v.h + u.g + 1) Then

v.f = v.h + u.g + 1

End If

Else If (v in closelist)

continue

Else

v.g = u.g + 1

v.h = evaluate(v.status)

v.f = v.g + v.h

openlist.insert(v)

End If

End For

End While

源码：

#include <cstring>

#include <cmath>

#include <queue>

#include <stack>

#include <algorithm>

#include <iostream>

using namespace std;

struct node{

int f,h,g;

int x,y;

char map[3][3];

friend bool operator< (const node &a,const node &b){

if(a.f==b.f)

return a.g<b.g;

return a.f>b.f;

}

};

node start;

int Hash[15]; // Hash[i] 存 i! 1~9

int pos[][2]= {{0,0},{0,1},{0,2},{1,0},{1,1},{1,2},{2,0},{2,1},{2,2}};// 0~8 的目标位置的坐标

int to[4][2]={0,-1,0,1,-1,0,1,0};

bool vis[500000];

//判断不可能的状况

bool check(){

int s[20];

int cnt = 0;

for(int i = 0; i<3; i++){

for(int j = 0; j<3; j++){ P

s[3\*i+j] = start.map[i][j];

if(s[3\*i+j] == 'x')

continue;

for(int k = 3\*i+j-1; k>=0; k--){

if(s[k] == 'x')

continue;

if(s[k]>s[3\*i+j])

cnt++;

}

}

}

if(cnt%2)

return false;

return true;

}

//康托

int solve(node a){

int s[20];

int ans = 0;

for(int i = 0; i<3; i++){

for(int j = 0; j<3; j++){

s[3\*i+j] = a.map[i][j];

int cnt = 0;

for(int k = 3\*i+j-1; k>=0; k--){

if(s[k]>s[3\*i+j])

cnt++; //前面比s[3\*i+j]大的，有多少个

}

ans = ans+Hash[i\*3+j]\*cnt;

}

}

return ans;//第ans大

} M’

//得到H值，曼哈顿距离之和

int get\_h(node a){

int ans = 0;

for(int i = 0; i<3; i++){

for(int j = 0; j<3; j++){

if(a.map[i][j] == 'x')

continue;

int k = a.map[i][j]-'1';

ans+=abs(pos[k][0]-i)+abs(pos[k][1]-j);

}

}

return ans;

}

int bfs(){

memset(vis,0,sizeof(vis)); // queue<node> Q;

priority\_queue<node> Q;

start.g = 0;

start.h = get\_h(start);

start.f = start.h;

vis[solve(start)]=true;

if(solve(start)==0) return 0;

Q.push(start);

node next;

while(!Q.empty()){

node a = Q.top();

Q.pop(); // node a = Q.front();

int k\_s = solve(a);

vis[k\_s]=true;

for(int i = 0; i<4; i++){

next = a;

next.x+=to[i][0];

next.y+=to[i][1];

if(next.x < 0 || next.y < 0 || next.x>2 || next.y > 2)

continue;

next.map[a.x][a.y] = a.map[next.x][next.y];

next.map[next.x][next.y] = 'x';

next.g+=1;

next.h = get\_h(next);

next.f = next.g+next.h;

int k\_n = solve(next);

if(vis[k\_n])

continue;

Q.push(next);

if(k\_n == 0)

return next.g;

}

}

}

int main(){

Hash[0] = 1;

for(int i = 1; i<=9; i++)

Hash[i] = Hash[i-1]\*i;

int t=0;

cin>>t;

char a=0;

while(t--){

for (int i=0;i<3;i++){

for (int j=0;j<3;j++){

cin>>a;

start.map[i][j]=a;

if(a=='0'){

start.map[i][j]='x';

start.x=i;

start.y=j;

}

}

}

if(!check()){

cout<<"No Solution!"<<endl;

}

else cout<<bfs()<<endl;

}

return 0;

}

怎样判断**八数码问题**是否有解

在分析之前，先引进逆序和逆序数的概念：对于棋子数列中任何一个棋子c[i](1≤i≤8)，如果有j>i且c[j]<c[i]，那么c[j]是c[i]的一个逆序，或者说c[i]和c[j]构成一个逆序对。定义棋子c[i]的逆序数为c[i]的逆序个数，棋子数列的逆序数为该数列所有棋子的逆序数总和。如果交换任何两个相邻的棋子，那么棋子数列的逆序数将发生奇偶性互变。其证明很简单，假设交换的是c[i]和c[i+1]，c[j](1≤j≤i-1或i+2≤j≤8)的逆序数并不改变。若交换之前c[i]<c[i+1]，那么交换之后c[i]的逆序数不变，而c[i+1]的逆序数加1；若交换之前c[i]>c[i+1]，那么交换之后c[i]的逆序数减1，而c[i+1]的逆序数不变。显然，空格与相邻棋子的交换不会改变棋局中棋子数列的逆序数的奇偶性。由于最终的逆序数是0，所以当初始状态棋局的棋子数列的逆序数是奇数时**无解**，偶数时有解。