**A\*算法求解八数码问题**

1. 八数码问题描述

所谓八数码问题起源于一种游戏:在一个3×3的方阵中放入八个数码1、2、3、4、5、6、7、8，其中一个单元格是空的。将任意摆放的数码盘（城初始状态）逐步摆成某个指定的数码盘的排列（目标状态），如图1所示



图1 八数码问题的某个初始状态和目标状态

对于以上问题，我们可以把数码的移动等效城空格的移动。如图1的初始排列，数码7右移等于空格左移。那么对于每一个排列，可能的一次数码移动最多只有4中，即空格左移、空格右移、空格上移、空格下移。最少有两种（当空格位于方阵的4个角时）。所以，问题就转换成如何从初始状态开始，使空格经过最小的移动次数最后排列成目标状态。

1. 八数码问题的求解算法

2.1 盲目搜索

宽度优先搜索算法、深度优先搜索算法

2.2 启发式搜索

启发式搜索算法的基本思想是：定义一个评价函数f，对当前的搜索状态进行评估，找出一个最有希望的节点来扩展。

先定义下面几个函数的含义：

f\*(n)=g\*(n)+h\*(n) (1)

式中g\*(n)表示从初始节点s到当前节点n的最短路径的耗散值；h\*(n)表示从当前节点n到目标节点g的最短路径的耗散值，f\*(n)表示从初始节点s经过n到目标节点g的最短路径的耗散值。

评价函数的形式可定义如(2)式所示：

f(n)=g(n)+h(n) (2)

其中n是被评价的当前节点。f(n)、g(n)和h(n)分别表示是对f\*(n)、g\*(n)和h\*(n)3个函数值的估计值。

利用评价函数f(n)=g(n)+h(n)来排列OPEN表节点顺序的图搜索算法称为算法A。在A算法中，如果对所有的x，

h(x)<=h\*(x) (3)

成立，则称好h(x)为h\*(x)的下界，它表示某种偏于保守的估计。采用h\*(x)的下界h(x)为启发函数的A算法，称为A\*算法。

针对八数码问题启发函数设计如下：

f(n)=d(n)+p(n) (4)

其中A\*算法中的g(n)根据具体情况设计为d(n)，意为n节点的深度，而h(n)设计为

把S放入OPEN表，记f=h

OPEN=NULL?

是

失败

扩展BESTNODE,产生其后继结点SUCCESSOR

选取OPEN表上未设置过的具有最小f值的节点

BESTNODE,放入CLOSED表

BESTNODE是目标节点

建立从SUCCESSOR返回BESTNODE的指针

计算g(SUC)=g(BES)+k(BES,SUC)

SUC∈OPEN

开始

g(SUC)<g(OLD)

SUC=OLD,把它添加到BESTNDOE的后继结点表中

重新确定OLD的父辈节点为BESTNODE，并修正父辈节点的g值和f值，记下g(OLD)

是

成功

SUC∈CLOSED

把SUCCESSOR放入OPEN表，添进BESTNODE的后裔表

计算f值

是

否

是

否

是

否

否

否

图2 A\*算法流程图

p(n)，意为放错的数码与正确的位置距离之和。

由于实际情况中，一个将牌的移动都是单步进行的，没有交换拍等这样的操作。所以要把所有的不在位的将牌，移动到各自的目标位置上，至少要移动从他们各自的位置到目标位置的距离和这么多次，所以最有路径的耗散值不会比该值小，因此该启发函数h(n)满足A\*算法的条件。

1. A\*算法流程图,如图2
2. A\*算法总结

4.1，把起始状态添加到开启列表。

4.2，重复如下工作：

a) 寻找开启列表中f值最低的节点，我们称它为BESTNOE

b) 把它切换到关闭列表中。

c) 对相邻的4个节点中的每一个

\*如果它不在开启列表，也不在关闭列表，把它添加到开启列表中。把BESTNODE作为这一节点的父节点。记录这一节点的f和g值

\*如果它已在开启或关闭列表中，用g值为参考检查新的路径是否更好。更低的g值意味着更好的路径。如果这样，就把这一节点的父节点改为BESTNODE，并且重新计算这一节点的f和g值，如果保持开启列表的f值排序，改变之后需要重新对开启列表排序。

d) 停止

把目标节点添加到关闭列表，这时候路径被找到，或者没有找到路径，开启列表已经空了，这时候路径不存在。

4.3，保存路径。从目标节点开始，沿着每一节点的父节点移动直到回到起始节点。这就是求得的路径。

5、数据结构

采用结构体来保存八数码的状态、f和g的值以及该节点的父节点；

struct Node{

int s[3][3];//保存八数码状态，0代表空格

int f,g;//启发函数中的f和g值

struct Node \* next;

struct Node \*previous;//保存其父节点

};

6、实验结果，如图3所示

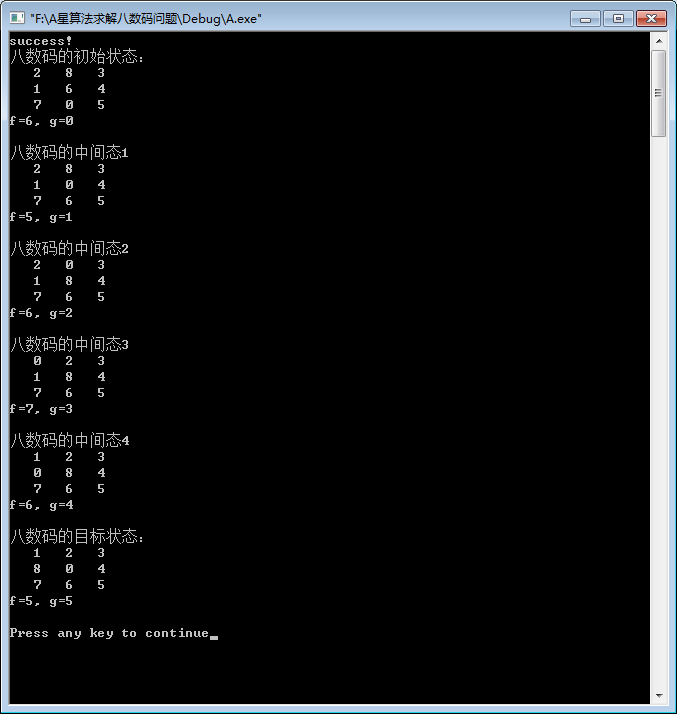


图3 A\*算法求解八数码问题实验结果

7、源代码

//-----------------------------------------------------------------------------

//代码：利用A\*算法求解八数码问题。

//八数码问题的启发函数设计为：f(n)=d(n)+p(n),其中A\*算法中的g(n)根据具体情况设计为d(n)，意为n节点的深度，而h(n)设计为p(n)，意为放错的数码与正确的位置距离之和。

//后继结点的获取：数码的移动等效为空格的移动。首先判断空格上下左右的可移动性，其次移动空格获取后继结点。

//-----------------------------------------------------------------------------

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

//八数码状态对应的节点结构体

struct Node{

int s[3][3];//保存八数码状态，0代表空格

int f,g;//启发函数中的f和g值

struct Node \* next;

struct Node \*previous;//保存其父节点

};

int open\_N=0; //记录Open列表中节点数目

//八数码初始状态

int inital\_s[3][3]={

2,8,3,

1,6,4,

7,0,5

};

//八数码目标状态

int final\_s[3][3]={

1,2,3,

8,0,4,

7,6,5

};

//------------------------------------------------------------------------

//添加节点函数入口，方法：通过插入排序向指定表添加

//------------------------------------------------------------------------

void Add\_Node( struct Node \*head, struct Node \*p)

{

struct Node \*q;

if(head->next)//考虑链表为空

{ q = head->next;

if(p->f < head->next->f){//考虑插入的节点值比链表的第一个节点值小

p->next = head->next;

head->next = p;

}

else {

while(q->next)//考虑插入节点x，形如a<= x <=b

{

if((q->f < p->f ||q->f == p->f) && (q->next->f > p->f || q->next->f == p->f)){

p->next = q->next;

q->next = p;

break;

}

q = q->next;

}

if(q->next == NULL) //考虑插入的节点值比链表最后一个元素的值更大

q->next = p;

}

}

else head->next = p;

}

//------------------------------------------------------------------------

//删除节点函数入口

//------------------------------------------------------------------------

void del\_Node(struct Node \* head, struct Node \*p )

{

struct Node \*q;

q = head;

while(q->next)

{

if(q->next == p){

q->next = p->next;

p->next = NULL;

if(q->next == NULL) return;

// free(p);

}

q = q->next;

}

}

//------------------------------------------------------------------------

//判断两个数组是否相等函数入口

//------------------------------------------------------------------------

int equal(int s1[3][3], int s2[3][3])

{

int i,j,flag=0;

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

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

if(s1[i][j] != s2[i][j]){flag = 1; break;}

if(!flag)

return 1;

else return 0;

}

//------------------------------------------------------------------------

//判断后继节点是否存在于Open或Closed表中函数入口

//------------------------------------------------------------------------

int exit\_Node(struct Node \* head,int s[3][3], struct Node \*Old\_Node)

{

struct Node \*q=head->next;

int flag = 0;

while(q)

if(equal(q->s,s)) {

flag=1;

Old\_Node->next = q;

return 1;}

else q = q->next;

if(!flag) return 0;

}

//------------------------------------------------------------------------

//计算p(n)的函数入口

//其中p(n)为放错位的数码与其正确的位置之间距离之和

//具体方法：放错位的数码与其正确的位置对应下标差的绝对值之和

//------------------------------------------------------------------------

int wrong\_sum(int s[3][3])

{

int i,j,fi,fj,sum=0;

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

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

{

for(fi=0; fi<3; fi++)

for(fj=0; fj<3; fj++)

if((final\_s[fi][fj] == s[i][j])){

sum += fabs(i - fi) + fabs(j - fj);

break;

}

}

return sum;

}

//------------------------------------------------------------------------

//获取后继结点函数入口

//检查空格每种移动的合法性，如果合法则移动空格得到后继结点

//------------------------------------------------------------------------

int get\_successor(struct Node \* BESTNODE, int direction, struct Node \*Successor)//扩展BESTNODE，产生其后继结点SUCCESSOR

{

int i,j,i\_0,j\_0,temp;

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

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

Successor->s[i][j] = BESTNODE->s[i][j];

//获取空格所在位置

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

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

if(BESTNODE->s[i][j] == 0){i\_0 = i; j\_0 = j;break;}

switch(direction)

{

case 0: if((i\_0-1)>-1 ){

temp = Successor->s[i\_0][j\_0];

Successor->s[i\_0][j\_0] = Successor->s[i\_0-1][j\_0];

Successor->s[i\_0-1][j\_0] = temp;

return 1;

}

else return 0;

case 1: if((j\_0-1)>-1){

temp = Successor->s[i\_0][j\_0];

Successor->s[i\_0][j\_0] = Successor->s[i\_0][j\_0-1];

Successor->s[i\_0][j\_0-1] = temp;

return 1;

}

else return 0;

case 2: if( (j\_0+1)<3){

temp = Successor->s[i\_0][j\_0];

Successor->s[i\_0][j\_0] = Successor->s[i\_0][j\_0+1];

Successor->s[i\_0][j\_0+1] = temp;

return 1;

}

else return 0;

case 3: if((i\_0+1)<3 ){

temp = Successor->s[i\_0][j\_0];

Successor->s[i\_0][j\_0] = Successor->s[i\_0+1][j\_0];

Successor->s[i\_0+1][j\_0] = temp;

return 1;

}

else return 0;

}

}

//------------------------------------------------------------------------

//从OPen表获取最佳节点函数入口

//------------------------------------------------------------------------

struct Node \* get\_BESTNODE(struct Node \*Open)

{

return Open->next;

}

//------------------------------------------------------------------------

//输出最佳路径函数入口

//------------------------------------------------------------------------

void print\_Path(struct Node \* head)

{

struct Node \*q, \*q1,\*p;

int i,j,count=1;

p = (struct Node \*)malloc(sizeof(struct Node));

//通过头插法变更节点输出次序

p->previous = NULL;

q = head;

while(q)

{

q1 = q->previous;

q->previous = p->previous;

p->previous = q;

q = q1;

}

q = p->previous;

while(q)

{

if(q == p->previous)printf("八数码的初始状态：\n");

else if(q->previous == NULL)printf("八数码的目标状态：\n");

else printf("八数码的中间态%d\n",count++);

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

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

{

printf("%4d",q->s[i][j]);

if(j == 2)printf("\n");

}

printf("f=%d, g=%d\n\n",q->f,q->g);

q = q->previous;

}

}

//------------------------------------------------------------------------

//A\*子算法入口:处理后继结点

//------------------------------------------------------------------------

void sub\_A\_algorithm(struct Node \* Open, struct Node \* BESTNODE, struct Node \* Closed,struct Node \*Successor)

{

struct Node \* Old\_Node = (struct Node \*)malloc(sizeof(struct Node));

Successor->previous = BESTNODE;//建立从successor返回BESTNODE的指针

Successor->g = BESTNODE->g + 1;//计算后继结点的g值

//检查后继结点是否已存在于Open和Closed表中，如果存在：该节点记为old\_Node，比较后继结点的g值和表中old\_Node节点

//g值，前者小代表新的路径比老路径更好，将Old\_Node的父节点改为BESTNODE，并修改其f，g值，后者小则什么也不做。

//即不存在Open也不存在Closed表则将其加入OPen表，并计算其f值

if( exit\_Node(Open, Successor->s, Old\_Node) ){

if(Successor->g < Old\_Node->g){

Old\_Node->next->previous = BESTNODE;//将Old\_Node的父节点改为BESTNODE

Old\_Node->next->g = Successor->g;//修改g值

Old\_Node->next->f = Old\_Node->g + wrong\_sum(Old\_Node->s);//修改f值

//排序~~~~~~~~~~~~~~~~~~

del\_Node(Open, Old\_Node);

Add\_Node(Open, Old\_Node);

}

}

else if( exit\_Node(Closed, Successor->s, Old\_Node)){

if(Successor->g < Old\_Node->g){

Old\_Node->next->previous = BESTNODE;

Old\_Node->next->g = Successor->g;

Old\_Node->next->f = Old\_Node->g + wrong\_sum(Old\_Node->s);

//排序~~~~~~~~~~~~~~~~~~

del\_Node(Closed, Old\_Node);

Add\_Node(Closed, Old\_Node);

}

}

else {

Successor->f = Successor->g + wrong\_sum(Successor->s);

Add\_Node(Open, Successor);

open\_N++;

}

}

//------------------------------------------------------------------------

//A\*算法入口

//八数码问题的启发函数为：f(n)=d(n)+p(n)

//其中A\*算法中的g(n)根据具体情况设计为d(n)，意为n节点的深度，而h(n)设计为p(n)，

//意为放错的数码与正确的位置距离之和

//------------------------------------------------------------------------

void A\_algorithm(struct Node \* Open, struct Node \* Closed) //A\*算法

{

int i,j;

struct Node \* BESTNODE, \*inital, \* Successor;

inital = (struct Node \* )malloc(sizeof(struct Node));

//初始化起始节点

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

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

inital->s[i][j] = inital\_s[i][j];

inital->f = wrong\_sum(inital\_s);

inital->g = 0;

inital->previous = NULL;

inital->next = NULL;

Add\_Node(Open, inital);//把初始节点放入OPEN表

open\_N++;

while(1)

{

if(open\_N == 0){printf("failure!"); return;}

else {

BESTNODE = get\_BESTNODE(Open);//从OPEN表获取f值最小的BESTNODE，将其从OPEN表删除并加入CLOSED表中

del\_Node(Open, BESTNODE);

open\_N--;

Add\_Node(Closed, BESTNODE);

if(equal(BESTNODE->s, final\_s)) {//判断BESTNODE是否为目标节点

printf("success!\n");

print\_Path(BESTNODE);

return;

}

//针对八数码问题，后继结点Successor的扩展方法：空格（二维数组中的0）上下左右移动，

//判断每种移动的有效性，有效则转向A\*子算法处理后继节点，否则进行下一种移动

else{

Successor = (struct Node \* )malloc(sizeof(struct Node)); Successor->next = NULL;

if(get\_successor(BESTNODE, 0, Successor))sub\_A\_algorithm( Open, BESTNODE, Closed, Successor);

Successor = (struct Node \* )malloc(sizeof(struct Node)); Successor->next = NULL;

if(get\_successor(BESTNODE, 1, Successor))sub\_A\_algorithm( Open, BESTNODE, Closed, Successor);

Successor = (struct Node \* )malloc(sizeof(struct Node)); Successor->next = NULL;

if(get\_successor(BESTNODE, 2, Successor))sub\_A\_algorithm( Open, BESTNODE, Closed, Successor);

Successor = (struct Node \* )malloc(sizeof(struct Node)); Successor->next = NULL;

if(get\_successor(BESTNODE, 3, Successor))sub\_A\_algorithm( Open, BESTNODE, Closed, Successor);

}

}

}

}

//------------------------------------------------------------------------

//main()函数入口

//定义Open和Closed列表。Open列表：保存待检查节点。Closed列表：保存不需要再检查的节点

//------------------------------------------------------------------------

void main()

{

struct Node \* Open = (struct Node \* )malloc(sizeof(struct Node));

struct Node \* Closed = (struct Node \* )malloc(sizeof(struct Node));

Open->next = NULL ; Open->previous = NULL;

Closed->next =NULL; Closed->previous = NULL;

A\_algorithm(Open, Closed);

}