(1)实验环境：Windows XP

(2)实验编程工具：VC++6.0

**3.2 数据结构**

1)定义状态图中的结点数据结构

typedef struct Node

{

status data;//结点所存储的状态

struct Node \*parent;//指向结点的父亲结点

struct SpringLink \*child;//指向结点的后继结点

struct Node \*next;//指向open或者closed表中的后一个结点

int fvalue;//结点的总的路径

int gvalue;//结点的实际路径

int hvalue;//结点的到达目标的苦难程度

}NNode , \*PNode;

2)定义存储指向结点后继结点的指针的地址

typedef struct SpringLink

{

struct Node \*pointData;//指向结点的指针

struct SpringLink \*next;//指向兄第结点

}SPLink , \*PSPLink;

3)初始化一个空链表

void initLink(PNode &Head)

**附录—源代码及其注释**

#include "iostream"

#include "stdlib.h"

#include "conio.h"

#define size 3

using namespace std;

//定义二维数组来存储数据表示某一个特定状态

typedef int status[size][size];

struct SpringLink;

//定义状态图中的结点数据结构

typedef struct Node

{

status data;//结点所存储的状态

struct Node \*parent;//指向结点的父亲结点

struct SpringLink \*child;//指向结点的后继结点

struct Node \*next;//指向open或者closed表中的后一个结点

int fvalue;//结点的总的路径

int gvalue;//结点的实际路径

int hvalue;//结点的到达目标的苦难程度

}NNode , \*PNode;

//定义存储指向结点后继结点的指针的地址

typedef struct SpringLink

{

struct Node \*pointData;//指向结点的指针

struct SpringLink \*next;//指向兄第结点

}SPLink , \*PSPLink;

PNode open;

PNode closed;

//开始状态与目标状态

status startt = {0,1,2,3,4,5,6,7,8};

status target = {1,4,2,3,5,8,6,7,0};

//初始化一个空链表

void initLink(PNode &Head)

{

Head = (PNode)malloc(sizeof(NNode));

Head->next = NULL;

}

//判断链表是否为空

bool isEmpty(PNode Head)

{

if(Head->next == NULL)

return true;

else

return false;

}

//从链表中拿出一个数据

void popNode(PNode &Head , PNode &FNode)

{

if(isEmpty(Head))

{

FNode = NULL;

return;

}

FNode = Head->next;

Head->next = Head->next->next;

FNode->next = NULL;

}

//向结点的最终后继结点链表中添加新的子结点

void addSpringNode(PNode &Head , PNode newData)

{

PSPLink newNode = (PSPLink)malloc(sizeof(SPLink));

newNode->pointData = newData;

newNode->next = Head->child;

Head->child = newNode;

}

//释放状态图中存放结点后继结点地址的空间

void freeSpringLink(PSPLink &Head)

{

PSPLink tmm;

while(Head != NULL)

{

tmm = Head;

Head = Head->next;

free(tmm);

}

}

//释放open表与closed表中的资源

void freeLink(PNode &Head)

{

PNode tmn;

tmn = Head;

Head = Head->next;

free(tmn);

while(Head != NULL)

{

//首先释放存放结点后继结点地址的空间

freeSpringLink(Head->child);

tmn = Head;

Head = Head->next;

free(tmn);

}

}

//向普通链表中添加一个结点

void addNode(PNode &Head , PNode &newNode)

{

newNode->next = Head->next;

Head->next = newNode;

}

//向非递减排列的链表中添加一个结点

void addAscNode(PNode &Head , PNode &newNode)

{

PNode P;

PNode Q;

P = Head->next;

Q = Head;

while(P != NULL && P->fvalue < newNode->fvalue)

{

Q = P;

P = P->next;

}

//上面判断好位置之后，下面就是简单的插入了

newNode->next = Q->next;

Q->next = newNode;

}

//计算结点额h值

int computeHValue(PNode theNode)

{

int num = 0;

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

{

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

{

if(theNode->data[i][j] != target[i][j])

num++;

}

}

return num;

}

//计算结点的f，g，h值

void computeAllValue(PNode &theNode , PNode parentNode)

{

if(parentNode == NULL)

theNode->gvalue = 0;

else

theNode->gvalue = parentNode->gvalue + 1;

theNode->hvalue = computeHValue(theNode);

theNode->fvalue = theNode->gvalue + theNode->hvalue;

}

//初始化函数，进行算法初始条件的设置

void initial()

{

//初始化open以及closed表

initLink(open);

initLink(closed);

//初始化起始结点，令初始结点的父节点为空结点

PNode NULLNode = NULL;

PNode Start = (PNode)malloc(sizeof(NNode));

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

{

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

{

Start->data[i][j] = startt[i][j];

}

}

Start->parent = NULL;

Start->child = NULL;

Start->next = NULL;

computeAllValue(Start , NULLNode);

//起始结点进入open表

addAscNode(open , Start);

}

//将B节点的状态赋值给A结点

void statusAEB(PNode &ANode , PNode BNode)

{

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

{

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

{

ANode->data[i][j] = BNode->data[i][j];

}

}

}

//两个结点是否有相同的状态

bool hasSameStatus(PNode ANode , PNode BNode)

{

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

{

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

{

if(ANode->data[i][j] != BNode->data[i][j])

return false;

}

}

return true;

}

//结点与其祖先结点是否有相同的状态

bool hasAnceSameStatus(PNode OrigiNode , PNode AnceNode)

{

while(AnceNode != NULL)

{

if(hasSameStatus(OrigiNode , AnceNode))

return true;

AnceNode = AnceNode->parent;

}

return false;

}

//取得方格中空的格子的位置

void getPosition(PNode theNode , int &row , int &col)

{

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

{

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

{

if(theNode->data[i][j] == 0)

{

row = i;

col = j;

return;

}

}

}

}

//交换两个数字的值

void changeAB(int &A , int &B)

{

int C;

C = B;

B = A;

A = C;

}

//检查相应的状态是否在某一个链表中

bool inLink(PNode spciNode , PNode theLink , PNode &theNodeLink , PNode &preNode)

{

preNode = theLink;

theLink = theLink->next;

while(theLink != NULL)

{

if(hasSameStatus(spciNode , theLink))

{

theNodeLink = theLink;

return true;

}

preNode = theLink;

theLink = theLink->next;

}

return false;

}

//产生结点的后继结点(与祖先状态不同)链表

void SpringLink(PNode theNode , PNode &spring)

{

int row;

int col;

getPosition(theNode , row , col);

//空的格子右边的格子向左移动

if(col != 2)

{

PNode rlNewNode = (PNode)malloc(sizeof(NNode));

statusAEB(rlNewNode , theNode);

changeAB(rlNewNode->data[row][col] , rlNewNode->data[row][col + 1]);

if(hasAnceSameStatus(rlNewNode , theNode->parent))

{

free(rlNewNode);//与父辈相同，丢弃本结点

}

else

{

rlNewNode->parent = theNode;

rlNewNode->child = NULL;

rlNewNode->next = NULL;

computeAllValue(rlNewNode , theNode);

//将本结点加入后继结点链表

addNode(spring , rlNewNode);

}

}

//空的格子左边的格子向右移动

if(col != 0)

{

PNode lrNewNode = (PNode)malloc(sizeof(NNode));

statusAEB(lrNewNode , theNode);

changeAB(lrNewNode->data[row][col] , lrNewNode->data[row][col - 1]);

if(hasAnceSameStatus(lrNewNode , theNode->parent))

{

free(lrNewNode);//与父辈相同，丢弃本结点

}

else

{

lrNewNode->parent = theNode;

lrNewNode->child = NULL;

lrNewNode->next = NULL;

computeAllValue(lrNewNode , theNode);

//将本结点加入后继结点链表

addNode(spring , lrNewNode);

}

}

//空的格子上边的格子向下移动

if(row != 0)

{

PNode udNewNode = (PNode)malloc(sizeof(NNode));

statusAEB(udNewNode , theNode);

changeAB(udNewNode->data[row][col] , udNewNode->data[row - 1][col]);

if(hasAnceSameStatus(udNewNode , theNode->parent))

{

free(udNewNode);//与父辈相同，丢弃本结点

}

else

{

udNewNode->parent = theNode;

udNewNode->child = NULL;

udNewNode->next = NULL;

computeAllValue(udNewNode , theNode);

//将本结点加入后继结点链表

addNode(spring , udNewNode);

}

}

//空的格子下边的格子向上移动

if(row != 2)

{

PNode duNewNode = (PNode)malloc(sizeof(NNode));

statusAEB(duNewNode , theNode);

changeAB(duNewNode->data[row][col] , duNewNode->data[row + 1][col]);

if(hasAnceSameStatus(duNewNode , theNode->parent))

{

free(duNewNode);//与父辈相同，丢弃本结点

}

else

{

duNewNode->parent = theNode;

duNewNode->child = NULL;

duNewNode->next = NULL;

computeAllValue(duNewNode , theNode);

//将本结点加入后继结点链表

addNode(spring , duNewNode);

}

}

}

//输出给定结点的状态

void outputStatus(PNode stat)

{

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

{

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

{

cout << stat->data[i][j] << " ";

}

cout << endl;

}

}

//输出最佳的路径

void outputBestRoad(PNode goal)

{

int deepnum = goal->gvalue;

if(goal->parent != NULL)

{

outputBestRoad(goal->parent);

}

cout << "第" << deepnum-- << "层的状态：" << endl;

outputStatus(goal);

}

void AStar()

{

PNode tmpNode;//指向从open表中拿出并放到closed表中的结点的指针

PNode spring;//tmpNode的后继结点链

PNode tmpLNode;//tmpNode的某一个后继结点

PNode tmpChartNode;

PNode thePreNode;//指向将要从closed表中移到open表中的结点的前一个结点的指针

bool getGoal = false;//标识是否达到目标状态

long numcount = 1;//记录从open表中拿出结点的序号

initial();//对函数进行初始化

initLink(spring);//对后继链表的初始化

tmpChartNode = NULL;

cout << "从open表中拿出的结点的状态及相应的值" << endl;

while(!isEmpty(open))

{

//从open表中拿出f值最小的元素,并将拿出的元素放入closed表中

popNode(open , tmpNode);

addNode(closed , tmpNode);

cout << "第" << numcount++ << "个状态是：" << endl;

outputStatus(tmpNode);

cout << "其f值为：" << tmpNode->fvalue << endl;

cout << "其g值为：" << tmpNode->gvalue << endl;

cout << "其h值为：" << tmpNode->hvalue << endl;

//如果拿出的元素是目标状态则跳出循环

if(computeHValue(tmpNode) == 0)

{

getGoal = true;

break;

}

//产生当前检测结点的后继(与祖先不同)结点列表，产生的后继结点的parent属性指向当前检测的结点

SpringLink(tmpNode , spring);

//遍历检测结点的后继结点链表

while(!isEmpty(spring))

{

popNode(spring , tmpLNode);

//状态在open表中已经存在，thePreNode参数在这里并不起作用

if(inLink(tmpLNode , open , tmpChartNode , thePreNode))

{

addSpringNode(tmpNode , tmpChartNode);

if(tmpLNode->gvalue < tmpChartNode->gvalue)

{

tmpChartNode->parent = tmpLNode->parent;

tmpChartNode->gvalue = tmpLNode->gvalue;

tmpChartNode->fvalue = tmpLNode->fvalue;

}

free(tmpLNode);

}

//状态在closed表中已经存在

else if(inLink(tmpLNode , closed , tmpChartNode , thePreNode))

{

addSpringNode(tmpNode , tmpChartNode);

if(tmpLNode->gvalue < tmpChartNode->gvalue)

{

PNode commu;

tmpChartNode->parent = tmpLNode->parent;

tmpChartNode->gvalue = tmpLNode->gvalue;

tmpChartNode->fvalue = tmpLNode->fvalue;

freeSpringLink(tmpChartNode->child);

tmpChartNode->child = NULL;

popNode(thePreNode , commu);

addAscNode(open , commu);

}

free(tmpLNode);

}

//新的状态即此状态既不在open表中也不在closed表中

else

{

addSpringNode(tmpNode , tmpLNode);

addAscNode(open , tmpLNode);

}

}

}

//目标可达的话，输出最佳的路径

if(getGoal)

{

cout << endl;

cout << "最佳路径长度为：" << tmpNode->gvalue << endl;

cout << "最佳路径为：" <<endl;

outputBestRoad(tmpNode);

}

//释放结点所占的内存

freeLink(open);

freeLink(closed);

getch();

}

int main()

{

AStar();

return 0;

}