3-8源代码：

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

int X, Y;

int k;

struct node

{

int q[3];

};

vector<node> s;

int q[500][3];

//用于存放搜索结点，q[][0]是左岸传教士人数

//q[][1]是左岸野蛮人人数，q[][2]是左岸船的数目

//q[][3]用于搜索中的父亲结点序号。

int ans=0;

int op\_num = 0;

int go[500][2];

int fx[500][500];

//安全状态：左岸中，传教士都在or都不在or传教士人数等于野人人数

int is\_safe(int state[3])

{

if ((state[0]==0||state[0]==X||state[0]==state[1])&&(state[1]>=0)&&(state[1]<=Y))

{

return 1;

}

return 0;

}

//是否到达目标状态

int is\_success(int state[3])

{

if (state[0]==0&&state[1]==0)

return 1;

return 0;

}

//该状态是否已经访问过

int vis(int state[3])

{

for (vector<node>::iterator it = s.begin(); it != s.end(); it++)

if ((\*it).q[0] == state[0] && (\*it).q[1] == state[1] && (\*it).q[2] == state[2])

return 1;

return 0;

}

int f1(int state[3])

{

return state[0]+state[1];

}

int f2(int state[3])

{

return state[0]+state[1]-2\*state[2];

}

int find\_max(int cur)

{

int max = -1;

int op = -1;

for (int j = 0; j < op\_num; j++)//分别考虑可能的动作

{

if (fx[cur+1][j] > max)

{

max = fx[cur+1][j];

op = j;

}

}

if (max == -1)

op = -1;

return op;

}

//过河操作

int search(int cur)

{

if (is\_success(q[cur]))

{

ans = cur;

return 1;

}

int state[3];

int j;

//cout<<"第"<<cur<<"层搜索"<<endl;

//获取当前搜索结点

//cout<<"展开结点"<<cur<<":"<<q[cur][0]<<' '<<q[cur][1]<<' '<<q[cur][2]<<endl;

if (q[cur][2])//船在左边

{

for (j = 0; j < op\_num; j++)//分别考虑可能的动作

{

state[0]=q[cur][0]-go[j][0];

state[1]=q[cur][1]-go[j][1];

state[2]=0;//船到了右边

fx[cur+1][j]=f2(state);

}

j = find\_max(cur);

while (j != -1)

{

fx[cur+1][j] = -1;

state[0]=q[cur][0]-go[j][0];

state[1]=q[cur][1]-go[j][1];

state[2]=0;//船到了右边

if (is\_safe(state)&&!vis(state))//如果是安全状态//判断与之前展开结点是否相同

{

node nd;

nd.q[0]=q[cur+1][0]=state[0];

nd.q[1]=q[cur+1][1]=state[1];

nd.q[2]=q[cur+1][2]=state[2];

s.push\_back(nd);

//cout<<"合法结点:"<<state[0]<<' '<<state[1]<<' '<<state[2]<<endl;

if (search(cur+1))

return 1;

}

j = find\_max(cur);

}

}

else //船在右边

{

for (j = 0; j < op\_num; j++)//分别考虑可能的动作

{

state[0]=q[cur][0]+go[j][0];

state[1]=q[cur][1]+go[j][1];

state[2]=1;

fx[cur+1][j]=f2(state);

}

j = find\_max(cur);

while (j != -1)

{

fx[cur+1][j] = -1;

state[0]=q[cur][0]+go[j][0];

state[1]=q[cur][1]+go[j][1];

state[2]=1; //船回到左边

if (is\_safe(state)&&!vis(state))//如果是安全状态且与之间状态不同

{

node nd;

nd.q[0]=q[cur+1][0]=state[0];

nd.q[1]=q[cur+1][1]=state[1];

nd.q[2]=q[cur+1][2]=state[2];

s.push\_back(nd);

//cout<<"合法结点:"<<state[0]<<' '<<state[1]<<' '<<state[2]<<endl;

if(search(cur+1))

return 1;

}

j = find\_max(cur);

}

}

return 0;

}

int main()

{

int n;

cout<<"请输入N：";

cin>>n;

cout<<"请输入k：";

cin>>k;

X = Y = n;

int state[3];

//初始状态

node nd;

nd.q[0]=state[0]=q[0][0]=X;

nd.q[1]=state[1]=q[0][1]=Y;

nd.q[2]=state[2]=q[0][2]=1;

s.push\_back(nd);

//初始化操作

cout<<"合法的操作组有："<<endl;

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

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

{

if (j >= i-j || j == 0)

{

go[op\_num][0] = j;

go[op\_num][1] = i-j;

cout<<go[op\_num][0]<<' '<<go[op\_num][1]<<endl;

op\_num++;

}

}

cout<<endl;

if (!search(0))

{

cout<<"无解"<<endl;

return 0;

}

cout<<"找到的解为:"<<endl;

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

{

//cout<<q[i][0]<<' '<<q[i][1]<<' '<<q[i][2]<<endl;

if (i > 0)

{

cout<<abs(q[i][0]-q[i-1][0])<<"个传教士和"<<abs(q[i][1]-q[i-1][1])<<"个野人";

if (q[i][2])

cout<<"从右岸乘船至左岸"<<endl;

else

cout<<"从左岸乘船至右岸"<<endl;

cout<<"左岸有"<<q[i][0]<<"个传教士和"<<q[i][1]<<"个野人"<<endl;

cout<<"右岸有"<<n-q[i][0]<<"个传教士和"<<n-q[i][1]<<"个野人"<<endl<<endl;

}

}

cout<<"本次搜索所花费的费用："<<ans<<endl;

return 0;

}

15数码问题实现

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#define N 4

typedef struct QNode{

int data[N][N]; //数据

int ancent; //标记方向左上右下分别为 1234 ，5为可以任意方向

int x; //标记0的横坐标

int y; //标记0的纵坐标

int gone; //是否遍历该节点，0未遍历，1遍历过

int value; //和目标的状态差=不在位将牌距离和+深度

int deep; //深度

struct QNode \*father; //存放前一节点在"store"数组中的位置

struct QNode \*next; //存放下一节点在"store"数组中的位置

}QNode, \*QueuePtr;

typedef struct{

QueuePtr head; //头结点

QueuePtr rear; //尾结点

}LinkQueue;

int A[N][N]={ //目标状态

{1,2,3,4},

{5,6,7,8},

{9,10,11,12},

{13,14,15,0}

};

int B[N][N]={ //初始状态

{5,1,2,4},

{9,6,3,8},

{13,15,10,11},

{14,0,7,12}

};

int x,y;

QueuePtr min; //存放最小的结点

bool begin\_opint(){ //判断输入的数据是否合法

int i,j;

for(i=0;i<N;i++){

for(j=0;j<N;j++){

if(A[i][j]==0){

x=i;y=j;

return true;}

}

}

return false;

}

bool compare(int a[N][N]){ //比较函数，判断当前状态是否与目标状态相等

int i,j;

for(i=0;i<N;i++){

for(j=0;j<N;j++){

if(a[i][j]!=B[i][j])

return false;

}

}

return true;

}

bool moveleft(int a[N][N],QueuePtr \*b,int x,int y){//向左移动函数

int k,i,j;

if(y==0)

return false;

for(i=0;i<N;i++){

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

(\*b)->data[i][j]=a[i][j];

}

k=(\*b)->data[x][y];

(\*b)->data[x][y]=(\*b)->data[x][y-1];

(\*b)->data[x][y-1]=k;

(\*b)->x=x;

(\*b)->y=y-1;

return true;

}

bool moveup(int a[N][N],QueuePtr \*b,int x,int y){//向上移动函数

int k,i,j;

if(x==0)

return false;

for(i=0;i<N;i++){

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

(\*b)->data[i][j]=a[i][j];

}

k=(\*b)->data[x][y];

(\*b)->data[x][y]=(\*b)->data[x-1][y];

(\*b)->data[x-1][y]=k;

(\*b)->x=x-1;

(\*b)->y=y;

return true;

}

bool movedown(int a[N][N],QueuePtr \*b,int x,int y){ //向下移动函数

int k,i,j;

if(x==N-1)return false;

for(i=0;i<N;i++){

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

(\*b)->data[i][j]=a[i][j];

}

k=(\*b)->data[x][y];

(\*b)->data[x][y]=(\*b)->data[x+1][y];

(\*b)->data[x+1][y]=k;

(\*b)->x=x+1;

(\*b)->y=y;

return true;

}

bool moveright(int a[N][N],QueuePtr \*b,int x,int y){ //向右移动函数

int k,i,j;

if(y==N-1)

return false;

for(i=0;i<N;i++){

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

(\*b)->data[i][j]=a[i][j];

}

k=(\*b)->data[x][y];

(\*b)->data[x][y]=(\*b)->data[x][y+1];

(\*b)->data[x][y+1]=k;

(\*b)->x=x;

(\*b)->y=y+1;

return true;

}

bool copy(QueuePtr \*a){ //复制函数

int i,j;

for(i=0;i<N;i++){

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

(\*a)->data[i][j]=A[i][j];

}

return true;

}

void output(QueuePtr \*p){ //输出函数

int i,j;

long int n=0;

for(;(\*p)->father!=NULL;(\*p)=(\*p)->father,n++){

for(i=0;i<N;i++){

for(j=0;j<N;j++){

printf(" %d",(\*p)->data[i][j]);

}printf("\n");

}printf("\n");

}

printf("step is %d\n",n-1);

}

int getvalue(QueuePtr \*p){ //计算耗散值函数

int count=0;//保存距离

bool test=true; //若已找到一个位置的值则继续找下一个

//计算不在位的距离和

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

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

test=true;

for(int k=0;k<N;k++){

for(int l=0;l<N;l++){

if((i!=(\*p)->x||j!=(\*p)->y)&&(\*p)->data[i][j]==B[k][l]){

count=count+abs(i-k)+abs(j-l);

test=false;

}

if(test==false) break;

}

if(test==false) break;

}

}

}

count=count+(\*p)->deep;//加上深度值

return count;

}

void main()

{

QueuePtr closed,p,q;

LinkQueue open;

if(!begin\_opint()){

printf("no 0 opint!!\n"); //确定0点

exit(0);

}

open.head=open.rear=(QueuePtr)malloc(sizeof(QNode));//头结点

open.head->father=NULL;

open.rear->next=open.head->next=NULL;

closed=(QueuePtr)malloc(sizeof(QNode));//头结点

closed->next=NULL;

closed->father=NULL;

p=(QueuePtr)malloc(sizeof(QNode));//S0进open表

copy(&p);

p->x=x;

p->y=y;

p->ancent=5;

p->deep=0; //s0的深度为0

p->gone=0;

p->father=open.head;

p->value=getvalue(&p);

p->next=open.head->next;

open.head->next=p;

open.rear=open.head;

if(compare(p->data)){

output(&p);

exit(0);

}

while(open.head->next!=NULL){

//寻找最小状态

for(min=q=open.head->next;q!=NULL;q=q->next){

if(q->value<=min->value&&q->gone==0){

min=q;

break;

}

}

min->gone=1; //改最小状态已遍历

min->father->next=min->next; //在open表中删除找到的最小态

min->next=closed->next; //插入closed表的表头

closed->next=min;

//空格向4个方向移动

switch(closed->next->ancent){

case 1:p=(QueuePtr)malloc(sizeof(QNode));//祖先结点从右来 if(moveleft(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=1;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(moveup(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=2;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(movedown(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=3;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

break;

case 2:p=(QueuePtr)malloc(sizeof(QNode));//祖先结点从下来 if(moveleft(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=1;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(moveup(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=2;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(moveright(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=4;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

break;

case 3:p=(QueuePtr)malloc(sizeof(QNode));//祖先结点从上来 if(moveleft(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=1;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(movedown(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=3;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(moveright(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=4;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

break;

case 4:p=(QueuePtr)malloc(sizeof(QNode));//祖先结点从左边来

if(moveup(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=2;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(movedown(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=3;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

if(compare(p->data)){ //比较输出结果

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(moveright(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=4;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

if(compare(p->data)){ //比较输出结果

output(&p);

exit(0);

}

}else free(p);

break;

default:p=(QueuePtr)malloc(sizeof(QNode));//初始情况

if(moveleft(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=1;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

//比较输出结果

if(compare(p->data)){

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode)); if(moveup(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=2;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

if(compare(p->data)){ //比较输出结果

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode)); if(movedown(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=3;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

if(compare(p->data)){ //比较输出结果

output(&p);

exit(0);

}

}else free(p);

p=(QueuePtr)malloc(sizeof(QNode));

if(moveright(closed->next->data,&p,closed->next->x,closed->next->y)){

p->father=closed->next;

p->ancent=4;

p->gone=0;

p->deep=min->deep+1;

p->value=getvalue(&p);

p->next=open.rear->next;

open.rear->next=p;

open.rear=p;

if(compare(p->data)){ //比较输出结果

output(&p);

exit(0);

}

}else free(p);

break;

}

}printf("error: no answer!\n");

}