深度优先搜索：

#include <stdio.h>

#include <string.h>

#include<stdlib.h>

typedef long UINT64;

typedef struct

{

char x; //位置x和位置y上的数字换位

char y; //其中x是0所在的位置

} EP\_MOVE;

#define SIZE 3 //8数码问题，理论上本程序也可解决15数码问题，

#define NUM SIZE \* SIZE //但move\_gen需要做很多修改，输入初始和结束状态的部分和check\_input也要修改

#define MAX\_NODE 1000000

#define MAX\_DEP 100

#define XCHG(a, b) { a=a + b; b=a - b; a=a - b; }

#define TRANS(a, b)

/\*{ long iii; (b)=0; for(iii=0; iii < NUM; iii++) (b)=((b) << 4) + a[iii]; }\*/ //将数组a转换为一个64位的整数b

#define RTRANS(a, b) \

{ \

long iii; \

UINT64 ttt=(a); \

for(iii=NUM - 1; iii >= 0; iii--) \

{ \

b[iii]=ttt & 0xf; \

ttt>>=4; \

} \

} //将一个64位整数a转换为数组b

//

typedef struct EP\_NODE\_Tag

{ UINT64 v; //保存状态，每个数字占4个二进制位，可解决16数码问题

struct EP\_NODE\_Tag \*prev; //父节点

struct EP\_NODE\_Tag \*small, \*big;

} EP\_NODE;

EP\_NODE m\_ar[MAX\_NODE];

EP\_NODE \*m\_root;

long m\_depth; //搜索深度

EP\_NODE m\_out[MAX\_DEP]; //输出路径

//

long move\_gen(EP\_NODE \*node, EP\_MOVE \*move)

{long pz; //0的位置

UINT64 t=0xf;

for(pz=NUM - 1; pz >= 0; pz--)

{if((node->v & t) == 0)

{ break; //找到0的位置

}

t<<=4;

}

switch(pz)

{case 0:

move[0].x=0;

move[0].y=1;

move[1].x=0;

move[1].y=3;

return 2;

case 1:

move[0].x=1;

move[0].y=0;

move[1].x=1;

move[1].y=2;

move[2].x=1;

move[2].y=4;

return 3;

case 2:

move[0].x=2;

move[0].y=1;

move[1].x=2;

move[1].y=5;

return 2;

case 3:

move[0].x=3;

move[0].y=0;

move[1].x=3;

move[1].y=6;

move[2].x=3;

move[2].y=4;

return 3;

case 4:

move[0].x=4;

move[0].y=1;

move[1].x=4;

move[1].y=3;

move[2].x=4;

move[2].y=5;

move[3].x=4;

move[3].y=7;

return 4;

case 5:

move[0].x=5;

move[0].y=2;

move[1].x=5;

move[1].y=4;

move[2].x=5;

move[2].y=8;

return 3;

case 6:

move[0].x=6;

move[0].y=3;

move[1].x=6;

move[1].y=7;

return 2;

case 7:

move[0].x=7;

move[0].y=6;

move[1].x=7;

move[1].y=4;

move[2].x=7;

move[2].y=8;

return 3;

case 8:

move[0].x=8;

move[0].y=5;

move[1].x=8;

move[1].y=7;

return 2;

}

return 0;

}

long mov(EP\_NODE \*n1, EP\_MOVE \*mv, EP\_NODE \*n2)

//走一步，返回走一步后的结果

{

char ss[NUM];

RTRANS(n1->v, ss);

XCHG(ss[mv->x], ss[mv->y]);

TRANS(ss, n2->v);

return 0;

}

long add\_node(EP\_NODE \*node, long r)

{

EP\_NODE \*p=m\_root;

EP\_NODE \*q;

while(p)

{ q=p;

if(p->v == node->v) return 0;

else if(node->v > p->v) p=p->big;

else if(node->v < p->v) p=p->small;

}

m\_ar[r].v=node->v;

m\_ar[r].prev=node->prev;

m\_ar[r].small=NULL;

m\_ar[r].big=NULL;

if(node->v > q->v)

{ q->big= &m\_ar[r];

}

else if(node->v < q->v)

{ q->small= &m\_ar[r];

}

return 1;

}

/\*得到节点所在深度\*/

long get\_node\_depth(EP\_NODE \*node)

{ long d=0;

while(node->prev)

{ d++;

node=node->prev;

}

return d;

}

/\*返回值：成功－返回搜索节点数，节点数不够－(-1)，无解－(-2)\*/

long bfs\_search(char \*begin, char \*end)

{ long h=0, r=1, c, i, j;

EP\_NODE l\_end, node, \*pnode;

EP\_MOVE mv[4]; //每个局面最多4种走法

TRANS(begin, m\_ar[0].v);

TRANS(end, l\_end.v);

m\_ar[0].prev=NULL;

m\_root=m\_ar;

m\_root->small=NULL;

m\_root->big=NULL;

while((h < r) && (r < MAX\_NODE - 4))

{ c=move\_gen(&m\_ar[h], mv);

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

{ mov(&m\_ar[h], &mv[i], &node);

node.prev= &m\_ar[h];

if(node.v == l\_end.v)

{ pnode= &node;

j=0;

while(pnode->prev)

{ m\_out[j]=\*pnode;

j++;

pnode=pnode->prev;

}

m\_depth=j;

return r;

}

if(add\_node(&node, r)) r++; //只能对历史节点中没有的新节点搜索，否则会出现环

}

h++;

printf("\rSearch...%9d/%d @ %d", h, r, get\_node\_depth(&m\_ar[h]));

}

if(h == r)

{ return -2; }

else

{return -1; }

}

long check\_input(char \*s, char a, long r)

{ long i;

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

{ if(s[i] == a - 0x30) return 0; }

return 1;

}

long check\_possible(char \*begin, char \*end)

{ char fs;

long f1=0, f2=0;

long i, j;

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

{ fs=0;

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

{

if((begin[i] != 0) && (begin[j] != 0) && (begin[j] < begin[i])) fs++;

}

f1+=fs;

fs=0;

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

{ if((end[i] != 0) && (end[j] != 0) && (end[j] < end[i])) fs++;

}

f2+=fs;

}

if((f1 & 1) == (f2 & 1)) return 1;

else

return 0;

}

void output(void)

{ long i, j, k;

char ss[NUM];

for(i=m\_depth - 1; i >= 0; i--)

{ RTRANS(m\_out[i].v, ss);

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

{ for(k=0; k < SIZE; k++)

{ printf("%2d", ss[SIZE \* j + k]);

}

printf("\n");

}

printf("\n");

}

}

int main(void)

{ char s1[NUM];

char s2[NUM];

long r;

char a;

printf("请输入开始状态:");

r=0;

while(r < NUM)

{ a=getchar();

if(a >= 0x30 && a < 0x39 && check\_input(s1, a, r))

{ s1[r++]=a - 0x30;

printf("%c", a);

}

}

printf("\n请输入结束状态:");

r=0;

while(r < NUM)

{ a=getchar();

if(a >= 0x30 && a < 0x39 && check\_input(s2, a, r))

{ s2[r++]=a - 0x30;

printf("%c", a);

}

}

printf("\n");

if(check\_possible(s1, s2))

{ r=bfs\_search(s1, s2);

printf("\n");

if(r >= 0)

{ printf("查找深度=%d,所有的方式=%ld\n", m\_depth, r);

output();

}

else if(r == -1)

{ printf("没有找到路径.\n");

}

else if(r == -2)

{printf("这种状态变换没有路径到达.\n");

}

else

{printf("不确定的错误.\n");

}

}

else

{ printf("不允许这样移动!\n");

}

return 0;

}

A算法：

#include <iostream>

#include <ctime>

#include <vector>

using namespace std;

const int ROW = 3;

const int COL = 3;

const int MAXDISTANCE = 10000;

const int MAXNUM = 10000;

int abs(int a)

{

if (a>0) return a;

else return -a;

}

typedef struct \_Node{

int digit[ROW][COL];

int dist; // 距离

int dep; // 深度

int index; // 索引值

} Node;

Node src, dest;

vector<Node> node\_v; // 储存节点

bool isEmptyOfOPEN() { //判断Open表是否空

for (int i = 0; i < node\_v.size(); i++) {

if (node\_v[i].dist != MAXNUM)

return false;

}

return true;

}

bool isEqual(int index, int digit[][COL]) { //判断节点是否与索引值指向的节点相同

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

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

if (node\_v[index].digit[i][j] != digit[i][j])

return false;

}

return true;

}

ostream& operator<<(ostream& os, Node& node) {

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

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

os << node.digit[i][j] << ' ';

os << endl;

}

return os;

}

void PrintSteps(int index, vector<Node>& rstep\_v) { //输出步骤

rstep\_v.push\_back(node\_v[index]);

index = node\_v[index].index;

while (index != 0) {

rstep\_v.push\_back(node\_v[index]);

index = node\_v[index].index;

}

for (int i = rstep\_v.size() - 1; i >= 0; i--)

cout << "Step " << rstep\_v.size() - i

<< endl << rstep\_v[i] << endl;

}

void Swap(int& a, int& b) { //交换

int t;

t = a;

a = b;

b = t;

}

void Assign(Node& node, int index) { //获取节点

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

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

node.digit[i][j] = node\_v[index].digit[i][j];

}

int GetMinNode() { //获取启发值最小的节点

int dist = MAXNUM;

int loc; // the location of minimize node

for (int i = 0; i < node\_v.size(); i++) {

if (node\_v[i].dist == MAXNUM)

continue;

else if ((node\_v[i].dist + node\_v[i].dep) < dist) {

loc = i;

dist = node\_v[i].dist + node\_v[i].dep;

}

}

return loc;

}

bool isExpandable(Node& node) { //判断是否可扩展

for (int i = 0; i < node\_v.size(); i++) {

if (isEqual(i, node.digit))

return false;

}

return true;

}

int Distance(Node& node, int digit[][COL]) { //计算距离

int distance = 0;

bool flag = false;

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

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

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

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

if (node.digit[i][j] == digit[k][l]) {

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

flag = true;

break;

}

else

flag = false;

}

if (flag)

break;

}

return distance;

}

int MinDistance(int a, int b) { //二者取小

return (a < b ? a : b);

}

void ProcessNode(int index) { //展开节点

int x, y;

bool flag;

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

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

if (node\_v[index].digit[i][j] == 0) {

x =i; y = j;

flag = true;

break;

}

else flag = false;

}

if(flag)

break;

}

Node node\_up; //上移操作

Assign(node\_up, index);

int dist\_up = MAXDISTANCE;

if (x > 0) {

Swap(node\_up.digit[x][y], node\_up.digit[x - 1][y]);

if (isExpandable(node\_up)) {

dist\_up = Distance(node\_up, dest.digit);

node\_up.index = index;

node\_up.dist = dist\_up;

node\_up.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_up);

}

}

Node node\_down; //下移操作

Assign(node\_down, index);

int dist\_down = MAXDISTANCE;

if (x < 2) {

Swap(node\_down.digit[x][y], node\_down.digit[x + 1][y]);

if (isExpandable(node\_down)) {

dist\_down = Distance(node\_down, dest.digit);

node\_down.index = index;

node\_down.dist = dist\_down;

node\_down.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_down);

}

}

Node node\_left;//左移操作

Assign(node\_left, index);

int dist\_left = MAXDISTANCE;

if (y > 0) {

Swap(node\_left.digit[x][y], node\_left.digit[x][y - 1]);

if (isExpandable(node\_left)) {

dist\_left = Distance(node\_left, dest.digit);

node\_left.index = index;

node\_left.dist = dist\_left;

node\_left.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_left);

}

}

Node node\_right; //右移操作

Assign(node\_right, index);

int dist\_right = MAXDISTANCE;

if (y < 2) {

Swap(node\_right.digit[x][y], node\_right.digit[x][y + 1]);

if (isExpandable(node\_right)) {

dist\_right = Distance(node\_right, dest.digit);

node\_right.index = index;

node\_right.dist = dist\_right;

node\_right.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_right);

}

}

node\_v[index].dist = MAXNUM;

}

int main() {

int number;

cout << "输入初始状态:" << endl;

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

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

cin >> number;

src.digit[i][j] = number;

}

src.index = 0;

src.dep = 1;

cout << "输入目标状态" << endl;

for (int m = 0; m < ROW; m++)

for (int n = 0; n < COL; n++) {

cin >> number;

dest.digit[m][n] = number;

}

node\_v.push\_back(src);

while (1) {

if (isEmptyOfOPEN()) {

cout << "找不到解!" << endl;

return -1;

}

else {

int loc; // the location of the minimize node

loc = GetMinNode();

if(isEqual(loc, dest.digit)) {

vector<Node> rstep\_v;

cout << "初始状态：" << endl;

cout << src << endl;

PrintSteps(loc, rstep\_v);

cout << "成功!" << endl;

break;

}

else

ProcessNode(loc);

}

}

return 0;

}