

## 课程设计报告

课程名称： 《算法分析与设计》

设计题目： 棋盘布局问题

院 系： 悉尼工商学院

设 计 者： 王昆

学 号： 19124349

指导教师： 喻钢

设计时间： 2012.5.30

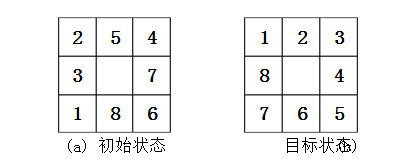
|  |
| --- |
| 课程题目：  在棋盘中实现从初始布局到目标布局的转变 |
| 个人小结：  遇到的问题  （1）本次课程设计过程是两种搜索方式分开开发，因此在综合在一起的时候两种方式的代码合在一起略显杂糅，程序可读性较差。  （2）在做启发式搜索时，发现效率并未达到预期，查阅资料得知h值取得不好，没有完全切合搜索规律，h值应为每个方格到其正确位置的路径之和，而我仅统计的是错位之和同时发现若h值取得越大其搜索速度越快单难以保证得出的是最优解。因此我设计了一个系数1000来进行加速。  （3）在设计广度优先搜索时指针操作出现失误，导致进度一度停滞，后放弃改为数组操作才完成程序  经验小结  (1)广度优先搜索是一种十分浪费空间的搜索方法，有时会陷入死循环。同时在时间效率上广度优先搜索效率不稳定，若搜索深度过大则时间效率异常缓慢。但可确定找到最优解。  (2)启发式搜索效率较稳定，当搜素深度较小时时间效率不及广度优先但在深度搜索表现优异，同时启发式搜索有选择的创建节点，大大节省空间效率。 |



**课程设计报告**

**1.题目描述与设计要求**

在3×3的方格棋盘上，摆有八个棋子，每一个棋子上标有1至8的某一数字。棋盘中留有一个空格。空格周围的棋子可以移到空格中。



给出一种初始状态和目标状态，找到一种移动方法，实现从初始状态到目标状态的转变。

* 1. **输入要求**

跟随指引，输入一行整数，0到8的随机排列，每个数字用空格隔开，表示从左至右，从上至下的数字顺序，0表示空格，此为初始状态。

之后输入另一行不同顺序的整数，此为目标状态。

* 1. **输出要求**

输出棋盘逐步从初始状态到目标状态变化的过程。

**2.算法设计**

**2.1问题分析**

由题目知给出一个初始状态和一个目标状态，找出一种从初始转变成目标状态的移动步骤。所谓问题的一个状态就是棋子在棋盘上的一种摆法。问题实际上就是找出从初始状态到达目标状态所经过的一系列中间过渡状态。可以采用广度优先搜索和启发式搜索两种方法进行求解。且棋盘通过空格的上、下、右四种操作来改变状态。所以需要解决的问题为每个状态的表示和每个状态变化的操作，设计出适合相应搜索的数据结构，完成相应算法思想。

2.1.1 任务定义

(1)完成棋盘状态表示。

(2)完成空格上下左右移动操作变化的表示。

(3)分别完成适应于广度优先搜索，启发式搜索的数据结构设计。

(4)得出两种搜索的搜索路径。

**2.2算法设计思路及求解过程**

**2.2.1 广度优先搜索**

从初始节点h开始逐层的对其进行扩展，并考察其是否为目的节点，若不是将其放入待考察队列中，在第n层节点没有完全进行考察并扩展完成前，不对第n+1层进行扩展。队列中节点总是按照先后顺序进行排列，先进的排在前面，后进的排在后面。

由此可知我们可以通过队列的思想完成搜索，其具体搜索过程如下。

1)初始化头结点front，front=real;

2)若front=null,问题无解,程序结束。

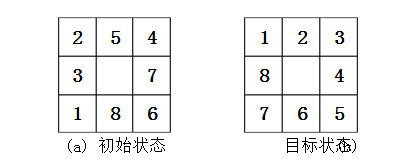
3)对front指向的节点进行考察，若为目的节点程序结束。

4)判断front指向的节点的空格是否可以上下左右移动若可以，扩展节点置于队尾。重复操作(2)。

(2)数据结构的选择和概要设计

1) 棋盘状态的表示

因为每一个棋盘状态是静止且确定的所以采取一个一维数组便可将其状态表示出来，数组按照下标顺序从左至右从上至下依次对应。空格可由零来表示。

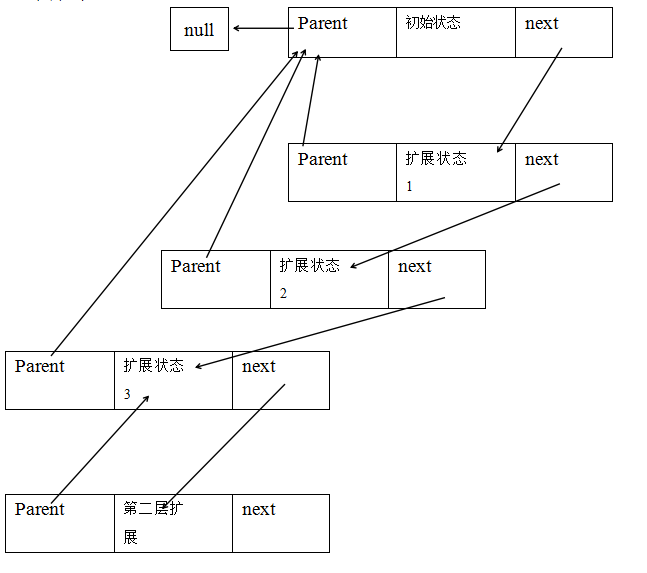


此状态可表示为int num[9]={2,5,4,3,0,7,1,8,6}。

1. 搜索结构设计

根据算法要求，需要对棋盘不断的进行动态扩展，且盲目搜索产生的数据巨大，所以不能采用静态链表或数组存储，因此选择采用动态链表来存储搜索过程中的每一个状态。

因为搜索是根据一种状态对其进行扩展，直至达到目的状态，由于要返回一个最优路径因此需要确定扩展关系，我选择用一个\*parent来记录此关系。此时存储结构可由此图表示



当扩展至目的状态时可通过parent指针找到路径。由此我们可以设置节点类型为

typedef struct Node

{

struct Node \*parent;

int num[9];

Node \*next;

}QNode；

**2.2.2 启发式搜索**

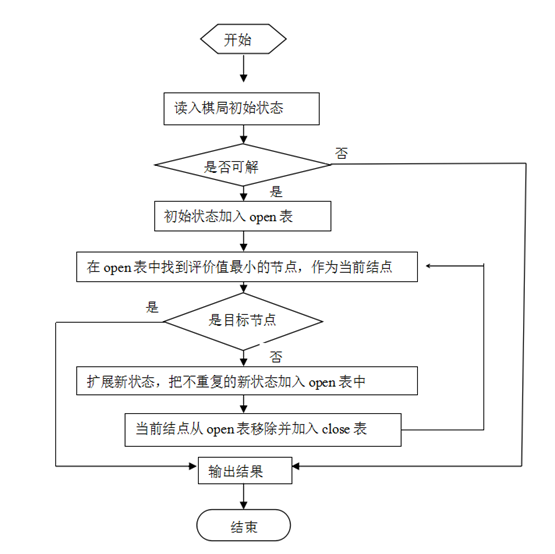
称盲目搜索，盲目搜索不考虑节点好坏，而问题的解决过程是有迹可循的，我们通过是否接近目标状态来判断节点的好坏，因此可以通过启发式搜索中的A\*算法来解决这个问题。

简单来说A\*就是将估值函数分成两个部分，一个部分是路径价值，另一个部分是一般性启发价值，合在一起算估整个结点的价值。在本实验中使用A\*算法求解。A\*搜索是一种效的搜索算法，它把到达节点的耗散g(n)和从该节点到目标节点的消耗h(n)结合起来对节点进行评价：f(n)=g(n)+h(n)。

由此设计如下

1. 把起始节点放到OPEN表中，并计算节点S的
2. 如果OPEN是空表，则失败退出，无解；
3. 从OPEN表中选择一个值最小的节点。如果有几个节点值相同，当其中有一个为目标节点时，则选择此目标节点；否则就选择其中任一个节点作为节点；
4. 把节点从 OPEN 表中移出，并把它放入 CLOSED 的已扩展节点表中;如果是个目标节点，则成功退出，求得一个解；
5. 扩展节点，生成其全部后继节点。对于的每一个后继节点：计算；如果 既不在OPEN表中，又不在CLOCED表中，则用估价函数把它添入OPEN表中。从加一指向其父节点的指针，以便一旦找到目标节点时记住一个解答路径；如果已在OPEN表或CLOSED表中，则比较刚刚对计算过的和前面计算过的该节点在表中的值。如果新的较小，则(I)以此新值取代旧值。(II)从指向，而不是指向他的父节点。(III)如果节点在CLOSED表中，则把它移回OPEN表中。转向（2）。

流程图如下



1. 数据结构的选择和概要设计

由于搜索特点，采用的基础存储结构和广度优先搜索相同，不同之处为启发式搜索使用两条链表，所以设计一个表结构，用来声明两条链表。

节点结构

typedef struct Node

{

double g,f;

struct Node \*parent;

int num[9];

}Node；

表结构

typedef struct Stack

{

Node \* npoint;//指向状态节点，相当于表节点的数据域。

struct Stack \* next;

}Stack；

**3.算法实现**

**3.1算法核心函数**

**3.1.1广度优先搜索**

（1）相关函数

1）空格移动函数

int move\_up(int num[]);

int move\_down(int num[]);

int move\_left(int num[]);

int move\_right(int num[]);

以上移为例，找到0的位置将其与下标比其小三的值交换，且下标为0,1,2的数字不可移动。

2）搜索过程函数

int bfs(int init[],int target[])；接收初始状态和目的状态总领搜索过程。

int cansolve(int num1[], int num2[])；判断两个数列的逆序数奇偶性，从而判断问题是否可解

void get\_num(QNode \*node, int temp[])；将节点中的数组获取出来。

void set\_num(QNode \*node, int temp[])；设置节点中数组的数据。

void print(QNode \*node)；输出节点中信息

（2)算法伪码

输入初始状态数组unit，目标状态数组target

If(问题可解)

While(队头指针非空时&&未找到目的节点)

If(队头节点为目的节点)

结束循环；

End if

对队头节点进行扩展，将扩展节点加入队尾；

队头指针后移；

End while

End if

**3.1.2启发式搜索设计**

（1）所用函数

int Equal(Node \* suc,Node \* goal)；判断节点是否相等，相等，不相等。

Node \* Belong(Node \* suc,Lstack \* list)判断节点是否属于OPEN表或CLOSED表，是则返回节点地址，否则返回空地址。

void Putinto(Node \* suc,Lstack \* list)把节点放入OPEN 或CLOSED 表中。

double Fvalue(Node suc, Node goal, float speed)计算f值。

double Distance(Node suc, Node goal, int i)计算方格的错位距离。

int BelongProgram(Lnode \* suc ,Lstack \* Open ,Lstack \* Closed ,Node goal ,float speed)判断子节点是否属于OPEN或CLOSED表并作出相应的处理。

int Canspread(Node suc, int n)判断空格可否向该方向移动，表示空格向上向下向左向右移。

void Spreadchild(Node \* child,int n)扩展child节点的字节点n表示方向表示空格向上向下向左向右移。

int Spread(Lnode \* suc, Lstack \* Open, Lstack \* Closed, Node goal, float speed)扩展后继节点总函数。

Node \* Process(Lnode \* org, Lnode \* goal, Lstack \* Open, Lstack \* Closed, float speed) 总执行函数。

void qf(int init[9],int target[9])启发式搜索总函数。

（2）搜索过程伪码

输入：初始状态，目标状态

输出：从初始状态到目标状态的一系列过程

算法描述：

Begin：

读入初始状态和目标状态，并计算初始状态评价函数值f；

根据初始状态和目标状态，判断问题是否可解；

If(问题可解)

把初始状态加如open表中；

While（未找到解&&状态表非空）

①在open表中找到评价值最小的节点，作为当前结点，；

②判断当前结点状态和目标状态是否一致，若一致，跳出循环；否则跳转到③；

③对当前结点，分别按照上、下、左、右方向移动空格位置来扩展新的状态结点，并在并计算新扩展结点的评价值f并记录其父节点若；

④对于新扩展的状态结点，判断其是否重复，若不重复，把其加入到open表中；

⑤把当前结点从open表中移至Close表；

End while

End if

输出结果；

End

**3.2算法的程序实现代码**

#include <iostream>

#include "stdio.h"

#include <cstdlib>

#include <cmath>

#include<process.h>

using namespace std;

//结构体定义

typedef struct Node

{

struct Node\* parent;

double f, g;

int num[9];

Node\* next;

}QNode, \* Lnode;

//OPEN CLOSED 表结构体

typedef struct Stack

{

Node\* npoint;

struct Stack\* next;

}Stack, \* Lstack;

//判断两个数组的逆序数奇偶性，从而判断问题是否可解

int cansolve(int num1[], int num2[])

{

int i, j;

int sum1 = 0, sum2 = 0;

for (i = 0; i < 9; i++)

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

{

if (num1[j] > num1[i] && num1[i] != 0) ++sum1;

if (num2[j] > num2[i] && num2[i] != 0) ++sum2;

}

if (sum1 % 2 == sum2 % 2) return 1;

else return 0;

}

//数据输入函数

void input(int num[])

{

int i = 0, j = 0, flag = 0;

char c;

while (i < 9)

{

while (((c = getchar()) != 10))

{

if (c == ' ')

{

if (flag >= 0)flag = 0;

}

else if (c >= '0' && c <= '8')

{

if (flag == 0)

{

num[i] = (c - '0');

flag = 1;

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

if (num[j] == num[i])flag = -2;

i++;

}

}

else if (flag >= 0)flag = -1;

}

if (flag < 0 || i < 9)

{

if (flag < 0)

{

if (flag == -1) cout<<"含有非法字符或数字!\n请重新输入:\n";

else if (flag == -2) cout<<"输入的数字有重复!\n请重新输入:\n";

}

else if (i < 9) cout<<"输入的有效数字不够!\n请重新输入:\n";

i = 0;

flag = 0;

}

}

}

//上移

int move\_up(int num[])

{

int i;

for (i = 0; i< 9; i++)

if (num[i] == 0) break;

if (i == 0 || i == 1 || i == 2)

return 0;

else

{

num[i] = num[i - 3];

num[i - 3] = 0;

return 1;

}

}

//下移

int move\_down(int num[])

{

int i;

for (i = 0; i < 9; i++)

if (num[i] == 0) break;

if (i == 6 || i == 7 || i == 8)

return 0;

else

{

num[i] = num[i + 3];

num[i + 3] = 0;

return 1;

}

}

//左移

int move\_left(int num[])

{

int i;

for (i = 0; i < 9; i++)

if (num[i] == 0) break;

if (i == 0 || i == 3 || i == 6)

return 0;

else

{

num[i] = num[i - 1];

num[i - 1] = 0;

return 1;

}

}

//右移

int move\_right(int num[])

{

int i;

for (i = 0; i < 9; i++)

if (num[i] == 0) break;

if (i == 2 || i == 5 || i == 8)

return 0;

else

{

num[i] = num[i + 1];

num[i + 1] = 0;

return 1;

}

}

//将节点中的数组获取出来

//获取到temp

void get\_num(QNode\* node, int temp[])

{

int i;

for (i = 0; i < 9; i++)

{

temp[i] = node->num[i];

}

}

//设置节点中数组的数据

//temp数据进入QNode

void set\_num(QNode\* node, int temp[])

{

int i;

for (i = 0; i < 9; i++)

node->num[i] = temp[i];

}

//打印

void print(QNode\* node)

{

int i;

for (i = 0; i < 9; i++)

{

cout << node->num[i]<<" ";

if ((i + 1) % 3 == 0)

cout << endl;

}

}

//判断目标节点数据与目标数组数据是否相等

int isequal(QNode\* node, int target[])

{

int i;

int flag = 1;

for (i = 0; i < 9 && flag; i++)

{

if (node->num[i] != target[i])

flag = 0;

}

return flag;

}

//退出

int exist(QNode\* node, int temp[])

{

QNode\* p;

for (p = node; p != NULL; p = p->parent)

if (isequal(node, temp))

return 1;

return 0;

}

//选取OPEN表上f值最小的节点，返回该节点地址

Node\* Minf(Lstack\* Open)

{

Lstack temp = (\*Open)->next, min = (\*Open)->next, minp = (\*Open);

Node\* minx;

while (temp->next != NULL)

{

if ((temp->next->npoint->f) < (min->npoint->f))

{

min = temp->next;

minp = temp;

}

temp = temp->next;

}

minx = min->npoint;

temp = minp->next;

minp->next = minp->next->next;

free(temp);

return minx;

}

//判断节点是否相等

//输入两个节点

//返回相等<1>，不相等<0>

int Equal(Node\* suc, Node\* goal)

{

for (int i = 0; i < 9; i++)

if (suc->num[i] != goal->num[i])return 0;

return 1;

}

//判断节点是否属于OPEN表或CLOSED表，是则返回节点地址，否则返回空地址

//输入节点地址<Node\* suc>，表地址<Lstack\* list>

Node\* Belong(Node\* suc, Lstack\* list)

{

Lstack temp = (\*list)->next;

if (temp == NULL)return NULL;

while (temp != NULL)

{

if (Equal(suc, temp->npoint))return temp->npoint;

temp = temp->next;

}

return NULL;

}

//把节点放入OPEN 或CLOSED 表中

//输入节点地址<Node\* suc>，表地址<Lstack\* list>

void Putinto(Node\* suc, Lstack\* list)

{

Stack\* temp;

temp = (Stack\*)malloc(sizeof(Stack));

temp->npoint = suc;

temp->next = (\*list)->next;

(\*list)->next = temp;

}

//计算方格的错位距离

//输入初始节点<Node suc>和目标节点 <Node goal>,i节点数

double Distance(Node suc, Node goal, int i)

{

int k, h1, h2;

for (k = 0; k < 9; k++)

{

if (suc.num[k] == i)h1 = k;

if (goal.num[k] == i)h2 = k;

}

return double(fabs(h1 / 3 - h2 / 3) + fabs(h1 % 3 - h2 % 3));

}

//计算f值

//输入初始节点<Node suc>和目标节点 <Node goal>,speed搜索速度

double Fvalue(Node suc, Node goal, float speed)

{

double Distance(Node, Node, int);

double h = 0;

for (int i = 1; i <= 8; i++)

h = h + Distance(suc, goal, i);

return h \* speed + suc.g; //(speed值增加时搜索过程以找到目标为优先因此可能不会返回最优解)

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*扩展后继节点部分的函数-以下\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//判断子节点是否属于OPEN或CLOSED表并作出相应的处理

//输入节点地址<Lnode\* suc>，表地址<Lstack\* Open> <Lstack\* Closed> 目标节点 <Node goal>,speed搜索速度

int BelongProgram(Lnode\* suc, Lstack\* Open, Lstack\* Closed, Node goal, float speed)

{

Node\* temp = NULL;

int flag = 0;

if ((Belong(\*suc, Open) != NULL) || (Belong(\*suc, Closed) != NULL))

{

if (Belong(\*suc, Open) != NULL) temp = Belong(\*suc, Open);

else temp = Belong(\*suc, Closed);

if (((\*suc)->g) < (temp->g))

{

temp->parent = (\*suc)->parent;

temp->g = (\*suc)->g;

temp->f = (\*suc)->f;

flag = 1;

}

}

else

{

Putinto(\*suc, Open);

(\*suc)->f = Fvalue(\*\*suc, goal, speed);

}

return flag;

}

//判断空格可否向该方向移动，表示空格向上向下向左向右移

//输入节点<node suc>, i节点数

int Canspread(Node suc, int n)

{

int i, flag = 0;

for (i = 0; i < 9; i++)

if (suc.num[i] == 0)break;

switch (n)

{

case 1:

if (i / 3 != 0)flag = 1; break;

case 2:

if (i / 3 != 2)flag = 1; break;

case 3:

if (i % 3 != 0)flag = 1; break;

case 4:

if (i % 3 != 2)flag = 1; break;

default:break;

}

return flag;

}

//扩展child节点的子节点

//n表示方向 , 表示空格向上向下向左向右移

void Spreadchild(Node\* child, int n)

{

int i, loc, temp;

for (i = 0; i < 9; i++)

child->num[i] = child->parent->num[i];

for (i = 0; i < 9; i++)

if (child->num[i] == 0)break;

if (n == 0)

loc = i % 3 + (i / 3 - 1) \* 3;

else if (n == 1)

loc = i % 3 + (i / 3 + 1) \* 3;

else if (n == 2)

loc = i % 3 - 1 + (i / 3) \* 3;

else

loc = i % 3 + 1 + (i / 3) \* 3;

temp = child->num[loc];

child->num[i] = temp;

child->num[loc] = 0;

}

//扩展后继节点总函数

//输入节点地址<Lnode\* suc>，表地址<Lstack\* Open> <Lstack\* Closed> 目标节点 <Node goal>,speed搜索速度

int Spread(Lnode\* suc, Lstack\* Open, Lstack\* Closed, Node goal, float speed)

{

int i;

Node\* child;

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

{

if (Canspread(\*\*suc, i + 1)) //判断某个方向上的子节点可否扩展

{

child = (Node\*)malloc(sizeof(Node)); //扩展子节点

child->g = (\*suc)->g + 1; //算子节点的g值

child->parent = (\*suc); //子节点父指针指向父节点

Spreadchild(child, i); //向该方向移动空格生成子节点

if (BelongProgram(&child, Open, Closed, goal, speed)) // 判断子节点是否属于OPEN或CLOSED表并作出相应的处理

free(child);

}

}

return 1;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*扩展后继节点部分的函数-以上\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//总执行函数

//输入节点地址<Lnode\* org>，表地址<Lstack\* Open> <Lstack\* Closed> 目标节点 <Lnode\* goal>,speed搜索速度

Node\* Process(Lnode\* org, Lnode\* goal, Lstack\* Open, Lstack\* Closed, float speed)

{

while (1)

{

if ((\*Open)->next == NULL)return NULL; //判断OPEN表是否为空，为空则失败退出

Node\* minf = Minf(Open); //从OPEN表中取出f值最小的节点

Putinto(minf, Closed); //将节点放入CLOSED表中

if (Equal(minf, \*goal))return minf; //如果当前节点是目标节点，则成功退出

Spread(&minf, Open, Closed, \*\*goal, speed);

//当前节点不是目标节点时扩展当前节点的后继节点

}

}

//启发式搜索

//初始表数组<init[9]>,目标表数组<target[9]>

void qf(int init[9], int target[9])

{

Lstack Open = (Stack\*)malloc(sizeof(Stack));

Open->next = NULL;

Lstack Closed = (Stack\*)malloc(sizeof(Stack));

Closed->next = NULL;

Node\* org = (Node\*)malloc(sizeof(Node));

Node\* p;

org->parent = NULL; //初始状态节点

org->f = 1;

org->g = 1;

int count = 0;

Node\* goal = (Node\*)malloc(sizeof(Node)); //目标状态节点

Node\* result;

set\_num(org, init);

set\_num(goal, target);

float speed = 1000;//speed搜索速度

Putinto(org, &Open);

result = Process(&org, &goal, &Open, &Closed, speed); //进行剩余的操作print(result)

for (p = result; p != NULL; p = p->parent)

{

print(p);

cout<<"------\n";

count++;

}

cout << "得出最优解共需" << count << "个节点\n";

cout<<endl;

cout<<"Press Enter key to exit!";

}

//广度优先搜索

//初始表数组<init[]>,目标表数组<target[]>

int bfs(int init[], int target[])

{

int temp[9];

int find = 0;

int step = 1, count = 1;

QNode\* front, \* rear, \* new\_node, \* p;

if (!cansolve(init, target))

{

cout<<"不能实现\n";

return 0;

}

front = (QNode\*)malloc(sizeof(QNode));

set\_num(front, init);

front->parent = NULL;

front->next = NULL;

rear = front;

while (front != NULL && !find)

{

if (isequal(front, target))

{

find = 1;

break;

}

get\_num(front, temp);

if (move\_up(temp) && !exist(front, temp))

{

new\_node = (QNode\*)malloc(sizeof(QNode));

set\_num(new\_node, temp);

new\_node->parent = front;

new\_node->next = NULL;

rear->next = new\_node;

rear = new\_node;

step++;

}

get\_num(front, temp);

if (move\_left(temp) && !exist(front, temp))

{

new\_node = (QNode\*)malloc(sizeof(QNode));

set\_num(new\_node, temp);

new\_node->parent = front;

new\_node->next = NULL;

rear->next = new\_node;

rear = new\_node;

step++;

}

get\_num(front, temp);

if (move\_down(temp) && !exist(front, temp))

{

new\_node = (QNode\*)malloc(sizeof(QNode));

set\_num(new\_node, temp);

new\_node->parent = front;

new\_node->next = NULL;

rear->next = new\_node;

rear = new\_node;

step++;

}

get\_num(front, temp);

if (move\_right(temp) && !exist(front, temp))

{

new\_node = (QNode\*)malloc(sizeof(QNode));

set\_num(new\_node, temp);

new\_node->parent = front;

new\_node->next = NULL;

rear->next = new\_node;

rear = new\_node;

step++;

} front = front->next;

count++;

}

if (find)

{

count = 0;

for (p = front; p != NULL; p = p->parent)

{

print(p);

cout<<"------\n";

count++;

}

cout<<"得出最优解共需"<<count<<"个节点\n";

}

}

int main()

{

int init[9], target[9];

cout << "输入初始状态:" << endl;

input(init);

cout << "输入目标状态:" << endl;

input(target);

if (cansolve(init, target))

{

cout << "使用广度优先进行搜索:" << endl;

bfs(init, target);

cout << endl;

cout << "使用启发式进行搜索:" << endl;

qf(init, target);

}

else {

cout << "无解" << endl;

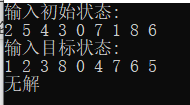
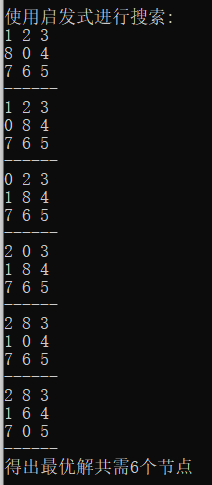
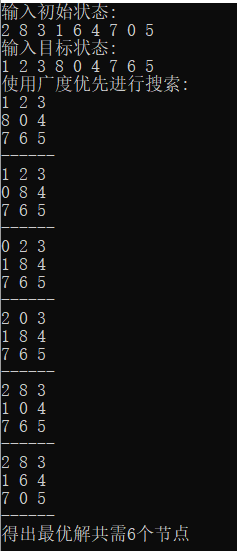
}

return 0;

}

**3.3 算法运行效果**

**3.3.1 运行截屏**



输入初始状态，目标状态后会分别输出广度优先搜索和启发式搜索棋盘逐步从初始状态到目标状态变化的过程。并给出所需要的节点数。如果无解也会给出相应提示。

**3.3.2 结果分析**

（1）实验结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验数据 | 广广度优先 | | 启发式搜索 | |
|  | 时间 | 最优节点数 | 时间 | 最优节点数 |
| 初始状态283164705  目标状态123804765 | 4ms | 6 | 109ms | 6 |
| 初始状态254307186  目标状态123804765 | 无解 | 无解 | 无解 | 无解 |
| 初始状态123456780  目标状态123405678 | 4399ms | 15 | 196ms | 15 |
| 初始状态  123456780  目标状态  013825476 | 48ms | 9 | 128ms | 9 |
| 初始状态  123456780  目标状态  123045678 | 12537ms | 16 | 300ms | 28 |
| 初始状态  183265470  目标状态  836107245 | 112ms | 11 | 347ms | 25 |

（2）结果分析。

当最优解步骤较小时，广度优先时间效能更好。造成这种现象的原因为当搜素深度小时，启发式搜索由于有较多的查重和链表操作导致时间性能下降，而当搜索深度较深时启发式搜索对节点优劣的考察使其在时间上展现出较大优势。但启发式搜索可能无法获取最优解。

(3)时间复杂度

问题状态最多为9！，使用广度优先算法，每次插入或者查找的复杂度为Log(n)，所以，如果整个算法遍历了所有状态，所需要的复杂度为O（n\*Log(n)）。

使用启发式搜索，用深度作为g(n)，对于该问题，启发函数可以用两种状态不同数字的数目，满足h(n)>h’(n),h’(n)为从当前节点到目标点的实际的最优代价值。每次扩展的节点的f值大于等于父节点的f值小。每移动一次，最多使得一个数字回归，或者说不在位减一个。h最多减小1，而g认为是深度，每次会增加1。所以，f=g+h， 自然非递减。时间复杂度为O(Log(n))。

(4) 空间复杂度

广度优先搜索是一种十分浪费空间的搜索方法，有时会陷入死循环,由于广搜需要记录状态，并且需要判重，每次图的状态转换为一个字符串

启发式搜索有选择的创建节点，大大节省空间效率。