###### 实验一：启发式搜索算法

|  |  |
| --- | --- |
| 选课序号 | 62 |
| 专业班级 | 计算机科学与技术2019-3班 |
| 学号 | 2220192813 |
| 姓名 | 胡聿鑫 |

##### 1、实验环境

CodeBlocks

Python3.8

##### 2、实验目的和要求

###### 2.1实验目的

熟悉掌握启发式搜索算法A\*及其可采纳性

###### 2.2 实验内容

* 编写程序实现8数码和15数码问题，采用至少两种估价函数
* 分析估价函数求解问题时候的效率差别，分析估价函数对搜索算法的影响

##### 3、解题思路、代码

###### 3.1 8-数码问题

###### 3.1.1解题思路

在A算法中，已经给出了估价函数的定义，即： f(n) = g(n) + h(n)，其中n为待估节点，f(n)为从初始状态节点s，经由节点n到达目标节点估计的最小路径代价，g(n)为从s到n的当前实际的路径代价，h(n)为节点n到目标节点估计的最小路径代价，也即启发式函数。A\*算法要求在A算法的基础上，h(n)≤h\*(n)，由于A\*算法是可采纳的，因此它总能得到最短解答路径。

针对8数码问题，可采用如下两种不同的估价函数：

1. 估价函数一：将错位的棋牌数作为h(n);
2. 估计函数二：将当前节点和实际节点所对应的直线距离作为h(n)。

具体实现过程：

1.设置OPEN表以存放搜索图中的叶节点（也即已生成但未扩展的节点），设置CLOSE表存放图中的非叶节点（也即已生成且已扩展的节点）。

2.对于OPEN表中的节点，每次均取第一个元素n进行扩展，若n即为目标节点，代表算法已经成功得到一个解，否则继续对节点n进行扩展。

3.节点n可能会进一步扩展出子节点m，若该子节点既不在OPEN表也不在CLOSE表中，则将其加入OPEN表；若m已经在CLOSE表中，这代表初始节点到m存在两条路径，为了选择最优路径，需要将新的路径和原路径进行比较，若新路径代价更小则需要将m取出并存入OPEN表，否则无需做任何改变。

4.重复以上步骤直至求解完毕，若OPEN表为空则说明算法无解。

###### 3.1.2代码

#include <iostream>

#include <time.h>

#include<cmath>

#define MAXLISTSIZE 10000

#define MAXSTEPSIZE 100

using namespace std;

struct ENode

{

int status[9];//8数码排列状态

int G;//第G次移动

int H;//不在正确位置的棋牌位置

int F;//代价

int Z;//0所在位置

int step;//移动方式，规定1234分别代表左右上下

ENode \*Parent;

};//status为8数码的排列状态

//正确排列情况

int CorrectSituaiton[9] = { 1, 2, 3, 8, 0, 4, 7, 6, 5 };

ENode \*Node;

ENode OPEN[MAXLISTSIZE];//定义OPEN表

ENode CLOSE[MAXLISTSIZE];//定义CLOSE表

int open = 0;

int close = 0;

//估价函数一：以错位的棋牌数为h(n)

int Valuation(int \*status)

{

int H = 0;

int i;

for (i = 0; i <= 8; i++)

{

if (CorrectSituaiton[i] != status[i])

{

H++;

}

}

return H;

}

//估价函数二：以当前节点和实际节点所对应的直线距离作为h(n)

/\*int Valuation(int \*status)

{

int H = 0;

int i, j;

for (i = 0; i <= 8; i++)

{

for (j = 0; j <= 8; j++)

{

if (FinalStatus[j] == status[i])

break;

}

H += sqrt(pow((i%3 - j%3), 2) + pow((i/3 - j/3), 2));

}

return H;

}\*/

ENode \*ENodeInit(int \*status, int zero, int g, ENode \*parent, int step)

{//对节点进行初始化

int i;

Node = new ENode;

for (i = 0; i <= 8; i++)

{

Node->status[i] = status[i];

}

Node->Z = zero;

Node->G = g;

Node->H = Valuation(Node->status);

Node->F = Node->G + Node->H;

Node->Parent = parent;

Node->step = step;

return Node;

}

int \*Left(int \*s, int z)

{//将0棋牌左移一次

int temp, i;

static int status[9];

for (i = 0; i <= 8; i++)

{

status[i] = s[i];

}

temp = status[z - 1];

status[z - 1] = 0;

status[z] = temp;

return status;

}

int \*Right(int \*s, int z)

{//将0棋牌右移一次

int temp, i;

static int status[9];

for (i = 0; i <= 8; i++)

{

status[i] = s[i];

}

temp = status[z + 1];

status[z + 1] = 0;

status[z] = temp;

return status;

}

int \*Up(int \*s, int z)

{//将0棋牌上移一次

int temp, i;

static int status[9];

for (i = 0; i <= 8; i++)

{

status[i] = s[i];

}

temp = status[z - 3];

status[z - 3] = 0;

status[z] = temp;

return status;

}

int \*Down(int \*s, int z)

{//将0棋牌下移一次

int temp, i;

static int status[9];

for (i = 0; i <= 8; i++)

{

status[i] = s[i];

}

temp = status[z + 3];

status[z + 3] = 0;

status[z] = temp;

return status;

}

int Exist(ENode \*N)

{//判断新节点是否已经存在于OPEN表或CLOSE表中

int i, j;

int H = 0; //错位棋牌个数

int status[9];

Node = new ENode;

Node = N;

for (i = 0; i <= 8; i++)

{

status[i] = Node->status[i];

}

for (i = 0; i <= open - 1; i++)//判断节点是否已经在OPEN表

{

for (j = 0; j <= 8; j++)

{

if (status[j] != OPEN[i].status[j])

{

H++;

}

}

if (H == 0)//H=0则表中已有该节点

{

return i + 1;

}

H = 0;

}

for (i = 0; i <= close - 1; i++)//判断节点是否已经在CLOSE表

{

for (j = 0; j <= 8; j++)

{

if (status[j] != CLOSE[i].status[j])

{

H++;

}

}

if (H == 0)//H=0证明在表中找到该节点

{

return (-i) - 1;

}

H = 0;

}

return 0;

}

void ExistAndOperate(ENode \*N)

{//判断子节点是否在OPEN或CLOSE中

int i;

int inList;

Node = new ENode;

Node = N;

if (Node->G == 1)//若为第一步的节点，直接加入OPEN表中

{

OPEN[open] = \*Node;

open++;

return;

}

inList = Exist(Node);//值>0 说明该节点只在OPEN表，<0 则只在CLOSE表，=0表明不在任何一表中

if (inList == 0)//节点不在两个表中，加入OPEN表

{

OPEN[open] = \*Node;

open++;

}

else if (inList > 0)//在OPEN表中，选择代价小的路径

{

if (OPEN[inList - 1].F > Node->F)//该判断语句为真则用新节点替换原节点

{

OPEN[inList - 1] = \*Node;

}

}

else if (inList < 0) //如果在CLOSE中，说明初始节点到该节点有两条路径，如果新找到的路径耗散值大，什么都不做，如果较小，将其从CLOSE中取出放入OPEN中

{

inList = -inList;

if (CLOSE[inList - 1].F > Node->F) //如果较小

{

OPEN[open] = \*Node; //将其取出放入OPEN

open++;

}

for (i = inList - 1; i <= close - 1; i++) //将其在CLOSE中释放

{

CLOSE[i] = CLOSE[i + 1];

}

close--;

}

}

ENode \*Search()

{//寻找代价最小路径

int \*status;

int i, j;

ENode \*Temp;

while (1)

{

Temp = new ENode;

for (i = open - 1; i > 0; i--)

{

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

{

if (OPEN[j].F > OPEN[j + 1].F)

{

\*Temp = OPEN[j + 1];

OPEN[j + 1] = OPEN[j];

OPEN[j] = \*Temp;

}

}

}

Node = new ENode;

\*Node = OPEN[0];//从OPEN表中取出第一个元素进行扩展

if (!Valuation(Node->status))//判断该节点是否为目标节点

{

break;

}

Temp = Node;

CLOSE[close] = \*Node;//将已扩展完成的节点放入CLOSE

close++;

for (i = 0; i <= open - 1; i++) //将节点从OPEN中取出

{

OPEN[i] = OPEN[i + 1];

}

open--;

if ((Temp->Z) % 3 >= 1)//判断语句为真，则左移构造新结点

{

Node = new ENode;

status = Left(Temp->status, Temp->Z);

Node = ENodeInit(status, Temp->Z - 1, (Temp->G) + 1, Temp, 1);

ExistAndOperate(Node);

}

if ((Temp->Z) % 3 <= 1)//判断语句为真，则右移构造新结点

{

Node = new ENode;

status = Right(Temp->status, Temp->Z);

Node = ENodeInit(status, Temp->Z + 1, (Temp->G) + 1, Temp, 2);

ExistAndOperate(Node);

}

if (Temp->Z >= 3)//判断语句为真，则上移构造新结点

{

Node = new ENode;

status = Up(Temp->status, Temp->Z);

Node = ENodeInit(status, Temp->Z - 3, (Temp->G) + 1, Temp, 3);

ExistAndOperate(Node);

}

if (Temp->Z <= 5)//判断语句为真，则下移构造新结点

{

Node = new ENode;

status = Down(Temp->status, Temp->Z);

Node = ENodeInit(status, Temp->Z + 3, (Temp->G) + 1, Temp, 4);

ExistAndOperate(Node);

}

if (open == 0)//算法无解

return NULL;

}

return Node;

}

void ShowStep(ENode \*Node)

{

int STEP[MAXSTEPSIZE];

int STATUS[MAXSTEPSIZE][9];

int step = 0;

int i, j;

int totalStep = Node->G;

while (Node)

{

STEP[step] = Node->step;

for (i = 0; i <= 8; i++)

{

STATUS[step][i] = Node->status[i];

}

step++;

Node = Node->Parent;

}

for (i = step - 1; i >= 0; i--)

{

if (STEP[i] == 1)

cout << "左";

else if (STEP[i] == 2)

cout << "右";

else if (STEP[i] == 3)

cout << "上";

else if (STEP[i] == 4)

cout << "下";

else if (STEP[i] == 0)

cout << "\n0棋牌移动次序：";

cout << " ";

}

cout << "\n共"<<totalStep <<"步"<< endl;

cout << "具体移动过程如下：" << endl;

for (i = step - 1; i >= 0; i--)

{

for (j = 0; j <= 8; j++)

{

cout << STATUS[i][j];

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

cout << endl;

else

cout << " ";

}

cout << "~~~~~~~~~~~~~~~~~~~~" << endl;

}

cout << "——移——动——完——毕——" << endl;

}

int main()

{

int fstatus[9];

int i, beginTime, endTime;

ENode \*FNode;

ENode \*EndNode;

cout << "请输入初始状态（数字0-8用空格隔开）" << endl;

for (i = 0; i <= 8; i++)

{

cin >> fstatus[i];

}

beginTime = clock();

for (i = 0; i <= 8; i++)

{

if (fstatus[i] == 0)

break;

}

FNode = ENodeInit(fstatus, i, 0, NULL, 0);

OPEN[open] = \*FNode;

open++;

EndNode = Search();

if (!EndNode)

cout << "无解" << endl;

else

ShowStep(EndNode);

endTime = clock();

cout << "运行时间:" << endTime - beginTime << "ms" << endl;

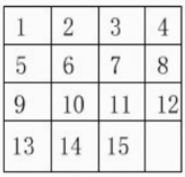
return 0;

}

###### 3.2 15-数码问题

###### 3.2.1 解题思路

15数码问题和8数码问题类似，但棋盘扩充为4×4棋盘，并且最终的目标状态如下：



在搜索的每一步都利用估价函数f(n)= g(n)+h(n)对Open表中的节点进行排序, 找出一个代价最小的节点作为下一次扩展的节点，并且满足条件：h(n)≤h\*(n)。其中g(n) 是在状态空间中从初始状态到状态n的实际代价，h(n) 是从状态n到目标状态的最佳路径的估计代价。

具体的移动方式仍然和8数码问题相同，在此不做赘述。该问题设计两种估价函数：

**估价函数一**：以每一个棋牌的位置和它最终正确位置之间横纵坐标的距离之和为估价函数，即曼哈顿距离。

**估价函数二**：以每一个棋牌的位置和它最终正确位置的直线距离为估价函数，即欧氏距离。

###### 3.2.2 代码

import heapq

import copy

import time

import math

import argparse

# 初始状态

S0 = [[5, 1, 2, 4],

[9, 6, 3, 8],

[13, 15, 10, 11],

[14, 0, 7, 12]]

# 最终状态

SG = [[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12],

[13, 14, 15, 0]]

# 规定棋牌0只能上下左右四个方向移动

MOVE = {'up': [1, 0],

'down': [-1, 0],

'left': [0, -1],

'right': [0, 1]}

# 建立一个空的OPEN表

OPEN = []

# 节点总数

SUM\_NODE\_NUM = 0

# 状态节点

class State(object):

def \_\_init\_\_(self, deepth=0, rest\_dis=0.0, state=None, hash\_value=None, father\_node=None):

'''

其中deepth表示初始到当前节点的步数，rest\_dis为启发式距离，state为节点当前存储状态，

hash\_value为哈希值（用于判断重复），father\_node是父节点的指针

'''

self.deepth = deepth

self.rest\_dis = rest\_dis

self.fn = self.deepth + self.rest\_dis

self.child = []

self.father\_node = father\_node

self.state = state

self.hash\_value = hash\_value

def \_\_lt\_\_(self, other): # 返回距离最小的

return self.fn < other.fn

def \_\_eq\_\_(self, other): # 判断是否相等

return self.hash\_value == other.hash\_value

def \_\_ne\_\_(self, other): # 判断是否不等

return not self.\_\_eq\_\_(other)

def cal\_M\_distence(cur\_state):

#以曼哈顿距离作为启发式函数

M\_cost = 0

for i in range(4):

for j in range(4):

if cur\_state[i][j] == SG[i][j]:

continue

num = cur\_state[i][j]

if num == 0:

x, y = 3, 3

else:

x = num / 4

y = num - 4 \* x - 1

M\_cost += (abs(x - i) + abs(y - j))

return M\_cost

def cal\_E\_distence(cur\_state):

#以欧氏距离作为启发式函数

E\_cost = 0

for i in range(4):

for j in range(4):

if cur\_state[i][j] == SG[i][j]:

continue

num = cur\_state[i][j]

if num == 0:

x, y = 3, 3

else:

x = num / 4

y = num - 4 \* x - 1

E\_cost += math.sqrt((x - i)\*(x - i) + (y - j)\*(y - j))

return E\_cost

def generate\_child(sn\_node, sg\_node, hash\_set, open\_table, cal\_distence):

'''

其中sn\_node表示当前节点， sg\_node表示最终的状态节点，hash\_set表示哈希表

open\_table表示OPEN表， cal\_distence为距离函数

'''

if sn\_node == sg\_node:

heapq.heappush(open\_table, sg\_node)

print('已找到终止状态！')

return

for i in range(0, 4):

for j in range(0, 4):

if sn\_node.state[i][j] != 0:

continue

for d in ['up', 'down', 'left', 'right']:

x = i + MOVE[d][0]

y = j + MOVE[d][1]

if x < 0 or x >= 4 or y < 0 or y >= 4:

continue

state = copy.deepcopy(sn\_node.state)

state[i][j], state[x][y] = state[x][y], state[i][j]

h = hash(str(state))

if h in hash\_set:

continue

hash\_set.add(h)

# 记录扩展节点的个数

global SUM\_NODE\_NUM

SUM\_NODE\_NUM += 1

deepth = sn\_node.deepth + 1

rest\_dis = cal\_distence(state)

node = State(deepth, rest\_dis, state, h, sn\_node)

sn\_node.child.append(node)

heapq.heappush(open\_table, node)

def show\_block(block, step):

print("~~~~~~~~~~~~~~~")

for b in block:

print(b)

def print\_path(node):

print("搜索过程如下：")

steps = node.deepth

stack = []

while node.father\_node is not None:

stack.append(node.state)

node = node.father\_node

stack.append(node.state)

step = 0

while len(stack) != 0:

t = stack.pop()

show\_block(t, step)

step += 1

return steps

def A\_start(start, end, distance\_fn, generate\_child\_fn):

'''

其中start为起始状态，end为终止状态， distance\_fn为距离函数

generate\_child\_fn为产生孩子节点的函数，A\_start最终返回最优路径长度

'''

root = State(0, 0, start, hash(str(S0)), None)

end\_state = State(0, 0, end, hash(str(SG)), None)

if root == end\_state:

print("start == end !")

OPEN.append(root)

heapq.heapify(OPEN)

node\_hash\_set = set()

node\_hash\_set.add(root.hash\_value)

while len(OPEN) != 0:

top = heapq.heappop(OPEN)

if top == end\_state:

return print\_path(top)

generate\_child\_fn(sn\_node=top, sg\_node=end\_state, hash\_set=node\_hash\_set,

open\_table=OPEN, cal\_distence=distance\_fn)

print("搜索失败!")

return -1

if \_\_name\_\_ == '\_\_main\_\_':

# 可配置式运行文件

parser = argparse.ArgumentParser(description='请选择启发式函数')

parser.add\_argument('--method', '-m', help='method 启发式函数(cal\_E\_distence or cal\_M\_distence)', default = 'cal\_E\_distence')

args = parser.parse\_args()

method = args.method

time1 = time.time()

if method == 'cal\_E\_distence':

length = A\_start(S0, SG, cal\_E\_distence, generate\_child)

else:

length = A\_start(S0, SG, cal\_M\_distence, generate\_child)

time2 = time.time()

if length != -1:

if method == 'cal\_E\_distence':

print("选择 |欧式距离| 作为启发函数")

else:

print("选择 |曼哈顿距离| 作为启发函数")

print("总步骤：", length,"步")

print("总用时：", (time2 - time1), "s")

print("共检测节点", SUM\_NODE\_NUM,"个")

##### 4、实验步骤

###### 4.1 8-数码问题

###### 4.1.1输入：

1 0 3 8 6 5 7 2 4

###### 4.1.2输出：

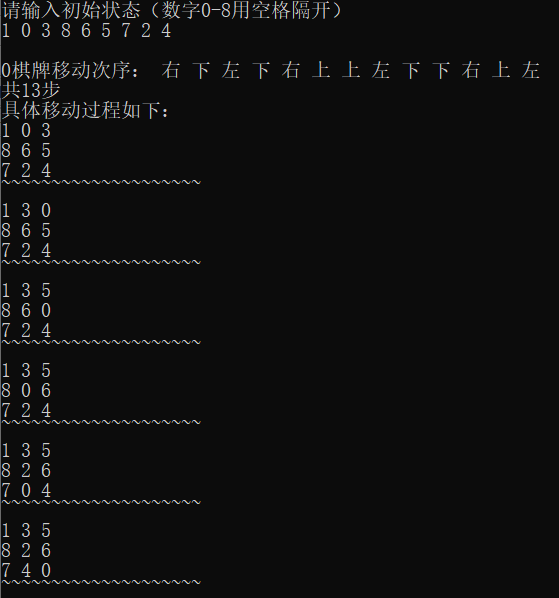
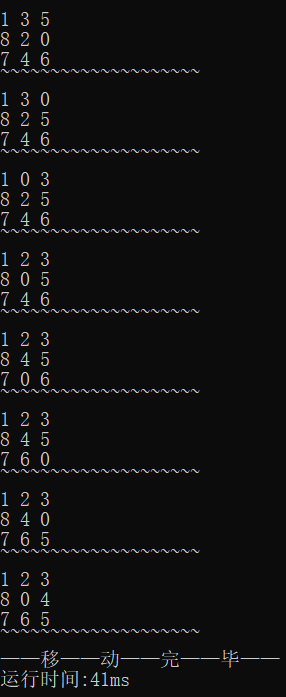
最终输出为：

1 2 3

8 0 4

7 6 5

给出一次运行的结果图：

该输入所需移动的步骤及过程如下图所示：



两种估价函数运行一次的时间分别如下图所示：

估计函数一：  估价函数二： 

###### 4.2 15-数码问题

###### 4.2.1 输入

[5, 1, 2, 4],

[9, 6, 3, 8],

[13, 15, 10, 11],

[14, 0, 7, 12]

###### 4.2.2 输出

最终输出正确排列为：

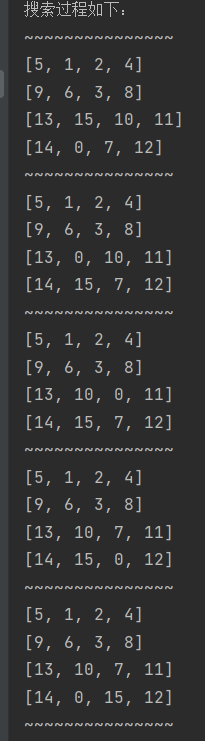
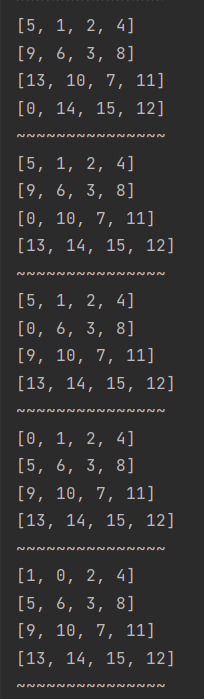
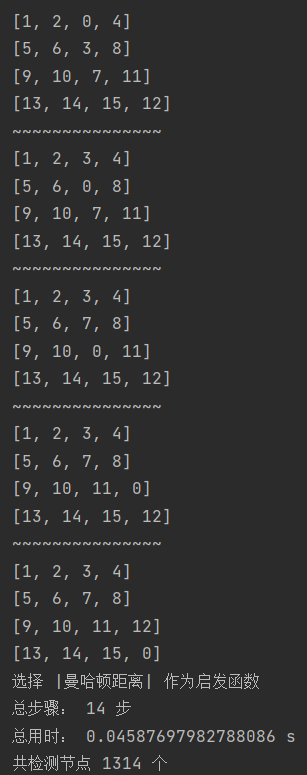
[1, 2, 3, 4],

[5, 6, 7, 8],

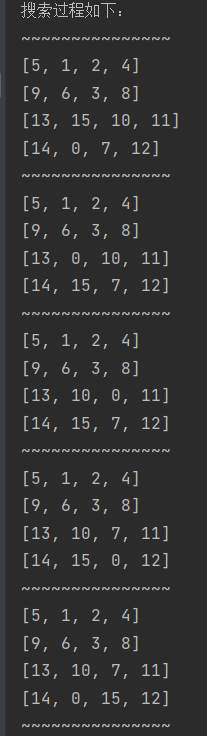
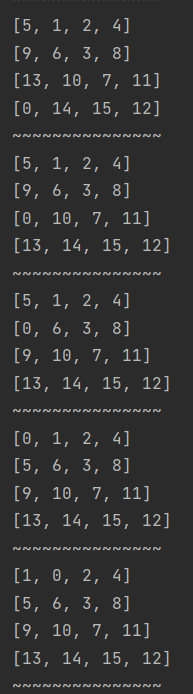
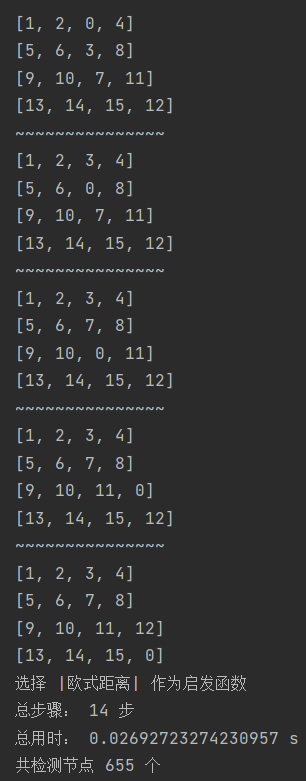
[9, 10, 11, 12],

[13, 14, 15, 0]

采用**曼哈顿距离**为估价函数，结果如下：

采用**欧氏距离**为估价函数，结果如下：

##### 5、讨论分析

###### 5.1 8-数码问题

从运行时间可以看出，8数码问题估价函数的选择对算法效率的影响至关重要，估价函数一的运行时间要高于估价函数二，说明在解决8数码问题时采用当前节点和实际节点所对应的直线距离（欧氏距离）作为h(n)要更加合理。

在错位棋牌数较少的情况下，无论采用哪种估价函数，运行时间都较快，因此没有绝对的优劣，只有在错位棋牌数较多时，采用估价函数二才有明显的效率提升。

###### 5.2 15-数码问题

在运行时间上，欧式距离作为启发式函数仍然具有更高的效率，并且检测的节点数也比曼哈顿距离作为启发式函数时要更少，也即在更加复杂的15数码问题上，欧氏距离只需花费更小的代价就能得到最短路径，因此采用欧氏距离能够显著提升算法效率。

针对15数码问题，只需要在8数码问题中修改相应的移动函数，并重新定义数组（给予足够空间）即可实现。为练习Python使用方式，在15数码问题上采用python进行编程，将8数码问题相应的C++代码转换为python并在此基础上进行更改，从而实现了15数码问题算法。

由于对Python应用不够熟练，在编写15数码问题时没有选择直接由用户输入初始状态的矩阵，而是直接在程序中编写，因此使得程序的灵活性有所降低。在算法的选择上，也是直接在程序中规定的，若能够设置用户选择估价函数则可以使得程序更加合理高效。