## **广州大学学生实验报告**

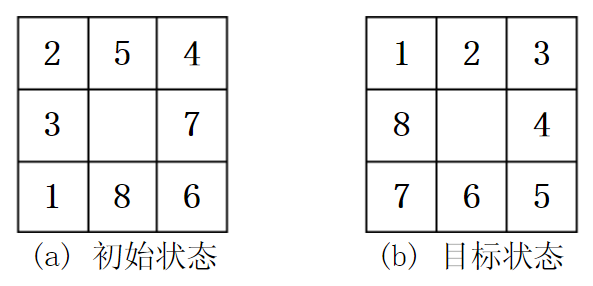
**开课学院及实验室：**计算机科学与工程实验室 **2021年10月2日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | **计算机科学与网络工程学院** | **年级/专业/班** | **网络193** | **姓名** | 吴伟俊 | **学号** | 1906200107 |
| **实验课程名称** | **人工智能导论实验** | | | | | **成绩** |  |
| **实验项目名称** | **基于图搜索技术的八数码问题求解** | | | | | **指导老师** | 张少宏 |

(\*\*\*报告只能为文字和图片,老师评语将添加到此处,学生请勿作答\*\*\*)

**一、实验内容**

**问题描述**：八数码，在3×3的方格棋盘上，摆放着1到8这八个数码，有1个方格是空的，其初始状态如图1所示，要求对空格执行空格左移、空格右移、空格上移和空格下移这四个操作使得棋盘从初始状态到目标状态。



**内容提要**：分别用广度优先搜索策略、深度优先搜索策略和启发式搜索算法（至少两种）求解八数码问题；分析估价函数对启发式搜索算法的影响；探究讨论各个搜索算法的特点。

**二、实验设备**

1. 实验设备：计算机；

2. 平台：Windows操作系统，Visual C++ 6.0

**三、实验步骤**

* 随机生成一个八数码问题分布，设计一个可解的目标状态（要求棋盘9个位置都不同）。

代码如下：

#随机生成3\*3矩阵

def gen():

list = []

arr = []

for i in range(3):

col = []

for j in range(3):

a = random.randint(0, 8)

while 1:

if a not in list:

list.append(a)

col.append(a)

break;

else:

a = random.randint(0, 8)

arr.append(col)

return arr

* 分别用广度优先搜索策略、深度优先搜索策略和至少两种启发式搜索算法求解八数码问题。
* 分析估价函数对启发式搜索算法的影响。

估价函数就是评价函数，它用来评价子节点的好坏，因为准确评价是不可能的，所以成为估值。这就是所谓的有信息搜索。如果估价函数只考虑节点的某种性能上的价值，而不考虑深度，比较有名的就是有序搜索，它看重看好能否找出解，而不看解离起始点的距离。估价函数的形式为：f(n)=g(n)+h(n)，g(n)为起点到当前位置的实际路径长度，h(n)为所在位置到终点的最佳路径的估计距离。这个估算，保证了所找到的路径是最短路径

估价函数中，某位置到终点的最佳路径的估计距离h(x)是估价函数的关键。当该距离等于0时，意味着此时是盲目搜索，该算法则相当于是一个BFS；对于Astar算法来说，该值总小于等h’(x)，h’(x)为一个最短路径的值，因此该算法找到的解总是最优的。

* 探究讨论各个搜索算法的特点。

BFS：这是一种基于队列这种数据结构的搜索方式，它的特点是由每一个状态可以扩展出许多状态，然后再以此扩展，直到找到目标状态或者队列中头尾指针相遇，即队列中所有状态都已处理完毕。

DFS：基于递归的搜索方式，它的特点是由一个状态拓展一个状态，然后不停拓展，直到找到目标或者无法继续拓展结束一个状态的递归。

启发式搜索:它是利用问题拥有的启发信息来引导搜索，达到减少搜索范围、降低问题复杂度的目的. 在A\*算法中，估价函数为f(n)=g\*(n)+h\*(n)。这里面的h\*(n)的附加条件为h\*(n)<=h‘(n)，h’(n)为n到目标的直线最短距离，也就说A\*算法中挑选的启发函数是最优的。A算法效率较高，当没有要求最短路径只求通路时，A算法较好；A\*算法有时会走在一个较长的通路，但在抵达终点之前，肯定会舍弃这条路，走更短的路，而A算法则不一定。

* \*扩展选做题：从初始状态到目标状态的变换，符合什么规律才可解（提示参考：逆序数）

通过逆序数奇偶性判断能否到达

**四、分析说明（包括核心代码及解释）**

1.DFS，BFS，Astar算法：

**import copy**

**import math**

**#棋盘的类，实现移动和扩展状态**

**class grid:**

**def \_\_init\_\_(self,stat):**

**self.pre=None**

**#目标状态**

**#self.target = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]**

**self.target = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]**

**#stat是一个二维列表**

**self.stat=stat**

**self.find0()**

**self.update()**

**#更新启发函数的相关信息**

**def update(self):**

**self.fH()**

**self.fG()**

**self.fF()**

**#H是和目标状态距离之和**

**def fH(self):**

**self.H=0**

**for i in range(3):**

**for j in range(3):**

**targetX=self.target[i][j]**

**nowP=self.findx(targetX)**

**#曼哈顿距离之和**

**#self.H+=abs(nowP[0]-i)+abs(nowP[1]-j)**

**#欧式距离之和**

**self.H+=math.sqrt((nowP[0]-i)\*(nowP[0]-i)+(nowP[1]-j)\*(nowP[1]-j))**

**#欧式距离的平方之和**

**#self.H += (nowP[0] - i) \* (nowP[0] - i) + (nowP[1] - j) \* (nowP[1] - j)**

**#F是启发函数，F=H**

**def fF(self):**

**self.F=self.H**

**#以三行三列的形式输出当前状态**

**def see(self):**

**for i in range(3):**

**print(self.stat[i])**

**print("F=",self.F,"H=",self.H)**

**print("-"\*10)**

**#查看找到的解是如何从头移动的**

**def seeAns(self):**

**ans=[]**

**ans.append(self)**

**p=self.pre**

**while(p):**

**ans.append(p)**

**p=p.pre**

**ans.reverse()**

**for i in ans:**

**i.see()**

**#找到数字x的位置**

**def findx(self,x):**

**for i in range(3):**

**if(x in self.stat[i]):**

**j=self.stat[i].index(x)**

**return [i,j]**

**#找到0，也就是空白格的位置**

**def find0(self):**

**self.zero=self.findx(0)**

**#扩展当前状态，也就是上下左右移动。返回的是一个状态列表，也就是包含stat的列表**

**def expand(self):**

**i=self.zero[0]**

**j=self.zero[1]**

**gridList=[]**

**if(j==2 or j==1):**

**gridList.append(self.left())**

**if(i==2 or i==1):**

**gridList.append(self.up())**

**if(i==0 or i==1):**

**gridList.append(self.down())**

**if(j==0 or j==1):**

**gridList.append(self.right())**

**return gridList**

**#deepcopy多维列表的复制，防止指针赋值将原列表改变**

**#move只能移动行或列，即row和col必有一个为0**

**#向某个方向移动**

**def move(self,row,col):**

**newStat=copy.deepcopy(self.stat)**

**tmp=self.stat[self.zero[0]+row][self.zero[1]+col]**

**newStat[self.zero[0]][self.zero[1]]=tmp**

**newStat[self.zero[0]+row][self.zero[1]+col]=0**

**return newStat**

**def up(self):**

**return self.move(-1,0)**

**def down(self):**

**return self.move(1,0)**

**def left(self):**

**return self.move(0,-1)**

**def right(self):**

**return self.move(0,1)**

**#判断状态g是否在状态集合中，g是对象，gList是对象列表**

**#返回的结果是一个列表，第一个值是真假，如果是真则第二个值是g在gList中的位置索引**

**def isin(g,gList):**

**gstat=g.stat**

**statList=[]**

**for i in gList:**

**statList.append(i.stat)**

**if(gstat in statList):**

**res=[True,statList.index(gstat)]**

**else:**

**res=[False,0]**

**return res**

**#计算逆序数之和**

**def N(nums):**

**N=0**

**for i in range(len(nums)):**

**if(nums[i]!=0):**

**for j in range(i):**

**if(nums[j]>nums[i]):**

**N+=1**

**return N**

**#根据逆序数之和判断所给八数码是否可解**

**def judge(src,target):**

**N1=N(src)**

**N2=N(target)**

**if(N1%2==N2%2):**

**return True**

**else:**

**return False**

**#Astar算法的函数**

**def Astar(startStat):**

**#open和closed存的是grid对象**

**open=[]**

**closed=[]**

**#初始化状态**

**g=grid(startStat)**

**#检查是否有解**

**if(judge(startStat,g.target)!=True):**

**print("所给八数码无解，请检查输入")**

**exit(1)**

**open.append(g)**

**#time变量用于记录遍历次数**

**time=0**

**#当open表非空时进行遍历**

**while(open):**

**#根据启发函数值对open进行排序，默认升序**

**open.sort(key=lambda G:G.F)**

**#找出启发函数值最小的进行扩展**

**minFStat=open[0]**

**#检查是否找到解，如果找到则从头输出移动步骤**

**if(minFStat.H==0):**

**print("found and times:",time,"moves:",minFStat.G)**

**minFStat.seeAns()**

**break**

**#走到这里证明还没有找到解，对启发函数值最小的进行扩展**

**open.pop(0)**

**closed.append(minFStat)**

**expandStats=minFStat.expand()**

**#遍历扩展出来的状态**

**for stat in expandStats:**

**#将扩展出来的状态（二维列表）实例化为grid对象**

**tmpG=grid(stat)**

**#指针指向父节点**

**tmpG.pre=minFStat**

**#初始化时没有pre，所以G初始化时都是0**

**#在设置pre之后应该更新G和F**

**tmpG.update()**

**#查看扩展出的状态是否已经存在与open或closed中**

**findstat=isin(tmpG,open)**

**findstat2=isin(tmpG,closed)**

**#在closed中,判断是否更新**

**if(findstat2[0]==True and tmpG.F<closed[findstat2[1]].F):**

**closed[findstat2[1]]=tmpG**

**open.append(tmpG)**

**time+=1**

**#在open中，判断是否更新**

**if(findstat[0]==True and tmpG.F<open[findstat[1]].F):**

**open[findstat[1]]=tmpG**

**time+=1**

**#tmpG状态不在open中，也不在closed中**

**if(findstat[0]==False and findstat2[0]==False):**

**open.append(tmpG)**

**time+=1**

**stat = [[2, 8, 3], [1, 0, 4], [7, 6, 5]]**

**Astar(stat)**

2.GBFS算法：

**import copy**

**import math**

**#棋盘的类，实现移动和扩展状态**

**class grid:**

**def \_\_init\_\_(self,stat):**

**self.pre=None**

**#目标状态**

**#self.target = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]**

**self.target = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]**

**#stat是一个二维列表**

**self.stat=stat**

**self.find0()**

**self.update()**

**#更新启发函数的相关信息**

**def update(self):**

**self.fH()**

**self.fG()**

**self.fF()**

**#H是和目标状态距离之和**

**def fH(self):**

**self.H=0**

**for i in range(3):**

**for j in range(3):**

**targetX=self.target[i][j]**

**nowP=self.findx(targetX)**

**#曼哈顿距离之和**

**#self.H+=abs(nowP[0]-i)+abs(nowP[1]-j)**

**#欧式距离之和**

**self.H+=math.sqrt((nowP[0]-i)\*(nowP[0]-i)+(nowP[1]-j)\*(nowP[1]-j))**

**#欧式距离的平方之和**

**#self.H += (nowP[0] - i) \* (nowP[0] - i) + (nowP[1] - j) \* (nowP[1] - j)**

**#F是启发函数，F=H**

**def fF(self):**

**self.F=self.H**

**#以三行三列的形式输出当前状态**

**def see(self):**

**for i in range(3):**

**print(self.stat[i])**

**print("F=",self.F,"H=",self.H)**

**print("-"\*10)**

**#查看找到的解是如何从头移动的**

**def seeAns(self):**

**ans=[]**

**ans.append(self)**

**p=self.pre**

**while(p):**

**ans.append(p)**

**p=p.pre**

**ans.reverse()**

**for i in ans:**

**i.see()**

**#找到数字x的位置**

**def findx(self,x):**

**for i in range(3):**

**if(x in self.stat[i]):**

**j=self.stat[i].index(x)**

**return [i,j]**

**#找到0，也就是空白格的位置**

**def find0(self):**

**self.zero=self.findx(0)**

**#扩展当前状态，也就是上下左右移动。返回的是一个状态列表，也就是包含stat的列表**

**def expand(self):**

**i=self.zero[0]**

**j=self.zero[1]**

**gridList=[]**

**if(j==2 or j==1):**

**gridList.append(self.left())**

**if(i==2 or i==1):**

**gridList.append(self.up())**

**if(i==0 or i==1):**

**gridList.append(self.down())**

**if(j==0 or j==1):**

**gridList.append(self.right())**

**return gridList**

**#deepcopy多维列表的复制，防止指针赋值将原列表改变**

**#move只能移动行或列，即row和col必有一个为0**

**#向某个方向移动**

**def move(self,row,col):**

**newStat=copy.deepcopy(self.stat)**

**tmp=self.stat[self.zero[0]+row][self.zero[1]+col]**

**newStat[self.zero[0]][self.zero[1]]=tmp**

**newStat[self.zero[0]+row][self.zero[1]+col]=0**

**return newStat**

**def up(self):**

**return self.move(-1,0)**

**def down(self):**

**return self.move(1,0)**

**def left(self):**

**return self.move(0,-1)**

**def right(self):**

**return self.move(0,1)**

**#判断状态g是否在状态集合中，g是对象，gList是对象列表**

**#返回的结果是一个列表，第一个值是真假，如果是真则第二个值是g在gList中的位置索引**

**def isin(g,gList):**

**gstat=g.stat**

**statList=[]**

**for i in gList:**

**statList.append(i.stat)**

**if(gstat in statList):**

**res=[True,statList.index(gstat)]**

**else:**

**res=[False,0]**

**return res**

**#计算逆序数之和**

**def N(nums):**

**N=0**

**for i in range(len(nums)):**

**if(nums[i]!=0):**

**for j in range(i):**

**if(nums[j]>nums[i]):**

**N+=1**

**return N**

**#根据逆序数之和判断所给八数码是否可解**

**def judge(src,target):**

**N1=N(src)**

**N2=N(target)**

**if(N1%2==N2%2):**

**return True**

**else:**

**return False**

**#Astar算法的函数**

**def Astar(startStat):**

**#open和closed存的是grid对象**

**open=[]**

**closed=[]**

**#初始化状态**

**g=grid(startStat)**

**#检查是否有解**

**if(judge(startStat,g.target)!=True):**

**print("所给八数码无解，请检查输入")**

**exit(1)**

**open.append(g)**

**#time变量用于记录遍历次数**

**time=0**

**#当open表非空时进行遍历**

**while(open):**

**#根据启发函数值对open进行排序，默认升序**

**open.sort(key=lambda G:G.F)**

**#找出启发函数值最小的进行扩展**

**minFStat=open[0]**

**#检查是否找到解，如果找到则从头输出移动步骤**

**if(minFStat.H==0):**

**print("found and times:",time,"moves:",minFStat.G)**

**minFStat.seeAns()**

**break**

**#走到这里证明还没有找到解，对启发函数值最小的进行扩展**

**open.pop(0)**

**closed.append(minFStat)**

**expandStats=minFStat.expand()**

**#遍历扩展出来的状态**

**for stat in expandStats:**

**#将扩展出来的状态（二维列表）实例化为grid对象**

**tmpG=grid(stat)**

**#指针指向父节点**

**tmpG.pre=minFStat**

**#初始化时没有pre，所以G初始化时都是0**

**#在设置pre之后应该更新G和F**

**tmpG.update()**

**#查看扩展出的状态是否已经存在与open或closed中**

**findstat=isin(tmpG,open)**

**findstat2=isin(tmpG,closed)**

**#在closed中,判断是否更新**

**if(findstat2[0]==True and tmpG.F<closed[findstat2[1]].F):**

**closed[findstat2[1]]=tmpG**

**open.append(tmpG)**

**time+=1**

**#在open中，判断是否更新**

**if(findstat[0]==True and tmpG.F<open[findstat[1]].F):**

**open[findstat[1]]=tmpG**

**time+=1**

**#tmpG状态不在open中，也不在closed中**

**if(findstat[0]==False and findstat2[0]==False):**

**open.append(tmpG)**

**time+=1**

**stat = [[2, 8, 3], [1, 0, 4], [7, 6, 5]]**

**Astar(stat)**

核心启发算法解释：

1 读取初始状态和目标状态，通过逆序数奇偶性判断能否到达。

2 将初始节点压入open表中

3 取出open表中估计值最小的节点，放入close表

4 判断该节点是否为目标节点，不是的话，拓展该节点，将子节点放入open表，返回上一步

5 将该节点压入栈中，并将指针指向父节点

6 如果父节点不为空，继续5

7 如果栈不为空，出栈并输出该节点。

一些解释：

在启发式搜索中，每次找到当前“最有希望是最短路径”的状态进行扩展。对于每个状态的我们用函数F来估计它是否有希望。

F=G+H

G：就是普通宽度优先搜索中的从起始状态到当前状态的代价，比如在这次的问题中，G就等于从起始状态到当前状态的最少步数。当G=0时，该算法则为贪婪算法。

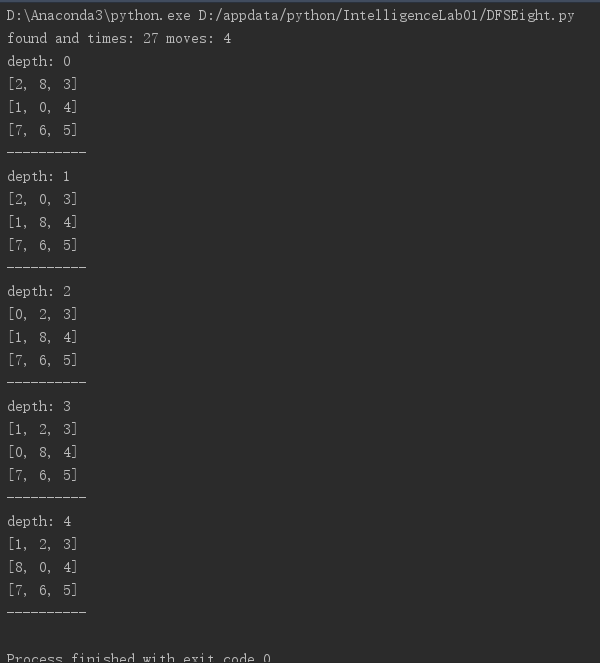
H：是一个估计的值，表示从当前状态到目标状态估计的代价。

H是由我自己设计的，H函数设计的好坏决定了A\*算法的效率。H值越大，算法运行越快。但是在设计评估函数时，需要注意一个很重要的性质：评估函数的值一定要小于等于实际当前状态到目标状态的代价。否则虽然程序运行速度加快，但是可能在搜索过程中漏掉了最优解。相对的，只要评估函数的值小于等于实际当前状态到目标状态的代价，就一定能找到最优解。所以，在这个问题中我将评估函数设定为1-8八数字当前位置到目标位置的曼哈顿距离之和。

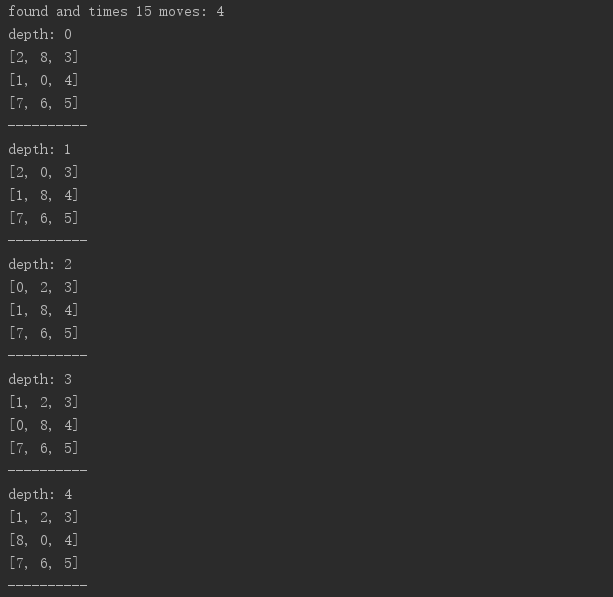
**五、总结心得**

**实验结果：**

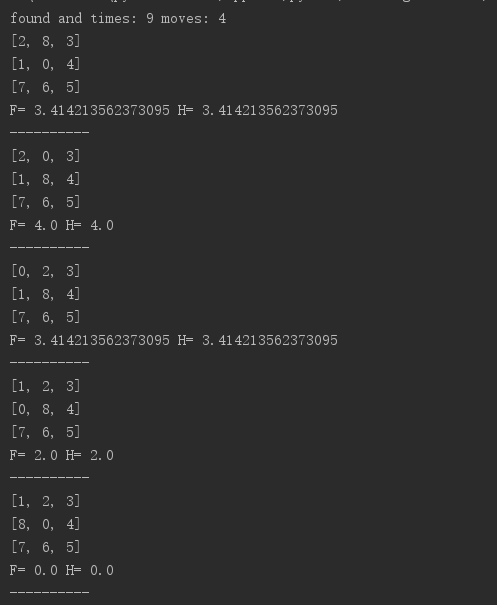
**DFS:**



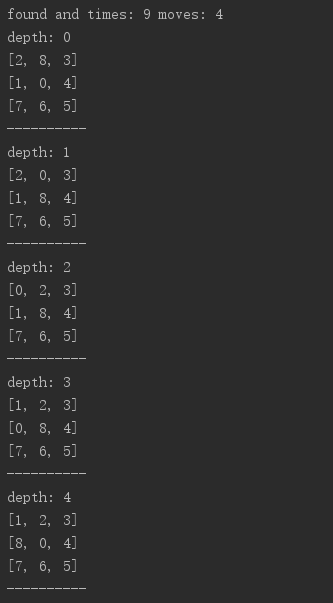
**BFS:**



**GBFS:**



**Astar:**



实验总结：

通过本次实验，我了解并熟悉了多种算法，如BFS，DFS，GBFS，AStar等算法，并体会到了估价函数对启发式算法的影响。比如，当h(x)=0时，AStar算法可以变成BFS算法。学习到了估计函数的h(x)的计算有不同的方法，如欧式距离，曼哈顿距离，欧氏距离的平方等方法。希望今后能学到更多有用的知识。