盲目搜索策略：深度优先搜索解决“八数码”问题

# 1实验目的

1. 了解盲目的图搜索策略；
2. 了解深度优先搜索策略；
3. 体会深度优先搜索策略的优缺点；
4. 比较深度优先搜索策略与宽度优先搜索策略的特点。

# 2实验要求

本次试验后，要求学生能：

1. 理解深度优先搜索策略的搜索过程、特点；
2. 理解深度限制对深度优先搜索结果的影响；
3. 用深度优先搜索策略编程写python程序，解决“八数码”问题；
4. 理解深度优先搜索与宽度优先搜索的异同。

# 3实验原理

## 3.1实验介绍

本实验将用宽度优先搜索策略编程写python程序，解决“八数码”问题：

八数码问题（重排九宫格问题）是在一个3×3的方格盘上，放有1~8的数码，余下一个为空（用“0”表示）。空格四周上下左右的数码皆可移动到空格里。需要找到一个数码移动序列，使初始无序数码转变为一些特殊排列。

例如，图1即为八数码问题的两个状态，需要找到从初始状态到目的状态的路径。

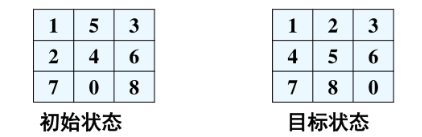


图1 八数码问题举例

本实验需要大家有一定的Python语言编程基础。

## 3.2深度优先搜索策略

深度优先搜索策略（depth-first search , DFS）属于盲目的图搜索策略，类似于树的先根遍历。搜索从初始状态S0出发，沿一个方向一直扩展下去；如果未能找到目的状态，或者无法再扩展，便回溯到另一个方向继续沿着这个新的方向一直搜索下去；如果还未找到目的状态或无法继续扩展，再回溯到另一方向……

当搜索到某一状态时，该状态的所有子状态及子状态的后裔状态都必须先于该状态被搜索。应尽可能地往深处探索，只有再也找不到某状态的后裔状态或到达深度限制时，才考虑该状态的兄弟状态。

## 3.2深度限制

显然，深度优先搜索法不一定得到最优解，并且可能由于深度的限制，会找不到解。

但是对深度优先搜索策略进行深度限制是十分有必要的，如果不加以限制，可能会沿着一条错误路径一直无限地扩展下去，这当然是不希望的。为了保证能找到解，就应该选择合适的深度限制，或者采取不断加大深度限制值的办法，反复搜索，直到找到解。

# 4实验步骤

## 4.1程序实现

（1）深度优先搜索的程序实现与宽度优先搜索大同小异，只需将宽度优先搜索中队列结构先进先出的opened表，改成先进后出的堆栈结构，即可实现深度优先搜索。

与上一章中搜索过程的区别在于，选取current节点时，不再像宽度优先那样选择opened表的第一个节点作为current，而是选opened表的最后一个节点

# 上一章中宽度优先算法搜索时是current = opened.pop(0)

current = opened.pop()

closed.append(current)

（2）需对深度优先搜索进行深度的限制，可以在每个子状态被生成时，将该状态所在的层数记录在集合gn中，当深度达到限制时，停止当前方向的搜索，回溯到另一方向继续搜索。

gn = {} # gn用于存放各个状态所在深度

修改expand（）函数：当到达深度限制时，停止扩展

def expend(state): # state为扩展前的状态

# 检验当前深度是否已经超出上限，若超出上限，则不进行扩展

if gn[state] > int(depth\_limit):

return None

expended = []

k = state.index("0") # k为0所在的位置

for a in range(0, len(movs[k])):

i = k # i为0的位置

j = movs[i][a] # j为待交换元素的位置

if i > j:

i, j = j, i

new = state[: i] + state[j] + state[i + 1: j] + state[i] + state[j + 1:] # 扩展出的一个新状态

expended.append(new)

return expended

搜索时，在将未出现过的子状态加入opened表的同时，在gn中更新其深度值。

newStates = expend(current)

if newStates is None:

continue

for s in newStates:

if s not in opened and s not in closed:

gn[s] = gn[current] + 1

parent[s] = current

opened.append(s)

## 4.2各函数完整代码

# 扩展出不同状态

def expend(state): # state为扩展前的状态

# 检验当前深度是否已经超出上限，若超出上限，则不进行扩展

if gn[state] > int(depth\_limit):

return None

expended = []

k = state.index("0") # k为0所在的位置

for a in range(0, len(movs[k])):

i = k # i为0的位置

j = movs[i][a] # j为待交换元素的位置

if i > j:

i, j = j, i

new = state[: i] + state[j] + state[i + 1: j] + state[i] + state[j + 1:] # 扩展出的一个新状态

expended.append(new)

return expended

# 计算逆序数，判断是否有解，打印路径。这部分与宽度优先一样。

def reverse\_number(state):

Sum = 0

for i in range(1, 9):

num = 0

for j in range(0, i):

if state[j] > state[i] != '0':

num = num + 1

Sum += num

return Sum

def is\_solvable(S0):

i = reverse\_number(S0)

j = reverse\_number(goal)

if i % 2 == j % 2:

return True

else:

return False

def print\_result(state):

# 根据parent中的索引，找出路径

results = [state] # 用来存放路径

while parent[state] != -1:

state = parent[state]

results.append(state)

results.reverse() # 逆序

print("可求解，求解过程如下：")

i = -1

for result in results:

i = i + 1

print("")

print("step----" + str(i))

print(result[:3])

print(result[3:6])

print(result[6:])

print("")

# 深度优先搜索

def search\_depth(S0):

global parent, gn, limit

sum = limit

opened = []

closed = []

# S0加入opened表

opened.append(S0)

# 开始搜索

while opened:

# 检验搜索次数是否超出限制

limit = limit - 1

search\_times = sum - limit

print("正在进行第%d次搜索" % search\_times)

if limit < 1:

return current

# opened表中删除第一个状态n，将n放入closed表，

current = opened.pop() # 宽度优先：opened表使用先进先出的堆栈结构，使搜索优先偏向先生成的状态

closed.append(current)

print("正在搜索第%d层" % gn[current])

print("curret:" + current)

print("goal:" + goal)

# 搜索成功，结束循环

if current == goal:

break

# 扩展当前状态，删除子状态中在opened表或closed表中出现过的，避免重复循环搜索

# 其余子状态加入opened表

newStates = expend(current)

if newStates is None:

continue

for s in newStates:

if s not in opened and s not in closed:

gn[s] = gn[current] + 1

parent[s] = current

opened.append(s)

# 检验是由于opened表为空而停止，还是因为搜索到了解路径而停止。

if not opened:

print("有解但是由于深度限制无法求出解！")

return current

## 4.3程序运行与测试

输入不同的初始状态、目标状态、搜索次数限制、搜索深度限制，比较结果有何不同，结合实验体会宽度优先搜索的特点。

同时，可以将相同的数据用上一节实现的宽度优先搜索进行实验，对比两种策略的搜索结果，体会两种搜索策略的不同特点。

主函数代码如下：

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

# 操作算子集合

movs = {0: [1, 3], 1: [0, 2, 4], 2: [1, 5], 3: [0, 4, 6], 4: [3, 1, 5, 7], 5: [4, 2, 8], 6: [3, 7], 7: [6, 4, 8],

8: [7, 5]}

gn = {} # gn用于存放各个状态所在深度

parent = {} # 用于存放各个状态的父状态，用于输出解路径

# 输入初始状态和目标状态

state0 = input("请输入初始状态（从左到右从上到下）：")

goal = input("请输入目标状态（从左到右从上到下）：")

# 输入搜索次数上限

limit = int(input("请输入搜索次数的上限（例如：50000）："))

# 输入深度上限

depth\_limit = int(input("请输入搜索深度的上限（例如：5000）："))

parent[state0] = -1 # 初始状态的父状态设置为-1

gn[state0] = 0 # 初始状态的深度设置为0

# 判断是否有解

if state0 == goal:

print("初始状态与目标状态一致，搜索结束。")

elif not is\_solvable(state0) or len(state0) != 9:

print("不可达，无解！")

else:

current = search\_depth(state0) # 开始搜索

print\_result(current) # 按格式输出结果

if limit == 0:

print("有解但搜索超时，建议更换搜索算法或目标序列！！！")

# 5实验结果

（1）输入初始状态（102345678）、目标状态（123456780），以及搜索次数上限（50000）、深度限制（5000）。

运行程序后，运行结果（部分）如下：

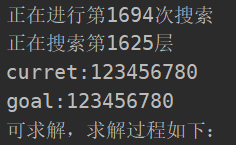


图2 部分运行结果

如图2可看出，一共进行了1694次搜索，在第1625层搜索得到了一条初始状态到目标状态的路径。层数和搜索次数接近，说明对深度限制要结合具体问题来设置，设置过低可能会导致无法求出解路径，或反而增加搜索时间。

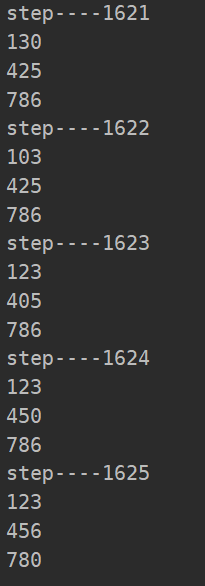


图3 搜索路径（部分）

（2）设置不同的搜索深度限制，搜索结果、搜寻时间会发生变化。

还是输入与之前相同的初始状态（102345678）、目标状态（123456780），以及搜索次数上限（50000）。但是将深度限制设置为10，再次进行实验。



图4 深度限制为10

由图4可以看出，当深度限制设置为10时，可能会找不出解。

（3）还可以尝试更多的初始状态、目标状态、深度限制的组合，比较结果的差异，同时和上一节的宽度优先搜索结果进行对比。