盲目搜索策略：宽度优先搜索解决“八数码”问题

# 1实验目的

1. 了解盲目的图搜索策略；
2. 了解宽度优先搜索策略；
3. 体会宽度优先搜索策略的优缺点。

# 2实验要求

本次试验后，要求学生能：

1. 理解宽度优先搜索策略的搜索过程、特点；
2. 用宽度优先搜索策略编程写python程序，解决“八数码”问题。

# 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。

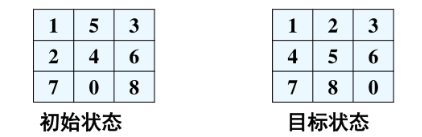


图1 八数码问题举例

本实验需要大家有一定的Python语言编程基础。

## 3.2宽度优先搜索策略

宽度优先搜索策略（breadth first search , BFS）类似于树的层次遍历，由初始状态S0生成状态1、2，然后扩展1，生成状态3、4、5，接着扩展2，生成6、7、8，该层全部搜索完成后才进入到下一层进行搜索，对状态3进行扩展，如此一层一层地扩展下去，指导找到目的状态（如果可以找到）。

由于宽度优先搜索总是在生成扩展完第N层的所有状态结点后才转向第N+1层，所以如果有解，宽度优先搜索总是能找到最优解。

在实际的宽度优先搜索中，为了保存状态空间搜索的轨迹，要用到opened、closed两个表，opened表包含已经生成出来但是子状态未被扩展的状态，opened表中状态的排列次序就是搜索的次序；closed表记录了已经被生成且被扩展过的状态。

需要注意：

（1）宽度优先搜索策略中的opened表是一个先进先出（FIFO）的队列结构。

（2）曾在opened表与closed表中出现过的状态都要及时删除，以免重复搜索。

（3）虽然宽度优先搜索总能找到最优解，但是，当图的分支太多，即状态的平均后裔数过大，这种组合爆炸会使算法耗尽资源，从而在可利用的空间中找不到解，同时这也会导致算法消耗极长的时间。所以要对搜索的时间或次数进行限制，避免浪费大量的时间空间。

## 3.3八数码问题有解性的判断

逆序数：在一个排列中，如果一对数的前后位置与大小顺序相反，即前面的数大于后面的数，那么它们就称为一个逆序。一个排列中逆序的总数就称为这个排列的逆序数。例如，“012345678”的逆序数为0，“102345678”的逆序数为1，“780123456”的逆序数为14。

八数码问题中，可以证明得出，任意数码进入空格都不会改变整体排序逆序数的奇偶性。因此，如果初始状态与目标状态的逆序数奇偶性相同，则此八数码问题必有解。

## 3.4八数码问题中状态扩展策略

八数码的任何一种摆法都是一种状态，对于八数码问题中的状态扩展，如果着眼于数码的移动，那么一共有4（方向）×8（数码）=32种可以扩展状态的操作算子，会十分繁琐。

但是如果着眼于空格上，那么操作算子可以简化成4个，根据空格（本题用“0”代表空格）所在的位置，即可选择合适的操作算子对状态进行扩展。

（1）0在左上角，可以向下或向右移动；

（2）0在最上面一排的中间，可以向左或向右或向下移动；

（3）0在右上角，可以向下或向右移动；

…… ……

# 4实验步骤

## 4.1判断问题是否有解

（1）表示各个状态

用0来表示空格，按从左到右、从上到下的顺序将各个状态的八数码存入一个String类型的变量中。

（2）计算状态逆序数

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

（3）判断问题是否有解

初始状态与目标状态的逆序数奇偶性相同时有解，即，逆序数除以2的余数相等时有解。

def is\_solvable(S0):

i = reverse\_number(S0)

j = reverse\_number(goal)

if i % 2 == j % 2:

return True

else:

return False

## 4.2扩展出不同状态

（1）创建操作算子集合，保存了空格（“0”）在九宫格各个位置可用的操作算子。例如，0在第2排中间（9宫格的正中间）时，0在此状态（String类型）中的序号为4，查询movs集合可得，0可以与该状态中需要为3、1、5、7的数码交换位置。

# 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]}

（2）根据空格的位置从当前状态扩展出子状态。

def expend(state): # state为扩展前的状态

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 expende

## 4.3宽度优先搜索

用到opened、closed两个列表，其中opened表为先进先出的队列结构。每次循环先取opened表中的第一个状态记为current，将它从opened表删除。然后对current进行扩展，生成current的所有子状态，将子状态中已经在opened或closed表中出现过的删除，将其余子状态按生成顺序加入opened表的表尾。最后将current加入closed表，然后继续循环。

由于前面已经判断过是否有解，所以必能搜索到解路径。但是搜索的时间空间开销可能很大，所以需要设置搜索次数上限。

def search\_width(S0):

global limit, parent

sum = limit # 搜索次数限制

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(0) # 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(current)

print(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)

for s in newStates:

if s not in opened and s not in closed:

parent[s] = current

opened.append(s)

return current

## 4.4打印搜索结果路径

在上一步的搜索中，每次由current生成新的子状态（记为new\_state）时，用parent保存（new\_state：current）关系，从而实现在最后输出搜索的完整路径。

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

print(result[:3])

print(result[3:6])

print(result[6:])

## 4.5 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，比较结果有何不同，结合实验体会宽度优先搜索的特点。

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]}

# 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）："))

parent = {state0: -1} # 存放各个状态的父状态，用于最后输出结果路径

# 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\_width(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）运行程序后，运行结果如下：

输入初始状态（102345678）、目标状态（140372685），以及搜索次数上限（100000）：

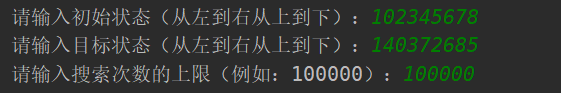


图2 输入初始与目标状态以及限制

得到搜索路径：

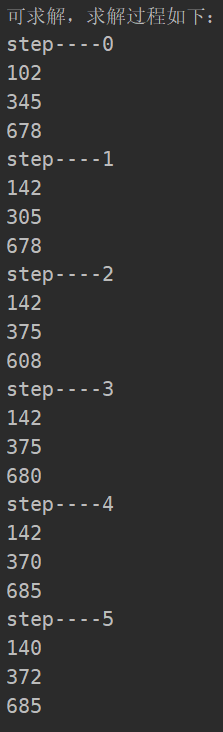


图3 搜索路径

在第5步搜索到解，说明目标状态在第5层。

一共进行了61次搜索：

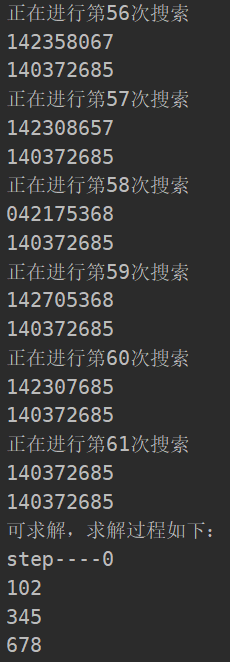


图4 搜索次数

（2）若初始状态到达目标状态的最短路径过长，尽管由初始状态和目标状态的逆序数判断出题目是有解的，但是宽度优先搜索策略难以在有限的时间和空间内完成搜索任务。

输入初始状态“102345678”、目标状态“123456780”、搜索次数限制50000，再次实验，实现结果如图5、图6.

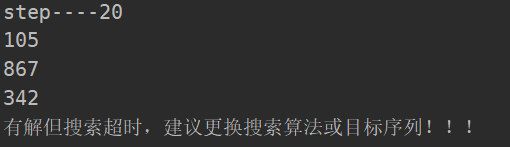


图5 搜索超时

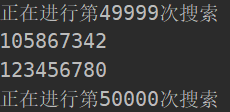


图6 搜索次数

如图所示，搜索了50000次，耗时约4分钟也未能找到结果（只是用于举例，实际上50000多次搜索也在可接受范围内）。