启发式搜索策略：A星搜索解决“八数码”问题

# 1实验目的

1. 理解启发式图搜索策略；
2. 理解A及A\*搜索算法；
3. 理解A\*算法的特性；
4. 比较启发式搜索算法与盲目搜索算法的差异；

# 2实验要求

本次试验后，要求学生能：

1. 理解启发式图搜索策略的搜索过程、特点；
2. 理解启发信息，能针对不同问题设计启发函数；
3. 用启发式图搜索策略编程编写python程序，解决“八数码”问题；
4. 理解A搜索算法与A\*搜索算法的关系；

# 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。

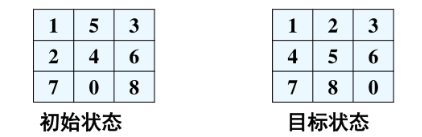


图1 八数码问题举例

本实验需要大家有一定的Python语言编程基础。

## 3.2启发式策略

### 3.2.1 简介

前两章的实验都是盲目搜索算法，其搜索的复杂性往往是很高的。为了提升搜索的效率，必须放弃用纯数学的方法来决定搜索次序，而需要对具体问题进行具体分析，利用与题目有关的信息，从中得到启发来引导搜索，从而减少搜索量。这就是启发式搜索策略，引导搜素的与题目有关的信息就是启发信息。

启发式信息可以用于在模糊的搜索中确定最合适的解，也可以在状态空间特别大的搜索问题中引导搜索向最有希望的方向搜索来降低复杂度。

但是启发式策略也极容易出错，仅仅只是对下一步搜索猜测性地指导，可能得到最优解也可能得到次优解，甚至可能一无所获。这是启发式策略的固有的局限性，这种局限性不可能通过所谓更好的启发式策略来彻底消除。

### 3.2.2启发信息与启发函数

启发式信息主要有：用于表述状态，使问题空间缩小的陈述性启发信息；用于构造操作算子，使操作算子少而精的过程性启发信息；表示控制策略，协调求救过程的控制性启发信息，这三种。

估价函数可以用来估计待搜索节点“有希望”的程度，主要要考虑已经付出的代价和将要付出的代价。估价函数f（n）一般定义为从初始节点经过n节点到达目的节点路径的最小代价的估计值，一般形式为：

f（n）= g（n）+ h（n）

其中，g（n）项表示初始节点到n的实际代价。保持了搜索宽度优先的成分，g（n）比重越大，搜索越倾向宽度优先搜索，有利于搜索的完备性，但是影响效率。

h（n）项表示n到目标节点的估计代价，体现了启发信息，h（n）越大，搜索的启发性越强，搜索效率越高，但是会影响搜索的完备性。

## 3.3 A搜索算法

A搜索算法是一种基于估价函数的加权启发式图搜索算法，搜索步骤与宽度优先、深度优先算法类似，区别在于：

（1）每次从opened表中选取新的current节点时，宽度优先搜索选取opened表的第一个节点；深度优先搜索选取opened表的最后一个节点；而A\*搜索算法选择opened表中估价函数值最小的节点作为current。

（2）对current状态进行扩展后，如果发现一个新状态已经在opened表或closed表中出现过，则只保留估价函数f（n）更小的一种（相同节点的启发函数h（n）相同，所以实际上比较深度g（n）即可）。

## 3.4 A\*搜索算法

A\*搜索是目前最佳的图搜索算法，定义h\*（n）为状态n到目标状态的最优路径，则当A搜索法中的启发函数h（n）小于等于h\*（n）时，就可以成为A\*搜索算法。

在八数码问题中，将h（n）设置为当前状态与目标状态的“错位数”（例如“123”与“321”错位数为2，“12345”与“54321”错位数为4），显然，此时的h（n）满足

h（n）≤h\*（n）

因此，这样构造出的八数码A搜索树也是A\*搜索树，可以找到最佳解。

如果某一问题有解，则A\*搜索一定能找到解，且一定能找到最优解。

# 4实验步骤

## 4.1程序实现

（1）扩展子状态、判断是否有解、打印路径的程序与宽度优先搜索一样。

# 扩展出不同状态

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 expended

# 计算逆序数并判断是否可解

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, goal):

i = reverse\_number(S0)

j = reverse\_number(goal)

if i % 2 == j % 2:

return True

else:

return False

# 打印路径

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:])

（2）计算当前状态与目标状态的错位数，作为启发函数hn（state）。

估价函数设置为：f（state）= hn（state）+ gn（state）

def hn(state):

hn = 0

for i in range(0, 9):

if state[i] != goal[i]:

hn +=1

return hn

（3）找出opened表中估价函数值最小的状态

def find\_min(opened):

temp = {}

for state in opened:

k = fn[state]

temp[state] = k

state\_min = min(temp, key=temp.get)

return state\_min

（4）搜索流程与宽度优先、深度优先搜索相似。

区别在于，选取current节点时，宽度优先搜索选取opened表的第一个节点；深度优先搜索选取opened表的最后一个节点；A\*搜索算法选择opened表中估价函数值最小的节点作为current。

current = find\_min(opened)

del fn[current]

opened.remove(current)

closed.append(current)

还有一个不同之处在于，每次生成新的子状态后，宽度优先搜索和深度优先搜索是只将从未在opened表、closed表中出现过的子状态加入opened表，不管已出现过的节点；而A\*搜索中，如果一个状态不是第一次出现（已在opened表或closed表中），要比较已在表中的那个状态s的gn1和新生成的状态s’的gn2（s与s’序列一样，只有gn对应的深度值不一样），只保留gn最小的那个状态，从而保证得到的是更优解。

for s in newStates:

# 如果扩展出的子状态不在opened表也不在closed表

# 计算该子状态的估值函数，加入到opened表中

if s not in opened and s not in closed:

gn[s] = gn[current] + 1

fn[s] = gn[s] + hn(s)

parent[s] = current

opened.append(s)

# 如果扩展出的子状态，已经在opened表或closed表中出现过了

# 比较gn，记录更短的路径走向及其估价函数值，并更新parent

else:

if s in opened:

if gn[s] > gn[current] + 1:

gn[s] = gn[current] + 1

# fn[s] = gn[s] + hn(s)

parent[s] = current

（5）搜索部分的完整代码

def search\_Astar(S0):

global parent, limit, gn, fn, opened, closed, current

sum = limit

# 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表中删除估价函数值最小的状态n，将n放入closed表，

current = find\_min(opened)

del fn[current]

opened.remove(current)

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表中出现过的状态，避免重复循环搜

索

# 其余子状态加入opened表

newStates = expend(current)

for s in newStates:

# 如果扩展出的子状态不在opened表也不在closed表

# 计算该子状态的估值函数，加入到opened表中

if s not in opened and s not in closed:

gn[s] = gn[current] + 1

fn[s] = gn[s] + hn(s)

parent[s] = current

opened.append(s)

# 如果扩展出的子状态，已经在opened表或closed表中出现过了

# 比较gn，记录更短的路径走向及其估价函数值

else:

if s in opened:

if gn[s] > gn[current] + 1:

gn[s] = gn[current] + 1

# fn[s] = gn[s] + hn(s)

parent[s] = current

return current

## 4.2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，结合实验体会为什么A\*被称为“最佳图搜索策略”。

同时，可以将相同的数据用之前两个章节中实现的宽度优先、深度优先搜索进行实验，对比搜索结果，体会启发式搜索的优越性。

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]}

opened = []

closed = []

gn = {}

fn = {}

parent = {}

# 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 # 初始状态的父状态设置为-1

gn[state0] = 0 # 初始状态已付代价为0

fn[state0] = gn[state0] + hn(state0) # 计算初始状态的估价函数值

# 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, goal) or len(state0) != 9:

print("不可达，无解！")

else:

current = search\_Astar(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

输入初始状态（283164705）、目标状态（123804765），进行A\*搜索，得到的结果如下：



图2-1 A\*搜索结果（搜索次数）

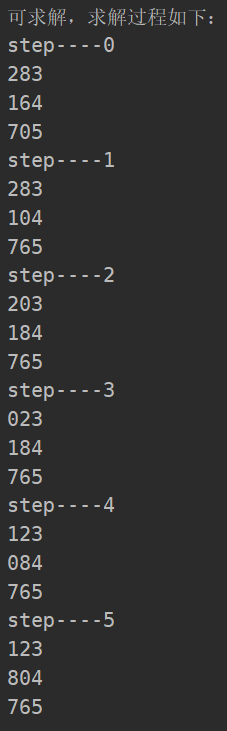


图2-2 A\*搜索结果（解路径）

可以更改初始状态、目标状态、估价函数进行更多的实验，同时将结果和前两章中的宽度优先搜索、深度优先搜索比较。

输入上一章用宽度优先搜索过的初始状态（“102345678”）、目标状态（“12345678”），结果如下：

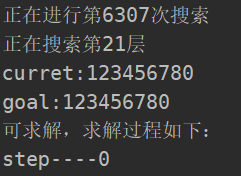


图3-1 另一组数据的搜索结果（搜索次数）

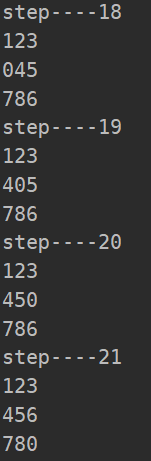


图3-2 另一组数据的搜索结果（部分路径）