

# 華中師範大學

## 算法设计与应用

报 告： 2019《算法设计》课程开卷考试试题

院 系： 国家数字化学习工程技术研究中心

姓 名： 鲍一鸣

学 号： 2019123389

时间：2019 年 12 月 16 日

1. 利用**动态规划**法求 0-1 背包问题。有  $n=20$  个物品，背包最大可装载  $M = 878$

Kg。物品重量和价值分别如下：878

$W=\{92, 4, 43, 83, 84, 68, 92, 82, 6, 44, 32, 18, 56, 83, 25, 96, 70, 48, 14, 58\}$ ,

$V=\{44, 46, 90, 72, 91, 40, 75, 35, 8, 54, 78, 40, 77, 15, 61, 17, 75, 29, 75, 63\}$ ,

求最优背包价值。

### 问题分析与算法设计思路

**动态规划核心：**计算并存储小问题的最优解，并将这些最优解组合成大问题的最优解。（将原问题分解为若干子问题，然后自底向上，先求解最小的子问题，把结果存储在表格中，再求解大的子问题时，直接从表格中查询小的子问题的解，避免重复计算，从而提高算法效率）

考虑原问题的一部分，设  $V(i, j)$  表示将前  $i$  ( $1 \leq i \leq n$ ) 个物品装入容量为  $j$  ( $1 \leq j \leq C$ ) 的背包获得最大价值，在决策  $x_i$  时，已经确定了  $(x_1, \dots, x_{i-1})$ ，则问题处于下列两种状态之一：

- (1) 背包容量不足以装入物品  $i$ ，则装入前  $i$  个物品得到的最大价值和装入前  $i-1$  个物品得到的最大价值是相同的，即  $x_i=0$ ，背包不增加价值
- (2) 背包容量可以装入物品  $i$ ，如果把第  $i$  个物品装入背包，则背包中物品的价值等于把前  $i-1$  个物品装入容量为  $j-w_i$  的背包中的价值加上第  $i$  个物品的价值  $v_i$ ；如果第  $i$  个物品没有装入背包，则背包中物品的价值等于把前  $i-1$  个物品装入容量为  $j$  的背包中所取得的价值。显然，取二者中价值较大者作为把前  $i$  个物品装入容量为  $j$  的背包中的最优解。得到如下递推式：

$$V(i, j) \begin{cases} V(i-1, j) \\ \max \{V(i-1, j), V(i-1, j-w_i) + v_i\} \end{cases}$$

### 算法实现部分：

```
# n 物品数量 int
# c 书包能承受的重重 int
# w 每个物品的重量 list
# v 每个物品的价值 list
def bag(n, c, w, v):
    value = [[0 for j in range(c + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, c + 1):
            if j < w[i - 1]:
                value[i][j] = value[i - 1][j]
            else:
                value[i][j] = max(value[i - 1][j], value[i - 1][j - w[i - 1]] + v[i - 1])
        # 背包总容量够放当前物体，取最大价值
    for x in value:
        print(x)
    return value
if __name__ == '__main__':
    n=20
    c=878
    w = [92, 4, 43, 83, 84, 68, 92, 82, 6, 44, 32, 18, 56, 83, 25, 96, 70, 48, 14, 58]
    v = [44, 46, 90, 72, 91, 40, 75, 35, 8, 54, 78, 40, 77, 15, 61, 17, 75, 29, 75, 63]
    bag(n, c, w, v)
```

生成查询表格（部分如下所示）：

[illegible]

源代码：

```
# -*- coding: utf-8 -*-
# @Time : 2019/12/15 16:30
# @Author : BaoBao
# @Mail : baobaotql@163.com
# @File : 0-1bags.py
# @Software: PyCharm

def bag(n, c, w, v):
    value = [[0 for j in range(c + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, c + 1):
            if j < w[i - 1]:
                value[i][j] = value[i - 1][j]
            else:
```

```

        value[i][j] = max(value[i -
1][j], value[i - 1][j - w[i - 1]] + v[i - 1])# 背包总容量够
放当前物体，取最大价值
    return value

```

```

def show(n, c, w, value):
    x = [0 for i in range(n)]
    j = c
    for i in range(n, 0, -1):
        if value[i][j] > value[i - 1][j]:
            x[i - 1] = 1
            j -= w[i - 1]

    print('背包中所装物品为:')

    weight = 0
    for i in range(n):
        if x[i]:

            print('第', i+1, '个 ', end='')

            print('重量为', w[i], ' ', end='')

            print('价值为', v[i], ' ', end='\n')
            weight = weight + w[i]

    print('最大价值为:', value[n][c])

    print('背包重量为: ', weight)

```

```

if __name__=='__main__':
    n = 20
    c = 878
    w = []
    v = []

    '''
    weight_i = [92, 4, 43, 83, 84, 68, 92, 82, 6, 44, 32,
18, 56, 83, 25, 96, 70, 48, 14, 58]
    value_j = [44, 46, 90, 72, 91, 40, 75, 35, 8, 54,
78, 40, 77, 15, 61, 17, 75, 29, 75, 63]

```

```
'''

for i in range(0, n):
    print('第', i + 1, '件物品重量:')
    weight_i = int(input())
    w.append(weight_i)

print("/*****/")

for j in range(0, n):
    print('第', j + 1, '件物品价格:')
    value_j = int(input())
    v.append(value_j)

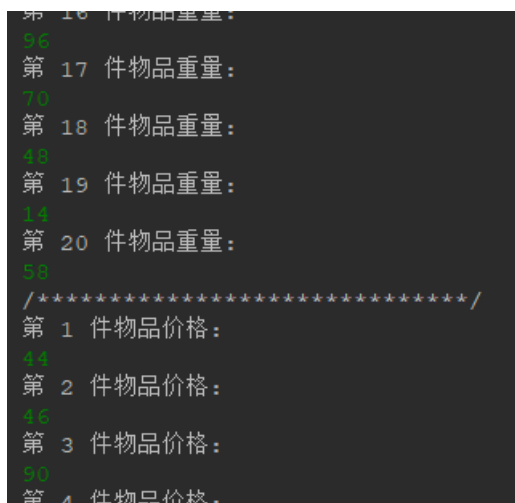
print('所有商品重量: ', w)

print('所有商品价格: ', v)

value = bag(n, c, w, v)
show(n, c, w, value)
```

运行截图：

(输入物品重量以及价格，最后得到背包重量以及最大价值)



```
第 16 件物品重量:
56
第 17 件物品重量:
70
第 18 件物品重量:
48
第 19 件物品重量:
14
第 20 件物品重量:
58
/****/
第 1 件物品价格:
44
第 2 件物品价格:
45
第 3 件物品价格:
90
第 4 件物品价格:
44
```

```

所有商品重量: [92, 4, 43, 83, 84, 68, 92, 82, 6, 44, 32, 18, 56, 83, 25, 96, 70, 48, 14, 58]
所有商品价格: [44, 46, 90, 72, 91, 40, 75, 35, 8, 54, 78, 40, 77, 15, 61, 17, 75, 29, 75, 63]
背包中所装物品为:
第 1 个 重量为 92  价值为 44
第 2 个 重量为 4   价值为 46
第 3 个 重量为 43  价值为 90
第 4 个 重量为 83  价值为 72
第 5 个 重量为 84  价值为 91
第 6 个 重量为 68  价值为 40
第 7 个 重量为 92  价值为 75
第 8 个 重量为 82  价值为 35
第 9 个 重量为 6   价值为 8
第 10 个 重量为 44  价值为 54
第 11 个 重量为 32  价值为 78
第 12 个 重量为 18  价值为 40
第 13 个 重量为 56  价值为 77
第 15 个 重量为 25  价值为 61
第 17 个 重量为 70  价值为 75
第 19 个 重量为 14  价值为 75
第 20 个 重量为 58  价值为 63
最大价值为: 1024
背包重量为: 871

```

2. 设计算法，求一个正整数  $n$  的整数划分表示。例如正整数 6 有如下 11 种不同的划分：

6;

5+1;

4+2, 4+1+1;

3+3, 3+2+1, 3+1+1+1;

2+2+2, 2+2+1+1, 2+1+1+1+1;

1+1+1+1+1+1。

### 问题分析与算法设计思路

在正整数  $n$  的所有划分中，将不大于  $m$  的划分个数记为  $p(n, m)$ ，其中  $m$  为最大加数。根据  $n$  和  $m$  的关系，考虑以下几种情况：建立  $p(n, m)$  的如下递归关系。

(1) 当  $m = 1$  时， $p(n, 1) = 1$ ， $n \geq 1$ 。当最大加数为 1 时，任何正整数  $n$  只有一种划分形式，即  $n = 1 + 1 + 1 + \dots + 1$ ;

(2) 当  $n=1$  时,  $p(1,m) = 1$ , 不论  $m$  的值为多少 ( $m>0$ ), 只有一种划分, 即 $\{1\}$ ;

(3) 当  $n < m$  时,  $p(n, m) = p(n, n)$ 。最大加数不能大于  $n$ , 划分中不可能出现负数, 因此就相当于  $p(n,n)$

(4) 当  $n=m$  时, 根据划分中是否包含最大加数  $n$ , 可以分为两种情况:

(a). 划分中包含最大加数  $n$  的情况, 只有 1 个, 即 $\{n\}$ ;

(b). 划分中不包含最大加数  $n$  的情况, 这时划分中最大的加数一定比  $n$  小, 即  $n$  的所有  $(n-1)$  划分。

因此  $p(n, n) = 1 + p(n, n-1)$ ;

(5) 当  $n>m$  时, 根据划分中是否包含最大加数  $m$ , 可以分为两种情况:

(a). 划分中包含最大加数  $m$  的情况, 即 $\{m, \{x_1, x_2, \dots, x_i\}\}$ , 其中 $\{x_1, x_2, \dots, x_i\}$  的和为  $n-m$ , 可能再次出现  $m$ , 因此是  $(n-m)$  的  $m$  划分, 因此这种划分的个数为  $p(n-m, m)$ ;

(b). 划分中不包含最大加数  $m$  的情况, 则划分中所有值都比  $m$  小, 即  $n$  的  $(m-1)$  划分, 个数为  $p(n, m-1)$ ;

因此  $p(n, m) = p(n-m, m) + p(n, m-1)$ ;

综合以上情况, 我们可以看出, 上面的结论具有递归定义特征, 其中 (1) 和

(2) 属于递归结束条件, (3) 和 (4) 属于特殊情况, 将会转换为情况 (5)。

而情况 (5) 为通用情况, 属于递推的方法, 其本质主要是通过减小  $m$  以达到递归结束条件, 从而解决问题。其递推表达式如下:

$$\begin{cases} 1 & n = 1 \quad \text{or } m = 1 \\ p(n, n) & n < m \\ 1 + p(n, n-1) & n = m \\ p(n, m-1) + p(n-m, m) & n > m > 1 \end{cases}$$

## 源代码

```
# -*- coding: utf-8 -*-
# @Time : 2019/12/15 18:00
# @Author : BaoBao
# @Mail : baobaotql@163.com
# @File : Integer_partition.py
# @Software: PyCharm

def partition(n, m, string):
    if n == 0:
        print(string)
    else:
        if m > 1:
            partition(n, m - 1, string)
        if m <= n:
            partition(n - m, m, str(m) + ' ' + string)

def divide(n, m):
    if (n == 1) or (m == 1):
        return 1
    if n < m:
        return divide(n, n)
    if n == m:
        return divide(n, n-1) + 1
    if n > m > 1:
        return divide(n, m-1) + divide(n-m, m)

if __name__ == '__main__':
    print('输入一个整数: ')
    n = int(input())
    m = n
    p = divide(n, m)

    print('划分方法有%d种' % p)

    print('%d 的划分如下: ' % n)
    partition(n, m, '')
```



## 运行截图

```
输入一个整数：
7
划分方法有15种
7的划分如下：
1 1 1 1 1 1 1
1 1 1 1 1 2
1 1 1 2 2
1 2 2 2
1 1 1 1 3
1 1 2 3
2 2 3
1 3 3
1 1 1 4
1 2 4
3 4
1 1 5
2 5
1 6
7
```

3. 利用启发式算法（可以是模拟退火、遗传算法等经典启发式算法，也可以自己原创）求解 30 个城市的 TSP 问题。两城市之间距离用直角坐标系中的两点距离公式。

{41,94},{37,84},{54,67},{25,62},{7,64},{2,99},{68,58},{71,44},{54,62},{83,69}  
{64,60},{18,54},{22,60},{83,46},{91,38},{25,38},{24,42},{58,69},{71,71},{74,  
78}{87,76},{18,40},{13,40},{82,7},{62,32},{58,35},{45,21},{41,26},{44,35},{4  
,50}

## 问题分析与算法设计思路

求解旅行商问题属于完全 NP 问题，精确的解决 TSP 问题只能通过穷举所有的路径组合，其时间复杂度为  $O()$

在该问题中我们采用的启发式算法为模拟退火算法来解决旅行商问题。首先我们对模拟退火算法进行简单的介绍：（1）随机挑选一个单元  $k$ ，并给他一个随机的位移，求出系统因此而产生的能量变化  $\Delta E_k$ （2）若  $\Delta E_k \leq 0$ ，该位移可

以采纳，而变化后的系统状态可作为下次变化的起点；若 $\Delta E_k > 0$ ，位移后的状态可以采纳的概率为

$$P_k = \frac{1}{1 + e^{-\Delta E_k/T}}$$

式中  $T$  为温度，然后从  $(0,1)$  区间均匀分布的随机数中挑选一个数  $RR$ 。若  $R < P_k$ ，则将变化后的状态作为下次的起点；否则，将变化前的状态作为下次的起点（3）转第（1）步继续执行，直到达到平衡状态为止。

模拟退火的关键在于一定概率  $p$  下接受比当前差的状态，而概率  $p$  又与当前的状态相关，当前状态越差，接受概率  $p$  越大，当前状态越好，接受概率  $p$  越小。

- （1） 初始化，模拟退火中用温度来观察程序进程，这里模拟温度，设置  $t=100$ ，从  $100^\circ$  开始降温，程序停止条件  $t_0=1$ ，温度下降的速率为  $\alpha=0.99$ ，在每个温度状态下，程序要尝试随机生成路径  $\text{markovlen}$  次， $\text{markovlen}=1000$
- （2） 创建城市坐标矩阵  $\text{distmat}$ ， $\text{distmat}[i][0]$  表示第  $i$  个城市的  $x$  坐标， $\text{distmat}[i][1]$  表示第  $i$  个城市的  $y$  坐标
- （3） 先随机生成一条路径  $\text{solutionnew}$ ，通过函数  $\text{getdistmat}()$  计算路径总长度  $\text{distmat}$
- （4） 再随机选择 2 个城市  $\text{loc1}$  和  $\text{loc2}$ ，交换  $\text{solutionnew}$  中这两个城市的位置，形成新的路径  $\text{solutioncurrent}$
- （5） 计算路径长度  $\text{solutioncurrent}$
- （6） ①如果新的路径长度更小，更新路径，并与当前最小的路径对比，如果

小于当前最小的路径，更新最小路径与最小值

②新的路径长度不小于现有路径的长度，则根据模拟退火概率去接

- (7) (4) (5) (6) 在某个  $t_0$  状态，迭代 markovlen 次，每次会交换两个城市生成的 solutionnew，然后  $t=t*\alpha$ ，温度状态改变，重复 4、5、6 知道  $t=1$ ，得到最优路径。
- (8) 打印最优解路径以及路径长度

**源代码：**

```
# -*- coding: utf-8 -*-
# @Time : 2019/12/16 11:32
# @Author : BaoBao
# @Mail : baobaotql@163.com
# @File : TSP_SA.py
# @Software: PyCharm

import numpy as np
import matplotlib.pyplot as plt
import imageio
import pdb

"旅行商问题 ( TSP , Traveling Salesman Problem )"
coordinates = np.array([
    [41, 94],
    [37, 84],
    [54, 67],
    [25, 62],
    [7, 64],
    [2, 99],
    [68, 58],
    [71, 44],
    [54, 62],
    [83, 69],
    [64, 60],
    [18, 54],
    [22, 60],
    [83, 46],
    [91, 38],
```

```
[25, 38],  
[24, 42],  
[58, 69],  
[71, 71],  
[74, 78],  
[87, 76],  
[13, 40],  
[18, 40],  
[82, 7],  
[62, 32],  
[58, 35],  
[45, 21],  
[41, 26],  
[44, 35],  
[4, 50]])
```

# 得到距离矩阵的函数

```
def getdistmat(coordinates):  
    '''  
  
    :param coordinates:  
    :return:  
    '''  
  
    num = coordinates.shape[0] # 30 个坐标点  
  
    distmat = np.zeros((30, 30)) # 30x30 距离矩阵  
  
    for i in range(num):  
        for j in range(i, num):  
            distmat[i][j] = distmat[j][i] =  
np.linalg.norm(coordinates[i] - coordinates[j])  
        return distmat
```

```
def initpara():  
    '''  
  
    :return:  
    '''  
  
    alpha = 0.99  
    t = (1, 100)  
    markovlen = 1000
```

```

        return alpha, t, markovlen

num = coordinates.shape[0]

distmat = getdistmat(coordinates) # 得到距离矩阵

solutionnew = np.arange(num)
# valuenew = np.max(num)

solutioncurrent = solutionnew.copy()
valuecurrent = 99000 # 取一个较大值作为初始值
# print(valuecurrent)

solutionbest = solutionnew.copy()
valuebest = 99000 # np.max

alpha, t2, markovlen = initpara()
t = t2[1]

result = [] # 记录迭代过程中的最优解
while t > t2[0]:
    for i in np.arange(markovlen):

        # 下面的两交换和三角换是两种扰动方式，用于产生新解

        if np.random.rand() > 0.5: # 交换路径中的这 2 个
            # 节点的顺序

            # np.random.rand()产生[0, 1)区间的均匀随
            # 机数

            while True: # 产生两个不同的随机数
                loc1 =
np.int(np.ceil(np.random.rand() * (num - 1)))
                loc2 =
np.int(np.ceil(np.random.rand() * (num - 1)))
                ## print(loc1,loc2)
                if loc1 != loc2:
                    break

```

```

        solutionnew[loc1], solutionnew[loc2] =
solutionnew[loc2], solutionnew[loc1]

    else: # 三交换
        while True:
            loc1 =
np.int(np.ceil(np.random.rand() * (num - 1)))
            loc2 =
np.int(np.ceil(np.random.rand() * (num - 1)))
            loc3 =
np.int(np.ceil(np.random.rand() * (num - 1)))

            if ((loc1 != loc2) & (loc2 !=
loc3) & (loc1 != loc3)):

                break

        # 下 loc1<loc2<loc3
        if loc1 > loc2:
            loc1, loc2 = loc2, loc1
        if loc2 > loc3:
            loc2, loc3 = loc3, loc2
        if loc1 > loc2:
            loc1, loc2 = loc2, loc1
        tmp1ist =
solutionnew[loc1:loc2].copy()
        solutionnew[loc1:loc3 - loc2 + 1 +
loc1] = solutionnew[loc2:loc3 + 1].copy()
        solutionnew[loc3 - loc2 + 1 +
loc1:loc3 + 1] = tmp1ist.copy()

        valuenew = 0
        for i in range(num - 1):
            valuenew +=
distmat[solutionnew[i]][solutionnew[i + 1]]
            valuenew +=
distmat[solutionnew[0]][solutionnew[29]]
        # print (valuenew)

        if valuenew < valuecurrent: # 接受该解

            # 更新 solutioncurrent 和 solutionbest
            valuecurrent = valuenew

```

```

        solutioncurrent = solutionnew.copy()

        if valuenew < valuebest:
            valuebest = valuenew
            solutionbest =
solutionnew.copy()

        else: # 按一定的概率接受该解

            if np.random.rand() < np.exp(-(
(valuenew - valuecurrent) / t):
                valuecurrent = valuenew
                solutioncurrent =
solutionnew.copy()
            else:
                solutionnew =
solutioncurrent.copy()
                t = alpha * t
                result.append(valuebest)

                print(t) #进展速度

#显示结果

print('最短路径为: ', np.array(result[-1]))

for i in solutionbest:
    print(i, end="-->")
print(solutionbest[0])

plt.plot(np.array(result))
plt.ylabel("bestvalue")
plt.xlabel("t")
plt.show()

```

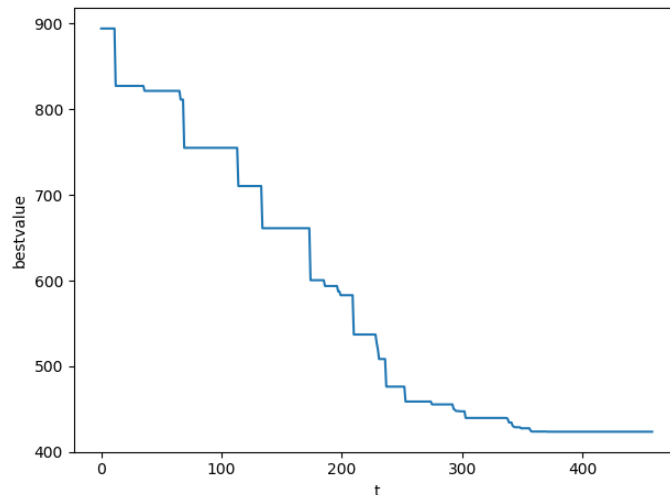
**运行截图：**

**(运行结果包含最短路径长度以及最短路径)**

```

1.0327938091311127
1.0224656730596016
1.0122410163290056
1.0021186061657157
0.9920974201040585
最短路径为: 423.740563133203
0-->5-->4-->3-->12-->11-->29-->21-->22-->16-->15-->28-->27-->26-->25-->24-->23-->14-->13-->7-->6
-->10-->9-->20-->19-->18-->17-->8-->2-->1-->0

```



## 求解 TSP 问题的的方法二 (2-opt)

使用 2-opt 思想解决该问题的算法如下 (首先设置参数最大迭代次数

maxCount, 初始化计数器 count 为 0):

1. 随机选择一条路线 (比方说是 A->B->C->D->E->F->G), 假设是最短路线 min;
2. 随机选择在路线 s 中不相连两个节点, 将两个节点之间的路径翻转过来获得新路径, 比方我们随机选中了 B 节点和 E 节点, 则新路径为 A->(E->D->C->B)->F->G, ()部分为被翻转的路径;
3. 如果新路径比 min 路径短, 则设新路径为最短路径 min, 将计数器 count 置为 0, 返回步骤 2, 否则将计数器 count 加 1, 当 count 大于等于 maxCount 时, 算法结束, 此时 min 即为最短路径, 否则返回步骤 2;

源代码:

```
# -*- coding: utf-8 -*-
# @Time : 2019/12/15 19:39
# @Author : BaoBao
# @Mail : baobaotql@163.com
# @File : TSP.py
```



```

# @Software: PyCharm
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt

MAXCOUNT = 2000

# 城市的坐标
cities = np.array([
    [41, 94],
    [37, 84],
    [54, 67],
    [25, 62],
    [7, 64],
    [2, 99],
    [68, 58],
    [71, 44],
    [54, 62],
    [83, 69],
    [64, 60],
    [18, 54],
    [22, 60],
    [83, 46],
    [91, 38],
    [25, 38],
    [24, 42],
    [58, 69],
    [71, 71],
    [74, 78],
    [87, 76],
    [13, 40],
    [18, 40],
    [82, 7],
    [62, 32],
    [58, 35],
    [45, 21],
    [41, 26],
    [44, 35],
    [4, 50]
])

def calDist(xindex, yindex):
    """

```

```

        :param xindex:
        :param yindex:
        :return:
        '''
        return (np.sum(np.power(cities[xindex] -
cities[yindex], 2))) ** 0.5

def calPathDist(indexList):
    '''
    :param indexList:
    :return:
    '''
    sum = 0.0
    for i in range(1, len(indexList)):
        sum += calDist(indexList[i], indexList[i - 1])
    return sum

def pathCompare(path1, path2):
    '''
    :param path1:
    :param path2:

    :return: path1 长度比 path2 短则返回 true
    '''
    if calPathDist(path1) <= calPathDist(path2):
        return True
    return False

def generateRandomPath(bestPath):
    '''
    :param bestPath:
    :return:
    '''
    a = np.random.randint(len(bestPath))
    while True:
        b = np.random.randint(len(bestPath))
        if np.abs(a - b) > 1:
            break
    if a > b:
        return b, a, bestPath[b:a + 1]
    else:
        return a, b, bestPath[a:b + 1]

```

```

def reversePath(path):
    '''
    :param path:
    :return: repath
    '''
    rePath = path.copy()
    rePath[1:-1] = rePath[-2:0:-1]
    return rePath

def updateBestPath(bestPath):
    '''
    :param bestPath:
    :return: bestpath
    '''
    count = 0
    while count < MAXCOUNT:
        print(calPathDist(bestPath))
        print(bestPath.tolist())
        start, end, path = generateRandomPath(bestPath)
        rePath = reversePath(path)
        if pathCompare(path, rePath):
            count += 1
            continue
        else:
            count = 0
            bestPath[start:end + 1] = rePath
    return bestPath

def draw(bestPath):
    '''
    :param bestPath:
    :return:
    '''
    ax = plt.subplot(111, aspect='equal')
    ax.plot(cities[:, 0], cities[:, 1], 'x', color='blue')
    for i, city in enumerate(cities):
        ax.text(city[0], city[1], str(i))
    ax.plot(cities[bestPath, 0], cities[bestPath, 1],
color='red')
    plt.show()

```

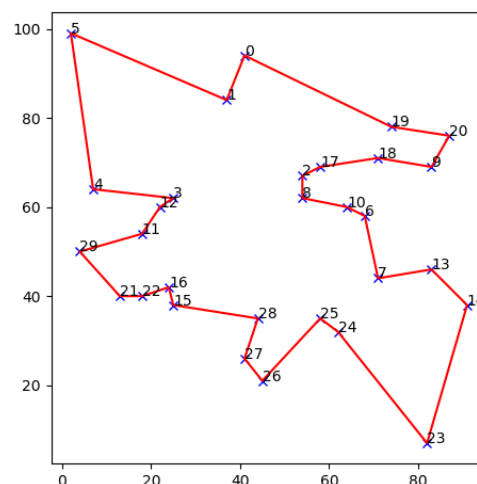
```
def opt2():
    # 随便选择一条可行路径
    bestPath = np.arange(0, len(cities))
    bestPath = np.append(bestPath, 0)
    bestPath = updateBestPath(bestPath)
    draw(bestPath)

if __name__ == '__main__':
    opt2()
```

运行截图：

（运行截图中包含路径以及路径长度，以及城市间的路径显示）

```
12, 3, 4, 5, 1, 0]
425.26670638827983
[0, 19, 20, 9, 18, 17, 2, 8, 10, 6, 7, 13, 14, 23, 24, 25, 26, 27, 28, 15, 16, 22, 21, 29, 11,
12, 3, 4, 5, 1, 0]
```



4. 在与自己研究相关的文献中，找一篇涉及经典算法或启发式算法或经典启发式算法（如模拟退火、禁忌搜索、遗传算法）应用的论文。用自己的语言解读该论文，内容要求包括问题描述、算法思想和具体算法过程。

**相关论文：**“Dynamic Simulated Annealing for solving the Traveling Salesman

Problem with Cooling Enhancer and Modified Acceptance Probability ” Akshay

Vyas, Dashmeet Kaur Chawla, Dr. Urjita Thakar Department of Computer

Engineering, Shri G.S. Institute of Technology & Science, 23 Park Road, Nr.

Lantern Square, Indore, Madhya Pradesh, India, PIN 45200

**论文阅读地址：**

[https://www.researchgate.net/publication/323750246\\_Dynamic\\_Simulated\\_Annealing\\_for\\_solving\\_the\\_Traveling\\_Salesman\\_Problem\\_with\\_Cooling\\_Enhancer\\_and\\_Modified\\_Acceptance\\_Probability](https://www.researchgate.net/publication/323750246_Dynamic_Simulated_Annealing_for_solving_the_Traveling_Salesman_Problem_with_Cooling_Enhancer_and_Modified_Acceptance_Probability)

**论文代码复现：** <https://github.com/baobaotql/Simulated-annealing-algorithm/tree/master>

**论文问题描述：**旅行商问题，即 TSP 问题（Traveling Salesman Problem），是求最短路径的问题，即“已给一个  $n$  个点的完全图，每条边都有一个长度，求总长度最短的经过每个顶点正好一次的封闭回路”。TSP 是组合优化问题，可以被证明具有 NPC 计算复杂性。如果希望暴力搜索其最佳解，其复杂度将是  $O(n!)$ ，其计算量随着  $n$  的增加将轻易超过目前计算机的可能算力。因此我们需要用更智能的方法求解。

于是我们先考虑局部搜索算法。局部搜索算法是贪心算法，他往往往邻域中最好的状态搜索，因此容易进入局部最优结果，而无法跳出局部最优的区域。

第二部分使用模拟退火算法。模拟退火算法从某一较高初温出发，伴随温度参数的不断下降,结合概率突跳特性在解空间中随机寻找目标函数的全局最优解,即在局部最优解能概率性地跳出并最终趋于全局最优。模拟退火算法比起局部搜索算法，赋予了一定跳出局部最优解的能力，但能否跳出局部最优解依然依赖随机性。

该项目主要是利用局部搜索算法（LS）和模拟退火算法（SA）解决 TSP 问题。

先是使用 LS 求解 TSP 问题，再尝试 SA 问题，比较两者，在效率上 SA 更占有。最后再在 LS 的基础上使用 SA，再优化 SA 部分算法，尝试求解 TSP 问题。选用的 TSP 测例为 eil101（有 101 个城市）

算法设计：

首先使用两种不同的局部搜索算法。

第一种选择邻域的方法是随机交换两个城市在序列中的顺序。每次循环中产生的候选序列为城市数（以下用  $C_s$  表示）\*10，并从中选择一个最优的（距离最短的）作为下一步。

第二种选择邻域的方法是随机交换三个城市在序列中的顺序。每次循环中产生的候选序列为  $C_s*10$ ，并从中选择一个最优的（距离最短的）作为下一步。

这两种算法都按以下步骤实现：

1. 录入初始状态，并打乱顺序产生一组随机状态，从这组状态（包括初始状态）中选最佳的状态作为起点；
2. Repeat:
  - a) 产生一个集合 S
  - b) Repeat  $10 * C_s$  times:
    - i. 将当前状态加入 S
    - ii. 产生 2 个（或 3 个）互不相同的、范围为[1, 城市数-1]的随机数
    - iii. 以这 2 个（或 3 个）随机数作为下标交换城市在序列中的顺序
    - iv. 将交换后的序列加入 S 中
  - c) 从 S 中选择一个最优的序列，作为当前状态
  - d) 如果当前状态与之前状态一样，则跳出循环。

可以知道，当当前状态与邻域中最佳状态一样时跳出循环，可以理解成到达局部最优解。虽然实际上这个邻域并没有完全覆盖当前状态的所有邻居，但覆盖全部邻居需要 $(C_s-1) * (C_s-2)$ （第二种邻域为 $(C_s-1) * (C_s-2) * (C_s-3)$ ）个数据，将加大每次循环的耗时，而且最终结果同样是会进入局部最优结果而无法跳出。

第二部分在 LS 的基础上加入 SA。

一开始我的 SA 流程如下：

1. 得到初始状态，设定初温  $T$ ，降温方式，结束条件
2. 外循环：
  - a) 当符合结束条件则跳出循环
  - b) 内循环：
    - i. 令当前解能量为  $D_0$
    - ii. 通过邻域搜索策略得到一组解并取其中最优（不包括当前状态）解能量为  $D_1$
    - iii. 令  $\Delta E = D_1 - D_0$ 
      1. If  $\Delta E \leq 0$ : 则使  $P = 1$   
Else: 使  $P$  为  $e^{-\Delta E/T}$ （或其他形式，其  $P$  应随着  $T$  降低而降低，而且  $\Delta E$  越小则越高）。
      2. 产生一个  $[0,1)$  的小数  $R$ , 若  $R < P$  则接受新状态, 否则不接受。
  - c) 降温

而本次实验使用了非传统的 SA——DSA-CE&MAP（本论文）

Step 1: Initialize cooling\_rate with a small value such as 0.001.

Generate an initial random tour  $x$ .

Step 2: Initialize  $T$  with a large value such as 100000.

Step 3: if totalCities < 30, then set coolingEnhancer = 0.5.

else if totalCities < 150 then set coolingEnhancer = 0.05.

else if totalCities < 750 then set coolingEnhancer = 0.005.

Otherwise, set coolingEnhancer = 0.0005.

Step 3: Repeat:

i. Generate next tour  $(x + \Delta x)$  by applying some operations on the current tour  $x$ .

ii. Evaluate  $\Delta E(x) = E(x + \Delta x) - E(x)$ , (i.e. neighborTourCost - currentTourCost):

if  $\Delta E(x) < 0$ , keep the new state (i.e. new path distance less than current distance);

otherwise, evaluate  $\Delta E' = E_{\text{bestSoFar}}(x) - E(x + \Delta x)$ , (i.e. bestTourCost - neighborTourCost) and then accept the new state with

acceptance probability,  $P = \frac{e^{-\Delta E/T}}{e^{-\Delta E'/T}}$

iii. If  $E_{\text{bestSoFar}}(x) > E(x + \Delta x)$ , then set  $E_{\text{bestSoFar}}(x) = E(x + \Delta x)$ .

iv. Set  $T = T - \Delta T$ ,  $\Delta T = T \times \text{coolingEnhancer} \times \text{cooling\_rate}$ .

until  $T$  is small enough.

(以上为 DSA-CE&MAP 论文中描述的过程)

使用该种策略能在经典 SA 的基础上更合理的降温且更合理的得到选择概率。

观察概率函数可以发现，新解不仅与当前解比较，还与最佳解比较。用到概

率函数的前提是当前解比新解好。当新解与当前解差距大的时候，分子会减

小， $P$  减小，符合策略。当新解与最优解差距大的时候（注意这里是最优解

— 新解），分母会增大， $P$  减小，符合策略。即，一个新解不仅考虑与当前解

的差距，还考虑与曾到达的最优解的差距。这样每次升温将考虑到更多因素，

使每次升温更慎重。

这里还引入了一个新的参数 coolingEnhancer 来影响降温策略。当城市越多

的时候，因为每个状态将更复杂，引入 coolingEnhancer 使其降温速度更慢，

使外循环迭代次数增加，增强算法的适应能力。

在 DSA-CE&MAP 的基础上，邻域搜索策略我再作了修正，由于前两种局部

搜索策略效果不佳，使用了第三种局部搜索策略（2-OPT）：

若  $W(l, l+1) + W(j, j+1) > W(l, j) + W(l+1, j+1)$

则用边  $(l, j)$  和  $(l+1, j+1)$  替换  $(l, l+1), (j, j+1)$



其中  $i, j$  为某两座城市下标,  $W(a,b)$  表示城市  $a$  到下一座城市  $b$  的距离。这种策略能很好的解决路线交叉的问题, 而上面两种交换城市的方法很难处理路线交叉。这种方法可以理解成用凸四边形的两条对边代替两条对角线 (好的效果)。

这种边的替代依赖于该问题中城市之间的距离是对称的 (即交换两个序列中相邻的城市的顺序不会影响两城市之间的距离)。

假设原本的顺序是  $i, i+1, s[n], j, j+1$ , 则边替换后则变成  $i, j, s[n], i+1, j+1$ , 其中  $i+1$  与  $j$  之间的路线将会因为先到达  $j$  再到达  $i+1$  而反转。我们观察可以发现  $i$  和  $j+1$  是没有变动的。  $S_2 = (i+1) + s[n] + (j)$  是整个反转了。因此我们只需要获得两个随机下标并将其中的城市序列反转即可得到新状态。

用与其他搜索同样的方法得到一个关于序列的集合, 并挑最优解。

由于 DSA-CE&MAP 中给出的初温过高, 因此将初温降低为 1000, 并将结束条件设置为  $T < 1$  (试运行发现  $T < 1$  后基本到达局部最优解), 以进一步提升速率。

在结合 DSA-CE&MAP (改良模拟退火) 与 2-OPT (局部搜索) 后, 达到了实验目标的 10%。

### 论文中得出的结论:

实际一开始使用的 SA 是局部搜索策略一、或策略一结合策略二得到的邻域配合经典 SA。由于该局部搜索策略过于简单, 导致即使调高邻域范围、调高初温、调低降温速率, 结果耗费大量时间 (一个小时) 迭代也无法进入 10% 的解。后来即使改进了 SA 的策略, 提升是有, 但也无法进入 10%, 而且还增加了时间消耗。最终以上方案轻易被 2-OPT 这个局部搜索解超过。因此选择

一个正确的局部搜索策略十分重要。可以看到 SA 对 LS 的提升是有的，但响应的，也会耗费更多的时间，耗费的时间主要取决与邻域的范围、初温、降温策略。因此需要多此调试得到最佳参数。