

“旅行商”问题实验报告

21215122, 何峙, 大数据与人工智能

实验内容

旅行商问题是指一名商人要到若干城市去推销商品, 已知城市个数和各城市间的路程(或旅费), 要求找到一条从最开始城市出发, 经过所有城市且每个城市只能访问一次, 最后回到最开始城市的路线, 使总的路程(或旅费)最小。

实验设计

本实验使用模拟退火算法求解。模拟退火算法的核心思想为: 算法迭代过程中, 以一定概率接受比当前解更差的解, 然后使用这个更差的解继续搜索。现利用模拟退火算法步骤对旅行商问题建模:

(1) 解空间

设经过每个城市形成一个路径为一个解, 形成某个排序 $s_i = \{c_1, c_2, c_3, \dots\}$, 其中 c_i 代表每个城市, 则解空间为所有这些排列的集合 $S = \{s_1, s_2, s_3, \dots\}$ 。

(2) 初始解

模拟退火算法的最优解与初始解无关, 故初始解可为随机生成的一个排列, 如 $s_0 = \{c_1, c_2, c_3, \dots\}$ 。

(3) 目标函数

目标函数即为所有城市的路径总长度(或总旅费):

$$\text{cost}(c_1, c_2, \dots, c_n) = \sum_{i=1}^{n+1} d(c_i, c_{i+1}) + d(c_1, c_n)$$

其中 d 为计算两个城市之间的距离函数。

(4) 新解的产生

本实验采用将两个城市的逆序变换作为新解: 任选城市序号 j 和 k , 交换 j 和 k 的访问顺序, 即若交换前解为 $s_i = \{c_1, c_2, \dots, c_j, \dots, c_k, \dots, c_n\}$, 则交换后解为 $s_{i'} = \{c_1, c_2, \dots, c_k, \dots, c_j, \dots, c_n\}$ 。

(5) 损失函数

目标函数差即为变换前的解和变化后的解的损失函数之差:

$$\Delta c = \text{cost}(s_{i'}) - \text{cost}(s_i)$$

(6) Metropolis 接受准则

$$\text{接受当前解的概率为 } p = \begin{cases} 1, & \Delta c > 0 \\ \exp(-\Delta c / t), & \Delta c < 0 \end{cases}$$

其中 t 用于控制退火速度, 其迭代规则为 $t := a * t$, $0 < a < 1$

实验结果

本实验使用一组随机创建的坐标数据对的列表作为每个城市的坐标, 形如:

coordinates = [[x1, y1], [x2, y2], ...,]。

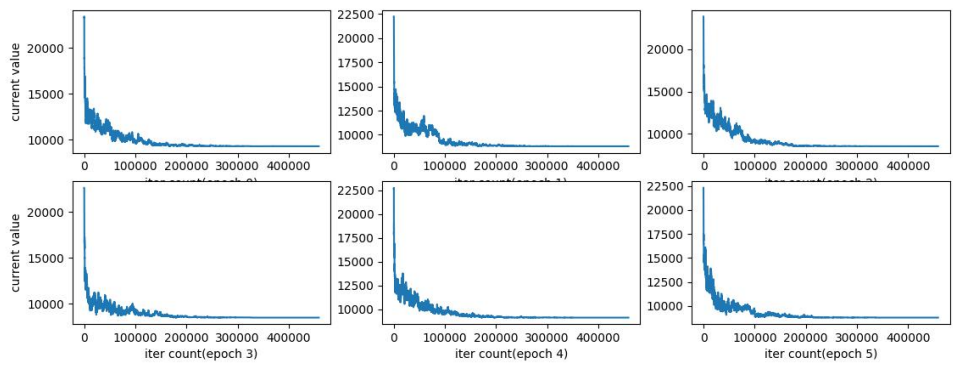


图 1 算法各次求最短距离变化趋势

图 1 显示了跑了 6 次模拟退火算法的旅行商最短距离随迭代次数变化的趋势。可以看出所求最短距离都是不断减少，符合算法预期，即模拟退火算法是可以求解旅行商问题的。

表 1 算法各次求得的最短距离

Epoch	1	2	3	4	5	6
最短距离	9268.03	8816.41	8521.08	8472.09	9136.44	8785.92

表 1 显示了算法每次所求的最短距离都不一样，说明模拟退火算法并不能求得精确解，只能逼近最优解。

附代码

```
import numpy as np
import os
import matplotlib.pyplot as plt

# 获取点坐标数据
def get_coordinates(file_path: str) -> np.ndarray:
    coordinates = np.array([
        [565.0,575.0],[25.0,185.0],[345.0,750.0],[945.0,685.0],[845.0,655.0],
        [880.0,660.0],[25.0,230.0],[525.0,1000.0],[580.0,1175.0],[650.0,1130.0],
        [1605.0,620.0],[1220.0,580.0],[1465.0,200.0],[1530.0, 5.0],[845.0,680.0],
        [725.0,370.0],[145.0,665.0],[415.0,635.0],[510.0,875.0],[560.0,365.0],
        [300.0,465.0],[520.0,585.0],[480.0,415.0],[835.0,625.0],[975.0,580.0],
        [1215.0,245.0],[1320.0,315.0],[1250.0,400.0],[660.0,180.0],[410.0,250.0],
        [420.0,555.0],[575.0,665.0],[1150.0,1160.0],[700.0,580.0],[685.0,595.0],
        [685.0,610.0],[770.0,610.0],[795.0,645.0],[720.0,635.0],[760.0,650.0],
        [475.0,960.0],[95.0,260.0],[875.0,920.0],[700.0,500.0],[555.0,815.0],
        [830.0,485.0],[1170.0, 65.0],[830.0,610.0],[605.0,625.0],[595.0,360.0],
        [1340.0,725.0],[1740.0,245.0]])

    return coordinates
```

计算点与点之间（城市之间）距离矩阵

```
def getdismat(coordinates: np.ndarray) -> np.ndarray:
```

```
    num = coordinates.shape[0]
```

```
    distmat = np.zeros((num, num))
```

```
    for i in range(num):
```

```
        place_i = coordinates[i]
```

```
        for j in range(num):
```

```
            place_j = coordinates[j]
```

```
            distmat[i][j] = np.sqrt(np.power(place_i[0] - place_j[0], 2) + np.power(place_i[1] - place_j[1],
```

```
2))
```

```
    return distmat
```

进行模拟退火过程

```
def sa_run():
```

```
    coordinates = get_coordinates('./a280.tsp')
```

```
    num = coordinates.shape[0]
```

```
    dist_mat = getdismat(coordinates)
```

```
    solution_new = np.arange(num)
```

```
    solution_current = solution_new.copy()
```

```
    solution_best = solution_new.copy()
```

```
    value_current = 9999999
```

```
    value_best = 9999999
```

```
    alpha = 0.99
```

```
    t_range = (1, 100)
```

```
    markovlen = 1000
```

```
    t = t_range[1]
```

```
    epochcount = 6
```

```
    epoch_best = []
```

```
    epoch_current = []
```

```
    for epoch in range(epochcount):
```

```
        result_best = [] # 记录迭代过程中的最优解
```

```
        result_current = []
```

```
        while t > t_range[0]:
```

```
            for i in range(markovlen):
```

```
                # 使用将两个坐标逆序的方式产生新解
```

```
                while True:
```

```
                    loc1 = int(np.ceil(np.random.rand() * (num - 1)))
```

```
                    loc2 = int(np.ceil(np.random.rand() * (num - 1)))
```

```
                    if loc1 != loc2:
```

```
                        break
```

```
                    solution_new[loc1], solution_new[loc2] = solution_new[loc2], solution_new[loc1]
```

```
    value_new = 0
```

```

for j in range(num - 1):
    value_new += dist_mat[solution_new[j]][solution_new[j + 1]]
value_new += dist_mat[solution_new[0]][solution_new[num - 1]]
if value_new < value_current:
    # 接受该解
    # print('accept1')
    value_current = value_new
    solution_current = solution_new.copy()

    if value_new < value_best:
        value_best = value_new
        solution_best = solution_new.copy()
else:
    # 以一定概率接受该解
    if np.random.rand() < np.exp(-(value_new - value_current) / t):
        # print('accept2')
        value_current = value_new
        solution_current = solution_new.copy()
    else:
        # print('not accept')
        solution_new = solution_current.copy()

result_current.append(value_current)

t = alpha * t
result_best.append(value_best)
print(f't: {t}, value: {value_current}')

epoch_current.append(result_current)
epoch_best.append(result_best)

print(f'epoch {epoch} finish!!!!!!!!')
solution_new = np.arange(num)
solution_current = solution_new.copy()
solution_best = solution_new.copy()
value_current = 9999999
value_best = 9999999
t = t_range[1]

# print(f'best values: {value_best}')
# print(f'best solution: {solution_best}')

fig_col = 3 # 每行多少个子图
fig_row = epochcount // fig_col

```

```
fig = plt.figure()
ax = fig.subplots(fig_row, fig_col)
for r in range(fig_row):
    for c in range(fig_col):
        index = r * fig_col + c
        if c == 0:
            ax[r, c].set_ylabel('current value')
            ax[r, c].set_xlabel(f'iter count(epoch {index})')
            ax[r, c].plot(np.array(epoch_current[index]))
```

```
plt.show()
```

```
for i in range(len(epoch_best)):
    print(f'best value {i}: {epoch_best[i][-1]}')
```

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