《算法设计与分析》课程实验报告

搜索算法实验

黎昱彤

学号：2022211414

班级：2022211312

计算机科学与技术

目录

1 实验介绍

1.1 实验任务

问题：

旅行商问题（Traveling Salesman Problem, TSP）是给定一系列城市及其之间的距离，要求找到一条最短路径，使得旅行商从某个城市出发，经过每个城市恰好一次并返回到出发点的城市。

该问题是一个 NP-困难的问题，目前没有已知的多项式时间算法可以解决所有问题实例。随着城市数量的增加，所有可能的路径组合会呈指数增长。例如，对于 n 个城市，可能的路径数为(n-1)!/2。

因此，对于较小的城市数（通常 $n \leq 20$），可以使用暴力搜索或动态规划方法；对于较大的城市数，常用启发式或近似算法。

基本工作：给出10个城市及其相互之间距离，找出其最短路径。

1) 用回溯法实现旅行商问题求解算法；
2) 用动态规划法的算法验证其正确性。

通过比较说明相关算法的特点，可能的改进（如果有，再实现加以验证）

可选扩展：

城市数目增加到 50 个以上，随机生成其邻接矩阵，用其他的一些算法找出其近似的最短路径。

本实验完成了基本工作和可选扩展。

1.2 实验原理

TSP 问题所求的路径是一个环，我们可以从任意的点出发去查找路径，旅行商问题只有当图是哈密顿图时才可能有解的。

回溯法通过逐步试错的方式来搜索解空间来求解。解决组合优化问题时，它通过穷举所有可能的路径，然后选择最短的路径。

动态规划法的核心思想是根据之前计算的结果来计算当前的最短路径长度，逐步构建出整个 dp 数组。最后通过查找 dp 数组中的最短路径来找到全局最优解。

扩展中，城市数目增加到 50 个以上，复杂度急剧上升。使用了其他的一些算法：启发式算法是一类在解决复杂问题时利用经验规则和启发式信息进行搜索的算法。与精确算法不同，启发式算法不保证找到最优解，但在很多情况下能找到一个较好的解，且计算效率较高。

本实验中选择的是**模拟退火算法**(Simulated Annealing，SA)。模拟退火算法模拟物理退火过程，从某一较高初温出发，随着温度的不断下降，以一定概率突跳在全局进行寻优，并最终趋于全局最优，搜索过程中趋于 0 概率的突跳特性可有效避免算法陷入局部最优。模拟退火算法依赖现有求解规则，是一种对已有规则进行改造的算法，它的解与初始值无关；其核心思想是以 1 概率接受较优解，以较小概率接受裂解（Metropolis 准则）

1.3 实验环境

Windows11

Visual Studio Code

Python

2 算法设计

2.1 回溯法求解

- 利用递归生成所有可能的路径，检查路径的代价并更新最优路径；
- 使用剪枝策略避免探索不可能的路径；

• 当所有城市都被访问过时，计算当前路径的总费用，并根据最优值更新结果。

伪代码如算法1所示。

Algorithm1 回溯法

```
Function Backtrack (current_city, depth, current_cost)
    if depth == TOTAL_CITIES:
        total_cost = current_cost + DISTANCE(current_city, START_CITY)
        if total_cost < BEST_COST:
            BEST_COST = total_cost
            BEST_PATH = CURRENT_PATH.copy()
        return
    for next_city in ALL_CITIES:
        if next_city is not visited:
            if current_cost + DISTANCE(current_city, next_city) >= BEST_COST:
                continue
            mark next_city as visited
            add next_city to CURRENT_PATH
            BACKTRACK(next_city, depth + 1, current_cost + DISTANCE(current_city, next_city))
            remove next_city from CURRENT_PATH
            unmark next_city as visited
End Function
Function SOLVE_TSP_BACKTRACK()
    initialize BEST_COST as infinity
    initialize BEST_PATH as empty
```

mark START_CITY as visited

add START_CITY to CURRENT_PATH

BACKTRACK(START_CITY, 1, 0)

return BEST_COST, BEST_PATH

End Function

2.2 动态规划法验证

• 状态表示：用位掩码 mask 表示城市访问状态；

•状态转移：从已访问城市出发，考虑每个未访问的城市，并更新相应的最小费用；

• 最终结果：通过最后状态(1 << n) - 1来遍历所有的城市，计算回到起点的最短路径费用。

伪代码如算法2所示。

Algorithm2 动态规划法

Function DP()

initialize DP_TABLE as a 2D array of size [2^TOTAL_CITIES][TOTAL_CITIES] with infinity

DP_TABLE[1][START_CITY] = 0

for each mask from 0 to (2^TOTAL_CITIES - 1):

for each current_city in ALL_CITIES:

if mask includes current_city:

for each next_city in ALL_CITIES:

if mask does not include next_city:

new_mask = mask | (1 << next_city)

DP_TABLE[new_mask][next_city] = min(

```
                DP_TABLE[new_mask][next_city],
                DP_TABLE[mask][current_city] + DISTANCE(current_city, next_city)
            )
    BEST_COST = infinity
    for each end_city in ALL_CITIES except START_CITY:
        BEST_COST = min(
            BEST_COST,
            DP_TABLE[(2^TOTAL_CITIES - 1)][end_city] + DISTANCE(end_city, START_CITY)
        )
    return BEST_COST
```

End Function

3 实验结果及分析

3.1 实验结果

10 个城市的结果如下：

```
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp.py
城市邻接矩阵
[0, 78, 49, 5, 85, 94, 75, 43, 92, 23]
[78, 0, 69, 66, 60, 33, 92, 8, 31, 41]
[49, 69, 0, 15, 21, 39, 29, 77, 25, 43]
[5, 66, 15, 0, 29, 40, 6, 68, 19, 13]
[85, 60, 21, 29, 0, 88, 98, 58, 41, 82]
[94, 33, 39, 40, 88, 0, 56, 82, 31, 53]
[75, 92, 29, 6, 98, 56, 0, 99, 3, 88]
[43, 8, 77, 68, 58, 82, 99, 0, 5, 79]
[92, 31, 25, 19, 41, 31, 3, 5, 0, 69]
[23, 41, 43, 13, 82, 53, 88, 79, 69, 0]
回溯法求解
最短路径长度: 209
最短路径:
[0, 3, 4, 2, 6, 8, 7, 1, 5, 9, 0]
[0, 9, 5, 1, 7, 8, 6, 2, 4, 3, 0]
动态规划求解
最短路径长度: 209
```

为简化答案，回溯法的输出是以城市 0 为起点的最短路径的所有解。动态规划法求得最短路径长度和回溯法求解结果一致，验证了正确性。

3.2 实验分析

1　回溯法 (Backtrack)

时间复杂度：$O(n \times n!)$

对于 n 个城市，有 n! 种路径，每次路径检查涉及 O(n)的邻接矩阵访问

空间复杂度：O(n)

sol 记录解和 visited 访问标记，栈递归深度为 n

2　动态规划法 (DP)

时间复杂度：$O(2^n \times n)$

有 $2^n \times n$ 个状态，每个状态的转移需要 O(n)

空间复杂度：$O(2^n \times n)$

状态压缩使用 $O(n \times 2^n)$的 dp 数组记录子问题解

对于更大的 n，回溯法复杂度增长更快，难以处理；动态规划法通过状态压缩和子问题重用，大幅减少了重复计算，但仍然需要访问解空间的每个状态以确保最优解，会受到状态空间大小的限制。

4 可能的改进

1.　回溯法改进

1.1　剪枝

如果当前费用已经超过当前最优值，则不继续搜索。

```
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp_adv.py
回溯法求解
最短路径长度: 129
运行时间: 6.063s
改进-回溯法求解
最短路径长度: 129
运行时间: 0.0165s
```

由于避免了不必要的递归，运行时间显著缩短。

1.2 预处理

使用贪心算法每次选择当前城市到未访问城市中距离最短的城市，预处理矩阵用贪心算法得到的的近似结果初始化 best_cost，从而减少初期的冗余搜索。

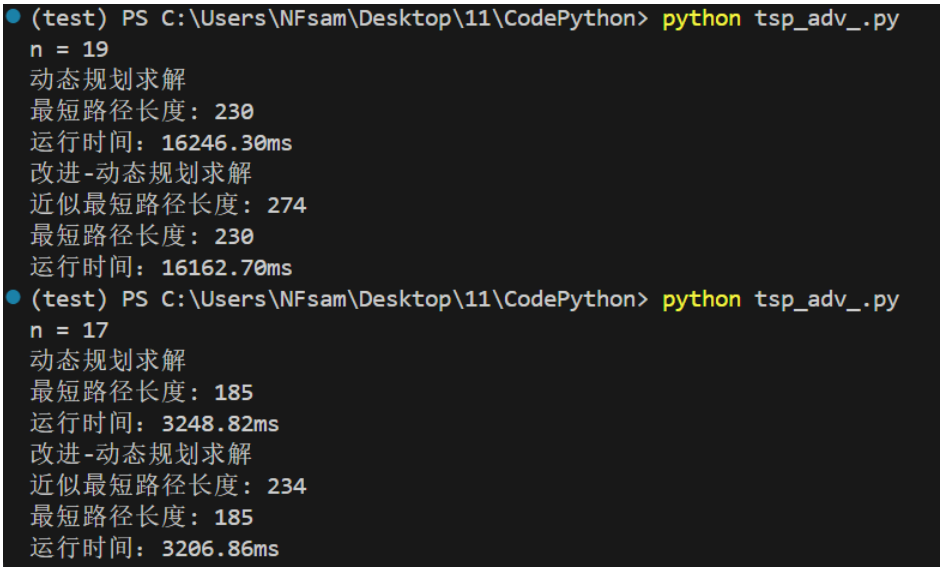
```
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp_adv.py
n = 15
回溯法求解
最短路径长度: 157
运行时间: 8124.33ms
改进-回溯法求解
近似最短路径长度: 222
最短路径长度: 157
运行时间: 7614.42ms
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp_adv.py
n = 13
回溯法求解
最短路径长度: 191
运行时间: 1499.31ms
改进-回溯法求解
近似最短路径长度: 304
最短路径长度: 191
运行时间: 1496.39ms
```

此改进在上一个剪枝基础上完成，改进-回溯法只有预处理，可见运行时间缩短了。贪心算法的解可能偏离最优解，但其时间复杂度 $O(n^2)$ 在大规模图上仍然具有非常快的计算速度。

2. 动态规划法改进

2.1 预处理

使用贪心算法预处理同回溯法。验证：



5 实验扩展

模拟退火通过随机搜索加收敛机制，以概率性替代了完全遍历，不保证全局最优解，但能以较低时间成本找到近似解。

模拟退火算法求解步骤：

1 贪心算法构造初始解作为当前解（SA对初始解不敏感，但贪婪算法构造的初始解能够提高求解效率）；
2 当前解生成邻域解，选择最优解作为当代最优解；
3 当代最优解优于历史最优解，则更新历史最优解和当前解，否则以一定概率接受当代最优解作为当前解；
4 执行退火操作，判断是否满足结束条件，满足则退出迭代，否则继续执行步骤2-3。

变量说明：

1 初始温度：initial_temperature

控制算法初始搜索的跳跃能力。太低会限制搜索范围，太高会增加无效尝试。

2 冷却速率：cooling_rate

决定温度降低的速度，影响全局搜索和局部搜索的平衡。

3　每温度迭代次数：iteration_per_temp

决定每个温度下的搜索深度。

小规模：

n=10，使用同一邻接矩阵，回溯法求解验证SA得到的答案并非最短，并且SA用时更长。

```
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp.py
城市邻接矩阵
[0, 77, 49, 47, 95, 56, 25, 72, 1, 68]
[77, 0, 20, 52, 27, 41, 30, 4, 89, 27]
[49, 20, 0, 68, 72, 36, 30, 40, 76, 71]
[47, 52, 68, 0, 47, 75, 25, 66, 44, 4]
[95, 27, 72, 47, 0, 85, 5, 96, 13, 54]
[56, 41, 36, 75, 85, 0, 50, 19, 87, 60]
[25, 30, 30, 25, 5, 50, 0, 52, 11, 79]
[72, 4, 40, 66, 96, 19, 52, 0, 56, 95]
[1, 89, 76, 44, 13, 87, 11, 56, 0, 51]
[68, 27, 71, 4, 54, 60, 79, 95, 51, 0]
回溯法求解
最短路径长度: 183
最短路径:
[0, 2, 5, 7, 1, 9, 3, 6, 4, 8, 0]
[0, 8, 4, 6, 3, 9, 1, 7, 5, 2, 0]
```

```
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp_SA.py
n = 10
模拟退火算法求解
近似的最短路径长度: 186
近似的最短路径: [0, 8, 4, 6, 2, 5, 7, 1, 9, 3, 0]
```

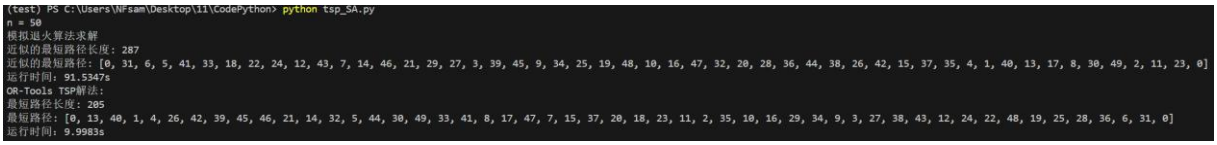
大规模：

OR-Tools（使用动态规划或其他高级算法求解TSP，速度快且接近最优解）结果和SA算法结果对比。

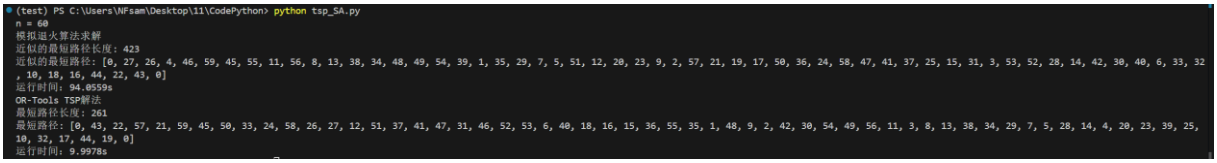
```
(test) PS C:\Users\NFsam\Desktop\11\CodePython> python tsp_SA.py
n = 50
模拟退火算法求解
近似的最短路径长度: 356
近似的最短路径: [0, 4, 18, 22, 34, 16, 25, 19, 28, 32, 46, 33, 13, 39, 5, 12, 36, 24, 8, 23, 49, 30, 7, 38, 3, 9, 35, 45, 47, 44, 48, 10, 20, 6, 11, 1, 14, 17, 42, 40, 31, 43, 26, 27, 41, 2, 37, 15, 21, 29, 0]
运行时间: 159.1525s
OR-Tools TSP解法:
最短路径长度: 228
最短路径: [0, 29, 9, 24, 33, 28, 32, 46, 49, 30, 19, 16, 34, 22, 25, 2, 41, 8, 35, 45, 47, 44, 5, 12, 36, 13, 39, 7, 38, 1, 40, 31, 43, 26, 27, 23, 6, 20, 10, 48, 17, 42, 37, 15, 21, 14, 11, 3, 18, 4, 0]
运行时间: 10.0191s
```

优化尝试：提前终止（稳定计数）；调整接受概率公式：增加温度对接受概率的影响（温度因子0.1）；自适应初始温度（max(max(G)) * n）。

n=50

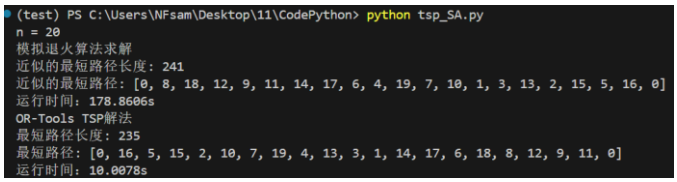


n=60

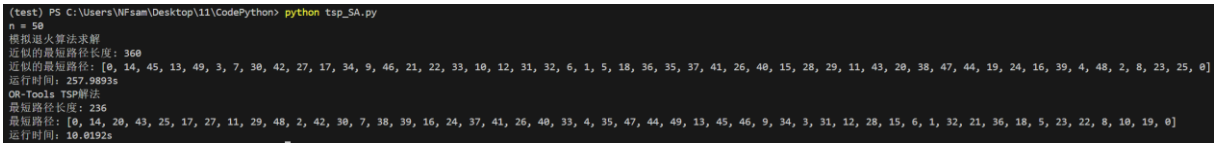


优化尝试：随机生成交换次数。

n=20



n=50



分析：

贪心算法复杂度$O(n^2)$

温度递减循环的循环次数T取决于cooling_rate和initial_temperature，循环到温度接近于零，设 k 为温度递减的次数，近似为$O(\log(T_{init}))$；每温度下的迭代次数iteration_per_temp固定设为 m ；每次迭代的邻域解是固定个解，对每个邻域解计算路径代价的复杂度为O(n)。综上，模拟退火的单次温度复杂度 $O(m\times n)$

总复杂度$O(n^2)+O(k\times m\times n)$，将 k 和 m 视为常数时，模拟退火算法的复杂度为近似线性O(n)，但若考虑 T_{init} 较大（如O(n)）时，总体复杂度可能接近$O(n^2)$

6 实验总结

本次实验旨在利用搜索算法解决TSP问题，并通过实现与测试不同算法的方法，对比其性能与效果，以加深对问题求解方法的理解。

基础工作我实现了两种适合小规模数据的TSP求解方法：回溯法和动态规划法，并做了很多优化测试，提高了求解效率。由于上述两种精确算法都遍历所有可能的解空间，在处理大规模TSP实例时计算复杂度过高，我在扩展任务中初次接触启发式算法，实现了模拟退火算法，用概率性替代完全遍历得到近似解。为了找参照评估求解结果，取Google的OR-Tools求解TSP。实验中我尝试了通过动态调整邻域解数量、引入稳定退出条件等方式，提升了算法的效率，但幅度有限。

附录

实验关键源码

```
# 随机生成城市的邻接矩阵
def generate_matrix(num_cities, max_distance):
    matrix = [[0 if i == j else random.randint(1, max_distance) for j in range(num_cities)]
for i in range(num_cities)]
    # 对称化（无向图）
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            matrix[j][i] = matrix[i][j]
    return matrix


class Traveling:
    def __init__(self, n, G):
        self.n = n                          # 图的顶点数
```

```
        self.sol = [i for i in range(n)]# 当前解
        self.best_sol = []              # 当前最优解
        self.G = G                      # 图的邻接矩阵
        self.cost = 0                   # 当前费用
        self.best_cost = float('inf')   # 当前最优值
        self.visited = [False] * n      # 访问标记

    # 回溯法
    def Backtrack(self, i):
        if i == self.n: # 当所有顶点都被访问完毕
            cost = self.cost + self.G[self.sol[-1]][self.sol[0]] # 回到起点的费用
            if cost < self.best_cost: # 更新最优值
                self.best_cost = cost
                self.best_sol = [self.sol[:]]
            elif cost == self.best_cost: # 添加解
                self.best_sol.append(self.sol[:])
        else:
            for j in range(self.n): # 遍历所有顶点
                if not self.visited[j]: # 判断是否可行
                    # 剪枝
                    if i > 0 and self.cost + self.G[self.sol[i - 1]][j] > self.best_cost:
                        continue
                    self.visited[j] = True
                    self.sol[i] = j
                    if i == 0:
                        self.cost = 0
                    else:
                        self.cost += self.G[self.sol[i - 1]][self.sol[i]]
                    self.Backtrack(i + 1) # 递归进入下一层
```

```
                    if i > 0:
                        self.cost -= self.G[self.sol[i - 1]][self.sol[i]]
                    self.visited[j] = False

    # 动态规划法
    def DP(self):
        # dp[mask][i] 表示访问过状态 mask 并且当前在城市 i 时的最小费用
        dp = [[float('inf')] * self.n for _ in range(1 << self.n)]
        dp[1][0] = 0 # 起点城市到自己的费用为 0

        for mask in range(1 << self.n):
            for i in range(self.n):
                if mask & (1 << i): # 如果城市 i 在状态 mask 中被访问过
                    for j in range(self.n):
                        if mask & (1 << j) == 0: # 如果城市 j 没被访问过
                            new_mask = mask | (1 << j)
                            dp[new_mask][j] = min(dp[new_mask][j], dp[mask][i] +
self.G[i][j])

        min_cost = float('inf')
        for i in range(1, self.n):
            min_cost = min(min_cost, dp[(1 << self.n) - 1][i] + self.G[i][0])
        # 更新最优值
        self.best_cost = min_cost

# 贪心算法
def greedy_approximation(G, n):
    visited = [False] * n
    current_city = 0
```

```
    visited[current_city] = True
    total_cost = 0
    path = [current_city]

    for _ in range(n - 1):
        next_city = None
        min_cost = float('inf')
        for j in range(n):
            if not visited[j] and G[current_city][j] < min_cost:
                min_cost = G[current_city][j]
                next_city = j
        visited[next_city] = True
        total_cost += min_cost
        current_city = next_city
        path.append(current_city)

    total_cost += G[current_city][path[0]]
    path.append(path[0])
    return total_cost, path

# 模拟退火算法
def simulated_annealing(G, n, initial_temperature, cooling_rate, iteration_per_temp):
    # 初始解: 贪心算法
    _, path = greedy_approximation(G, n)
    cost = calculate_cost(G, path)

    best_path = path[:]
    best_cost = cost
```

```
    temperature = initial_temperature
    stable_iterations = 0 # 稳定计数

    while temperature > 1e-3: # 终止条件: 温度接近于零
        update = False
        for _ in range(iteration_per_temp):
            neighborhood = [] # 邻域
            for _ in range(100):
                new_path = path[:]
                i, j = random.sample(range(1, n), 2) # 保证起点不参与交换
                new_path[i], new_path[j] = new_path[j], new_path[i]
                neighborhood.append((calculate_cost(G, new_path), new_path))
            # 选择最优解作为当代最优解
            neighborhood.sort(key=lambda x: x[0])
            best_local_cost, best_local_path = neighborhood[0]
            # 判断是否接受当代最优解
            if best_local_cost < cost or random.random() < math.exp(-(best_local_cost -
cost) / (temperature * 0.1)):
                path = best_local_path
                cost = best_local_cost
                update = True
                # 更新全局最优解
                if cost < best_cost:
                    best_path = path[:]
                    best_cost = cost
        if not update:
            stable_iterations += 1
            if stable_iterations >= 10:
                break
```

```
        else:
            stable_iterations = 0
        # 温度递减
        temperature *= cooling_rate

    return best_cost, best_path
```

注：tsp.py 基础工作（回溯法+动态规划法）

tsp_adv.py 回溯法改进　tsp_adv_.py 动态规划法改进

tsp_SA.py 模拟退火算法