旅行商问题的若干解法及其分析

俞沣城 *，罗健文 †

（智能工程学院 2021 级智能科学与技术）

摘要：旅行商问题（TSP）是组合优化领域中的经典问题，广泛应用于物流、制造和交通等多个领域。本文探讨了三种求解 TSP 的方法：Hopfield 神经网络、动态规划（Held-Karp 算法）和遗传算法。通过对比分析这三种方法的计算复杂度、解的质量以及实现复杂度，揭示了各自的优势与局限。动态规划能够在小规模问题中提供精确解，但其计算复杂度随着问题规模的增加迅速上升，限制了其应用范围。Hopfield 神经网络在中小规模问题中表现良好，能够找到近似最优解，但实现和参数调节较为复杂，且容易陷入局部最优。遗传算法则展现出优越的扩展性和适应性，适用于大规模和多变种的 TSP 问题，能够在合理时间内提供高质量的近似解。最后，本文总结了在不同应用场景下选择合适求解方法的依据，为实际问题的解决提供了参考。

关键词：旅行商问题，Hopfield 神经网络，动态规划，遗传算法，优化算法

作业完成情况

- 按照要求采用 Hopfield 网络[1]实现 30 个城市路径的 TSP 问题优化, 并进行了 MATLAB 仿真, 见 section 2。
- 使用动态规划[2-3]算法求解 TSP, 并进行仿真, 见 section 3。
- 使用遗传算法[4-5]求解 TSP, 并进行仿真, 见 section 4。
- 分析对比三种方法, 见 section 5。
- 整理源代码和仿真结果并进行打包, 同时源代码也可见补充材料。

1 旅行商问题

1.1 什么是旅行商问题

旅行商问题（Traveling Salesman Problem，简称 TSP[6]）是运筹学和计算机科学中的经典问题，示例图如 fig. 1 所示，这个问题的经典描述为: 给定一系列城市以及它们之间的距离（或路径花费），要求找出一条从某个城市出发，访问所有城市且最终回到出发城市的最短路径（或最小花费路线）。

输入.

- n 个城市
- 这些城市之间的距离或代价矩阵（一般假设满足三角不等式，但并不一定都满足）

输出.

- 一条访问所有城市且只访问一次的封闭路线（哈密顿回路），使得总旅行距离（或总花费）最小。
- 最优路线的总距离/总花费。

问题类别.

- 旅行商问题被证明是一个 NP-困难（NP-hard）问题，且寻找其最优解是 NP-完全（NP-complete）性质的一个典型代表。这意味着当城市数量 n 很大时，想要在多项式时间内找到最优解是极其困难的（甚至认为是不可能的）。

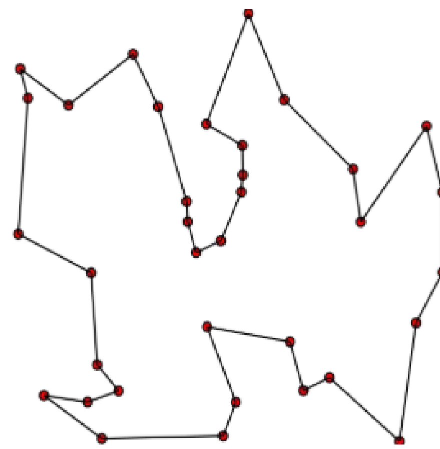

Fig. 1 旅行商问题求解结果演示

特点与挑战.

- 复杂度高: 当城市数目很少时，可以用穷举或动态规划等精确方法在合理的时间内求解; 随着城市数量增多，求解难度呈指数级上升。
- 应用范围广: 并不局限于“旅行”场景，还可以应用在诸如物流配送、印刷电路板钻孔、基因测序等需要路径优化的领域。
- 数学研究价值: 旅行商问题常被用作研究组合优化算法的“试金石”，新提出的各种算法、启发式方法常用于求解 TSP 来验证性能。

1.2 旅行商问题的基本分类

- 经典 TSP: CTSP 是在一个带权无向完全图中找一个权值最小的 Hamilton 回路。在各类 TSP 中，该类问题的研究成果最多。近几年来，研究者或者基于数学理论构造近似算法，或者使用各种仿自然的算法框架结合不同的局部搜索方法构造混合算法。
- 不对称 TSP: 若在 CTSP 模型中，两个顶点 i 和 j 间的距离 d 不一定相等，则称为 ATSP。ATSP 由于两点间

距离的不对称性，所以求解更困难，但由于现实生活中多数实际场景都为不对称的 TSP，所以对于基于实际交通网络的物流配送来说，其比 CTSP 更具有实际应用价值。

- 配送收集 TSP: TSPPD 是由 CTSP 适应物流配送领域的实际需求而产生的。这个问题涉及到两类顾客需要：一类是配送需求，要求将货物从配送中心送到需求点；另一类是收集需求，要求将货物从需求点运往配送中心。当所有的配送和收集需求都由一辆从配送中心出发、限定容量的车辆来完成时，怎样安排行驶路线才能构成一条行程最短的 Hamilton 回路。
- 多人旅行商问题: 即多个旅行商遍历多个城市，在满足每个城市被一个旅行商经过一次的前提下，求遍历全部城市的最短路径。解决 MTSP 对解决“车辆调度路径安排”问题具有重要意义。过去的研究大多将 MTSP 转化成多个 TSP，再使用求解 TSP 的算法进行求解。Hong Qu 等人结合胜者全取 (winner—take—all) 的竞争机制设计了一个柱形竞争的神经网络模型来求解 MTSP，并对网络收敛于可行解进行了分析和论证。
- 多目标旅行商问题: CRISP 的路径上只有一个权值 (即距离)，而 MoTSP 研究的是路径上有多个权值的 TSP，要求找一条通过所有顶点并最终回到起点的回路，使回路上的各个权值都尽可能小。由于在多目标情况下，严格最优解并不存在，研究 MoTSP 的目的是找到 Pareto 最优解，这是一个解集，而不是一个单一解。现阶段算法为构造一个求解单目标的遗传局部搜索算法，然后基于此求解多目标组合优化问题算法。

1.3 旅行商问题的常见求解方法

精确算法. 动态规划（Held-Karp 算法）[2-3], 时间复杂度大约为 $O(n^2 \cdot 2^n)$，仅能在城市数量较小（几十个以内）的情况下求解。**分支定界**（Branch and Bound）/**分枝剪枝**, 可以通过剪枝来减少搜索空间，但最坏情况下仍是指数级复杂度。

启发式算法（Heuristics）[7-9]. 这些方法并不保证一定能得到最优解，但通常能在较短时间内得到一个“近似最优解”。常见方法包括：**最近邻算法**（Nearest Neighbor）[10], 从任意城市出发，每次选择尚未访问且距离最近的城市。**贪心插入算法**（Greedy Insertion）, 不断将还未访问的城市插入到当前回路中最合适的位置。2-opt、3-opt **等邻域搜索**, 基于局部搜索思想，通过对现有路线做局部调整（例如交换两段路径），不断改进解。

元启发式算法（Metaheuristics）. 常见的如模拟退火 (Simulated Annealing)[11]、遗传算法 (Genetic Algorithm)[4-5]、蚁群算法 (Ant Colony Optimization)[12-13]、粒子群优化 (Particle Swarm Optimization)[14] 等。这些方法在大规模 TSP 实例中表现出良好的求解效率和可扩展性。

2 Hopfield 网络[1]求解旅行商问题

2.1 Hopfield 问题求解原理

网络结构. 在 Hopfield 神经网络中，每个神经元均与其他神经元相互连接，且网络具有一个能量函数 E，通过迭代更新神经元状态来最小化该能量函数，从而得到与之对应的最优解。

对于 TSP 问题，如果有 N 个城市，则可以构建一个 $N \times N$ 的神经网络矩阵 V，其中：

- 行索引 i 对应城市编号（$i = 1, 2, \ldots, N$）。
- 列索引 j 对应访问顺序（$j = 1, 2, \ldots, N$）。

因此:

- V_{ij} 表示城市 i 在第 j 步时是否被访问。理想情况下，$V_{ij} = 1$ 或 0。
- 每一行只允许有且仅有一个“1”（因为每个城市只能被访问一次）。
- 每一列只允许有且仅有一个“1”（因为在访问序列中，一个城市位置只能放一个城市）。

能量函数. 为强制网络逼近 TSP 最优解，需要在能量函数中加入不同约束的惩罚项：行约束, 即每个城市只被访问一次, 计算公式如 eq. (1) 所示。

$$E_1 = \frac{A}{2} \sum_{i=1}^{N} \left(\sum_{j=1}^{N} V_{ij} - 1 \right)^2 \tag{1}$$

其中 A 是惩罚系数，若某一行（某城市）访问次数越多或越少于 1，能量会增加。

列约束：在访问顺序中，每个位置只能放一个城市, 如 eq. (2) 所示。

$$E_2 = \frac{A}{2} \sum_{j=1}^{N} \left(\sum_{i=1}^{N} V_{ij} - 1 \right)^2 \tag{2}$$

同样，若某列（某个访问顺位）出现多于或少于一个城市，能量会增大。

距离约束：路径总距离越小越好, 如 eq. (3) 所示。

$$E_3 = \frac{D}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} d_{i,k} V_{i,j} V_{k,j+1} \tag{3}$$

这里 $d_{i,k}$ 是城市 i 与城市 k 之间的距离；$j+1$ 表示下一个访问位置（需适当循环取模形成回路）。D 为距离惩罚系数。

最终能量函数为 eq. (4) 所示。

$$\begin{gathered} E = E_1 + E_2 + E_3 = \\ \frac{A}{2} \sum_{i=1}^{N} \left(\sum_{j=1}^{N} V_{ij} - 1 \right)^2 + \\ \frac{A}{2} \sum_{j=1}^{N} \left(\sum_{i=1}^{N} V_{ij} - 1 \right)^2 + \\ \frac{D}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} d_{i,k} V_{i,j} V_{k,j+1}. \end{gathered} \tag{4}$$

状态更新方程. Hopfield 网络通过动力学方程迭代更新神经元输入（记为 U），输出（记为 V）通过激活函数, 如公式 eq. (5) 所示。

$$V_{ij} = \frac{1 + \tanh\left(\frac{U_{ij}}{U_0}\right)}{2} \tag{5}$$

其中 U_0 是一个缩放参数，用于控制 tanh 的陡峭程度。

网络的演化基于负梯度下降方法对 E 进行迭代，得到对 U_{ij} 的微分形式, 如 eq. (6) 所示。

$$\frac{dU_{ij}}{dt} = -\frac{\partial E}{\partial V_{ij}} \tag{6}$$

在代码实现中，我们离散化时间, 如 eq. (7) 所示。

$$U_{ij}(t+1) = U_{ij}(t) + \Delta t \cdot \left(-\frac{\partial E}{\partial V_{ij}}\right) \tag{7}$$

其中 Δt 为步长。

2.2 MATLAB 代码仿真

在本章节, 我会解释 MATLAB 仿真中重要的代码部分. 首先, 随机生成 N (本次实验中为 30) 个城市的坐标, 范围在 $[0, 1]$ 内, 然后利用 `dist` 函数, 计算所有城市之间两两距离矩阵 `distance` (大小 30×30)。

```
citys = rand(30, 2);
distance = dist(citys, citys');
```

然后初始化 TSP 与网络的核心参数。

```
N = size(citys, 1);    % N = 30
A = 200;               % 行列约束的惩罚系数
D = 100;               % 路径距离约束的惩罚系数
U0 = 0.1;              % 激活函数的缩放参数
step = 0.0001;         % 步长 (Delta t)
```

其中, A 和 D 分别控制行列约束、路径约束在能量函数中的权重。

然后, 随机初始化网络输入状态 U，并把它映射到输出 V 上，范围在 $(0, 1)$ 之间。后续会不断迭代更新。

```
delta = 2*rand(N, N) - 1; % 在 (-1,1) 之间随机初始化
U = U0 * log(N-1) + delta;% 初始神经元输入状态 U
V = (1 + tansig(U/U0))/2; % 初始神经元输出状态 V
```

核心迭代循环如下所示。

```
for k = 1:iter_num
    dU = diff_u(V, distance);   % 计算 dU, 即 Hopfield 网络的状态更新量
    U = U + dU*step;            % 更新输入神经元状态 U
    V = (1 + tansig(U/U0))/2;   % 通过双曲正切函数更新输出神经元状态 V
    e = energy(V,distance);     % 计算当前网络的能量函数值
    E(k) = e;                   % 记录本次迭代的能量函数值
end
```

在核心循环中, 每一轮都会进行以下的步骤:

- 调用 $diff_u(V, distance)$ 计算 $\frac{\partial E}{\partial V}$，得到对 U 的变化量 dU（由于 $dU = -\partial E/\partial V$）。
- 更新 U: $U \leftarrow U +$ step $\times\, dU$
- 更新 V 通过激活函数：$V_{ij} \leftarrow \frac{1+\tanh\left(\frac{U_{ij}}{U_0}\right)}{2}$
- 计算当前能量值 `e = energy(V, distance)` 并保存到数组 E 中

核心循环后, 即可以绘制结果 (如果存在合法的路径), 详细可见补充材料。

`is_valid` 函数详解

```
function [flag, V1] = is_valid(V, N)
% V 是最终的输出神经元矩阵 (N×N), 理想情况是:
%   - 在每一列 (对应城市在第几步访问), 只有一个输出为1
%   - 在每一行 (对应第 i 个城市), 也只有一个输出为1
[rows,cols] = size(V);
V1 = zeros(rows,cols);  % 先构造一个同样规模的 0 矩阵
[~,V_ind] = max(V);     % 对每一列找到最大值 (即最可能选中的神经元下标)
for j = 1:cols
    V1(V_ind(j),j) = 1; % 将 V1 对应位置赋值为 1, 构建确定解
end
C = sum(V1,1); % 每一列的求和, 判断是否都为1
R = sum(V1,2);  % 每一行的求和, 判断是否都为1
flag = isequal(C,ones(1,N)) & isequal(R',ones(1,N));
% 若每一列、每一行都是 1, 则说明是有效解
end
```

首先, 先对每一列找出最大值所在的行索引 V_ind，将对应位置设置为 1（得到一个二值矩阵 V1）。然后, 检查每列、每行是否都有且只有一个元素为 1, 如 eq. (8) 所示。

$$\begin{aligned} \textstyle\sum_{i=1}^{N} V_{ij} = 1 \quad (j) \\ \textstyle\sum_{j=1}^{N} V_{ij} = 1 \quad (i) \end{aligned} \tag{8}$$

若都成立则 $flag = 1$，否则为 0。

`diff_u(V, d)` 函数详解

```
function du=diff_u(V,d)
global A D;
% V 大小为 N×N, d 为距离矩阵
n=size(V,1);
% 1) sum(V,2) - 1 表示每一行之和距离 1 的偏差 (V 满足行约束)
%    sum_x 就是将这个偏差复制到 n×n 矩阵
sum_x=repmat(sum(V,2)-1,1,n);
% 2) sum(V,1) - 1 表示每一列之和距离 1 的偏差 (V 满足列约束)
%    sum_i 就是将这个偏差复制到 n×n 矩阵
sum_i=repmat(sum(V,1)-1,n,1);
% 3) 处理路径距离约束: 将输出矩阵 V 右移 1 列做比较
V_temp=V(:,2:n);
V_temp=[V_temp V(:,1)];
% 4) 求 d * V_temp, 即计算距离矩阵与下一列神经元对应元素相乘的加权和
sum_d=d*V_temp;
% 5) dU 即 Hopfield 神经网络的状态更新方程
du=-A*sum_x-A*sum_i-D*sum_d;
```

```
%    注意这里的负号是因为梯度下降法，对于能量函
     数 E, dU = -∂E/∂V
end
```

功能：根据能量函数对 V 求偏导，从而得到状态更新量 dU 。这里的 sum_x、sum_i 分别对应前面提到的行、列约束的偏差；而 sum_d 计算了路径距离约束相关的偏差, 最后再进行合并。

energy(V, d) 函数详解

```
function E=energy(V,d)
global A D
n=size(V,1);
% 1) sum(V,2) - 1 表示每行之和与 1 的偏差, 用
    sumsqr() 计算误差平方和
sum_x=sumsqr(sum(V,2)-1);
% 2) sum(V,1) - 1 表示每列之和与 1 的偏差
sum_i=sumsqr(sum(V,1)-1);
% 3) 处理路径距离约束: 同上, 将 V 右移 1 列形成
    环
V_temp=V(:,2:n);
V_temp=[V_temp V(:,1)];
sum_d=d*V_temp; % d 为距离矩阵
% 对应能量中路径部分
sum_d=sum(sum(V.*sum_d)); % V.*(d * V_temp) 并求
    和
% 4) 总能量函数为 0.5(A * sum_x + A * sum_i + D
    * sum_d)
%    其中系数 0.5 是因为 E = 1/2 * ∑(...), 可避
    免重复计算
E=0.5*(A*sum_x+A*sum_i+D*sum_d);
end
```

功能：计算当前网络输出 V 对应的总能量值，从行、列、距离三部分累加而来（附加上 0.5 是为了对冲求导时的重复系数）。

2.3 为什么 Hopfield 网络能够趋于稳定

Hopfield 网络之所以会收敛到一个稳定状态，可以用“负梯度下降”和“能量函数”这两个关键词来理解：

能量函数. 在 Hopfield 网络中，我们把需要优化的目标和约束（例如 TSP 的行、列、距离约束）都合并到一个能量函数 E 里。这个能量函数类似“势能”，网络的状态越接近可行且优的解，其能量就越低；相反，偏离可行解会导致能量上升。

负梯度下降. Hopfield 网络的状态更新规则是“沿着能量函数的负梯度”变化，也就是每一步都让能量降低。具体做法如 eq. (9) 所示。

$$U \leftarrow U - \eta \frac{\partial E}{\partial V}, \quad V = f(U) \tag{9}$$

其中 U 是网络神经元的输入，V 是神经元输出，η 是步长。更新后，能量不会上升，只会维持或下降。

收敛到稳定态. 不断迭代后，能量会趋于某个极小值，此时网络的状态不再显著变化，也就是到达了“稳定点”。因为能量无法再继续下降，网络就停留在此状态不动了。这种稳定态通常对应网络获得的一个可行解（在 TSP 上即是符合行列唯一性并尽量缩短距离的路径）。

2.4 仿真结果

本次实验, 在要求 30 个城市的基础上, 另外加入了 5 个, 10 个, 20 个和 40 个的仿真结果, 如 fig. 2 所示。

3 动态规划求解旅行商问题

动态规划的 Held-Karp[2-3] 算法是一种求解 TSP 的经典精确方法，通过状态压缩来记忆已访问的城市集合以及当前所在城市，从而实现比纯粹穷举更有效的搜索。然而，其复杂度仍是 $O(N^2 2^N)$ ，当 N 较大（如 20 以上）时，所需的时间和空间都会非常庞大。因此, 在本章节中, 我们不再进行 30 个城市的求解和计算, 这是因为设备内存有限和 MATLAB 的限制, 如 fig. 3 所示。

```
错误使用 ones
请求的 1073741824x30 (240.0GB)数组超过预设的最大数组大小(16.0GB)。这可能会导致 MATLAB 无响应。

出错 dp>tsp_dp (第 61 行)
dp = INF * ones(nStates, N, 'double');

出错 dp>DP_TSP_main (第 19 行)
[bestPath, bestCost] = tsp_dp(distance);

出错 dp (第 9 行)
DP_TSP_main();
```

Fig. 3 $N = 30$ 时, 程序无法运行

3.1 Held-Karp 动态规划算法原理

动态规划思路 对于 TSP 来说，如果有 N 个城市，可先固定一个城市作为起点（例如城市 1），并要求最后回到该点。动态规划的核心在于记住“已经访问过哪些城市”和“当前所在城市”即可确定后续最优策略。通过状态压缩来表示已经访问的城市集合。

首先是状态的定义, 定义为 dp[mask, i]。

- mask 是一个二进制掩码（bitmask），用 1 表示城市已被访问，0 表示未被访问。若 mask 的第 j 位为 1，则说明城市 j 已访问。
- i 为当前所处的城市编号。
- dp[mask, i] 表示：在访问过集合 mask 中的城市、当前位于城市 i 时，行走过的最短路径长度。

然后是状态转移, 若要从 (mask, i) 转移到 (mask', j)，其中 $\text{mask}' = \text{mask} \cup \{j\}$，则需要更新, 如 eq. (10) 所示。

$$\begin{aligned} \text{dp}[\text{mask} \cup \{j\}, j] = \\ \min\Big(\text{dp}[\text{mask} \cup \{j\}, j],\ \text{dp}[\text{mask}, i] + d(i, j)\Big), \end{aligned} \tag{10}$$

其中 $d(i, j)$ 是城市 i 与城市 j 间的距离。

在此期间, 需要设置边界条件, 初始边界条件: 只访问过城市 1，且在城市 1，$\text{dp}[1 << (1-1), 1] = 0$, 结束边界条件: 当所有城市都被访问，即 $\text{mask} = (1 << N) - 1$，还要从当前城市 i 回到城市 1，得到完整回路。

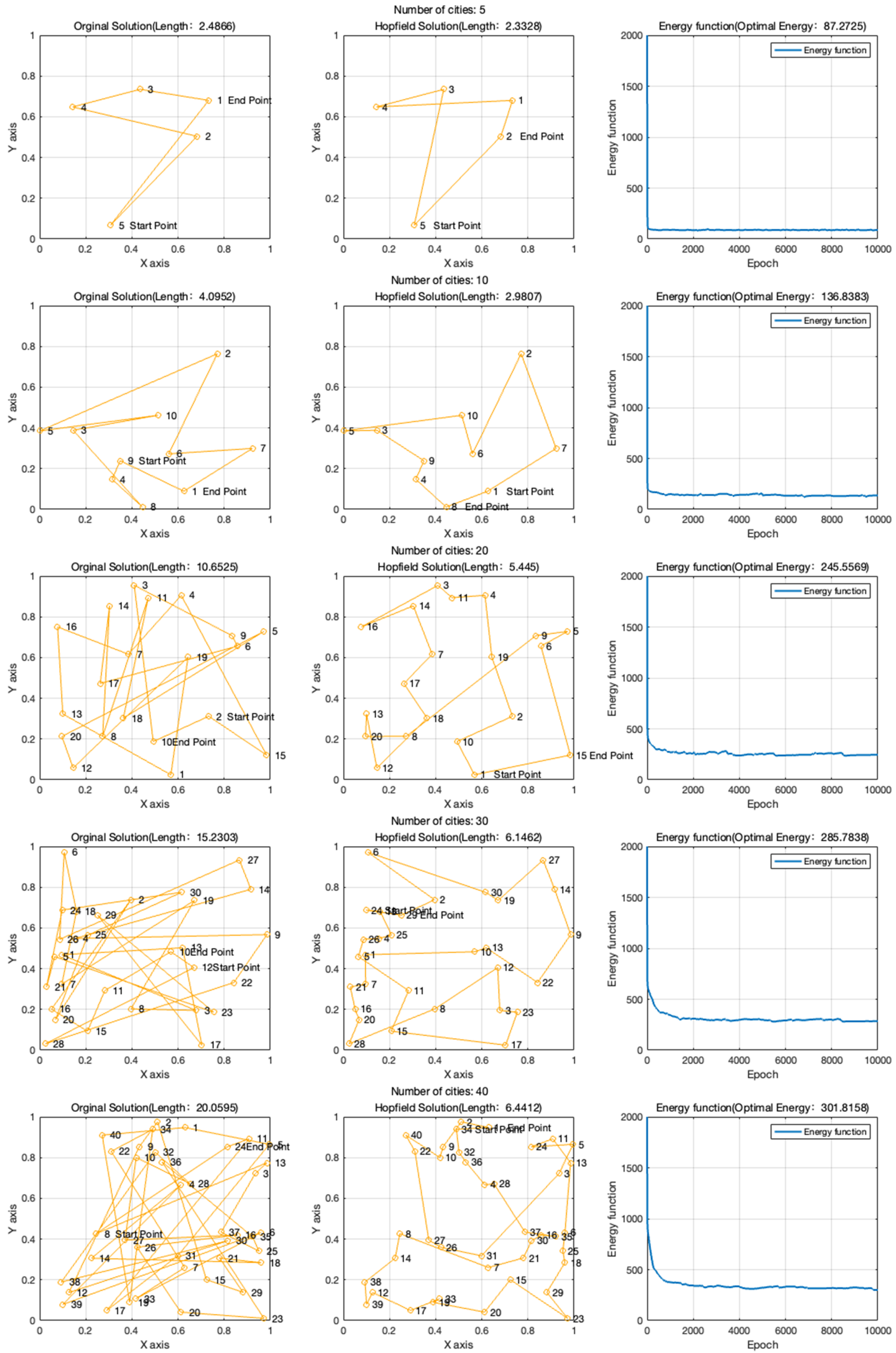

Fig. 2 Hopfield 优化方法所得到的 MATLAB 仿真结果

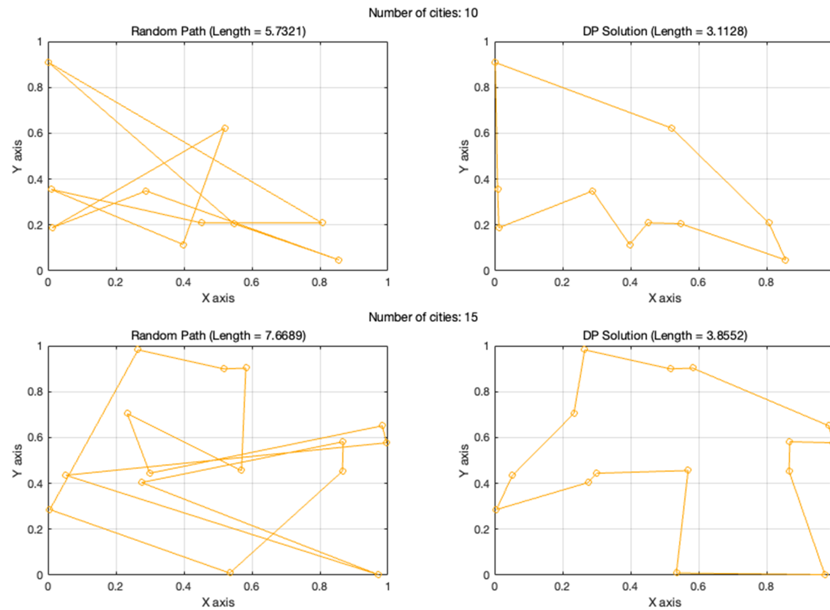

Fig. 4 动态规划优化方法所得到的 MATLAB 仿真结果

复杂度分析 Held-Karp 算法的时间与空间复杂度都是 $O(N^2 2^N)$。对于 $N = 10$ 或 $N = 15$ 等中小规模还算可行；一旦 $N = 20$ 或更大，内存和时间就会骤增，在普通机器上难以完成, 如 fig. 3 所示。

3.2 MATLAB 代码仿真

在本章节, 我会解释 MATLAB 仿真中重要的代码部分. 首先, 我把整个优化过程分为三个关键函数进行实现。

- `DP_TSP_main()`：主函数，生成随机城市、调用 `tsp_dp()` 求解、构造可视化需要的矩阵 V1，并最终调用 `plot_result` 绘图。
- `tsp_dp(distance)`：核心动态规划实现，返回最优路径 `bestPath` 以及对应的最短距离 `bestCost`。
- `plot_result(N, citys, V1)`：用于可视化初始随机路径与 DP 解路径的对比。

`tsp_dp` 函数部分代码如下所示。

```
function [bestPath, bestCost] = tsp_dp(distance)
    % ------ 初始化 ------
    N = size(distance, 1);
    INF = 1e10;
    nStates = 2^N;
    dp = INF * ones(nStates, N, 'double');
    parent = -1 * ones(nStates, N, 'int32');
    dp(1, 1) = 0;           % 起始状态: 只访问了城市1
        , 且当前在城市1
    % ------ 动态规划主体 ------
    for mask = 1 : nStates
        for lastCity = 1 : N
            if ~bitand(mask, bitshift(1,
                lastCity-1))
                continue;   % lastCity不在mask里
            end
            currCost = dp(mask, lastCity);
            if currCost >= INF
                continue;
            end
            for nextCity = 1 : N
                if bitand(mask, bitshift(1,
                    nextCity-1))
                    continue;   % nextCity已在
                        mask中
                end
                nextMask = bitor(mask, bitshift
                    (1, nextCity-1));
                newCost = currCost + distance(
                    lastCity, nextCity);
                if newCost < dp(nextMask,
                    nextCity)
                    dp(nextMask, nextCity) =
                        newCost;
                    parent(nextMask, nextCity) =
                         lastCity;
                end
            end
        end
    end
    % ------ 回到城市1, 找最优值 ------
    fullMask = bitshift(1, N) - 1;
    bestCost = INF;
    bestLastCity = -1;
    for i = 2 : N
        tempCost = dp(fullMask, i) + distance(i,
            1);
        if tempCost < bestCost
            bestCost = tempCost;
            bestLastCity = i;
        end
    end
    % ------ 回溯最优路径 ------
    bestPath = zeros(1, N, 'int32');
    bestPath(N) = 1;
    mask = fullMask;
    curCity = bestLastCity;
    for pos = N-1 : -1 : 1
        bestPath(pos) = curCity;
        prevCity = parent(mask, curCity);
```

```
        mask = bitxor(mask, bitshift(1, curCity
            -1));
        curCity = prevCity;
    end
end
```

首先是 dp 数组：dp(mask, i)，其中 mask 取值范围是 □ (2^N)，用二进制表示访问过的城市集合；i 表示当前所处城市；dp(mask, i) 存储访问了 mask 对应的城市、且当前在 i 时所需的最短路径距离。

然后是进行初始化。dp(1,1) = 0，表示只访问了城市 1（mask=1, 二进制仅最低位为 1）、且当前城市为 1 时，路程为 0。其余 dp(...) 默认设为一个极大值 INF。

然后是状态转移, 外层 mask 循环遍历所有已访问城市集合，内层 lastCity 再遍历可能的"当前城市"。若 lastCity 在 mask 中，则通过枚举 nextCity 来尝试前往还未访问过的城市。若 nextCity 未访问，则更新 dp(nextMask, nextCity), 状态转移公式如 eq. (11) 所示。

$$\begin{aligned} dp[\text{nextMask}, \text{nextCity}] \leftarrow \\ \min\Big(dp[\text{nextMask}, \text{nextCity}],\ dp[\text{mask}, \text{lastCity}] + . \\ d(\text{lastCity}, \text{nextCity})\Big) \end{aligned} \tag{11}$$

同时用 parent 记录是从哪个城市转移过来，方便最终回溯路径。

最后, 所有城市都被访问过时，mask = (1<<N) - 1；最后要加上从 i 返回城市 1 的距离来得到一条完整回路, 如 eq. (12) 所示。

$$\text{bestCost} = \min_{i \neq 1}\Big(dp[\text{fullMask}, i] + d(i, 1)\Big). \tag{12}$$

代码其他详细部分, 可见补充材料。

3.3 仿真结果

如 fig. 4 所示, 动态规划可以很好的完成这个任务。在实验中, N 的取值有 10 和 15。

4 遗传算法求解旅行商问题

4.1 遗传算法[4-5]原理

遗传算法是一种基于自然选择和遗传机制的优化搜索方法，广泛应用于解决复杂的优化和搜索问题。遗传算法模拟了自然界中生物进化的过程，通过不断迭代优化种群中的个体，以寻找最优解。遗传算法的核心思想源自达尔文的进化论，主要包括选择、交叉（也称为重组）和变异三个基本操作。这些操作在每一代中反复进行，逐步优化种群中的个体适应度，最终趋近于问题的最优解。

初始化种群. 首先，遗传算法需要生成一个初始种群。种群由多个个体（也称为染色体）组成，每个个体表示问题的一个潜在解。染色体通常用二进制串、实数串或其他适当的编码方式表示。

例如，假设我们有一个优化问题，其解可以用长度为 n 的二进制串表示，则初始种群可以表示为 eq. (13) 所示。

$$\text{Population } = \{C_1, C_2, \ldots, C_m\} \tag{13}$$

其中，m 是种群的大小，C_i 是第 i 个个体。

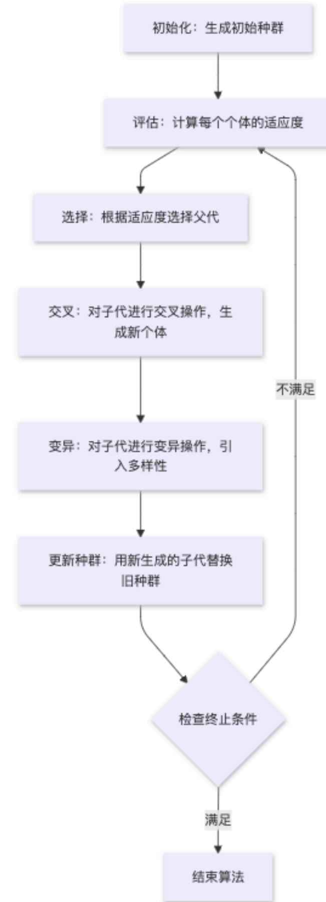

Fig. 5 遗传算法执行流程

适应度函数. 适应度函数（Fitness Function）用于评估每个个体的优劣。适应度函数根据问题的具体需求定义，通常目标是最大化或最小化某个评价指标。设适应度函数为 $f(C_i)$，则 $f(C_i) =$ 适应度值, 适应度值越高，表示该个体的解越优。

选择操作. 选择操作根据个体的适应度选择下一代的父代。常用的选择方法包括轮盘赌选择、竞赛选择和排名选择等。

轮盘赌选择是一种常见的方法，其选择概率与个体的适应度成正比。具体公式如 eq. (14) 所示。

$$P(C_i) = \frac{f(C_i)}{\sum_{j=1}^{m} f(C_j)} \tag{14}$$

其中，$P(C_i)$ 是个体 C_i 被选中的概率。

交叉操作. 交叉操作通过交换两个父代的部分基因，生成新的后代。常见的交叉方法包括单点交叉、多点交叉和均匀交叉等。单点交叉的过程如下：

- 在两个父代染色体上随机选择一个交叉点。
- 交换交叉点后的基因片段，生成两个新个体。

例如，假设两个父代染色体为：$C_1 = 11001010$ 和 $C_2 = 10110100$ 选择交叉点在第 4 位后，则生成的新个体为：$\text{Offspring}_1 = 1100 + 0100 = 11000100$ 和 $\text{Offspring}_2 = 1011 + 1010 = 10111010$ 。

变异操作. 变异操作随机地对个体的基因进行微小修改，以维持种群的多样性，防止过早收敛到局部最优解。变异通常以较低的概率进行，例如每个位点的变异概率为 p_m 。对于二进制编码的染色体，变异操作可以是翻转某个位的值：如果 $C_i = 0$ 则变为 1，反之亦然。

终止条件. 遗传算法通常在满足某个终止条件时停止，如达到预定的迭代次数、找到足够优的解或种群的适应度不再显著提升等。

遗传算法执行流程如 fig. 5 所示。

4.2 MATLAB 代码仿真

主要流程. 遗传算法主要执行流程如下所示。

1. 生成城市数据，同之前其他算法格式：`citys = rand(N, 2)`，其中 N 为城市的数量。
2. 初始化遗传算法的参数。
3. 初始化种群。
4. 计算初始种群的适应度
5. 开始迭代
6. 选择操作
7. 染色体交叉操作
8. 染色体变异操作
9. 筛选种群
10. 判断终止条件

主运行流程如所示。

```
function GA_TSP_main()
%
% ga function
%
citys = rand(50, 2); % 数据
N = size(citys, 1);
distance = pdist2(citys, citys, 'euclidean'); %
    计算城市间距离矩阵

% 遗传算法参数
populationSize = 50;       % 种群大小
generations = 5000;        % 迭代代数
break_generations = 100; % 退出条件，如果连续
    break_generations代bestCost没有变化，则退出
    循环
crossoverRate = 0.8;       % 交叉概率
mutationRate = 0.5;        % 变异概率

% 初始化种群
population = init_population(populationSize, N);
% 计算此时种群的适应度
fitness = eval_fitness(populationSize,
    population, distance);
% 记录初始种群最佳适应度
[bestFitness, bestIdx] = min(fitness);
bestPath = population(bestIdx, :);
bestCost = bestFitness;
costList = [];
% 开始迭代进化
for gen = 1:generations
    % 选择：基于轮盘赌选择
    selected = selection(population, fitness,
        populationSize);
    assert(is_population_valid(selected,
        populationSize), 'the population is
        invalid');
    % note: 选择完之后，种群里面一定有重复的元素
    % 交叉：有概率进行交叉
    offspring = crossover(selected,
        crossoverRate, N);
    assert(is_population_valid(offspring,
        populationSize), 'the population is
        invalid');
    % 变异
    offspring = mutation(offspring, mutationRate
        , N);
    assert(is_population_valid(offspring,
        populationSize), 'the population is
        invalid');
    % 计算当前种群的适应度
    offspringFitness = eval_fitness(
        populationSize, offspring, distance);

    % 精英保留：保留最好的 eliteCount 个个体
    % 将当前种群和后代种群的适应度和个体进行拼接
    combinedFitness = [fitness; offspringFitness
        ];
    combinedPopulation = [population; offspring
        ];

    % 对所有适应度进行排序，选择最优的
        populationSize 个个体
    [sortedFitness, sortedIdx] = sort(
        combinedFitness);
    population = combinedPopulation(sortedIdx(1:
        populationSize), :); % 选择fitness更小的
        50个对象
    fitness = sortedFitness(1:populationSize);

    % 更新最路径
    if fitness(1) < bestCost
        bestCost = fitness(1);
        bestPath = population(1, :);
    end
    costList = [costList, bestCost];
    % 可选：显示进度
    if mod(gen, 1) == 0
        fprintf('Generation %d/%d: Best Cost =
            %.4f\n', gen, generations, bestCost)
            ;
    end
end
% disp(bestPath); % 最佳路线
V1 = zeros(N, N);
for stepIdx = 1:N
```

```
    cityIdx = bestPath(stepIdx);
    V1(cityIdx, stepIdx) = 1;
end
plot_result(N, citys, V1, costList);
end
```

初始化种群操作 (init_population). 初始化种群部分代码非常简单, 使用 randperm 方法可以生成一个大小为 N, 数值为 $1-N$ 排列的数组即可。详细代码可见补充材料。

计算适应度函数 (eval_fitness).

通过 calculate_route_distance 方法计算种群中每一个特定排列路径的距离总和即可, 详细代码可见补充材料。

选择操作 (selection). 选择操作通过轮盘赌的方式进行选择。首先, 因为目前适应度为路线的距离总和, 是越小越好的。因此首先需要把适应度转换为越大越好的表示形式, 在进行选择。具体代码如下所示。

```
function selected = selection(population,
    fitness, populationSize)
%
% input:
%   population 当前种群; fitness当前种群的适应度
    ; populationSize当前种群的大小
% return:
%   selected 被选择过后的种群
%
% 将适应度转换为适应度值越大越好 (此处最小化问
    题, 逆转适应度)
maxFit = max(fitness);
adjustedFitness = maxFit - fitness + 1e-6;  % 防
    止为0
totalFit = sum(adjustedFitness);
prob = adjustedFitness / totalFit;
% 计算累积概率
cumProb = cumsum(prob);
% 选择
selected = zeros(size(population));
for i = 1:populationSize
    r = rand();
    selectedIdx = find(cumProb >= r, 1, 'first')
        ;
    selected(i, :) = population(selectedIdx, :);
end
end
```

交叉操作 (crossover). 交叉操作的设计比较复杂, 在本实验中, 采用 PMX(部分交叉) 方法进行实现, 在 crossover 中调用 pmx 函数完成两个个体的交叉操作。

```
function offspring = crossover(selected,
    crossoverRate, N)
    populationSize = size(selected, 1);
    offspring = selected;
    for i = 1:2:populationSize-1
        if rand() < crossoverRate
            parent1 = selected(i, :);
            parent2 = selected(i+1, :);
            [child1, child2] = pmx(parent1,
                parent2, N);
            offspring(i, :) = child1;
            offspring(i+1, :) = child2;
        end
    end
end
```

具体来说, 假设 parent1 = [1 3 4 5 2], parent2 = [4 5 2 1 3] ($N = 5$)。在 pmx 中, 首先随机生成两个数组下标, 表示需要进行交叉操作的范围, 并完成两个下标之间部分的交换。

```
% 随机选择交叉点
pt = sort(randperm(N, 2));
pt1 = pt(1);
pt2 = pt(2);
% 初始化子代
child1 = parent1;
child2 = parent2;
% 复制交叉区域
child1(pt1:pt2) = parent2(pt1:pt2);
child2(pt1:pt2) = parent1(pt1:pt2);
```

假设 pt(1)=3, pt(2)=5, 此时, child1 = [_ _ 2 1 3], child2 = [_ _ 4 5 2], 前两个数不能直接填写原 parent1, parent2 中的数值, 因此这样会造成个体不合法 (路径点重复)。因此需要处理这种重复情况, 代码如下所示, 通过计算目前 child1 或 child2 中仍然未存在数值, 来填写剩下的路径点数值。

```
benchmark = 1:N;
% parent
% parent1: 1 3 4 5 2
% parent2: 4 5 2 1 3
% fix duplicate
% child1: 1 3 2 1 3
% child2: 4 5 4 5 2
child1_left = setdiff(benchmark, child1(pt1:pt2)
    );
child2_left = setdiff(benchmark, child2(pt1:pt2)
    );

% 先处理 child1 中的问题
n1 = length(child1_left);
n2 = length(child2_left);
assert(n1 == n2);
randomIdx1 = randperm(n1);
randomIdx2 = randperm(n2);
child1_left_random = child1_left(randomIdx1);
child2_left_random = child2_left(randomIdx2);
insert_idx = 1;
for i=1:N
    if i < pt1 || i > pt2
        child1(i) = child1_left_random(
            insert_idx);
        child2(i) = child2_left_random(
            insert_idx);
        insert_idx = insert_idx + 1;
    end
end
assert(length(unique(child1)) == length(child1))
    ;
assert(length(unique(child2)) == length(child2))
    ;
```

个体变异操作 (mutation). 个体变异操作实现比较简单, 只需要针对需要变异的个体的路径数组中, 随机挑选两个元素进行交换即可, 完整代码可见补充材料。

判断种群合法性 (is_population_valid). 为了验证种群在进行选择、交叉和变异操作后, 是否出现不合法个体, 因此用 is_population_valid 来进行验证。同时, 这个方法也可以间接验证选择、交叉和变异操作的效果是符合预期的。

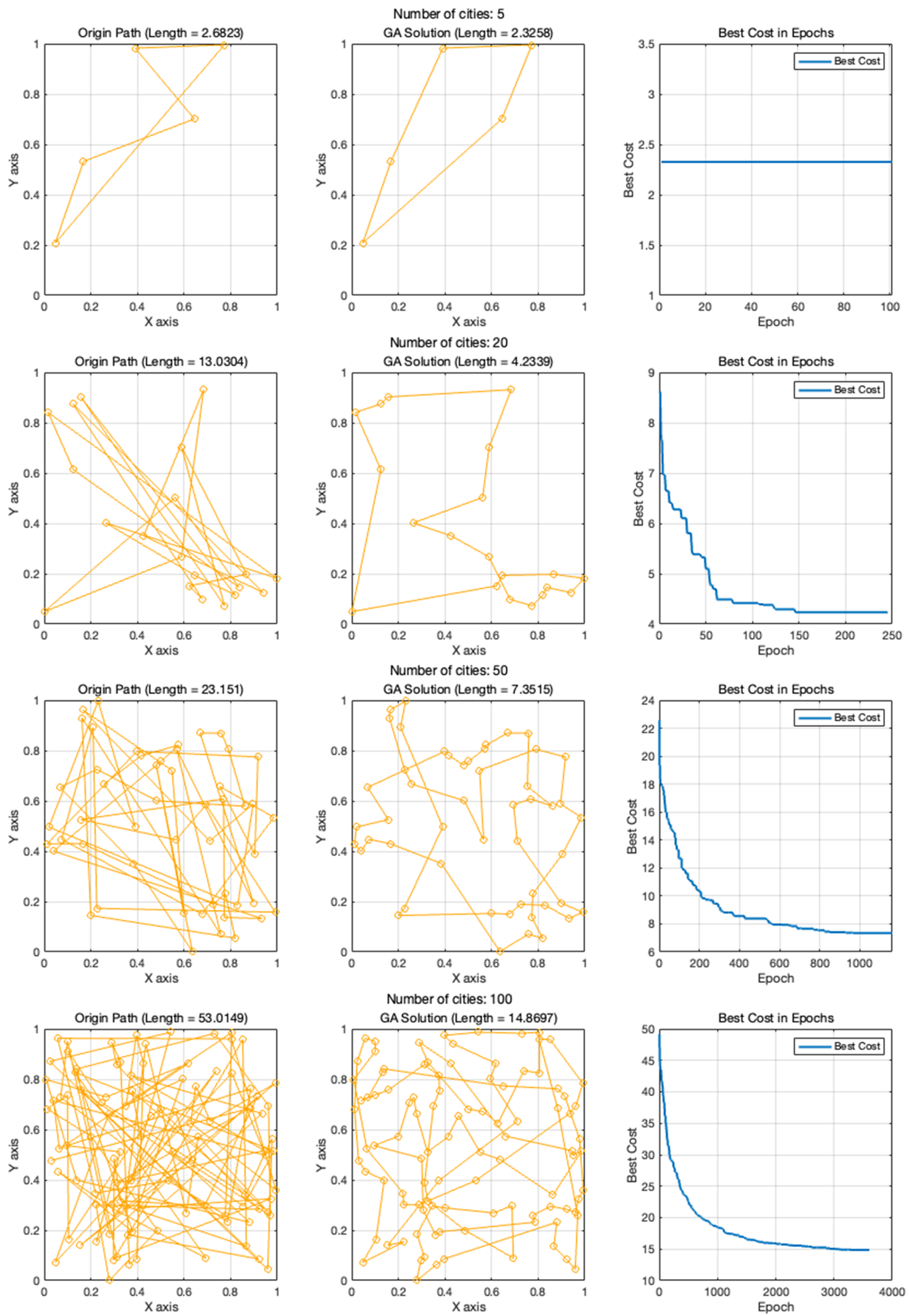

Fig. 6 遗传算法优化方法所得到的 MATLAB 仿真结果

4.3 仿真结果

在遗传算法的实验中, 运行城市数量为 5 个, 20 个, 50 个和 100 个的仿真结果, 如 fig. 6 所示。因此可以得出, 遗传算法对比动态规划来说, 空间复杂度小, 可以在牺牲时间的前提下, 进行更多城市数量的 TSP 问题的优化。

5 三种方法的比较

本报告采用了 Hopfield 神经网络、动态规划（Held-Karp 算法）和遗传算法三种方法来求解旅行商问题（TSP）。以下从计算复杂度、解的质量和实现复杂度三个方面对这三种方法进行比较。

5.1 计算复杂度

动态规划（Held-Karp 算法）. 该方法的时间和空间复杂度均为 $O(N^2 \cdot 2^N)$，其中 N 为城市数量。尽管通过状态压缩优化了计算，但随着城市数量的增加，其复杂度呈指数级增长，限制了其在大规模问题中的应用。

Hopfield 神经网络. Hopfield 网络的计算复杂度主要取决于网络的迭代次数和每次迭代中能量函数的计算。整体复杂度为 $O(T \cdot N^2)$，其中 T 为迭代次数。相比动态规划，Hopfield 网络在中小规模问题中具有更好的计算效率，但在大规模问题中仍需较多计算资源。

遗传算法. 遗传算法的时间复杂度通常为 $O(G \cdot P \cdot N)$，其中 G 为迭代代数，P 为种群规模，N 为城市数量。由于其线性增长的复杂度，遗传算法在处理大规模 TSP 问题时表现出更好的扩展性。

5.2 解的质量

动态规划. 能够找到 TSP 问题的精确最优解，保证了解的质量。然而，这一优势仅在城市数量较少时适用，对于大规模问题无法应用。

Hopfield 神经网络. 通常能够找到近似最优解，解的质量依赖于网络参数的设置和初始状态。存在陷入局部最优的风险，导致解的质量不稳定。

遗传算法. 通过种群的多样性和进化操作，能够在较大规模下找到高质量的近似解。虽然无法保证全局最优，但在实际应用中，其解的质量通常令人满意，特别是在参数优化后。

5.3 实现复杂度

动态规划. 实现相对直接，但需要大量内存来存储中间状态，尤其在处理较大规模问题时。此外，状态转移的实现较为复杂，需要细致的编程和优化。

Hopfield 神经网络. 涉及神经网络的设计和能量函数的定义，需要一定的神经网络知识。参数调节（如惩罚系数的设置）较为繁琐，增加了实现的复杂度。

遗传算法. 实现相对灵活，主要包括种群初始化、适应度评估、选择、交叉和变异等基本操作。尽管需要设计适应度函数和进化策略，但其模块化特性使得实现过程较为简便，且易于调整和扩展。

6 总结

本报告探讨了三种求解旅行商问题（TSP）的方法：Hopfield 神经网络、动态规划（Held-Karp 算法）和遗传算法。通过对比发现，动态规划能够在小规模问题中提供精确解，但其计算复杂度随着城市数量的增加迅速上升，限制了其应用范围。Hopfield 神经网络在中小规模问题中表现良好，能够找到近似最优解，但实现和参数调节较为复杂，且易陷入局部最优。遗传算法则展现出优越的扩展性和适应性，适用于大规模和多变种的 TSP 问题，能够在合理时间内提供高质量的近似解。综上所述，选择适当的方法应依据具体问题的规模、对解的精度要求及可用计算资源来决定。

参考文献

[1] HOPFIELD J J, TANK D W. “neural” computation of decisions in optimization problems[J/OL]. Biological Cybernetics, 1985, 52(3): 141-152. https://doi.org/10.1007/BF00339943.

[2] HELD M L, KARP R M. The traveling salesman problem and minimum spanning trees[J/OL]. Advances in Computers, 1962, 10: 212-217. https://doi.org/10.1016/S0065-2458(08)60024-3.

[3] BELLMAN R. Dynamic programming[M]. Princeton, NJ: Princeton University Press, 1957.

[4] SAMPSON J R. Adaptation in natural and artificial systems (john h. holland)[M]. Society for Industrial and Applied Mathematics, 1976.

[5] HOLLAND J H. Genetic algorithms[J]. Scientific american, 1992, 267(1): 66-73.

[6] HOFFMAN K L, PADBERG M, RINALDI G, et al. Traveling salesman problem[J]. Encyclopedia of operations research and management science, 2013, 1: 1573-1578.

[7] LIN S, KERNIGHAN B W. An effective heuristic algorithm for the traveling salesman problem[J/OL]. Operations Research, 1973, 21(2): 498-516. https://www.jstor.org/stable/1677106. DOI: 10.1287/opre.21.2.498.

[8] BEASLEY J E. The traveling salesman problem and its variations[M/OL]. 1st ed. New York, NY: Springer-Verlag, 1994. https://www.springer.com/gp/book/9780387985739.

[9] CHRISTOFIDES N. Worst-case analysis of a new heuristic for the traveling salesman problem[J/OL]. Theoretical Computer Science, 1976, 1(1): 237-279. https://www.sciencedirect.com/science/article/pii/0304397576900278. DOI: 10.1016/0304-3975(76)90027-8.

[10] DANTZIG G B, FULKERSON D R, JOHNSON S M. Solution of a large-scale traveling-salesman problem[J/OL]. Operations Research, 1954, 2(4): 393-410. https://www.jstor.org/stable/1685314. DOI: 10.1287/opre.2.4.393.

[11] KIRKPATRICK S, GELATT C D, VECCHI M P. Optimization by simulated annealing[J/OL]. Science, 1983, 220 (4598): 671-680. https://doi.org/10.1126/science.220.4598.671.

[12] DORIGO M. Optimization, learning and natural algorithms [J]. Ph. D. Thesis, Politecnico di Milano, 1992.

[13] DORIGO M, MANIEZZO V, COLORNI A. Ant system: optimization by a colony of cooperating agents[J]. IEEE transactions on systems, man, and cybernetics, part b (cybernetics), 1996, 26(1): 29-41.

[14] KENNEDY J, EBERHART R. Particle swarm optimization[C]//Proceedings of ICNN'95-international conference on neural networks: Vol. 4. ieee, 1995: 1942-1948.

旅行商问题的若干解法及其分析

补充材料

俞沣城 *，罗健文 †

（智能工程学院 2021 级智能科学与技术）

A Hopfield 求解代码

```
%
% hopfield.m
% write by Yufc: https://github.com/ffengc, 2024-12-24
%

clear all; clc;
global A D;

tic;  % 开始计时
%------------------------------
% 1. 生成城市位置数据
%------------------------------
citys=rand(40, 2);

%------------------------------
% 2. 计算城市之间的距离矩阵
%------------------------------
distance = dist(citys,citys'); % 这里使用 dist 函数，计算 citys 与其自身的欧氏距离矩阵

%------------------------------
% 3. 初始化神经网络参数
%------------------------------
N = size(citys, 1);          % 城市的数量
A = 200;                     % 能量函数中的惩罚系数，用于约束行、列的约束项
D = 100;                     % 能量函数中的惩罚系数，用于约束路径距离的项
U0 = 0.1;                    % tan-sigmoid 函数的规模参数
step = 0.0001;               % 每次迭代更新的步长
delta = 2 * rand(N,N) - 1;   % 在 (-1,1) 之间随机初始化
U = U0 * log(N-1) + delta;   % 初始输入神经元状态 U
V = (1 + tansig(U/U0))/2;    % 初始输出神经元状态 V，映射到 (0,1) 之间
iter_num = 10000;            % 迭代次数
E = zeros(1,iter_num);       % 用于记录每次迭代的能量值

%------------------------------
% 4. 迭代寻优
%------------------------------
for k = 1:iter_num
    dU = diff_u(V, distance);    % 计算 dU，即 Hopfield 网络的状态更新量
    U = U + dU*step;             % 更新输入神经元状态 U
    V = (1 + tansig(U/U0))/2;    % 通过双曲正切函数更新输出神经元状态 V
    e = energy(V,distance);      % 计算当前网络的能量函数值
    E(k) = e;                    % 记录本次迭代的能量函数值
end

%------------------------------
% 5. 判断网络输出是否为有效解
%------------------------------
[valid_flag, V1] = is_valid(V, N);
if valid_flag == 1
    disp('Get a optimal path');
    plot_result(N, citys, V1, iter_num, E);
else
    disp('The optimal path is invalid');
end

elapsedTime = toc;  % 结束计时，并返回运行时间（单位为秒）
fprintf('Opt Time: %.4f', elapsedTime);

%% functions
```

```
% --------------------------------------------
% 函数部分 1: is_valid(V, N)
% 判断路径有效性，即是否满足“每个城市只访问一次”并构成完整回路
% --------------------------------------------
function [flag, V1] = is_valid(V, N)
% V 是最终的输出神经元矩阵 (N×N)，理想情况是:
%   - 在每一列（对应城市在第几步访问），只有一个输出为1
%   - 在每一行（对应第 i 个城市），也只有一个输出为1
[rows,cols] = size(V);
V1 = zeros(rows,cols);  % 先构造一个同样规模的 0 矩阵
[~,V_ind] = max(V);     % 对每一列找到最大值（即最可能选中的神经元下标）
for j = 1:cols
    V1(V_ind(j),j) = 1; % 将 V1 对应位置赋值为 1, 构建确定解
end
C = sum(V1,1);          % 每一列的求和, 判断是否都为1
R = sum(V1,2);          % 每一行的求和, 判断是否都为1
flag = isequal(C,ones(1,N)) & isequal(R',ones(1,N));
% 若每一列、每一行都是 1, 则说明是有效解
end

% --------------------------------------------
% 函数部分 2: plot_result(N, citys, V1, iter_num, E)
% 绘制初始随机路径 VS. Hopfield 输出路径, 并绘制能量函数收敛曲线
% --------------------------------------------
function plot_result(N, citys, V1, iter_num, E)
%-----------------------------
% 2.1 先计算一个随机路径的长度
%-----------------------------
sort_rand = randperm(N);            % 随机打乱城市顺序
citys_rand = citys(sort_rand,:);    % 得到随机的城市访问顺序
Length_init = dist(citys_rand(1,:),citys_rand(end,:)');
% 计算随机路径长度
for i = 2:size(citys_rand,1)
    Length_init = Length_init+dist(citys_rand(i-1,:),citys_rand(i,:)');
end
%-----------------------------
% 2.2 绘制随机路径
%-----------------------------
figure('Position', [100, 100, 1000, 300]);
hold on;
subplot(1, 3, 1);
plot([citys_rand(:,1);citys_rand(1,1)],[citys_rand(:,2);citys_rand(1,2)],'o-','LineWidth',1, 'Color
', '#FFA500');
for i = 1:length(citys)
    text(citys(i,1),citys(i,2),['   ' num2str(i)])
end
text(citys_rand(1,1),citys_rand(1,2),['       Start Point' ])
text(citys_rand(end,1),citys_rand(end,2),['       End Point' ])
title(['Orginal Solution(Length: ' num2str(Length_init) ')'])
axis([0 1 0 1])
grid on;
xlabel('X axis');
ylabel('Y axis');
%-----------------------------
% 2.3 使用 Hopfield 确定的路径
%-----------------------------
[~,V1_ind] = max(V1);           % 每列选出最大值的索引, 得到具体访问顺序
citys_end = citys(V1_ind,:);    % 据此得到优化后的城市访问顺序

% 计算该最优路径长度
Length_end = dist(citys_end(1,:),citys_end(end,:)');
for i = 2:size(citys_end,1)
    Length_end = Length_end+dist(citys_end(i-1,:),citys_end(i,:)');
end
disp('Hopfield Solution Matrix');   % 控制台显示提示
% 绘制最优路径
% figure(2)
subplot(1, 3, 2);
plot([citys_end(:,1);citys_end(1,1)],...
    [citys_end(:,2);citys_end(1,2)],'o-','LineWidth',1, 'Color', '#FFA500');
```

```
for i = 1:length(citys)
    text(citys(i,1),citys(i,2),['   ' num2str(i)])
end
text(citys_end(1,1),citys_end(1,2),['       Start Point' ])
text(citys_end(end,1),citys_end(end,2),['       End Point' ])
title(['Hopfield Solution(Length: ' num2str(Length_end) ')'])
axis([0 1 0 1])
grid on;
xlabel('X axis');
ylabel('Y axis');
%----------------------------
% 2.4 绘制能量函数迭代曲线
%----------------------------
subplot(1, 3, 3);
plot(1:iter_num,E,'LineWidth',2);
ylim([0 2000])
title(['Energy function(Optimal Energy: ' num2str(E(end)) ')']);
xlabel('Epoch');
ylabel('Energy function');
grid on;
legend('Energy function');

sgtitle(sprintf('Number of cities: %d', N));
hold off;
end

% ------------------------------------------
% 函数部分 3: diff_u(V, d)
% 计算 Hopfield 网络中状态 U 的导数 dU
% ------------------------------------------
function du=diff_u(V,d)
global A D;
% V 大小为 N×N,  d 为距离矩阵
n=size(V,1);
% 1) sum(V,2) - 1 表示每一行之和距离 1 的偏差 (V 满足行约束)
%    sum_x 就是将这个偏差复制到 n×n 矩阵
sum_x=repmat(sum(V,2)-1,1,n);
% 2) sum(V,1) - 1 表示每一列之和距离 1 的偏差 (V 满足列约束)
%    sum_i 就是将这个偏差复制到 n×n 矩阵
sum_i=repmat(sum(V,1)-1,n,1);
% 3) 处理路径距离约束: 将输出矩阵 V 右移 1 列做比较
V_temp=V(:,2:n);
V_temp=[V_temp V(:,1)];
% 4) 求 d * V_temp, 即计算距离矩阵与下一列神经元对应元素相乘的加权和
sum_d=d*V_temp;
% 5) dU 即 Hopfield 神经网络的状态更新方程
du=-A*sum_x-A*sum_i-D*sum_d;
%    注意这里的负号是因为梯度下降法, 对于能量函数 E, dU = -∂E/∂V
end


% ------------------------------------------
% 函数部分 4: energy(V, d)
% 计算当前输出 V 对应的能量值 E
% ------------------------------------------
function E=energy(V,d)
global A D
n=size(V,1);
% 1) sum(V,2) - 1 表示每行之和与 1 的偏差, 用 sumsqr() 计算误差平方和
sum_x=sumsqr(sum(V,2)-1);
% 2) sum(V,1) - 1 表示每列之和与 1 的偏差
sum_i=sumsqr(sum(V,1)-1);
% 3) 处理路径距离约束: 同上, 将 V 右移 1 列形成环
V_temp=V(:,2:n);
V_temp=[V_temp V(:,1)];
sum_d=d*V_temp; % d 为距离矩阵
% 对应能量中路径部分
sum_d=sum(sum(V.*sum_d)); % V.*(d * V_temp) 并求和
% 4) 总能量函数为 0.5(A * sum_x + A * sum_i + D * sum_d)
%    其中系数 0.5 是因为 E = 1/2 * ∑(...), 可避免重复计算
```

```
E=0.5*(A*sum_x+A*sum_i+D*sum_d);
end
```

B 动态规划求解代码

```
%
% dp.m
% write by Yufc: https://github.com/ffengc, 2024-12-24
%

clear;clc;
close all;

DP_TSP_main();

function DP_TSP_main()
% =========== 1. 生成城市位置 & 距离矩阵 ===========
citys = rand(5, 2);                    % n个城市的随机坐标
N = size(citys, 1);                     % N=30
distance = dist(citys, citys');         % 计算城市间距离矩阵

% =========== 2. 动态规划求解TSP ===========
% 这里返回最优路径 bestPath, 以及其对应的最短距离 bestCost
[bestPath, bestCost] = tsp_dp(distance);

% =========== 3. 将bestPath转换为与plot_result相同格式的V1矩阵 ===========
%   V1是一个 N×N 的0-1矩阵, 第 j 列为第 j 步访问的城市
%   若第 j 步访问的是 bestPath(j), 则 V1( bestPath(j), j ) = 1
V1 = zeros(N, N);
for stepIdx = 1:N
    cityIdx = bestPath(stepIdx);
    V1(cityIdx, stepIdx) = 1;
end

% =========== 4. 绘图结果 ===========
%   直接调用之前的plot_result函数, 展示初始随机路径 vs DP获得的最优路径
plot_result(N, citys, V1);

% 控制台输出
disp(['DP best route:  ', mat2str(bestPath)]);
disp(['DP shortest distance:  ', num2str(bestCost)]);
end


function [bestPath, bestCost] = tsp_dp(distance)
% TSP_DP 使用 Held-Karp 动态规划算法求解TSP
%   distance: N×N 距离矩阵
%   bestPath: 最优路线 (长度 N 的向量), 从城市1出发并回到城市1
%   bestCost: 对应的最优路长

% ------ 一些初始化 ------
N = size(distance, 1);
INF = 1e10;    % 大数

% 我们假定: 从城市1开始, 并最终回到城市1
% 状态定义: dp[mask, i] 表示在“已访问城市集合”为 mask, 且“最后访问城市”为 i 时的最小路径距离
% mask 用二进制表示哪几个城市已访问过

nStates = 2^N;
dp = INF * ones(nStates, N, 'double');
% 用于回溯路径
parent = -1 * ones(nStates, N, 'int32');

% 初始状态: 只访问了城市1, 最后停在城市1
dp(1, 1) = 0;

% ------ 动态规划: 遍历所有子集和可能的末尾城市 ------
for mask = 1 : nStates
    % 对应已访问过的城市集合: mask
    % 找出集合mask中包含哪些城市
```

```
    % 这里通过位运算来判断
    for lastCity = 1 : N
        % 若lastCity不在mask中, 则跳过
        if ~bitand(mask, bitshift(1, lastCity-1))
            continue;
        end
        % 当前dp[mask, lastCity]
        currCost = dp(mask, lastCity);
        if currCost >= INF
            continue;
        end
        % 枚举下一个要访问的城市 nextCity
        for nextCity = 1 : N
            % 如果 nextCity 已在集合mask中, 则跳过
            if bitand(mask, bitshift(1, nextCity-1))
                continue;
            end
            % 新的子集
            nextMask = bitor(mask, bitshift(1, nextCity-1));
            % 更新dp
            newCost = currCost + distance(lastCity, nextCity);
            if newCost < dp(nextMask, nextCity)
                dp(nextMask, nextCity) = newCost;
                parent(nextMask, nextCity) = lastCity;
            end
        end
    end
end

% ------ 最后, 还需回到城市1 ------
% mask = (1<<N)-1 表示所有城市都已访问
fullMask = bitshift(1, N) - 1;
bestCost = INF;
bestLastCity = -1;

for i = 2 : N
    tempCost = dp(fullMask, i) + distance(i, 1);
    if tempCost < bestCost
        bestCost = tempCost;
        bestLastCity = i;
    end
end

% ------ 回溯得到最优路径 ------
bestPath = zeros(1, N, 'int32');
bestPath(N) = 1;            % 回到城市1
mask = fullMask;
curCity = bestLastCity;

for pos = N-1 : -1 : 1
    bestPath(pos) = curCity;
    prevCity = parent(mask, curCity);
    mask = bitxor(mask, bitshift(1, curCity-1));
    curCity = prevCity;
end

% 此时bestPath是[1,x2,x3,...,xN(=1)],
% 但若希望 strictly "从城市1出发"并在最后返回1,
% 我们可以把最末一个1去掉, 因为plot_result里只需要访问顺序
% 不过看你的需求, 本例里把N个城市顺序输出就行了
% (plot_result里会自动连回首城市)
end

function plot_result(N, citys, V1)
% ===== 1) 计算并绘制初始随机路径 =====
sort_rand = randperm(N);
citys_rand = citys(sort_rand, :);
Length_init = dist(citys_rand(1,:), citys_rand(end,:)');
for i = 2:N
    Length_init = Length_init + dist(citys_rand(i-1,:), citys_rand(i,:)');
```

```
end

figure('Position', [100, 100, 1000, 300]);
hold on;
subplot(1, 2, 1);
plot([citys_rand(:,1); citys_rand(1,1)], ...
    [citys_rand(:,2); citys_rand(1,2)], 'o-', ...
    'LineWidth',1,'Color','#FFA500');
title(['Random Path (Length = ' num2str(Length_init) ')']);
axis([0 1 0 1]); grid on; xlabel('X axis'); ylabel('Y axis');

% ===== 2) 由 V1 获取最终路径并绘制 =====
[~, V1_ind] = max(V1);   % 每列最大值索引
citys_end = citys(V1_ind, :);
Length_end = dist(citys_end(1,:), citys_end(end,:)');
for i = 2:N
    Length_end = Length_end + dist(citys_end(i-1,:), citys_end(i,:)');
end

subplot(1, 2, 2);
plot([citys_end(:,1); citys_end(1,1)], ...
    [citys_end(:,2); citys_end(1,2)], 'o-', ...
    'LineWidth',1,'Color','#FFA500');
title(['DP Solution (Length = ' num2str(Length_end) ')']);
axis([0 1 0 1]); grid on; xlabel('X axis'); ylabel('Y axis');
sgtitle(sprintf('Number of cities: %d', N));
hold off;
end
```

C 遗传算法求解代码

```
%
% ga.m
% Written by Yufc: https://github.com/ffengc, 2024-12-24
%

clear;clc;
close all;

GA_TSP_main();

function GA_TSP_main()
%
% ga function
%
citys = rand(100, 2); % 数据
N = size(citys, 1);
distance = pdist2(citys, citys, 'euclidean');  % 计算城市间距离矩阵

% 遗传算法参数
populationSize = 50;      % 种群大小
generations = 5000;       % 迭代代数
break_generations = 200;  % 退出条件，如果连续break_generations代bestCost没有变化，则退出循环
crossoverRate = 0.8;      % 交叉概率
mutationRate = 0.5;       % 变异概率
% eliteCount = 2;         % 精英保留数量

% 初始化种群
population = init_population(populationSize, N);
% 计算此时种群的适应度
fitness = eval_fitness(populationSize, population, distance);
% 记录初始种群最佳适应度
[bestFitness, bestIdx] = min(fitness);
bestPath = population(bestIdx, :);
bestCost = bestFitness;
costList = [];
% 开始迭代进化
for gen = 1:generations
    % 选择：基于轮盘赌选择
    selected = selection(population, fitness, populationSize);
```

```
    assert(is_population_valid(selected, populationSize), 'the population is invalid');
    % note: 选择完之后, 种群里面一定有重复的元素
    % 交叉: 有概率进行交叉
    offspring = crossover(selected, crossoverRate, N);
    assert(is_population_valid(offspring, populationSize), 'the population is invalid');
    % 变异
    offspring = mutation(offspring, mutationRate, N);
    assert(is_population_valid(offspring, populationSize), 'the population is invalid');
    % 计算当前种群的适应度
    offspringFitness = eval_fitness(populationSize, offspring, distance);

    % 精英保留: 保留最好的 eliteCount 个个体
    % 将当前种群和后代种群的适应度和个体进行拼接
    combinedFitness = [fitness; offspringFitness];
    combinedPopulation = [population; offspring];

    % 对所有适应度进行排序, 选择最优的 populationSize 个个体
    [sortedFitness, sortedIdx] = sort(combinedFitness);
    population = combinedPopulation(sortedIdx(1:populationSize), :); % 选择fitness更小的50个对象
    fitness = sortedFitness(1:populationSize);

    % 更新最佳路径
    if fitness(1) < bestCost
        bestCost = fitness(1);
        bestPath = population(1, :);
    end
    costList = [costList, bestCost];
    % 可选: 显示进度
    if mod(gen, 1) == 0
        fprintf('Generation %d/%d: Best Cost = %.4f\n', gen, generations, bestCost);
    end
    % break?
    if length(costList) > break_generations
        last_n_elements = costList(end - break_generations + 1 : end);
        if length(unique(last_n_elements)) == 1
            disp('Best Cost remains stable, break');
            break; % 长时间没有优化
        end
    end
end
% disp(bestPath); % 最佳路线
V1 = zeros(N, N);
for stepIdx = 1:N
    cityIdx = bestPath(stepIdx);
    V1(cityIdx, stepIdx) = 1;
end
plot_result(N, citys, V1, costList);
end

%% functions
function population = init_population(populationSize, N)
%
% input:
%   populationSize 需要初始化的种群大小
%   N 城市的数量
% return:
%   初始化种群, 返回一个二维数组, N列, populationSize行
%
for i = 1:populationSize
    population(i, :) = randperm(N);
end
end

function fitness = eval_fitness(populationSize, population, distance)
%
% input:
%   populationSize 种群大小; population种群; distance距离矩阵
% return:
%   fitness 长为 populationSize 的数组, 表示适应度
%
```

```
fitness = zeros(populationSize, 1);
for i = 1:populationSize
    route = population(i, :);
    fitness(i) = calculate_route_distance(route, distance);
end
end

function routeDistance = calculate_route_distance(route, distance)
%
% input:
%   route 路径; distance距离矩阵;
% return:
%   route_distance计算这个路径的距离
%
N = length(route);
routeDistance = 0;
for j = 1:N-1
    routeDistance = routeDistance + distance(route(j), route(j+1));
end
% 回到起点
routeDistance = routeDistance + distance(route(N), route(1));
end

function selected = selection(population, fitness, populationSize)
%
% input:
%   population 当前种群; fitness当前种群的适应度; populationSize当前种群的大小
% return:
%   selected 被选择过后的种群
%
% 将适应度转换为适应度值越大越好（此处最小化问题，逆转适应度）
maxFit = max(fitness);
adjustedFitness = maxFit - fitness + 1e-6;  % 防止为0
totalFit = sum(adjustedFitness);
prob = adjustedFitness / totalFit;

% 计算累积概率
cumProb = cumsum(prob);

% 选择
selected = zeros(size(population));
for i = 1:populationSize
    r = rand();
    selectedIdx = find(cumProb >= r, 1, 'first');
    selected(i, :) = population(selectedIdx, :);
end
end

function offspring = crossover(selected, crossoverRate, N)
%
% 交叉操作
%
populationSize = size(selected, 1);
offspring = selected;
for i = 1:2:populationSize-1
    if rand() < crossoverRate
        parent1 = selected(i, :);
        parent2 = selected(i+1, :);
        [child1, child2] = pmx(parent1, parent2, N);
        offspring(i, :) = child1;
        offspring(i+1, :) = child2;
    end
end
end

% PMX交叉实现
function [child1, child2] = pmx(parent1, parent2, N)
% 随机选择交叉点
pt = sort(randperm(N, 2));
pt1 = pt(1);
```

```
pt2 = pt(2);

% 初始化子代
child1 = parent1;
child2 = parent2;

% 复制交叉区域
child1(pt1:pt2) = parent2(pt1:pt2);
child2(pt1:pt2) = parent1(pt1:pt2);
benchmark = 1:N;
% parent
% parent1: 1 3 4 5 2
% parent2: 4 5 2 1 3
% fix duplicate
% child1: 1 3 2 1 3
% child2: 4 5 4 5 2
child1_left = setdiff(benchmark, child1(pt1:pt2));
child2_left = setdiff(benchmark, child2(pt1:pt2));

% 先处理 child1 中的问题
n1 = length(child1_left);
n2 = length(child2_left);
assert(n1 == n2);
randomIdx1 = randperm(n1);
randomIdx2 = randperm(n2);
child1_left_random = child1_left(randomIdx1);
child2_left_random = child2_left(randomIdx2);
insert_idx = 1;
for i=1:N
    if i < pt1 || i > pt2
        child1(i) = child1_left_random(insert_idx);
        child2(i) = child2_left_random(insert_idx);
        insert_idx = insert_idx + 1;
    end
end
assert(length(unique(child1)) == length(child1));
assert(length(unique(child2)) == length(child2));
end

function offspring = mutation(offspring, mutationRate, N)
%
% 变异操作
% input:
%   offspring后代种群; mutationRate变异率; N城市个数;
%
populationSize = size(offspring, 1);
for i = 1:populationSize
    if rand() < mutationRate
        % 随机选择两个基因位置进行交换
        swapIdx = randperm(N, 2);
        % 交换
        temp = offspring(i, swapIdx(1));
        offspring(i, swapIdx(1)) = offspring(i, swapIdx(2));
        offspring(i, swapIdx(2)) = temp;
    end
end
end

function flag = is_population_valid(population, populationSize)
for i = 1:populationSize
    cur_route = population(i,:);
    cur_route_unique = unique(cur_route);
    if length(cur_route_unique) ~= length(cur_route)
        flag = false;
        return;
    end
    flag = true;
end
end
```

```matlab
function plot_result(N, citys, V1, costList)
% ===== 1) 计算并绘制初始随机路径 =====
sort_rand = randperm(N);
citys_rand = citys(sort_rand, :);
Length_init = dist(citys_rand(1,:), citys_rand(end,:)');
for i = 2:N
    Length_init = Length_init + dist(citys_rand(i-1,:), citys_rand(i,:)');
end

figure('Position', [100, 100, 1000, 300]);
hold on;
subplot(1, 3, 1);
plot([citys_rand(:,1); citys_rand(1,1)], ...
    [citys_rand(:,2); citys_rand(1,2)], 'o-', ...
    'LineWidth',1,'Color','#FFA500');
title(['Origin Path (Length = ' num2str(Length_init) ')']);
axis([0 1 0 1]); grid on; xlabel('X axis'); ylabel('Y axis');

% ===== 2) 由 V1 获取最终路径并绘制 =====
[~, V1_ind] = max(V1);   % 每列最大值索引
citys_end = citys(V1_ind, :);
Length_end = dist(citys_end(1,:), citys_end(end,:)');
for i = 2:N
    Length_end = Length_end + dist(citys_end(i-1,:), citys_end(i,:)');
end

subplot(1, 3, 2);
plot([citys_end(:,1); citys_end(1,1)], ...
    [citys_end(:,2); citys_end(1,2)], 'o-', ...
    'LineWidth',1,'Color','#FFA500');
title(['GA Solution (Length = ' num2str(Length_end) ')']);
axis([0 1 0 1]); grid on; xlabel('X axis'); ylabel('Y axis');
subplot(1, 3, 3);
plot(1:length(costList), costList, "LineWidth", 2);
grid on;
legend("Best Cost");
xlabel("Epoch");
ylabel("Best Cost");
title("Best Cost in Epochs");
sgtitle(sprintf('Number of cities: %d', N));
hold off;
end
```