学 士 学 位 论 文

摘　要

蚁群算法(Ant Colony Optimization, ACO), 是一种模拟蚂蚁觅食过程中利用信息素的释放相互合作、选择路径的行为的仿生优化算法。最常见的应用场景是求解 TSP 问题。

在基本蚁群算法中，针对给定的 TSP 问题，需要对蚁群设置众多的参数，而参数的选取与算法的全局收敛性和收敛速度有着密切的关联。但由于蚁群算法参数空间的庞大性和各参数之间的关联性，如何确定最优组合参数使蚁群算法求解性能最佳一直是一个极其复杂的优化问题，目前上没有完善的理论依据，大多情况下都是根据经验而定。

遗传算法(Genetic Algorithm, GA)是一种被广泛应用于多目标参数优化、机器学习等领域之中的算法。本文的工作是基于遗传算法，对基本蚁群算法的参数最优组合进行估算。在此过程中，合理的分析并做出了蚁群的适应度函数、蚁群基因的二进制编码解码方式等方面的讨论。

关键字：　TSP 问题、蚁群算法、遗传算法、多目标组合优化、参数估计

Abstract

Ant Colony Optimization is a bionic optimization algorithm witch

Key word: Travelling Salesman Problem, Ant Colony Optimization, Genetic Algorithm, Multi-objective combinational optimization, parameter estimation

目　录

1 绪论

旅行商问题(Travelling Salesman Problem, TSP)是数学领域中著名问题之一。问题的描述为：在一个n阶带权完全图中，求一条哈密顿(Hamiltonian)回路，使的该回路经过的所有边权和最小。

TSP 问题在通讯、交通、军事等诸多领域均有着广泛的应用价值，然而它却是一个 NP 完全问题，人们无法给出一个有效的算法以在多项式时间内对其求解。在规模稍大的 TSP 模型下，获取最优解几乎是不可能做到的。因此，寻找一种能快速获取较优解的算法具有重要的科研意义。

蚁群算法(Ant Colony Optimization, ACO), 是由 Marco Dorigo 于 1991 年在他的博士论文[1]中提出的一种仿生算法。该算法通过模拟蚂蚁觅食过程中利用信息素的释放相互合作，选择路径的行为，在求解 TSP 问题中取得了十分卓越的成果。

在基本蚁群算法中，针对给定的 TSP 问题，需要对蚁群设置包括：信息素强度Q、信息素挥发系数ρ、信息启发式因子α、期望启发式因子β、蚁群规模m等参数。参数的选取与算法的全局收敛性和收敛速度有着密切的关联。但由于蚁群算法参数空间的庞大性和各参数之间的关联性，如何确定最优组合参数使蚁群算法求解性能最佳一直是一个极其复杂的优化问题，目前上没有完善的理论依据，大多情况下都是根据经验而定[2]。

蚁群算法参数的选取，可以看成一个多目标组合优化问题。遗传算法(Genetic Algorithm, GA)对于这类问题的求解卓有成效。遗传算法模拟自然界中生物进化的过程，以生物个体的适应度作为标准，对生物群体分别作用于选择、交叉、变异算子，得到下一代种群。随着遗传代数的增加，群体中的个体逐渐接近最优解。

本文将基于遗传算法对基本蚁群算法进行合理的参数优化，尝试给出一个合理的、普适的蚁群算法参数选取的方法。

2 蚁群算法的基本原理与编程实现

2.1 蚁群算法的基本原理

2.1.1 TSP 问题概述

TSP 问题（又称旅行商问题、货郎担问题），是数学领域中的一个著名问题。该问题的描述为：一个旅行商欲不重复无遗漏的路过 n 个城市并最终回到起点，怎样设计路线使得其行程达到最短。即：在一个节点数为 n 的无向完全图中，如何构造一条最短的哈密顿回路。

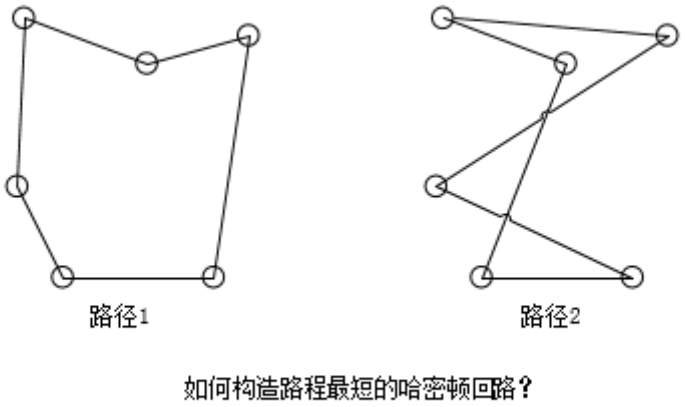

图 2-1 TSP 问题示意图

对于 n 个城市的 TSP 问题，所有可能的路径选择有 $\frac{(n-1)!}{2}$ 种。当 n 较大时，这将是一个天文数字！TSP 问题已被证明是一个 NP 问题，也即是说无法找到一个有效的算法，能够在多项式的时间复杂度以内找到问题的精确解。因此，对于大规模的 TSP 问题，人们往往采用近似算法以得到问题的较优解。

2.1.2 蚁群算法简介

蚁群算法(Ant Colony Optimization, ACO)是一种模拟进化算法，由 Marco Dorigo 于 1992 年在他的博士论文中提出。该算法模拟自然界中蚂蚁觅食寻找路线时表现出的协作行为，广泛应用于各类优化问题的求解。而该算法最常见的

应用场景便是对 TSP 问题进行求解，本文对蚁群算法的若干讨论也将基于 TSP 问题的求解之上。

2.1.3 蚁群算法的数学模型

蚂蚁通过信息素的释放，与其他的个体之间产生协作。由于信息素的挥发作用，距离食物较近的路线将逐渐积累较高浓度的信息素，而信息素浓度越高的路线又具有较大的概率被其它蚂蚁所选择。在这样一种正反馈作用下，蚁群将逐渐聚集到最优的路径上。这就是蚁群算法的基本思想。

根据信息素更新策略的不同，Marco Dorigo 提出了三种一群算法模型，分别为 Ant-Cycle 模型、Ant-Quantity 模型以及 Ant-Density 模型。这三个模型之间的主要差别在于信息素挥发时机的不同。在处理 TSP 问题时，通常情况下 Ant-Cycle 模型具有较好的求解性能与收敛速度，因此在本文中关于蚁群算法的讨论将基于 Ant-Cycle 基本蚁群算法模型。

下面给出基本蚁群算法的 Ant-Cycle 数学模型：

设 n 为 TSP 问题中的城市数量， m 为蚁群中的蚂蚁数量， $\tau_{ij}(t)$ 为 t 时刻路径 (i,j) 上的信息素含量， d_{ij} 为路径 (i,j) 的长度， C 为所有节点的集合， $tabu_k$ 为第 k 只蚂蚁已经经过的节点集合， ${p_{ij}}^k(t)$ 为第 k 只蚂蚁在 t 时刻从节点 i 转移到节点 j 的概率

$$
{p_{ij}}^k(t)=\begin{cases}\dfrac{[\tau_{ij}(t)]^{\alpha}/{d_{ij}}^{\beta}}{\sum_{s\in C-tabu_k}[\tau_{is}(t)]^{\alpha}/{d_{is}}^{\beta}} & j\in C-tabu_k\\ 0 & j\in tabu_k\end{cases} \tag{2-1}
$$

公式 2-1 中， α 为信息启发因子，表示在蚂蚁选择路径时，信息素的重要性，该值越大，表明蚂蚁越重视其它蚂蚁释放的信息素提供的信息。 β 为期望启发因子，表示在蚂蚁选择路径时，路程长短的重要性，该值越大，蚂蚁将趋于选择较短路径前进，也即整个蚁群系统更倾向于贪心算法求解问题。

在 Ant-Cycle 模型中，蚂蚁将在遍历所有城市一遍之后更新信息素，记 $\Delta{\tau_{ij}}^k$ 为第 k 只蚂蚁遍历所有城市后，在路径 (i,j) 上所释放的信息素量

$$
\Delta{\tau_{ij}}^k=\begin{cases}Q/L_k & \text{若蚂蚁经过路径}(i,j)\\ 0 & \text{否则}\end{cases} \tag{2-2}
$$

则在所有蚂蚁遍历完城市一遍后，所有路径上的信息素总量将更新为

$$
\tau_{ij}(t+n)=(1-\rho)\tau_{ij}(t)+\sum_{k=1}^{m}\Delta{\tau_{ij}}^k \tag{2-3}
$$

公式 2-3 中，ρ为信息素挥发系数，其值越小，信息素的积累效果将越显著。

以上为蚁群算法对 TSP 问题的一次搜索过程。通过多次迭代该搜索过程，由于信息素逐渐积累，蚁群将逐渐收敛至该 TSP 问题的某个局部最优解上。该解即为蚁群算法得出的 TSP 问题的较优解。

2.2 蚁群算法的编程实现

这里给出 Ant-Cycle 蚁群算法的实现步骤流程图（以求解 TSP 问题为例）：

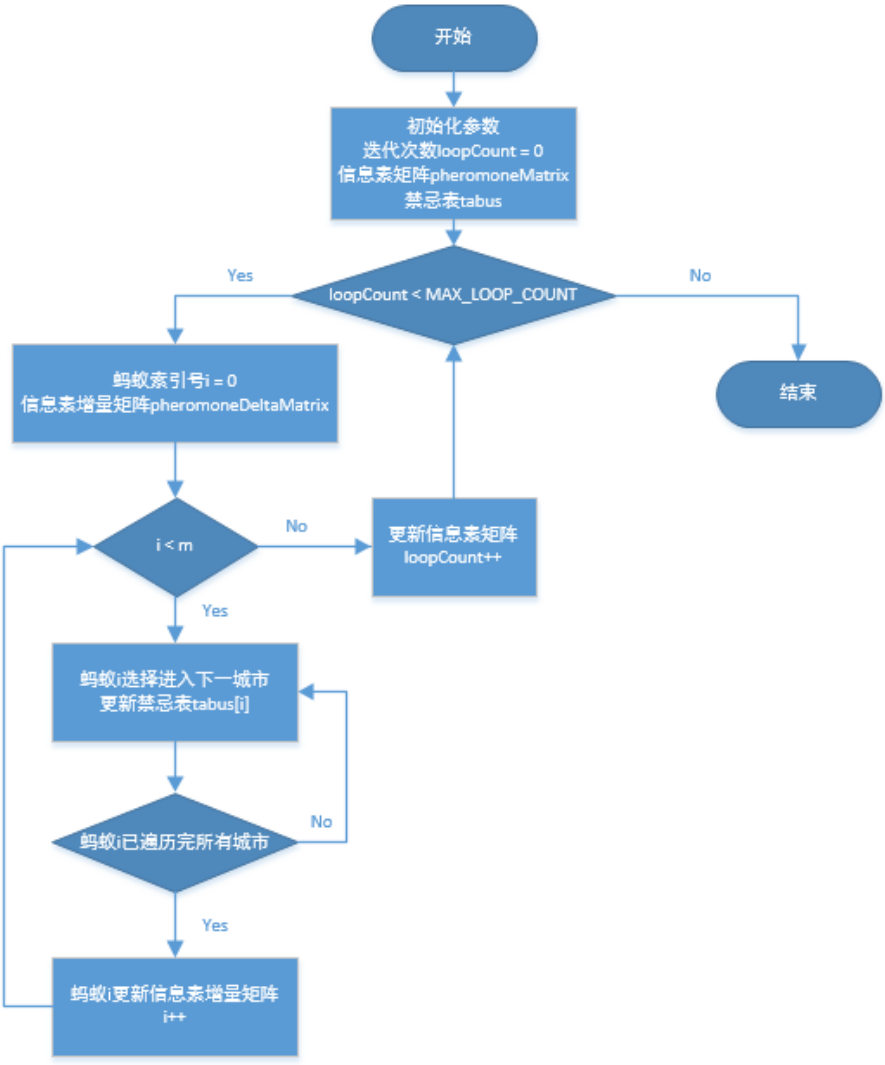

图 2-2 蚁群算法流程图

图 2-2 所示算法步骤的文字描述如下：

1.初始化参数，令迭代次数 loopCount = 0,设置最大迭代次数 MAX_LOOP_COUNT，初始化信息素矩阵 pheromoneMatrix,其中 pheromoneMatrix[i][j] = const，const 为常量，初始化禁忌表 tabus，tabus[i][j]记录第 i 只蚂蚁是否经过节点 j 的信息，进入步骤 2。

2.若 loopCount > MAX_LOOP_COUNT,算法结束，否则，令蚂蚁索引号为 i = 0，初始化信息素增量矩阵 pheromoneDeltaMatrix（初始为零矩阵），进入步骤 3。

3.蚂蚁按照公式 2-1 选择移动到下一个城市，更新禁忌表 tabus[i]，若尚有城市未被遍历，重复本步骤，否则进入步骤 4。

4.蚂蚁按照公式 2-2 释放信息素，更新信息素增量矩阵。蚂蚁索引号 i = i + 1,若 i < m，进入步骤 3，否则进入步骤 6。

5.按照公式 2-3 更新所有路径上的信息素，进入步骤 6。

6.循环次数 loopCount = loopCount + 1，进入步骤 2。

3 遗传算法的基本原理与编程实现

3.1 遗传算法的基本原理

3.1.1 遗传算法简介

遗传算法是一种模拟自然界中种群生物进化的优化算法，常被应用与多目标参数优化问题。遗传算法在问题的潜在解集中选取部分个体并视作一个种群(Population)，将个体的差异性特征（即问题的参数）进行适当的编码作为基因(Gene)，通过定义合适的适应度函数，对种群中的个体进行自然选择。然后对于选择算子作用后的种群进行个体的交叉与变异，最终得到下一代种群。经过一定代数的进化，种群中的个体的平均适应度将明显优于初代种群。

3.1.2 遗传算法的数学模型

由编码方式的不同，遗传算法通常被分为二进制编码遗传算法与浮点数编码遗传算法。在本文中，将采用二进制编码的遗传算法进行相关讨论。

遗传算法的一代进化过程可以抽象为三个作用于种群的算子：

1.选择算子：

选择运算是遗传算法中最为关键的一个步骤，其中，适应度函数的定义是否合理，将直接影响到算法的全局收敛性与收敛速度。适应度函数是一个种群间个体到非负实数之间的映射函数，体现了个体对于环境的适应程度，适应度越高的个体被选择到下一代的概率也就越大（采用轮盘赌方式进行个体选择）。

2. 交叉算子：

对于被选择出的个体，进行两两随机配对。将配对后的个体之间进行基因交叉互换，得到新的个体，用得到的新个体取代旧个体，这个过程称为交叉运算。

3. 变异算子：

为了防止算法陷于局部最优解中，经过交叉算子作用后的种群中每个个体将以一定的概率产生基因突变，突变后得到的种群将作为新一代的种群。

迭代上述进化过程，适应度高的个体所携带的优良基因将得以保留，适应度低的个体基因将被逐渐淘汰，种群的基因将趋于最优。

下面给出二进制编码遗传算法的数学模型：

设所求问题为求解能使非负多元函数 $f(X)$ 在可行域上 U 的达到最大值的 X，其中 $X=(x_1,x_2,\cdots,x_n)$ 为所求的参数向量。

在可行域 U 中选取 m 个适当的可行解作为遗传算法的初始种群，设他们的决策变量（基因）集合为 $P(t)=(X_1,X_2,\cdots,X_m)$。

将决策变量 X_i 编码为长度为 k 的二进制数 $X_i=b_{i1}b_{i2}\cdots b_{ik}$。对种群 $P(t)$ 分别作用以 *selection* 选择算子， *crossover* 交叉算子， *mutation* 变异算子，得到下一代种群 $P(t+1)$。

$$P(t+1)=mutation(crossover(selection(P(t)))) \tag{3-1}$$

公式 3-1 中，各算子定义如下：

selection 选择算子：根据种群 $P(t)$ 中个体的适应度，对种群进行 m 次选择，每次各个体被选择到下一代的概率

$$P_i=\frac{f(X_i)}{\sum_{t=0}^{m}f(X_m)} \tag{3-2}$$

经过上述选择过程后，得到 $selection(P(t))$。

crossover 交叉算子：在种群 $selection(P(t))$ 中进行随机两两配对，对每对个体 (X_i,X_j) 进行基因重组，即

$$\begin{cases} X_i=b_{i1}b_{i2}\cdots b_{it}b_{j(t+1)}\cdots b_{jk} \\ X_j=b_{j1}b_{j2}\cdots b_{jt}b_{i(t+1)}\cdots b_{ik} \end{cases} \tag{3-3}$$

其中，交叉位置t随机选取。

将经过上述交叉过程后，得到$crossover(selection(P(t)))$

$mutation$变异算子：对交叉算子作用后的$crossover(selection(P(t)))$，其中各个体$X_i = b_{i1}b_{i2}\cdots b_{ik}$以一定概率按位产生突变。得到$P(t+1)$。

3.2 遗传算法的编程实现

这里给出基本遗传算法的实现步骤流程图：

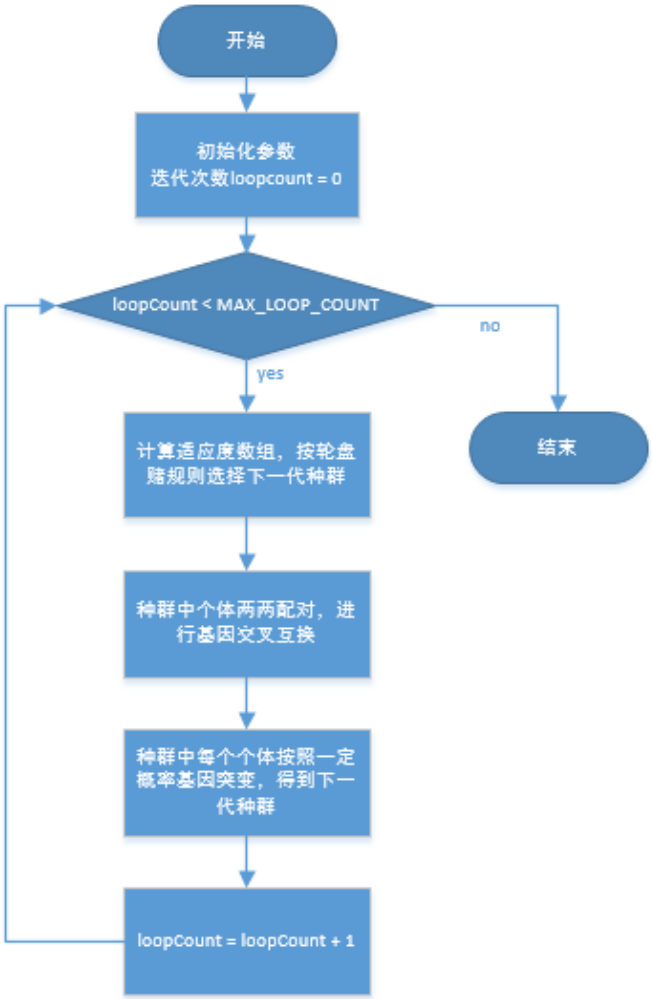

图 3-1 遗传算法流程图

图 3-1 所示算法步骤的文字描述如下：

1. 初始化参数，令迭代次数 loopCount = 0,设置最大迭代次数 MAX_LOOP_COUNT。选择 m 个合适的可行解作为初代群体。进入步骤 2。

2. 计算种群中所有个体的适应度，得到种群适应度数组 fitnessArray。进入步骤 3。

3. 按照轮盘赌规则，以适应度数组为权重，进行 m 次选择，得到被选择算子作用后的种群。进入步骤 4。

4. 将种群中的个体随机两两配对。每对个体之间进行基因交叉互换。得到被交叉算子作用后的种群。进入步骤 5。

5. 对种群中所有的个体进行基因突变，得到被变异算子作用后的种群。迭代次数 loopCount = loopCount + 1。进入步骤 6。

6. 若 loopCount < MAX_LOOP_COUNT，跳转至第 2 步，否则算法结束。

4 基于遗传算法对蚁群算法进行参数优化

4.1 蚁群算法存在的缺陷

在蚁群算法中，需要包括蚂蚁数量m、信息素强度Q、信息素挥发系数ρ、信息启发因子α、期望启发因子β等众多初始参数。这些参数的选取对于蚁群算法的全局收敛性和求解效率都有着显著的影响。如何确定这些参数的最佳组合是一个极其复杂的优化问题。通常而言，蚁群算法的参数选取往往凭借经验以及反复试凑得到。这显然是蚁群算法的一个重要的缺陷。

本文基于遗传算法，对蚁群算法的参数进行优化。尝试为蚁群算法的参数提供一个普适的、科学的选取方法。

4.2 蚁群的适应度函数

4.2.1 蚁群算法的性能评价指标

要采用遗传算法对蚁群算法进行参数优化，合理定义蚁群的适应度函数是整个算法的核心。

评价一个蚁群求解 TSP 问题性能的好坏，我们给出以下三个指标：

1.最佳性能指标

定义基本蚁群算法的相对误差作为最佳性能指标E_O

$$E_O = \frac{c_b - c^*}{c^*} \times 100\% \tag{4-1}$$

其中c_b为蚁群多次运算后得到的最优解，c^*表示所求 TSP 问题的理论最优解（若理论最优解未知，可用当前已知最优解替代）。该指标代表了蚁群算法对问题的优化性能，其值越小，表明该蚁群的优化性能越好。

2.时间性能指标

定义基本蚁群算法的时间性能指标E_T如下：

$$E_T = \frac{I_a T_0}{I_{\max}} \times 100\% \tag{4-2}$$

其中，I_a为本次计算首次搜索到最优解时的迭代次数，$I_{\max}$为给定的最大迭代次数，T_0为迭代一次所需的平均时间。时间性能指标用来衡量蚁群算法的收敛速度，其值越小，表明算法的收敛速度越快。

3.鲁棒性能指标

定义基本蚁群算法的鲁棒性能指标E_R如下：

$$E_R = \frac{c_a - c^*}{c^*} \times 100\% \tag{4-3}$$

其中c_a为蚁群多次运算后得到的解的平均值，c^*表示所求 TSP 问题的理论最优解（若理论最优解未知，可用当前已知最优解替代）。该指标代表了蚁群算法对算法随机性的依赖程度。

综上所述，我们可以定义蚁群算法的综合性能指标E如下：

$$E = \alpha_O E_O + \alpha_T E_T + \alpha_R E_R \tag{4-4}$$

其中α_O、α_T和α_R分别为最佳性能指标、时间性能指标和鲁棒性能指标的权重，满足$\alpha_O + \alpha_T + \alpha_R = 1$。综合性能指标$E$值越小，表明蚁群算法的性能越好。

4.2.2 蚁群算法的适应度函数

定义蚁群的适应度函数

$$f(C) = \frac{1}{E} = \frac{1}{\alpha_O \dfrac{c_b - c^*}{c^*} + \alpha_T \dfrac{I_a T_0}{I_{\max}} + \alpha_R \dfrac{c_a - c^*}{c^*}} \tag{4-5}$$

其中，令算法平均一次迭代时间$T_0 = \frac{m}{n}$（蚁群规模与城市数量之比）。

4.2.3 适应度的尺度变换

遗传算法中，个体被选择到下一代的概率是由个体的适应度所决定的。然而应用实践表明，单纯使用公式 4-5 计算出的结果作为个体的适应度，在算法的后期，由于个体的适应度都处于一个较高的平均水平，有轮盘赌计算出的概率相差不大，选择算子的作用效果将会减弱。

为了改善这一现象，对种群中个体适应度做合理的尺度变换，以提高选择算子的作用效果。个体 C_i 变换后的适应度定义为 $f'(C_i)$

$$f'(C_i)=\begin{cases} f(C_i)-f_{\min} & f(C_i)>f_{\min} \\ 0 & f(C_i)\le f_{\min} \end{cases} \tag{4-6}$$

其中，定义 $f_{\min}=\mu-3\sigma$ ， μ 为种群适应度的均值， σ 为种群适应度的标准差。

4.3 蚁群基因的二进制编码与解码

4.3.1 格雷编码的引入

遗传算法的部搜索能力不强，导致这个问题的主要原因是新一代的种群是有上一代种群经过交叉、变异后得到的。即便上一代种群个体已经接近了最优解，经过变异算子作用后，参数值可能会产生较大的变化，导致种群无法继续向最优解方向收敛。

格雷编码是广泛应用于通信、模拟-数字信号转换等领域的一种编码方式。格雷码具有这样一个特性：任意两个相邻整数对应的格雷码之间汉明距离为 1。正是由于这个特性的存在，使得格雷码十分适用于遗传算法。编码过后的基因经过突变后，其对应的原始数据只会产生细微的变化。这样可以大大提高遗传算法的局部搜索能力。

对于给定的自然二进制编码 $b_1b_2\cdots b_n$ ，其与对应的格雷码 $g_1g_2\cdots g_n$ 之间由以下公式进行转换。

自然二进制码至格雷码：

$$\begin{cases} g_1=b_1 \\ g_i=b_{i-1}\oplus b_i & i>1 \end{cases} \tag{4-7}$$

格雷码至自然二进制码：

$$\begin{cases} b_1=g_1 \\ b_i=b_{i-1}\oplus g_i & i>1 \end{cases} \tag{4-8}$$

4.3.2 编码与解码

本文对蚁群算法的参数采用以下二进制编码、解码方式：

1.编码

设 x 为待编码实数，区间 $[a,b)$ 为 x 的取值范围， n 为指定的编码长度。则将 x 编码为

$$x_g = gray\left(\frac{(x-a)2^n}{b-a}\right) \tag{4-9}$$

其中，函数 $gray(x)$ 为格雷编码函数。

2.解码

设 x_g 为待解码数，区间 $[a,b)$ 为 x 的取值范围， n 为指定的编码长度。则将 x_g 编码为

$$x = a + \frac{(b-a)gray^{-1}(x_g)}{2^n} \tag{4-10}$$

其中，函数 $gray^{-1}(x)$ 为格雷解码函数。

4.4 基于遗传算法的蚁群算法参数优化

4.4.1 初始数据的选取

蚁群求解的 TSP 问题设定为 EIL51 问题，该 TSP 问题的数据如下：

表 4-1 EIL51 问题数据

序号	x	y	序号	x	y	序号	x	y
1	37	52	18	17	33	35	62	63
2	49	49	19	13	13	36	63	69
3	52	64	20	57	58	37	32	22
4	20	26	21	62	42	38	45	35
5	40	30	22	42	57	39	59	15
6	21	47	23	16	57	40	5	6
7	17	63	24	8	52	41	10	17
8	31	62	25	7	38	42	21	10
9	52	33	26	27	68	43	5	64
10	51	21	27	30	48	44	30	15

11	42	41	28	43	67	45	39	10
12	31	32	29	58	48	46	32	39
13	5	25	30	58	27	47	25	32
14	12	42	31	37	69	48	25	55
15	36	16	32	38	46	49	48	28
16	52	41	33	46	10	50	56	37
17	27	23	34	61	33	51	30	40

已知该问题最优解路径为

$$\text{起点}\rightarrow 1\rightarrow 22\rightarrow 8\rightarrow 26\rightarrow 31\rightarrow 28\rightarrow 3\rightarrow 36\rightarrow 35\rightarrow 20\rightarrow$$
$$2\rightarrow 29\rightarrow 21\rightarrow 16\rightarrow 50\rightarrow 34\rightarrow 30\rightarrow 9\rightarrow 49\rightarrow 10\rightarrow$$
$$39\rightarrow 33\rightarrow 45\rightarrow 15\rightarrow 44\rightarrow 42\rightarrow 40\rightarrow 19\rightarrow 41\rightarrow 13\rightarrow$$
$$25\rightarrow 14\rightarrow 24\rightarrow 43\rightarrow 7\rightarrow 23\rightarrow 48\rightarrow 6\rightarrow 27\rightarrow 51\rightarrow$$
$$46\rightarrow 12\rightarrow 47\rightarrow 18\rightarrow 4\rightarrow 17\rightarrow 37\rightarrow 5\rightarrow 38\rightarrow 11\rightarrow 32\rightarrow \text{终点}$$

该路径长度为 426。

令 $m\in\{20,80\}$、$Q\in\{100,800\}$、$\alpha\in\{1,5\}$ $\beta\in\{1,5\}$ $\rho\in\{0.1,0.5\}$，由这些初始参数进行组合，产生下表中 32 个蚁群个体作为遗传算法的初始种群。

表 4-2 初始种群参数表

序号	m	Q	α	β	ρ	序号	m	Q	α	β	ρ
1	20	100	1	1	0.1	17	80	100	1	1	0.1
2	20	100	1	1	0.5	18	80	100	1	1	0.5
3	20	100	1	5	0.1	19	80	100	1	5	0.1
4	20	100	1	5	0.5	20	80	100	1	5	0.5
5	20	100	5	1	0.1	21	80	100	5	1	0.1
6	20	100	5	1	0.5	22	80	100	5	1	0.5
7	20	100	5	5	0.1	23	80	100	5	5	0.1
8	20	100	5	5	0.5	24	80	100	5	5	0.5
9	20	800	1	1	0.1	25	80	800	1	1	0.1
10	20	800	1	1	0.5	26	80	800	1	1	0.5
11	20	800	1	5	0.1	27	80	800	1	5	0.1
12	20	800	1	5	0.5	28	80	800	1	5	0.5
13	20	800	5	1	0.1	29	80	800	5	1	0.1
14	20	800	5	1	0.5	30	80	800	5	1	0.5
15	20	800	5	5	0.1	31	80	800	5	5	0.1
16	20	800	5	5	0.5	32	80	800	5	5	0.5

蚁群适应度函数的计算公式 4-5 中，通过大量实践，发现时间性能指标 E_T 对于算法结果有不良影响。由于该指标趋于将蚁群算法的收敛速度达到最快，

应用加入该指标的适应度计算公式后，种群将收敛至近贪婪算法。因此，本文计算蚁群适应度时将该指标权重设置为 0。即 $(\alpha_O,\alpha_T,\alpha_R)=(0.5,0,0.5)$

4.4.2 遗传算法的计算结果

根据前文所述计算方法，本文利用 Java 语言编写了蚁群算法与遗传算法的相应程序（程序代码见附录）。将上述给出的初始数据代入程序进行计算，得到的结果如下：

表 4-3 遗传算法各代种群结果

序号	m		Q		α		β		ρ	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
0	50	30	450	350	3	2	3	2	0.3	0.2
1	53.1	26.6	558	332	3.34	2.58	5.29	2.02	0.33	0.23
2	64.1	20.6	571	310	3.94	2.55	5.89	1.93	0.37	0.28
3	65	11.9	562	311	4.54	2.63	6.62	1.96	0.42	0.3
4	66.4	12.3	495	340	3.65	2.57	6.11	1.65	0.37	0.27
5	58.6	15.9	495	320	2.96	2.14	6.25	1.43	0.36	0.24
6	59.4	14.2	528	344	2.19	1.84	6.76	1.64	0.51	0.28
7	63.3	12.1	559	333	2.19	2.08	6.15	1.12	0.65	0.27
8	63.6	12.6	564	322	1.71	1.69	6.98	1.53	0.65	0.27
9	58.2	18.6	599	358	1.28	0.54	6.81	1.54	0.62	0.27
10	57.3	18.4	574	367	1.3	0.59	6.97	1.65	0.54	0.18
11	58.8	18.2	466	338	1.46	0.78	6.75	1.27	0.55	0.14
12	59	16.7	406	295	1.57	0.9	6.95	1.37	0.52	0.09
13	63.8	16.7	290	261	1.52	1.07	6.93	1.34	0.53	0.12
14	68	15.9	274	230	1.36	0.91	6.67	1.29	0.51	0.13
15	68.4	13.2	318	281	1.45	0.85	6.92	1.34	0.5	0.17
16	63.7	14	323	227	1.3	0.76	7	1.46	0.59	0.1
17	66.2	21	289	240	1.11	0.66	6.98	1.24	0.56	0.16
18	70.2	22.6	280	218	1.32	0.61	7.09	1.03	0.58	0.12
19	68.8	21.7	334	243	1.36	0.75	7.35	1.15	0.6	0.08
20	66.3	22.3	370	219	1.36	0.94	7.45	1.2	0.6	0.1
21	72.6	21.1	413	271	1.04	0.27	7.69	1.19	0.63	0.12
22	77.3	19.9	316	240	1.39	1.03	7.38	1.29	0.62	0.11
23	80.2	20.3	345	277	1.17	0.27	7.13	1.32	0.6	0.06
24	77	17.3	360	326	1.04	0.3	6.65	1.11	0.65	0.09
25	72.9	18.7	423	339	1.14	0.64	7.13	1.09	0.69	0.14
26	71.8	12.2	468	328	1.42	0.98	7.57	1.05	0.72	0.16
27	67.1	12.3	544	362	1.26	0.75	7.41	1.05	0.77	0.16
28	65.8	16.8	631	310	1.43	0.99	7.43	1.12	0.74	0.17

29	71	14.5	585	327	1.37	0.74	7.4	1.25	0.69	0.14
30	70.6	12.6	626	300	1.09	0.26	7.43	1.43	0.68	0.13

上表中给出了遗传算法计算得到的前 30 代种群的各参数的统计数据。

其中，m、Q、α、β、ρ分别为蚁群算法中蚂蚁数量、信息素强度、信息启发因子、期望启发因子与信息素挥发系数。各参数下μ列与σ列分别表示各代种群中对应参数的均值与标准差。

5 参数优化的初步结论

根据上面给出的数据，可以分别绘制出蚁群算法的参数m、Q、α、β、ρ随代数增长的曲线图如下：

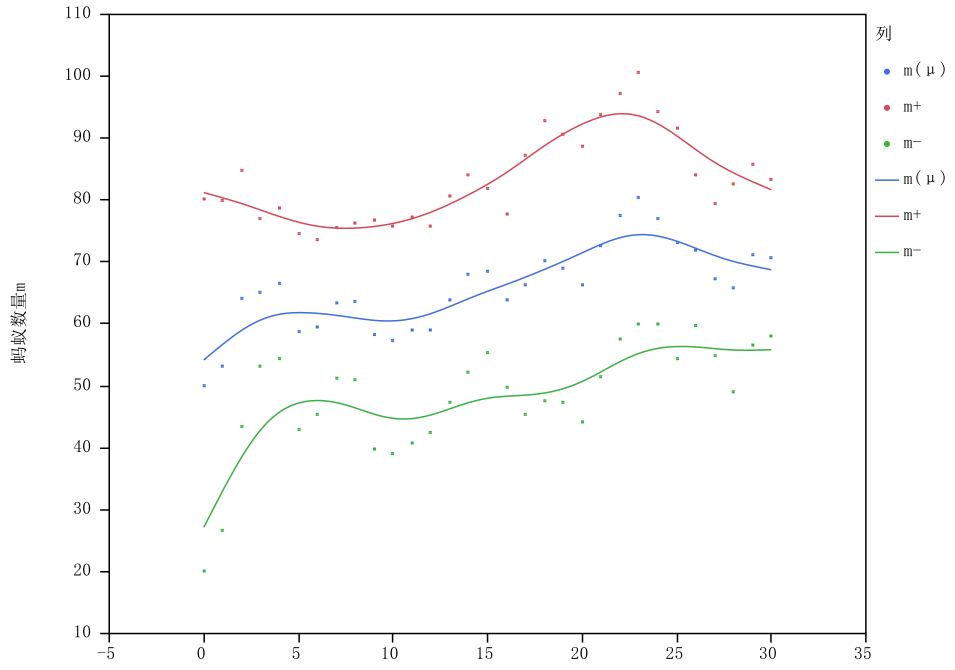

图 5-1 蚂蚁数量-遗传代数曲线图

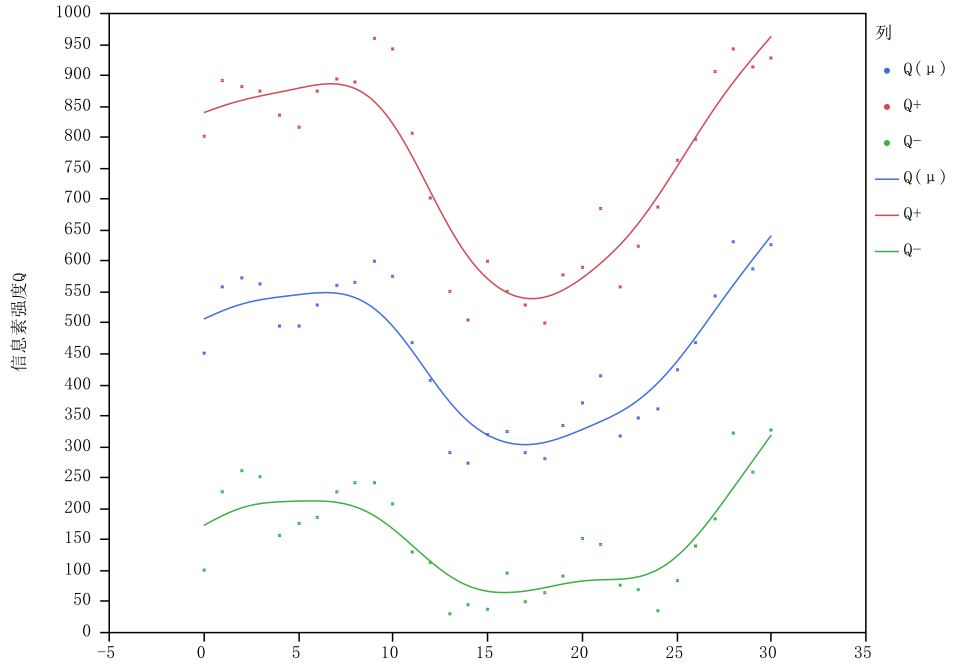

图 5-2 信息素强度-遗传代数曲线图

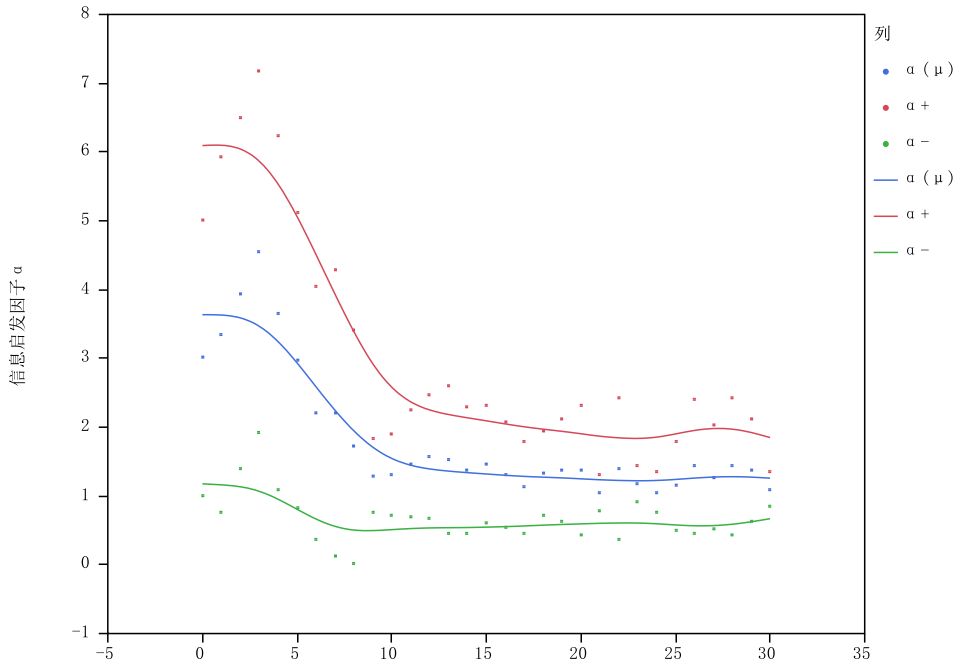

图 5-3 信息启发因子-遗传代数曲线图

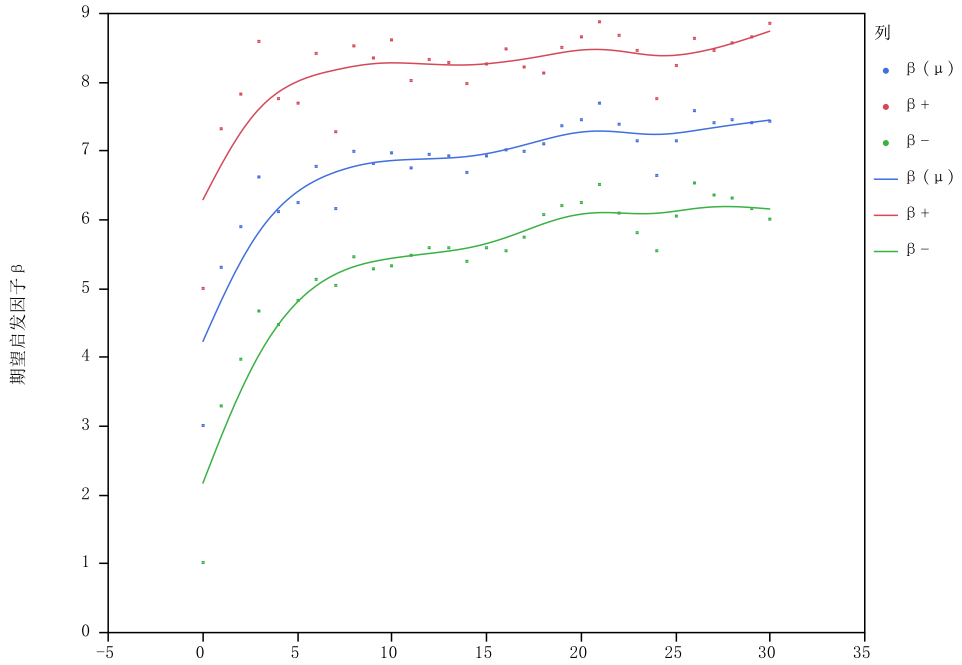

图 5-4 期望启发因子-遗传代数曲线图

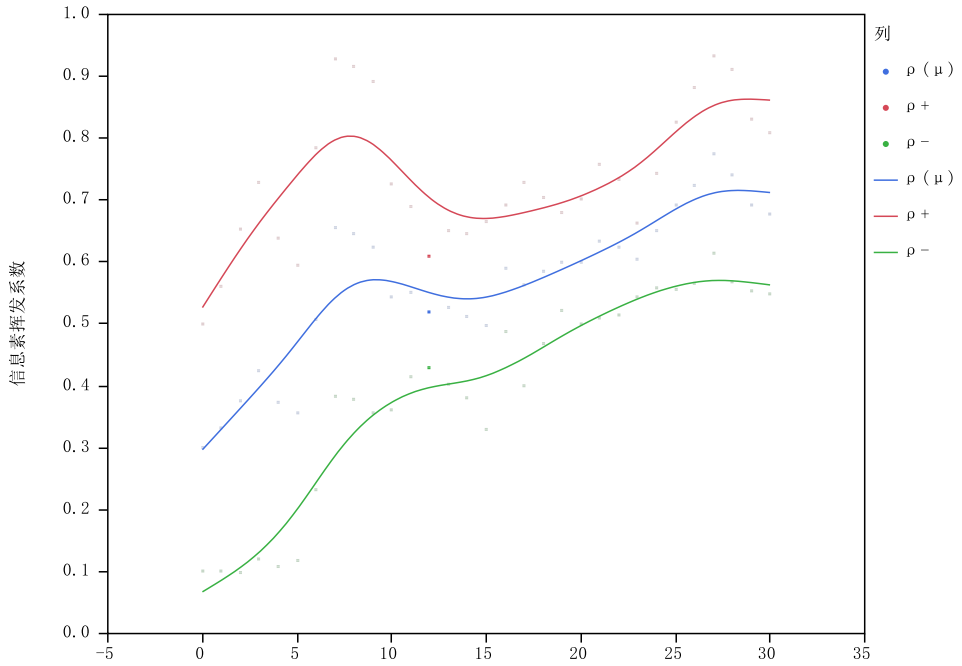

图 5-5 信息素挥发系数-遗传代数曲线图

以上图例中，蓝线表示因变量参数的均值 μ，红线为均值加一个标准差 $\mu+\sigma$，绿线为均值减一个标准差 $\mu-\sigma$。

通过观察上面各参数经过遗传后变化的曲线图，容易发现，蚂蚁数量m信息启发因子α、期望启发因子β、信息素挥发系数ρ均有较明显的收敛趋势，而信息素浓度Q并没有显著的收敛于某个值附近。

通过上述计算，我们可以初步给出蚁群算法针对 EIL51 问题的最优参数估计区间为：$m\in[50,80]$、$\alpha\in[1,2]$、$\beta\in[6,8]$、$\rho\in[0.5,0.8]$，而由于信息素强度并无明显的收敛趋势，按照人们大量实验得到的经验，这里给出其较优的取值区间为$Q\in[300,800]$。

6 存在的不足

本文通过遗传算法对蚁群算法的初始参数进行优化，并且获得了多个参数的较优取值区间。但是，由于尚有许多因素未考虑周全，因此本文所做的工作仍然存在许多不足与改进的空间。现将部分不足之处列举如下：

1．定义蚁群算法适应度函数时，直接忽略了算法的时间性能指标。虽然时间性能指标权重过高会导致蚁群算法向近贪心算法收敛，但该指标也不应直接忽视。而应通过多次试验，给出一个较小的合理权重。

2．在蚁群基因的二进制编码解码过程中，虽然使用了格雷码，但仍然无法完全回避基因突变导致的原参数急剧变化的情况。这也导致算法难以收敛到局部最优解。该问题可以通过更改基因突变策略或使用浮点型编码遗传算法来加以改进。

3．本文中对于蚁群算法的性能指标均是参照求解同一个 TSP 问题(EIL51)来评价的。然而蚁群算法的众多参数必然与具体的 TSP 问题相关。这也导致所求得到的结果并非普适于所有类型的 TSP 问题。对于这一问题，应当将各参数设为以具体 TSP 问题为变量的函数。然而该解决方法将大大增加计算的复杂度与计算时间。本文仅提出该思路，暂不做讨论。

7 致谢

历时近两月，本文方才基本定稿。在本文的撰写过程中遇到了许多困难与障碍，都在老师的指导和同学的帮助下度过了。在这里尤其要感谢我的论文指导老师——杜大刚老师。他在我撰文过程中给予了无私的指导。

感谢这篇论文所涉及到的各位学者。本文引用了数位学者的研究文献，如果没有各位学者的研究成果的帮助和启发，本篇论文的写作将很难完成。

感谢各位同学和朋友，在我翻译外文文献的时候提供了许多参考意见，还在论文的撰写和排版的过程中提供热情的帮助。

由于本人学术水平有限，所写论文难免有不足之处，恳请各位老师和学友批评和指正！

8 参考文献

[1]Colorni A, Dorigo M, Maniezzo V, etal. Distributed optimization by ant colonies. Proceedings of the 1st European Conference on Artificial Life, 1991, 134~142

[2]Dorigo M, Maniezzo V, Colorni A. Ant system: optimization by a colony of cooperating angent. IEEE Transaction on Systems, Man, and Cybernetics-Part B, 1996, 26(1): 29~41

[3]段海滨.蚁群算法原理及其应用[M].北京：科学出版社，2015.12

[4]

[5]周明，孙树栋.遗传算法原理及其应用[M].北京：国防工业出版社，1999.6

[6]张彤，张华，王子才.浮点数编码的遗传算法及其应用[J].哈尔滨工业大学学报，2000，32(4)：59~61

9 附录

A 蚁群算法核心代码

```
/**
 *  求解给定的 TSP 问题
 *
 *  @param tsp 给定的 tsp 问题
 *  @return 求得的解
 **/
public TSPSolution solveTSP(final TSP tsp) {
    TSPSolution solution = new TSPSolution(tsp);
    // 初始化信息素矩阵
    double[][] pheromoneMatrix = new double[tsp.n][tsp.n];
```

```
for (int i = 0; i < tsp.n; i++) {
    for (int j = 0; j < tsp.n; j++) {
        pheromoneMatrix[i][j] = q / tsp.averageDistance * tsp.n;
    }
}
// 开始迭代求解
for (int loopCount = 0; loopCount < MAX_LOOP_COUNT; loopCount++) {
    // 信息素增量矩阵
    double[][] pheromoneDeltaMatrix = new double[tsp.n][tsp.n];
    // 记录蚂蚁路径
    int[][] paths = new int[m][tsp.n];
    // 记录蚂蚁禁忌表
    boolean[][] tabus = new boolean[m][tsp.n];
    // 初始化路径记录与禁忌表
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < tsp.n; j++) {
            paths[i][j] = -1;
            tabus[i][j] = false;
        }
    }
    // 记录所有蚂蚁的最短行程
    double currentMinDistance = Double.MAX_VALUE;
    int[] currentMinPath = new int[tsp.n];
    // 对所有蚂蚁进行遍历操作
    for (int i = 0; i < m; i++) {
        // 随机初始化蚂蚁位置
        int position = (int) (Math.random() * tsp.n);
        // 更新路径
        paths[i][0] = position;
        // 更新禁忌表
        tabus[i][position] = true;
        // 遍历城市
        for (int j = 1; j < tsp.n; j++) {
            // 当前所在位置
```

```
                int currentPosition = paths[i][j - 1];
                // 计算权重
                double[] weightArray = new double[tsp.n];
                for (int k = 0; k < weightArray.length; k++) {
                    if (!tabus[i][k]) {
                        weightArray[k] = Math.pow(pheromoneMatrix[currentPosition][k], alpha) /
                                Math.pow(tsp.distanceMatrix[currentPosition][k], beta);
                    }
                    else {
                        weightArray[k] = 0;
                    }
                }
                // 选取下一个城市
                int nextPosition = Utils.roulette(weightArray, tabus[i]);
                // 更新路径
                paths[i][j] = nextPosition;
                // 更新禁忌表
                tabus[i][nextPosition] = true;
            }
            // 计算总路程
            double distance = tsp.calcDistance(paths[i]);
            // 释放信息素
            double pheromoneDelta = q / distance;
            for (int j = 0; j < tsp.n - 1; j++) {
                pheromoneDeltaMatrix[paths[i][j]][paths[i][j + 1]] += pheromoneDelta;
            }
            pheromoneDeltaMatrix[paths[i][tsp.n - 1]][paths[i][0]] += pheromoneDelta;
            // 若该蚂蚁找到较短的路径，则更新路程信息
            if (distance < currentMinDistance) {
                currentMinDistance = distance;
                currentMinPath = paths[i];
            }
        }
        // 更新信息素
```

```
            for (int i = 0; i < tsp.n; i++) {
                for (int j = 0; j < tsp.n; j++) {
                    // 信息素挥发
                    pheromoneMatrix[i][j] *= 1 - lambda;
                    // 信息素增加
                    pheromoneMatrix[i][j] += pheromoneDeltaMatrix[i][j];
                }
            }
            // 记录本次迭代路程信息
            solution.addPath(currentMinDistance, currentMinPath);
        }
        return solution;
    }
```

B 遗传算法核心代码

```
/**
 *  进化
 * */
public void evolution() {
    /**
     *  选择
     * */
    List<T> selectedPopulation = select(currentPopulation);

    /**
     *  交叉
     * */
    List<T> crossoverPopulation = crossover(selectedPopulation);

    /**
     *  变异，记录当前种群
     * */
    currentPopulation = mutate(crossoverPopulation);
```

```
        /**
         *  记录种群历史
         * */
        populationHistory.add(currentPopulation);
    }

    /**
     *  选择
     * */
    @SuppressWarnings("unchecked")
    private List<T> select(List<T> population) {
        List<T> resultPopulation = new ArrayList<>();
        // 计算适应度
        double[] fitnessArray = new double[populationSize];
        for (int i = 0; i < populationSize; i++) {
            fitnessArray[i] = population.get(i).calcFitness();
        }
        // 计算适应度均值
        double sum = 0;
        for (double fitness : fitnessArray) {
            sum += fitness;
        }
        double avg = sum / populationSize;
        // 计算适应度标准差
        double sqSum = 0;
        for (double fitness : fitnessArray) {
            sqSum += (fitness - avg) * (fitness - avg);
        }
        double sd = Math.sqrt(sqSum / populationSize);
        // 适应度尺度变换
        double fMin = avg - sd;
        for (int j = 0; j < populationSize; j++) {
            fitnessArray[j] = fitnessArray[j] > fMin ? fitnessArray[j] - fMin : 0;
```

```
        }

        // 选择个体
        for (int i = 0; i < populationSize; i++) {
            int index = Utils.roulette(fitnessArray);
            Individual individual = population.get(index);
            resultPopulation.add((T) individual.createIndividual(individual.getGenome()));
        }
        return resultPopulation;
    }

    /**
     *  交叉
     * */
    @SuppressWarnings("unchecked")
    private List<T> crossover(List<T> population) {
        List<T> resultPopulation = new ArrayList<>();
        // 随机打乱
        Collections.shuffle(population);
        // 两两配对
        for (int i = 0; i < populationSize / 2; i++) {
            Individual individual1 = population.get(i);
            Individual individual2 = population.get(populationSize - 1 - i);

            // 获取两者基因组
            List<boolean[]> genome1 = individual1.getGenome();
            List<boolean[]> genome2 = individual2.getGenome();

            List<boolean[]> crossGenome1 = new ArrayList<>();
            List<boolean[]> crossGenome2 = new ArrayList<>();
            assert genome1.size() == genome2.size();
            // 基因交叉
            for (int j = 0; j < genome1.size(); j++) {
                boolean[] gene1 = genome1.get(j);
```

```
                boolean[] gene2 = genome2.get(j);
                // 随机选取交叉点
                assert gene1.length == gene2.length;
                int geneLength = gene1.length;
                int index = (int) (Math.random() * geneLength);
                // 基因交叉互换
                boolean[] crossGene1 = new boolean[geneLength];
                boolean[] crossGene2 = new boolean[geneLength];
                for (int k = 0; k < geneLength; k++) {
                    if (k < index) {
                        crossGene1[k] = gene1[k];
                        crossGene2[k] = gene2[k];
                    }
                    else {
                        crossGene1[k] = gene2[k];
                        crossGene2[k] = gene1[k];
                    }
                }
                crossGenome1.add(crossGene1);
                crossGenome2.add(crossGene2);
            }

            resultPopulation.add((T) individual1.createIndividual(crossGenome1));
            resultPopulation.add((T) individual2.createIndividual(crossGenome2));
        }
        return resultPopulation;
    }

    /**
     *  变异
     * */
    @SuppressWarnings("unchecked")
    private List<T> mutate(List<T> currentPopulation) {
        return currentPopulation.stream().map(
```

```
                individual -> (T) individual.createIndividual(individual.getGenome().stream().map(gene -> {
                    boolean[] mutateGene = new boolean[gene.length];
                    if (gene.length > 0) {
                        mutateGene[0] = gene[0];
                        for (int i = 1; i < gene.length; i++) {
                            mutateGene[i] = Math.random() < P_MUTATE ? (!gene[i]) : gene[i];
                        }
                    }
                    return mutateGene;
                }).collect(Collectors.toList()))
        ).collect(Collectors.toList());
    }
```