第 5 章

CHAPTER 5

智能优化算法

优化问题广泛地存在于信号处理、生产调度、任务分配、模式识别、图像处理、自动控制等众多领域。在电子、通信、计算机、经济学和管理学等众多学科中，不断地出现了许多复杂的组合优化问题。面对这些大型的优化问题，传统的优化算法往往需要遍历整个搜索空间，无法在短时间内完成搜索，且容易产生“组合爆炸”问题。

受人类智能、生物智能及自然现象等启发，人们通过模拟自然规律、生物群体的智能行为等发明了很多智能优化算法来解决上述复杂的优化问题，如模拟自然界生物进化的遗传算法、模拟鸟群捕食的粒子群优化算法和模拟蚂蚁觅食的蚁群算法等。本章主要学习一些经典的智能优化算法，并通过实例介绍这些算法的实现过程和具体应用。主要内容如下。

第 21 集
微课视频

5.1　优化算法

了解关于优化问题的一般概念以及优化算法的发展、分类及应用。

5.2　遗传算法

理解模拟生物自然进化过程的遗传算法的算法思想，掌握遗传算法流程及实现过程，能够运用遗传算法求解一般的优化问题，能够分析算法的优缺点和影响算法性能的主要因素，能够尝试对算法做一些改进。

5.3　群智能算法

了解群智能算法的核心思想，了解常见群智能算法的种类。

5.4　粒子群优化算法

理解粒子群优化算法的基本思想，掌握算法的实现过程。能够运用粒子群优化算法求解一般的优化问题，并分析算法的优缺点和影响算法性能的主要因素。

5.5　蚁群算法

理解群智能算法中蚁群算法的基本思想，掌握其流程和实现过程，并能够运用蚁群算法求解经典的旅行商问题，能够分析算法的优缺点和影响算法性能的主要因素。

5.1　优化算法

优化算法是计算机科学和数学领域中的一个重要分支，是一类用于在给定约束条件下寻找最优解（或近似最优解）的方法和策略。优化算法具有简单、通用、便于并行处理等特点，能够提高系统效率，降低能耗，合理利用资源，并且随着处理对象规模的增大，这种效果

也会更加明显。

鉴于许多实际工程问题的复杂性、非线性、约束性以及建模困难等诸多因素，寻求高效的优化算法成为工程领域的主要研究内容之一。

5.1.1 优化算法的产生与发展

最早的优化问题可以追溯到古代的农业、建筑和战争中的资源分配问题。17 世纪，牛顿和莱布尼茨的微积分理论为优化算法提供了理论基础。

1. 线性规划

20 世纪 30 年代，学者们开始研究线性规划问题，其目标函数和约束条件都是线性的。1947 年，George Dantzig 提出的单纯形算法是解决线性规划问题的第一个有效算法。

2. 非线性规划

随着问题复杂性的增加，非线性规划问题开始受到关注，其目标函数或约束条件是非线性的。20 世纪 50 年代，出现了多种非线性规划算法，如梯度下降法、牛顿法等。

3. 组合优化

20 世纪 60 年代，旅行商问题(travelling salesman problem，TSP)等组合优化问题开始被广泛研究。组合优化问题涉及从有限集合中选择元素以优化某个目标函数，这类问题普遍存在于运筹学和工程领域。

4. 启发式算法

随着问题规模的不断增大，精确算法在计算上变得不可行。20 世纪 70—80 年代，启发式算法逐步兴起，如禁忌搜索、遗传算法、模拟退火算法等，这些算法在求解组合优化问题时表现得更高效。

5. 群智能算法

20 世纪末，基于群体智能的群智能算法，如蚁群算法、粒子群优化算法等，受到广泛关注，通过个体之间的协作和信息交流解决大规模优化问题。

6. 机器学习与人工智能

21 世纪以来，随着机器学习和人工智能的兴起，深度学习的发展极大地推动了优化算法的研究。例如，随机梯度下降(stochastic gradient descent，SGD)及其变体(如 Adagrad、Adadelta 和 RMSProp 等)被广泛应用于神经网络的训练。此外，一些自适应优化算法(如 Adam 和 Nadam)也在深度学习中取得了较好的效果。

当前，优化算法正在与大数据、云计算等技术结合，以解决更大规模和更复杂的优化问题。量子计算的发展为优化算法提供了新的计算平台，量子优化算法正在成为研究的热点。总的来说，优化算法的发展是一个不断演进和创新的过程，随着计算能力的增强和新理论的出现，新的优化算法也会不断涌现以满足应用场景的多样化需求。

5.1.2 优化算法的分类与应用

优化算法是用于寻找问题最优解或近似最优解的一系列方法。根据它们的特性和应用领域，优化算法可以被分为不同的类别。

(1) 线性规划算法：①单纯形法(simplex method)；②内点法(interior-point method)；③对偶单纯形法(dual simplex method)。

(2) 非线性规划算法：①梯度下降法(gradient descent method)；②牛顿法(Newton's method)。

(3) 动态规划算法：①贝尔曼-福特算法(Bellman-Ford algorithm)；②动态规划表(dynamic programming table)。

(4) 局部搜索算法：①爬山法(hill climbing method)；②模拟退火(simulated annealing)；③禁忌搜索(tabu search)。

(5) 群智能算法：①粒子群优化(particle swarm optimization)算法；②蚁群优化(ant colony optimization)算法；③人工蜂群算法(artificial bee colony algorithm)。

(6) 约束优化算法：①拉格朗日乘数法(Lagrange multiplier method)；②惩罚函数法(penalty function method)。

(7) 启发式算法：①遗传算法(genetic algorithm)；②贪心算法(greedy algorithm)。

(8) 基于深度学习的优化算法：随机梯度下降算法(stochastic gradient descent algorithm)。

(9) 动量算法(momentum algorithm)。

优化算法的分类并不是互斥的，某些优化算法可能同时属于多个类别。例如，粒子群优化算法是一种群智能算法，也是一种启发式算法。优化算法的选择通常取决于问题的性质、规模、求解的精度以及计算资源的可用性等，在实际应用中，可能需要结合多种算法或对现有算法进行适当调整以适应特定问题的需求。

第 22 集
微课视频

优化算法被广泛应用于经济学、生物学、工程学、物理学等多个学科领域，解决资源分配、路径规划、调度优化等问题，帮助人们在复杂问题中作出更优决策或找到更好的解决方案。

第 23 集
微课视频

5.2 遗传算法

遗传算法(genetic algorithm，GA)是一种通过模拟生物自然进化过程搜索最优解的智能优化算法。“遗传算法”这一概念由 J. D. Bagley 在 1967 年首次提出。1975 年，美国密歇根大学的 J. H. Holland 教授出版了著名的 *Adaptation in Natural and Artificial Systems*，标志着遗传算法的诞生，开启了遗传算法理论和方法的系统性研究。1985 年，在美国召开了第一届遗传算法的国际会议并成立了国际遗传算法学会(International Society of Genetic Algorithm，ISGA)。1988 年，Holland 的学生 D. J. Goldberg 出版了 *Genetic Algorithms in Search*，*Optimization*，*and Machine Learning*，对遗传算法及其应用作了全面而系统的论述。随后，遗传算法在多目标优化、生产调度、自动控制、机器人学、人工生命等方面得到了广泛的应用。

5.2.1 遗传算法的基本思想

遗传算法是基于达尔文的生物进化论的自然选择和遗传学机理的生物进化过程而提出的。

达尔文的生物进化论中最重要的是优胜劣汰，适者生存原理。物种的子代会继承父代的特征，但也会产生一些异于父代的新变化，当环境发生改变时，能适应环境的个体被保留，不能适应的则被淘汰。这种自然选择是长期的、缓慢的、连续的过程。

遗传学机理中最重要的是基因遗传原理。它认为遗传物质以基因形式包含在染色体内。每个基因都在其特殊的位置控制某种特征。因此，不同基因产生的个体对环境的某种适应性也不尽相同。基因杂交和基因突变可能产生更适应于环境的子代。经过优胜劣汰的自然选择，适应性高的基因能够延续并保留下来。

基于这些基本思想，遗传算法将问题的解表示成“染色体”，在算法实现中即以某种方法编码的串。首先，给出一群“染色体”（即搜索空间中的一组假设解）作为初始种群。然后，把这些假设解置于问题的“环境”中，按适者生存的原则从中选择出能较好适应环境的“染色体”进行复制，再通过交叉，变异过程产生能更好地适应环境的新一代“染色体”群。经过一代一代的进化（即算法过程的迭代），算法就会收敛到最适应环境的一个“染色体”上，它就是问题的最优解。表 5-1 给出了生物进化和遗传过程中的概念与遗传算法中要素的类比关系。

表 5-1 生物进化与遗传算法中各要素的类比关系

进化与遗传中的概念	遗传算法的要素
环境	适应度函数
种群	随机得到的初始解集合（解的个数即群体的规模）
个体	可行解
适应能力	解的适应度函数值
适者生存	适应值较大的解被选中的可能性大
染色体	解的编码串
基因	解的编码中的每个分量
婚配	交叉：选择两个染色体交换部分分量，产生一组新的染色体的过程
变异	编码的某一分量发生变化的过程
进化结束	算法终止

5.2.2 遗传算法的流程

遗传算法的流程如图 5-1 所示，主要包含以下主要步骤。

步骤 1：初始化种群。通常以随机方法从解空间中产生串或个体，构成初始解的集合。个体数量即为种群规模 N。问题的最优解将通过这些初始的假设解不断进化而得出。

步骤 2：适应度评价，计算种群中每个个体的适应度值。

步骤 3：选择。又称再生算子，依据某种方法（通常与适应度相关）从群体中选择优胜的个体，淘汰劣质个体的操作。常用的选择方法有适应度比例法、随机遍历抽样法、局部选择法等。

步骤 4：交叉。以概率 P_c 把两个父代染色体的部分分量加以替换重组而生成新个体。交叉算子在遗传算法中起到核心作用，通过交叉提升遗传算法的搜索能力。

步骤 5：变异。按概率 P_m 对某些染色体上的基因值进行扰动，使算法加速向最优解收敛。

步骤 6：终止条件判定，若不满足，则转至步骤 2，否则进入下一步。

步骤 7：输出种群中适应度值最优的染色体作为问题的满

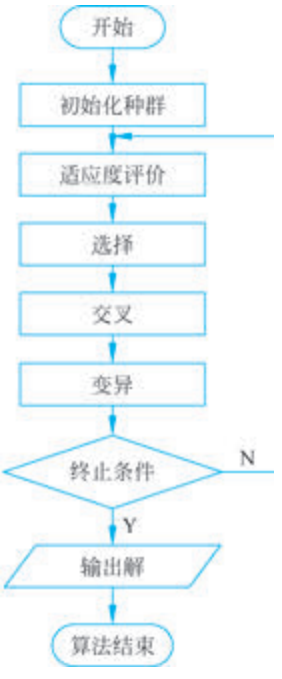

图 5-1 遗传算法的流程

意解或最优解。

5.2.3 遗传算法的实现过程

遗传算法搜索最优解的方法是模仿生物的进化过程，主要使用选择算子、交叉算子与变异算子模拟生物进化，从而产生一代又一代的种群。

根据遗传算法的思想和流程，在计算机上实现遗传算法时通常涉及编码机制、种群初始化、适应度函数、遗传算子（选择、交叉、变异）、控制参数等要素。

1. 编码机制

遗传算法中的编码是将问题的解空间映射到遗传空间的过程，即用特定的符号串将解空间中的解表示为遗传空间中的染色体的过程，如图 5-2 所示。

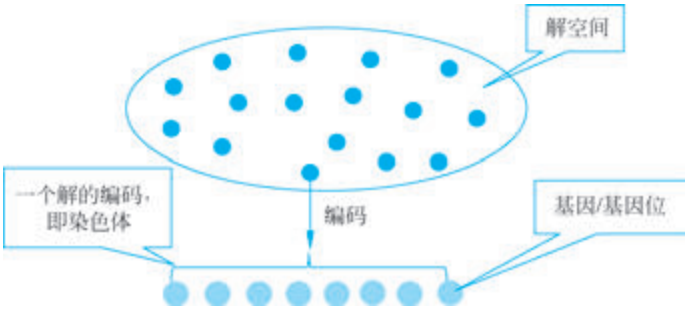

图 5-2 遗传算法中的编码

常见的编码方式主要有以下几种。

（1）二进制编码：最常见的编码方式之一，用 0 或 1 表示每位基因，将解的个体编码为二进制串，对应遗传空间中的染色体。这种编码方式简单直观，易于实现遗传操作，但在求解高维优化问题时，可能会因为编码串过长导致算法的搜索效率较低。

每个染色体是一个由 0 或 1 组成的编码串。假设解的取值范围为$[x_{\min}, x_{\max}]$，某个个体的编码是 $b_L b_{L-1} b_{L-2} \cdots b_2 b_1$，则对应的解码公式为

$$x = x_{\min} + \delta\left(\sum_{i=1}^{L} b_i 2^{i-1}\right) \tag{5-1}$$

$$\delta = \frac{x_{\max} - x_{\min}}{2^L - 1} \tag{5-2}$$

其中，串长 L 取决于求解精度。

（2）实数编码：直接采用实数编码，若干实数表示一个个体。使用这种编码方式，无须进行数制转换，可直接在解的表现型上进行遗传操作。这种编码方式适用于一些连续优化问题，但可能需要特殊的遗传操作。

（3）整数编码：将个体编码为整数串，每个整数表示一个基因。整数编码可以表示更广泛的数值范围，但需要更大的存储空间。

（4）符号编码：将问题的解表示为符号序列的编码方式。采用符号编码需要根据问题的特点和需求确定符号集，将问题的解转换为符号序列，每个符号代表解中的一个元素或特征。符号编码常用于序列排序、调度问题等。

除以上介绍的几种编码方式以外，遗传算法还有许多其他的编码方式，如多参数优化问

题中采用的多参数级联编码、模拟生物二倍体基因遗传的二倍体编码、格雷编码等。

合理的编码方式可以显著提高遗传算法的性能和效率。在实际应用中，要根据具体问题的特点和求解需求选择编码机制并结合适当的遗传操作和算法参数实现优化。

2. 种群初始化

遗传算法是对群体进行操作的，种群初始化是算法开始的第一步，它决定了算法搜索解空间的起点。通常采用随机的方法在解空间中生成一组解作为初始群体，如图 5-3 所示。

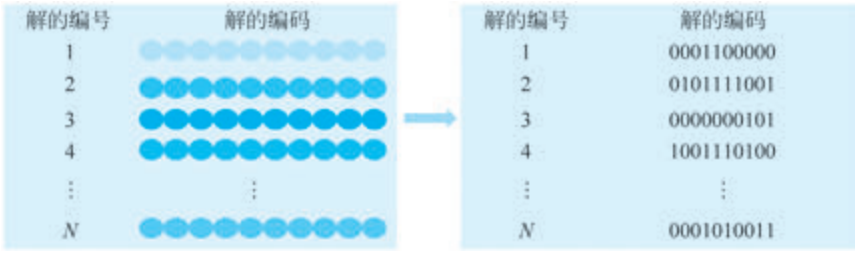

图 5-3 遗传算法中的种群初始化

为了使得算法具有更好的搜索性能，通常在初始化种群时需要考虑以下因素。

(1) **多样性**：确保初始种群中个体的多样性，避免算法过早收敛到局部最优解。

(2) **均匀分布**：理想情况下，初始种群中的个体应该在整个解空间中均匀分布。

(3) **种群规模**：种群中解的数量，通常取决于问题复杂度和计算资源。种群规模太小将导致搜索空间小，容易陷入局部最优解；种群规模太大，计算量也会随之增大，算法的搜索效率降低，同时可能会出现过拟合风险，一般取值为 20～100。

种群初始化的质量对遗传算法的性能有很大影响，需要在初始化过程中平衡随机性和多样性，良好的初始化种群有利于加速算法的收敛速度，并增大找到全局最优解的概率。

3. 适应度函数

在遗传算法中，适应度函数用来评估个体在问题求解中的优劣程度。运用适应度函数为每个个体计算一个适应度值，如表 5-2 所示，用于反映个体对于目标问题的满足程度。适应度值越高，表明个体越接近问题的最优解。

表 5-2 遗传算法中的适应度函数与适应度值

编　　号	个 体 编 码	解　　码	适 应 度 值
1	0001100000	X_1	$f(X_1)$
2	0101111001	X_2	$f(X_2)$
3	0000000101	X_3	$f(X_3)$
4	1001110100	X_4	$f(X_4)$
…	…	…	…
N	0001010011	X_N	$f(X_N)$

适应度函数的设计会直接影响遗传算法的性能和收敛速度。在一些简单的优化问题中，适应度函数可以直接是目标函数本身，如求解函数极值问题，而在求解复杂的优化问题时往往需要构造适应度函数以准确反映问题的目标和约束条件。在设计适应度函数时，通常要根据具体的实际问题考虑合理性、一致性、计算量等多方面因素。一个设计良好的适应度函数能够引导种群向更优解方向进化，帮助遗传算法有效地搜索解空间，找到满足问题要

求的最优解或近似最优解。

4. 遗传算子

1）选择

选择也称为复制操作，是指从当前群体中按照一定的概率选出优良的个体，用来产生下一代。基于个体的适应度计算其被选中的概率，通常，适应度越高的个体，被选中的概率也越大。常见的选择方法如下。

（1）轮盘赌选择：个体被选中的概率与其适应度值成正比。轮盘赌选择方法操作相对简单，在遗传算法中被广泛采用。其主要思想是将个体的适应度按比例转换为选择概率，按个体的比例值将轮盘进行扇区划分（即轮盘每个扇区的角度与个体的选择概率成比例），每次转动轮盘并待轮盘停止后，依据指针落停的扇区来决定被选中的个体。很显然，个体的适应度值越高，其所占比例就越大，被选中的概率自然也就越大。

具体实现方法：先计算出种群中个体的适应度值，然后计算该个体适应度值在该种群中所占的比例作为该个体的选择概率或生存概率。

$$p(I_i)=\frac{f(I_i)}{\sum_{i=1}^{N} f(I_i)} \tag{5-3}$$

其中，$p(I_i)$ 为个体 I_i 的选择概率；$f(I_i)$ 为个体 I_i 的适应度值；N 为种群大小。

（2）锦标赛选择：从种群中随机选择 k 个个体进行比较，适应度值最高的个体被选中。这一过程被反复执行，直到下一代的种群规模达到要求为止。参数 k 称为竞赛规模。

（3）排序选择：首先对种群中的个体按照适应度值进行排序，然后将事先设计好的概率按排序分配给个体，作为各自的选择概率。选择概率仅取决于个体在种群中的序位，而不是实际的适应度值。

选择操作的目的是让适应度高的个体有更多机会参与繁殖，从而将优良的基因传递给下一代，逐步引导种群向更优的方向进化。它在遗传算法中起着重要的作用，影响着算法的收敛速度和最终求解的质量。

2）交叉

交叉操作通常基于一定的交叉概率 p_c，对选中的两个父代个体交换某些基因位，形成新的子代个体的过程，也称为重组，是产生新个体的主要手段，用以增加种群的多样性，从而促进算法的搜索和优化能力。交叉方式主要有单点交叉、两点交叉、多点交叉、均匀交叉等。

（1）单点交叉：又称为简单交叉，随机产生一个有效的交叉位置，交换位于该交叉位置后的所有基因形成两个新的个体，如图 5-4 所示。

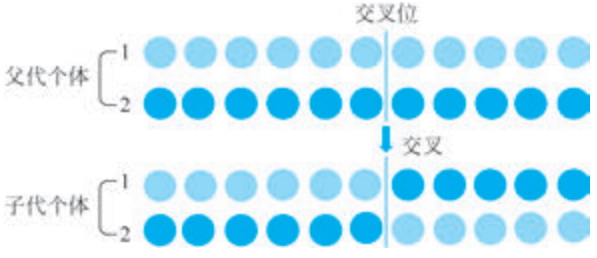

图 5-4　单点交叉

(2) 两点交叉：设置两个交叉点，将两个交叉点之间的串码相互交换。

(3) 多点交叉：原理与单点交叉类似，交叉点增加至多个，交换两个个体在所设定的多个交叉点之间的部分染色体。

(4) 均匀交叉：在父代个体的染色体上，对每个基因位都以一定的概率决定是否进行交换。

交叉操作的效果取决于交叉概率的设置、交叉方式的选择以及父代个体的特征。合适的交叉操作能够有效地引导遗传算法在解空间中进行搜索，提高找到最优解的可能性。

3) 变异

变异操作通常以一个较小的变异概率 p_m 发生，是一种对个体染色体上的基因进行随机改变的操作。通过变异提高种群的多样性，避免算法陷入局部最优，同时也有助于恢复种群中可能丢失的基因，维持种群基因库的丰富性。变异的方式主要包括位点变异、逆转变异、插入变异、高斯变异等。

(1) 位点变异：在个体的编码串中随机挑选一个或多个基因位，对这些基因位以变异概率 p_m 进行变动，如将二进制编码串中某一个基因位从 0 变为 1 或从 1 变为 0 如图 5-5 所示。

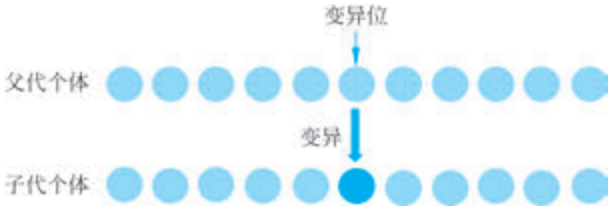

图 5-5 位点变异

(2) 逆转变异：在个体的编码串中随机选择两点作为逆转点，然后将两个逆转点之间的基因值以逆序插入原位置。

(3) 插入变异：在个体编码串中随机选择一个基因，然后将此基因插入随机选择的插入点形成新的个体。

(4) 高斯变异：高斯变异是改进的遗传算法中的一种变异方法。在进行变异操作时，用符合均值为 μ，方差为 σ^2 的正态分布的一个随机数来替换原基因值，即

$$x'_i = x_i + X_{\text{rand}} \tag{5-4}$$

其中，X_{rand} 是服从正态分布 $N(\mu,\sigma^2)$ 的随机数。

变异操作依赖变异概率，如果变异概率过高，可能会破坏已有的优良基因组合；如果变异概率过低，则可能无法有效地发挥其提高种群多样性的作用。变异操作是遗传算法中一个重要的补充机制，与选择和交叉等遗传算子协同工作，以提高算法搜索到全局最优解的能力。

5. 控制参数

遗传算法中的控制参数主要有种群规模(N)、编码长度(L)、交叉概率(p_c)、变异概率(p_m)。

1) 种群规模

种群规模(N)影响算法的搜索能力和运行效率。N 设置过大，搜索空间大，可以提高群体的多样性，从而提高算法的搜索能力，但同时增加算法的计算量，降低了算法的运行效

率。N 设置过小，虽然降低了计算量，但又限制了每次进化中群体包含更多较好染色体的能力。种群规模一般设置为 20～100。

2）**编码长度**

编码长度(L)会影响算法的计算量和交叉、变异操作的效果。L 的设置与优化问题密切相关，一般由问题定义的解的形式和所选的编码方法决定。一般情况下，二进制编码方法中，编码长度由解的取值范围和规定精度共同决定。实数编码方法中，编码长度与问题定义的解的维数相同。

3）**交叉概率**

交叉概率(p_c)决定了进化过程中参加交配的染色体数目。p_c 过大，会使具有高适应度的个体结构很快被破坏；p_c 过小，又会导致搜索过程缓慢，甚至停滞不前。其取值一般为 0.5～1，也可采用自适应方法调整交叉概率。

4）**变异概率**

变异概率(p_m)决定了进化过程中个体发生变异的数量，提高群体进化的多样性。一般地，p_m 的值不宜过大，因为变异对已搜索到的较优解有一定的破坏作用，如果 p_m 的值太大，可能会导致算法从目前处于的较好的搜索状态倒退回原来较差的状态。p_m 的取值一般为 0.001～0.1，也可采用自适应方法调整变异概率。

这些参数对算法的性能会产生较大的影响。上述给出的各参数取值只是大致的范围，在具体问题求解时，需要结合经验并通过大量的实验反复调整以确定合适的参数取值。

5.2.4 遗传算法应用实例

以下是使用遗传算法求函数 $f(x)=3x^2-6x+2, x\in[-5,5]$ 最小值的具体应用实例。

步骤 1：根据问题定义适应度函数。

```
import random
def objective_function(x):
    return 3 * x * * 2 - 6 * x + 2
def fitness_function(x):
    return - objective_function(x)      # 因为是求最小值,所以取负值作为适应度
```

步骤 2：设置遗传算法参数。

```
POPULATION_SIZE = 50                    # 种群大小
GENE_LENGTH = 8                         # 基因长度,用于编码 x 的值
CROSSOVER_RATE = 0.8                    # 交叉概率
MUTATION_RATE = 0.05                    # 变异概率
NUM_GENERATIONS = 100                   # 迭代次数
```

步骤 3：初始化种群。

```
def initialize_population():
population = [ ]
    for _ in range(POPULATION_SIZE):
        individual = [random.randint(0, 1) for _ in range(GENE_LENGTH)]
        population.append(individual)
    return population
```

这一步随机生成了一个包含 POPULATION_SIZE 个体的种群，每个个体由 GENE_LENGTH 个 0 或 1 组成。

步骤 4：解码个体基因。

```
def decode_individual(individual):
    decimal_value = 0
    for bit in individual:
        decimal_value = (decimal_value << 1) | bit
    min_value = -5 # 函数定义域下限
    max_value = 5 # 函数定义域上限
    scaled_value = min_value + decimal_value * (max_value - min_value) / (2 ** GENE_LENGTH - 1)
    return scaled_value
```

将个体的基因编码转换为对应的实数值 x。

步骤 5：选择操作。

```
def selection(population):
    fitness_values = [fitness_function(decode_individual(individual)) for individual in population]
    total_fitness = sum(fitness_values)
    probabilities = [fitness / total_fitness for fitness in fitness_values]
    selected_indices = random.choices(range(POPULATION_SIZE), weights=probabilities, k=POPULATION_SIZE)
    selected_population = [population[i] for i in selected_indices]
    return selected_population
```

根据个体的适应度计算选择概率，然后通过随机选择生成新的种群。适应度高的个体有更大的概率被选中。

步骤 6：交叉操作。以一定的概率对选中的父代个体进行交叉操作，生成子代个体。

```
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        crossover_point = random.randint(1, GENE_LENGTH - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
        return child1, child2
    else:
        return parent1, parent2
```

步骤 7：变异操作。以较小的概率对个体的基因进行变异。

```
def mutation(individual):
    for i in range(GENE_LENGTH):
        if random.random() < MUTATION_RATE:
            individual[i] = 1 - individual[i]
    return individual
```

步骤 8：运行遗传算法。

```
def genetic_algorithm():
    population = initialize_population()
    for generation in range(NUM_GENERATIONS):
```

```
        selected_population = selection(population)
        new_population = []
        for i in range(0, POPULATION_SIZE, 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1]
            child1, child2 = crossover(parent1, parent2)
            child1 = mutation(child1)
            child2 = mutation(child2)
            new_population.append(child1)
            new_population.append(child2)
        population = new_population
    best_individual = max(population, key=lambda x: fitness_function(decode_individual(x)))
    best_value = decode_individual(best_individual)
```

步骤 9：输出结果。

```
print("Best solution: x = ", best_value, "Function value:", objective_function(best_value))
genetic_algorithm()
```

在每次迭代中，依次进行选择、交叉和变异操作，生成新的种群。经过多次迭代，最终找到适应度最高（即函数值最小）的个体，并输出其对应的 x 变量值和函数值。

5.2.5 遗传算法的特点与改进

1. 遗传算法的特点

作为一种模拟自然选择和遗传学原理的搜索算法，与传统的其他优化算法相比，遗传算法具有以下特点。

（1）**全局搜索能力**。算法从问题解的串集开始搜索，而不是从单个解开始。这是遗传算法与传统优化算法的显著区别。从单个初始值迭代求最优解的传统优化算法容易误入局部最优解，遗传算法的初始解是一个集合，覆盖面大，有利于全局择优。

（2）**隐含的并行性**。遗传算法中的一些操作可以同时处理种群中的多个个体，具有内在的并行性，这有助于提高算法的效率。

（3）**鲁棒性**。遗传算法对初始种群不敏感，在不同的初始条件下都有可能收敛到较好的解，能够处理非线性、多峰等多种类型的优化问题。

（4）**易实现**。遗传算法求解时使用特定问题的信息极少，只需适应值和串编码等通用信息，操作相对简单，易于编程实现。

（5）**自适应性**。可以根据适应度函数自适应地调整搜索方向和范围，适应度高的个体有更大的机会参与遗传操作，从而引导算法向更优的解进化。

2. 遗传算法的局限性

遗传算法在解决优化问题时具备一些优势，但同时也存在一些局限性。

（1）**参数选择的敏感性**。遗传算法中的关键参数，如交叉概率、变异概率和种群大小，对算法的性能有较大影响，需要通过大量的实验和经验确定合适的参数值，可能导致算法性能的不确定性。

（2）**过早收敛**。在实际应用中，遗传算法容易产生早熟收敛的问题，即算法过早地集中在一个解的局部邻域内，而无法探索到更优的全局解。

(3) **较高的计算复杂度和时间成本**。在处理大规模或高维度优化问题时,评估个体的适应度和遗传操作过程需要耗费较多的计算资源和时间成本。

(4) **较弱的局部搜索能力**。相对于一些专门的局部搜索算法,遗传算法的局部搜索能力相对较弱,可能导致在求解大规模问题的进化后期搜索效率降低。

(5) **适应度评估的依赖性**。适应度函数的设计对于算法的效果至关重要,如果适应度评估不准确或不恰当,可能导致算法的错误引导。

3. 遗传算法的改进

为了克服基本遗传算法的缺点和局限性,在实际应用时需要根据具体问题对算法进行相应的调整或改进。随着算法应用和研究的进一步深入,出现了很多改进的遗传算法。

1) 自适应遗传算法

从本质上讲,遗传算法是一种自适应的动态搜索过程,而传统的遗传算法中,固定取值的交叉概率 p_c 和变异概率 p_m 参数值会限制其进化性能。自适应遗传算法的主要思想是在算法的不同阶段,根据种群的特性和个体的适应度等因素动态调整 p_c 和 p_m 的值。

例如,一种常见的自适应调整策略是根据种群的最大适应度值(F_{max})与平均适应度值(F_{avg})的关系调整 p_c 和 p_m。

$$p_c = k_1 \frac{F_{max} - F'}{F_{max} - F_{avg}} \tag{5-5}$$

$$p_m = k_2 \frac{F_{max} - F}{F_{max} - F_{avg}} \tag{5-6}$$

其中,F'是两个交叉个体中较大的个体适应度值;F 是变异个体的适应度值;k_1 和 k_2 是系数,通常小于 1.0。

自适应遗传算法可以更好地平衡全局搜索和局部搜索,提高算法的性能和效率,避免早熟收敛,提高找到更优解的可能性,但需要合理设计适应度函数和调整策略,并且不同的问题可能需要不同的自适应方式。

2) 双种群遗传算法

在基本遗传算法中,经过若干次迭代后,种群的特性就不会再有很大的变化,为了打破这种群内的平衡,可以建立多个种群,同时进化,交换多个种群之间的优秀个体的基因,有利于算法跳出局部最优解,提高算法的搜索性能。双种群遗传算法将种群分为两个子群体,每个子群体都有自己的进化过程,同时它们之间会进行信息交流。

两个子种群采用不同的进化策略或参数,有助于保持种群的多样性,避免算法过早收敛。例如,在一个子种群中可以使用较高的交叉概率和变异概率,以加强全局搜索能力,而在另一个子种群中使用较低的交叉概率和变异概率,侧重于局部搜索和微调。

双种群遗传算法在处理复杂问题时表现出色,已被广泛应用于各种优化问题,如函数优化、生产调度、路径规划等。

3) 双倍体遗传算法

双倍体遗传算法中的每个个体具有两条染色体,即双倍体结构。在该算法中,每条染色体都包含了基因信息,但在表达和遗传操作时有其独特的方式。通常,其中一条染色体处于显性状态,决定个体的表现性,而另一条染色体处于隐性状态,隐性染色体中可能包含了在当前环境下未表达但在未来可能有用的基因组合,从而丰富了种群的基因多样性。

双倍体遗传算法存在更多潜在的基因组合，能更好地应对环境变化和噪声干扰，降低了算法过早陷入局部最优解的可能性，但算法较复杂，计算复杂度也相对较高。

5.3 群智能算法

群智能(swarm intelligence，SI)算法是一类源于对自然界中生物群体智能行为的观察和模拟的启发式算法。其核心思想是利用个体之间的简单信息交互和协作，在没有集中控制和全局信息的情况下，利用简单的规则和行为产生复杂的智能优化效果，以在解空间中找到最优或近似最优解。群智能算法具有分布式控制、自组织和适应性等特点。常见的群智能算法主要有以下几种。

(1) 粒子群优化(particle swarm optimization，PSO)算法：模拟鸟群觅食的群体行为，每个粒子代表一个解，通过跟踪个体最优和全局最优位置来更新解。

(2) 蚁群优化(ant colony optimization，ACO)算法：模拟蚂蚁在寻找食物过程中释放信息素并根据信息素浓度选择路径的行为，多用于解决组合优化问题，如旅行商问题、车辆路径规划问题等。


第 24 集
微课视频

(3) 人工蜂群(artificial bee colony，ABC)算法：模拟蜜蜂的采蜜行为，模拟引领蜂、跟随蜂和侦察蜂的分工协作，实现优化的智能算法。

(4) 人工鱼群算法(artificial fish swarm algorithm，AFSA)：通过模拟鱼类的觅食、聚群、追尾、随机等行为的新型仿生优化算法。


第 25 集
微课视频

(5) 细菌觅食算法(bacterial foraging algorithm，BFA)：模拟细菌在环境中的觅食行为，是一种通过趋化、复制和驱散三种行为实现寻优的新型群体智能优化算法。

(6) 火烈鸟搜索算法(flamingo search algorithm，FSA)：模拟火烈鸟群体合作觅食的新型智能优化算法。

群智能算法在求解规模大、解空间复杂的优化问题时具有独特的优势，被广泛应用于函数优化、神经网络训练、路径规划、物流调度等领域。例如，蚁群优化算法被用来规划最优的物流配送路线；粒子群优化算法被用来优化神经网络训练中的网络参数。

5.4 粒子群优化算法

受人工生命研究结果的启发，Kennedy 和 Eberhart 博士于 1995 年在模拟鸟群觅食过程中的迁徙和群集行为时提出了一种基于群体智能的进化计算技术，即粒子群优化(PSO)算法。与遗传算法类似，粒子群优化算法也是从随机解出发，通过迭代寻找最优解，但是它比遗传算法的规则更简单，没有遗传算法的选择、交叉和变异操作，而是通过追随当前搜索到的最优值寻找全局最优。

5.4.1 粒子群优化算法的基本思想

粒子群优化算法的基本思想源于对鸟群觅食行为的模拟。假设一个场景：一群鸟在寻找食物，在远处有一块田有非常多的食物，所有鸟都不知道那块田的位置，但是它们知道自己当前距离那块田有多远，那么找到那块田的最优策略就是搜寻目前距离那块田最近的鸟

群所在位置。粒子群优化算法正是从鸟群觅食行为中得到启示形成的一种优化方法。

粒子群优化算法将每个潜在的解类比为搜索空间中的一只鸟，称为“粒子”，每个粒子都有自己的位置和速度，位置代表问题的一个可能解，问题的最优解就对应为鸟群要寻找的那块田。每个粒子设定一个初始位置和速度向量，根据目标函数计算当前所在位置的**适应度值**(fitness value)，可以将其理解为距离那块田的距离。每个粒子根据自身的历史最优位置和整个粒子群的全局最优位置不断调整自己的速度和位置，逐渐向最优解靠近。通过这种方式，粒子群中的粒子在解空间中进行协同搜索，最终找到问题的最优解或近似最优解。在每次迭代中，粒子根据两种“经验”更新自己的速度和位置：①自身的历史最优位置(个体最优解 P_{best})，即该粒子在之前的迭代中所达到的最优解；②整个粒子群所发现的最优位置(全局最优解 G_{best})，即整个粒子群在之前迭代中找到的最优解。

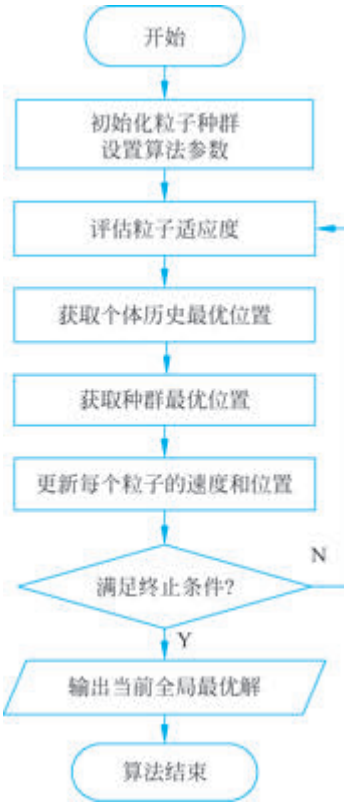

图 5-6 粒子群优化算法流程

5.4.2 粒子群优化算法的流程

粒子群优化算法流程如图 5-6 所示。

步骤 1：初始化粒子群并设置算法相关参数(粒子数量 N、速度范围 $v_{\max}$、惯性权重 w、学习因子 c_1 和 c_2)。

步骤 2：计算每个粒子位置的适应度值，并将其适应度值与其经过的最好位置 P_i 作比较，如果比 P_i 更优，则将其作为该粒子当前的最好位置 P_{best}。

步骤 3：根据各个粒子的历史最优位置 P_i 找到群体历史最优位置 G_{best}。

步骤 4：更新每个粒子的速度，并将其限制在 $v_{\max}$ 内，更新当前的位置。

步骤 5：判断是否满足终止条件。若满足，输出当前历史最优位置；否则返回步骤 2。

5.4.3 粒子群优化算法的实现过程

模拟鸟群觅食的行为粒子群优化算法以其简洁而有效的实现过程为解决复杂的优化问题提供了有效途径。

1. 粒子群初始化

确定具体问题搜索空间的维度、范围以及粒子数量(N)，并随机初始化每个粒子的位置和速度。位置表示解的可能取值，速度决定了粒子在搜索空间中的移动方向和步长。

粒子数量(N)相当于遗传算法的种群规模，会影响算法的搜索能力和收敛速度。一般来说，粒子数量越多，算法的搜索范围越广，但计算成本相应也越高。从经验上看，N 通常取 20～40，对复杂或特定问题可以取 100～200。

惯性权重(w)控制粒子对先前速度的继承程度，与物理中的惯性相似，取值范围为[0,1]。如果 w 过小，历史运动状态对当前运动影响较小，粒子的速度能够很快改变。如果

w 取值较大，虽然提高了搜索空间的范围，但离子运动方向不易改变，难以向较优位置收敛。也就是说，较大的 w 有利于全局搜索，较小的 w 有利于局部搜索。

学习因子(c_1 和 c_2)分别调节粒子向个体最优和全局最优学习的步长。学习因子过低会使粒子在目标区域外徘徊，过高则会导致粒子越过目标区域。从经验来看，一般地，c_1 和 c_2 均取值为 2.0 时，算法具有较好的收敛效果。

粒子最大速度($v_{\max}$)决定当前位置与最优位置之间区域的精度，通常设置为粒子的范围宽度。若设置过大，则粒子可能跨过最优位置；若设置过小，则容易陷入局部最优。限制最大速度可以有效防止计算溢出。

最大迭代次数($G_{\max}$)一般需要根据具体的工程问题而决定。

2. 适应度评估

根据问题的目标函数计算每个粒子的适应度值。适应度值反映了粒子所代表的解的优劣程度。

3. 更新个体最优解和全局最优解

个体最优解：对于每个粒子，将其当前适应度值与自身历史最优适应度值进行比较，如果当前的适应度值更优，则更新个体最优解为当前位置。

全局最优解：在所有粒子的个体最优位置中找出适应度值最好的位置作为全局最优解。

4. 更新个体的速度和位置

根据以下公式更新每个粒子的速度。

$$v_{id}^{t+1} = wv_{id}^{t} + c_1 r_1 (p_{id}^{t} - x_{id}^{t}) + c_2 r_2 (p_{gd}^{t} - x_{id}^{t}) \tag{5-7}$$

其中，v_{id}^{t+1} 是粒子 i 在第 d 维在第 t 次迭代的速度；w 是惯性权重；c_1 和 c_2 是学习因子；r_1 和 r_2 是在 $[0,1]$ 内的随机数；p_{id}^{t} 是粒子 i 在第 d 维的个体最优位置；p_{gd}^{t} 是全局最优位置；x_{id}^{t} 是粒子 i 在第 d 维在第 t 次迭代的位置。同时，还要对更新后的速度进行限制，确保其不超出 $v_{\max}$。

更新粒子的位置：

$$x_{id}^{t+1} = x_{id}^{t} + v_{id}^{t+1} \tag{5-8}$$

5. 终止条件判定

重复上述步骤，直到满足预设的终止条件。终止条件可以是达到最大迭代次数、适应度值达到一定精度，或者在连续若干次迭代中适应度值不再有明显改善等。

5.4.4 粒子群优化算法的应用实例

以下是一个应用粒子群优化算法求解函数 $f(x)=x^2-5x+6, x\in[-10,10]$ 最小值的具体应用实例。

(1) 定义目标函数。

```
import random
def objective_function(x): return x * * 2 - 5 * x + 6
```

(2) 设置算法相关参数并初始化粒子群。

```
num_particles = 50          # 粒子数量
max_iterations = 100        # 最大迭代次数
w = 0.5                     # 惯性权重
```

```
c1 = 1.5          # 学习因子 1
c2 = 1.5          # 学习因子 2
# 初始化粒子的位置和速度
particles_position = [random.uniform(-10, 10) for _ in range(num_particles)]
particles_velocity = [random.uniform(-1, 1) for _ in range(num_particles)]
# 初始化个体最优位置、全局最优位置、全局最优适应度
personal_best_position = particles_position.copy()
global_best_position = particles_position[0]
global_best_fitness = objective_function(global_best_position)
```

(3) 更新迭代。

```
for iteration in range(max_iterations):
    for i in range(num_particles):
        # 更新粒子的速度
        r1 = random.random()
        r2 = random.random()
        particles_velocity[i] = (w * particles_velocity[i] + c1 * r1 * (personal_best_
        position[i] - particles_position[i]) + c2 * r2 * (global_best_position - particles_
        position[i]))
        # 更新粒子的位置
        particles_position[i] += particles_velocity[i]
        # 计算当前粒子的适应度
        current_fitness = objective_function(particles_position[i])
        # 更新个体最优位置
        if current_fitness < objective_function(personal_best_position[i]):
            personal_best_position[i] = particles_position[i]
        # 更新全局最优位置
        if current_fitness < global_best_fitness:
            global_best_fitness = current_fitness
            global_best_position = particles_position[i]
```

(4) 输出全局最优解。

```
print("全局最优解:", global_best_position)
print("全局最优适应度值:", global_best_fitness)
```

5.4.5 粒子群优化算法的改进

粒子群优化算法收敛速度较快，但搜索精度不高，对于解决需要高精度的问题可能无法取得令人满意的结果。同时，算法具有较强的参数依赖性，需要通过大量的实验确定合适的参数值，提高了使用的难度和复杂性。为了提高粒子群优化算法的性能，国内外学者主要从以下几个方面对粒子群优化算法做了一些改进。

1. 惯性权重的动态调整

惯性权重 w 可以在算法运行过程中动态变化，如在前期较大以增强全局搜索能力，在后期较小以加强局部搜索能力。常见的动态调整方式有线性递减、非线性递减、自适应惯性权重等。例如，线性递减权重策略为

$$w = w_{\max} - \frac{k(w_{\max} - w_{\min})}{K} \tag{5-9}$$

其中，w 表示惯性权重；$w_{\max}$ 表示惯性最大权重；$w_{\min}$ 表示惯性最小权重；k 表示当前迭代次数；K 表示最大迭代次数。这种方法中，惯性权重从最大值线性减小到最小值。

2. 学习因子的自适应调整

学习因子 c_1 和 c_2 可以根据迭代次数或粒子的性能进行自适应调整，以平衡粒子自身经验和群体经验的影响。例如，可采用非对称学习因子法，即在搜索初期采用较大的自身学习因子值 c_1 和较小的群体经验学习因子值 c_2，增加群内粒子的多样性。随着迭代次数的增加，使 c_1 线性递减，c_2 线性递增，从而提高粒子向全局最优的收敛能力。

$$c_1 = c_{1\mathrm{i}} + \frac{k(c_{1\mathrm{f}} - c_{1\mathrm{i}})}{K} \tag{5-10}$$

$$c_2 = c_{2\mathrm{i}} + \frac{k(c_{2\mathrm{f}} - c_{2\mathrm{i}})}{K} \tag{5-11}$$

其中，k 表示当前迭代次数；K 表示最大迭代次数；$c_{1\mathrm{i}}$、$c_{2\mathrm{i}}$ 分别为 c_1、c_2 的初始值；$c_{1\mathrm{f}}$、$c_{2\mathrm{f}}$ 分别为 c_1、c_2 的最终值。

3. 引入变异操作

对粒子的位置或速度进行随机变异，提高种群的多样性，避免算法过早收敛到局部最优。例如，每次迭代中在粒子更新之后，以一个较小的概率随机改变粒子某个维度的值。引入变异操作后，能够使算法更好地应对复杂的优化问题，提高算法的求解性能。

4. 多种群策略

将粒子群划分为多个子群，每个子群独立进化，子群之间可以定期交流信息，有助于探索不同的搜索空间。

第 26 集
微课视频

5. 与其他算法结合

例如，与模拟退火算法、遗传算法等结合，克服单独采用一种算法的缺点，充分发挥各自的优势。

6. 邻域拓扑结构的改进

改变粒子之间信息交流的方式和范围，如动态调整领域大小、采用混合式的领域拓扑结构等。

5.5 蚁群算法

M. Dorigo 等在观察蚂蚁觅食的过程中发现了规律，即蚁群可以在一定时间内探寻出到达食物源的最短路径。他们受自然界蚂蚁觅食行为的启发，于 1992 年提出了**蚁群优化**(ACO)算法(简称蚁群算法)，该算法最早被应用于求解**旅行商问题**(TSP)。

5.5.1 蚁群算法的基本思想

蚁群算法的基本思想来源于自然界蚂蚁觅食的活动过程。在觅食开始时，单只蚂蚁不具备足够高的智慧，运动路径杂乱无章，但在一定时间后，蚁群中的蚂蚁会聚集在最短路径上。昆虫学家在研究蚂蚁觅食时发现蚂蚁之间最重要的通信媒介就是它们在移动过程中释放的特有的分泌物——信息素。蚂蚁们对信息素具有感知能力，当一只孤立的蚂蚁随机移动时能检测到其他同伴所释放的信息素，并沿着该路径移动，同时又释放自身的信息素，从而增强了该路径上的信息素浓度。信息素的浓度大小表征路径的长短，浓度越高，则表示有越多蚂蚁经过此处，也表示距离食物越近。随着时间推移，这条距离较近的道路信息素浓度

越来越高，越多蚂蚁选择这条道路，如图 5-7 所示。在这种正反馈机制下，“最短”的道路最终会被找到，即最佳路径。

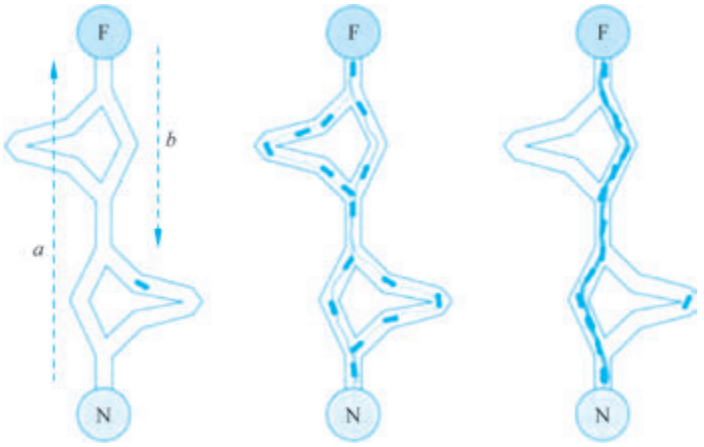

图 5-7 蚁群觅食行为示意图

用蚂蚁的行走路径表示待优化问题的可行解，整个蚂蚁群体的所有路径构成待优化问题的解空间。蚁群觅食行为与蚁群算法各要素的类比关系如表 5-3 所示。

表 5-3 蚁群觅食行为与蚁群算法各要素的类比关系

蚁 群 觅 食	蚁 群 算 法
从蚁巢到食物的行走路径	可行解
蚁巢到食物的最短路径	最优解
蚁巢到食物的所有路径	解空间
某路径上的信息素浓度	信息素矩阵
根据信息素选择路径	根据概率路径选择

5.5.2 蚁群算法的流程与实现过程

蚁群算法的一般流程如图 5-8 所示。运用蚁群算法求解优化问题的一般步骤如下。

步骤 1：初始化算法参数。

（1）设定蚂蚁数量(m)；

（2）设置信息素挥发系数(ρ)；

（3）确定信息素重要度(α)及启发式信息重要度(β)；

（4）设定最大迭代次数；

（5）初始化信息素矩阵：设置所有路径信息素浓度的初值，通常为一个较小的正数，确保所有路径都可能被搜索；

（6）放置蚂蚁：将所有蚂蚁放置在解空间的某个起始节点(一般为随机选择)。

步骤 2：构造解空间。

每只蚂蚁选择下一个要访问的节点。选择规则基于

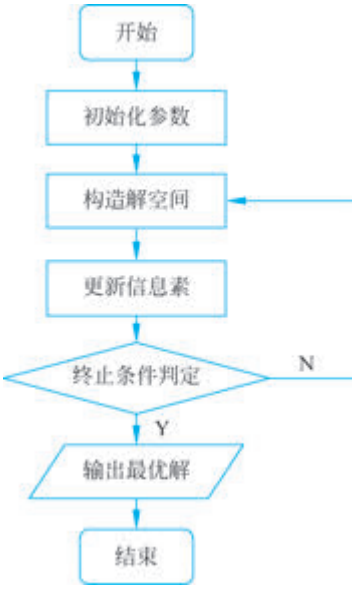

图 5-8 蚁群算法的一般流程

当前节点到其他未访问节点间的信息素浓度和启发式信息(如距离倒数)的综合考量。选择下一个节点的概率为

$$P(i \to j) = \frac{\tau_{ij}^{\alpha} d_{ij}^{-\beta}}{\sum_{k \in N(i)} (\tau_{ik}^{\alpha} d_{ik}^{-\beta})} \tag{5-12}$$

其中,τ_{ij} 是从节点 i 到节点 j 的信息素浓度;d_{ij} 是节点 i 到节点 j 的距离;α 和 β 是调整信息素和启发式信息影响力的参数;$N(i)$ 是节点 i 的邻居集合。

避免循环:确保蚂蚁不会重复访问同一节点,直到所有节点都被访问过一次。

结束条件:当蚂蚁构建了完整的路径(回到起点或满足特定条件),构造过程结束。

步骤 3:更新信息素。

(1) 信息素蒸发:所有路径上的信息素按照挥发系数 ρ 进行衰减(模拟自然界信息素挥发过程)。

(2) 信息素沉积:对于每只蚂蚁构建的路径,该路径上的信息素增加。通常,增加的量与路径长度的逆相关(即路径越短,增加的信息素越多)。

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij} \tag{5-13}$$

$$\Delta\tau_{ij} = \begin{cases} C_k^{-1}, & i,j \in \mathbb{R}^k \\ 0, & \text{其他} \end{cases} \tag{5-14}$$

其中,$\tau_{ij}(t+1)$ 为 $t+1$ 时刻的信息素浓度;$\tau_{ij}(t)$ 为 t 时刻的信息素浓度;ρ 为信息素挥发系数;$\Delta\tau_{ij}$ 为本次构建中增加的信息素;C_k 为路径长度,是 $\mathbb{R}^k$ 中所有边的长度之和。

步骤 4:判断是否满足终止条件(达到预定的迭代次数或满足其他终止条件,如解的质量不再显著提高),若满足,则进行下一步,否则返回步骤 2。

步骤 5:结果输出。在整个过程中记录并跟踪所有蚂蚁找到的最优路径,即全局最优解。最终输出找到的最优路径及其对应的总成本(如总距离)。

不同的蚁群算法实现可能会有一些细节上的差异,具体步骤和参数设置需要根据实际情况进行调整和优化。

5.5.3 蚁群算法的应用实例

假设五个城市之间的距离如表 5-4 所示,应用蚁群算法求解这五个城市的旅行商问题,即从某一城市出发,求解访问五个城市并回到起点的最短路径。

表 5-4 五个城市之间的距离

城市	距离				
	A	B	C	D	E
A	0	10	15	20	14
B	10	0	8	12	15
C	15	8	0	10	12
D	20	12	10	0	8
E	14	15	12	8	0

步骤 1：初始化算法参数。

```
import numpy as np
import random
num_cities = 5                              # 城市数量
num_ants = 10                               # 蚂蚁数量
alpha = 1                                   # 信息素重要程度
beta = 2                                    # 启发信息重要程度
rho = 0.5                                   # 信息素挥发系数
iter_num = 100                              # 迭代次数
distance_matrix = np.array([[0, 10, 15, 20, 14], [10, 0, 8, 12, 15], [15, 8, 0, 10, 12], [20,
12,10, 0, 8], [14, 15, 12, 8, 0]])          # 五个城市的距离矩阵
# 初始化信息素矩阵
pheromone_matrix = np.ones((num_cities, num_cities))
#计算路径的总距离
def cal_path_distance(path):
    distance = 0
for i in range(len(path) - 1):
    distance += distance_matrix[path[i]][path[i + 1]]
    distance += distance_matrix[path[-1]][path[0]]
    return distance
for iter in range(iter_num):
    paths = []
```

步骤 2：每只蚂蚁构建路径。

```
for ant in range(num_ants):
    visited = [False] * num_cities
    current_city = random.randint(0, num_cities - 1)
    visited[current_city] = True
    path = [current_city]
    while False in visited:
        unvisited = [i for i in range(num_cities) if not visited[i]]
        probabilities = []
        # 计算转移概率
        for next_city in unvisited:
            pheromone = pheromone_matrix[current_city][next_city]
            heuristic = 1 / distance_matrix[current_city][next_city]
            probability = (pheromone ** alpha) * (heuristic ** beta)
            probabilities.append(probability)
        sum_probabilities = sum(probabilities)
        probabilities = [p / sum_probabilities for p in probabilities]
    # 根据概率选择下一个城市
    next_city_index = np.random.choice(len(unvisited), p=probabilities)
    next_city = unvisited[next_city_index]
    visited[next_city] = True
    path.append(next_city)
    current_city = next_city
    paths.append(path)
```

步骤 3：更新信息素。

```
delta_pheromone_matrix = np.zeros((num_cities, num_cities))
for path in paths:
    distance = cal_path_distance(path)
for i in range(len(path) - 1):
```

```
        delta_pheromone_matrix[path[i]][path[i + 1]] += 1 / distance
        delta_pheromone_matrix[path[i + 1]][path[i]] += 1 / distance
    pheromone_matrix = (1 - rho) * pheromone_matrix + delta_pheromone_matrix
```

步骤 4：输出最优解。

```
best_path = None
best_distance = float(&amp;apos;inf&amp;apos;)
for path in paths:
    distance = cal_path_distance(path)
    if distance < best_distance:
        best_path = path
        best_distance = distance
print("最优路径:", best_path)
print("最优路径距离:", best_distance)
```

进行多次迭代，直到满足结束条件，可能得到的最优路径为 A→B→C→D→E→A，总距离为 50。

5.5.4 蚁群算法的改进

由于蚁群中的个体运动是随机的，当群体规模较大时，要找出一条较好的路径往往需要耗费较长的时间。在实际应用中，为提高蚁群算法的性能，往往需要根据具体问题对该算法进行调整，学者们主要从以下几个方面提出了一些改进策略。

1. 自适应参数调整

（1）动态调整信息素挥发系数。在算法运行初期，设置相对较大的挥发系数，让信息素快速扩散，扩大搜索空间，避免算法陷入局部最优；随着迭代次数增加，适当减小挥发系数，利于算法收敛。

（2）动态调整启发因子。根据算法迭代情况动态改变信息素重要程度因子和启发信息重要程度因子。例如，算法初期选择较大的启发信息因子，让蚂蚁更多依赖距离信息去探索，后期选择较大的信息素因子，让蚂蚁更多地依赖于前期积累的信息素进行探索。

2. 优化初始信息素

一般情况下，蚁群算法会将每条路径上的信息素浓度初始化为同一个值，即均匀初始信息素。导致在搜索前期有很大盲目性，无法引导蚂蚁朝着更优的路径去探索，初期搜索效率低。依据问题特性设置非均匀分布的初始信息素，让蚂蚁初始搜索就有一定倾向性，利于更快找到优解，提高算法搜索效率。

3. 融合其他算法

（1）与遗传算法融合。利用遗传算法的选择、交叉、变异操作处理蚁群算法产生的解，筛选出更优个体，增强全局搜索能力，克服蚁群易陷入局部最优的局限。

（2）与模拟退火算法的融合。借助模拟退火算法能以一定概率接受较差解跳出局部最优的特性，在蚁群算法的路径选择等环节按照一定概率参考模拟退火的这一机制，增大找到全局最优解的概率。

在实际应用中，需要根据具体问题对参数进行调优，以获得更好的求解效果。同时，为避免算法过早陷入局部最优，可以采用适当的设计策略，如精英保留策略、动态调整参数等。

另外，蚁群算法天然支持并行处理，每只蚂蚁的路径选择和信息素更新可以独立进行，这有助于提高算法的运行效率。

本章小结

本章主要介绍了智能优化算法的产生、发展和分类，详细介绍了进化计算中的遗传算法以及群智能算法中的粒子群优化算法和蚁群算法。这些智能优化算法已经被广泛应用在机器学习、智能控制、模式识别、网络安全等各个领域，在实际应用中，这些算法也在不断地改进和完善，以满足应用场景的多样化需求。优化算法的发展是一个不断演进和创新的过程，越来越强大的计算能力和新理论给智能优化算法带来了崭新的机遇，使智能算法呈现出融合化、自适应、多目标、并行化的发展趋势，也必将在更多的领域得到更广泛的应用。

本章习题

5-1 试阐述遗传算法中参数对算法性能的影响。

5-2 分别用遗传算法和粒子群优化算法编程求解以下函数的最大值。

$$f(x)=x_1{}^2-x_2,\quad x_i\in[-10,10],\quad i=1,2$$

5-3 目前对经典遗传算法的改进策略有哪些？

5-4 常见的群智能优化算法有哪些？

5-5 粒子群优化算法有哪些主要参数？试阐述每个参数对算法性能的影响。

5-6 请阐述蚁群算法的基本思想并分析其主要参数对算法性能的影响。

5-7 试给出蚁群算法的一种改进方案，并证明该方案在解决某类问题时的有效性。

5-8 除粒子群优化算法和蚁群算法外，你还了解哪些群智能算法？请阐述其主要思想。

5-9 试阐述智能优化算法未来的发展趋势。