第2章　基本遗传算法

本章从借助遗传算法优化一个简单函数的实例入手，分析遗传算法的基本特征。其次，我们将介绍和比较更多的遗传操作的算法。最后讨论一下遗传算法设计的若干问题。

2.1　简单函数优化的实例

考虑下列一元函数求最大值的优化问题：

$$f(x) = x\sin(10\pi \cdot x) + 2.0 \qquad x \in [-1,2] \tag{2.1}$$

该函数曲线如图2.1所示。由于$f(x)$在区间$[-1,2]$可微，我们首先用微分法求取$f(x)$的最大值。

$$f'(x) = \sin(10\pi \cdot x) + 10\pi \cdot x \cdot \cos(10\pi \cdot x) = 0 \tag{2.2}$$

即

$$\tan(10\pi \cdot x) = -10\pi \cdot x \tag{2.3}$$

上式的解有无穷多个：

$$\begin{cases} x_i = \dfrac{2i-1}{20} + \varepsilon_i, \quad i = 1,2,\cdots & (2.4) \\ x_0 = 0 & (2.5) \\ x_i = \dfrac{2i+1}{20} + \varepsilon_i, \quad i = -1,-2,\cdots & (2.6) \end{cases}$$

式中ε_i($i=1,2,\cdots$及$i=-1,-2,\cdots$)是一接近于0的实数递减序列。

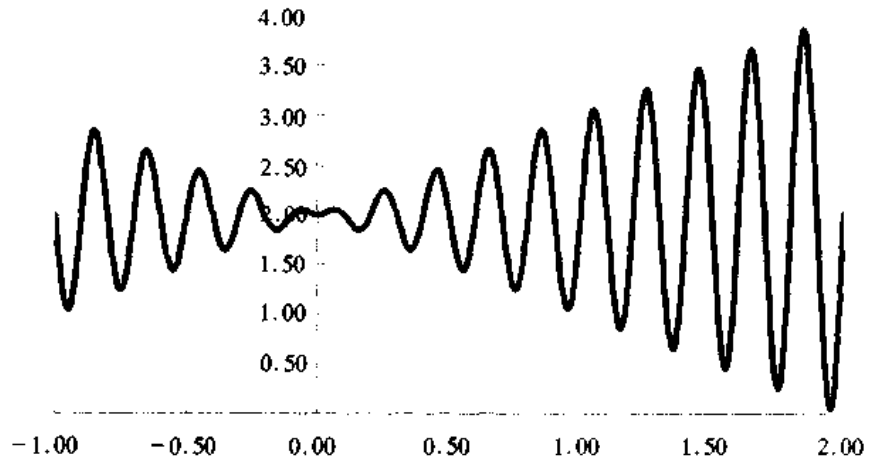

图2.1　函数$f(x)=x\sin(10\pi x)+2.0$的曲线

当i为奇数时x_i对应局部极大值点，i为偶数时x_i对应局部极小值点。显然x_{19}即为区间$[-1,2]$内的最大值点：

$$x_{19} = \frac{37}{20} + \varepsilon_{19} = 1.85 + \varepsilon_{19} \tag{2.7}$$

此时，函数最大值 $f(x_{19})$比 $f(1.85) = 3.85$ 稍大。

下面我们用遗传算法解决上述简单函数的最优化问题。

(1) **编码**　变量 x 作为实数，可以视为遗传算法的表现型形式。从表现型到基因型的映射称为编码。通常采用二进制编码形式，将某个变量值代表的个体表示为一个{0, 1}二进制串。当然，串长取决于求解的精度。如果设定求解精确到 6 位小数，由于区间长度为 2 - (- 1) = 3，必须将闭区间[- 1, 2]分为 3×10^6 等份。因为

$$2\ 097\ 152 = 2^{21} < 3 \times 10^6 \leqslant 2^{22} = 4\ 194\ 304$$

所以编码的二进制串长至少需要 22 位。

将一个二进制串$(b_{21}b_{20}\cdots b_0)$转化为区间[- 1, 2]内对应的实数值很简单，只需要采取以下两步：

① 将一个二进制串$(b_{21}b_{20}\cdots b_0)$代表的二进制数化为 10 进制数：

$$(b_{21}b_{20}\cdots b_0)_2 = (\sum_{i=0}^{21} b_i \cdot 2^i)_{10} = x' \tag{2.8}$$

② x'对应的区间[- 1, 2]内的实数：

$$x = -1.0 + x' \cdot \frac{2 - (-1)}{2^{22} - 1} \tag{2.9}$$

例如，一个二进制串 $s1 = <1000101110110101000111>$表示实数值 0.637 197。

$$x' = (1000101110110101000111)_2 = 2\ 288\ 967$$

$$x = -1.0 + 2\ 288\ 967 \cdot \frac{3}{2^{22} - 1} = 0.637\ 197$$

二进制串<0000000000000000000000>与<1111111111111111111111>，则分别表示区间的两个端点值 - 1 和 2。

(2) **产生初始种群**　一个个体由串长为 22 的随机产生的二进制串组成染色体的基因码，我们可以产生一定数目的个体组成种群，种群的大小(规模)就是指种群中的个体数目。

(3) **计算适应度**　对于个体的适应度计算，考虑到本例目标函数在定义域内均大于 0，而且是求函数最大值，所以直接引用目标函数作为适应度函数：

$$f(s) = f(x) \tag{2.10}$$

这里二进制串 s 对应变量 x 的值。

例如，有三个个体的二进制串为：

$$s_1 = <1000101110110101000111>$$

$$s_2 = <0000001110000000010000>$$

$$s_3 = <1110000000111111000101>$$

分别对应于变量值 $x_1 = 0.637\ 197$, $x_2 = -0.958\ 973$, $x_3 = 1.627\ 888$，个体的适应度计算如下：

$$f(s_1) = f(x_1) = 2.586\ 345$$

$$f(s_2) = f(x_2) = 1.078\ 878$$

$$f(s_3) = f(x_3) = 3.250\ 650$$

显然,三个个体中 s_3 的适应度最大,s_3 为最佳个体。

(4) **遗传操作** 这里介绍交叉和变异这两个遗传操作是如何工作的。

下面是经过选择操作(第 1 章中介绍过的轮盘赌选择方法)的两个个体,首先执行单点交叉,如

$$s_2 = < 00000 \mid 01110000000010000 >$$

$$s_3 = < 11100 \mid 00000111111000101 >$$

随机选择一个交叉点,例如第 5 位与第 6 位之间的位置,交叉后产生新的子个体:

$$s_2' = < 00000 \mid 00000111111000101 >$$

$$s_3' = < 11100 \mid 01110000000010000 >$$

这两个子个体的适应度分别为:

$$f(s_2') = f(-0.998\ 113) = 1.940\ 865$$

$$f(s_3') = f(1.666\ 028) = 3.459\ 245$$

我们注意到,个体s_3' 的适应度比其两个父个体的适应度高。

下面考察变异操作。假设已经以一小概率选择了 s_3 的第 5 个遗传因子(即第 5 位)变异,遗传因子由原来的 0 变为 1,产生新的个体为 $s_3' = <11101000001111111000101>$,计算该个体的适应度:$f(s_3') = f(1.721\ 638) = 0.917\ 743$,发现个体$s_3'$ 的适应度比其父个体的适应度减少了,但如果选择第 10 个遗传因子变异,产生新的个体为 $s_3'' = <11100000011111111000101>$,$f(s_3'') = f(1.630\ 818) = 3.343\ 555$,又发现个体$s_3''$ 的适应度比其父个体的适应度改善了。这说明了变异操作的"扰动"作用。

(5) **模拟结果** 设定种群大小为 50,交叉概率 $p_c = 0.25$,变异概率 $p_m = 0.01$,按照上述的基本遗传算法(Simple Genetic Algorithm, SGA),在运行到 89 代时获得最佳个体:

$$S_{max} = < 11010011111110011001111 >$$

$$x_{max} = 1.850\ 549, \qquad f(x_{max}) = 3.850\ 274$$

这个个体对应的解与微分法预计最优解的情况吻合,最大函数值比 3.85 略大,可以作为问题的近似最优解。表 2.1 列出了模拟世代的种群中最佳个体的演变情况。

表 2.1 模拟世代的种群中最佳个体的演变情况(150 代终止)

世代数	个体的二进制串	x	适应度
1	10001110000101100011111	1.831 624	3.534 806
4	00000110110001010011111	1.842 416	3.790 362
7	11101010111001110011111	1.854 860	3.833 280
11	01101010111001110011111	1.854 860	3.833 286
17	11101010111111101001111	1.847 536	3.842 004
18	00001110111111101001111	1.847 554	3.842 102
34	11000011011110110011111	1.853 290	3.843 402
40	11010010000100011001111	1.848 443	3.846 232
54	10001101101000110011111	1.848 699	3.847 155
71	01001101100001011001111	1.850 897	3.850 162
89	11010011111110011001111	1.850 549	3.850 274
150	11010011111110011001111	1.850 549	3.850 274

从上述简单函数优化的实例中可以看出，遗传算法是一种强大的随机搜索方法。由于数学函数优化问题不需要专门领域的知识，且能较好地反映算法本身的实际效能，所以常用于GA的测试问题。下面四个为具有相当复杂度的常用测试函数：

① Shubert 函数

$$f(x,y)=\left\{\sum_{i=1}^{5} i\cos[(i+1)x+1]\right\}\left\{\sum_{i=1}^{5} i\cos[(i+1)y+1]\right\}+0.5[(x+1.425\ 13)^2+(y+0.800\ 32)^2] \tag{2.11}$$

此函数有 760 个局部极小点，其中只有一个(－1.425 13，－0.800 32)为全局最小，最小值为 186.730 9。自变量取值范围 $-10<x,y<10$。此函数极易陷入局部极小值 186.34。

② Camel 函数

$$f(x,y)=(4-2.1x^2+\frac{x^4}{3})x^2+xy+(-4+4y^2)y^2 \tag{2.12}$$

此函数有 6 个局部极小点，其中有两个(－0.089 8，0.712 6)和(0.089 8，－0.712 6)为全局最小点，最小值为－1.031 628，自变量的取值范围为 $-100<x,y<100$。

③ De Jones's F_5(Shekel's Foxholes)函数

$$f_5(x_i)=0.02+\sum_{j=1}^{25}\frac{1}{j+\sum_{i=1}^{2}(x_i-a_{ij})^6} \tag{2.13}$$

其中$(a_{ij})_{2\times 25}=\begin{pmatrix}-32 & -16 & 0 & 16 & 32 & -32 & -16 & \cdots & 0 & 16 & 32\\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \cdots & 32 & 32 & 32\end{pmatrix}$。自变量取值范围为 $-65\ 536<x_i<65\ 536$。此函数有多个局部极大点，一般其函数值大于 1 即可认为收敛。

④ Shaffer's F_6 函数

$$f(x,y)=0.5-\frac{\sin^2\sqrt{x^2+y^2}-0.5}{(1+0.001(x^2+y^2))^2} \tag{2.14}$$

此函数有无限个局部极大点，其中只有一个(0，0)为全局最大，最大值为 1。自变量的取值范围为 $-100<x,y<100$。此函数最大值峰周围有一个圈脊，它们的取值均为 0.990 283，因此很容易停滞在此局部极大点。

除了以上四个函数之外，常用的测试函数还有 De Jones's $F_1\sim F_4$ 等。

2.2　遗传基因型

Holland 提出的遗传算法是采用二进制编码来表现个体的遗传基因型的，它使用的编码符号集由二进制符号 0 和 1 组成的，因此实际的遗传基因型是一个二进制符号串，其优点在于编码、解码操作简单，交叉、变异等遗传操作便于实现，而且便于利用模式定理进行理论分析等；其缺点在于，不便于反映所求问题的特定知识，对于一些连续函数的优化问题等，也由于遗传算法的随机特性而使得其局部搜索能力较差，对于一些多维、高精度要求的连续函数优化，二进制编码存在着连续函数离散化时的映射误差，个体编码串较短时，可能达不到精度要求；而

个体编码串的长度较长时，虽然能提高精度，但却会使算法的搜索空间急剧扩大。显然，如果个体编码串特别长时，会造成遗传算法的性能降低。后来许多学者对遗传算法的编码方法进行了多种改进，例如，为提高遗传算法局部搜索能力，提出了格雷码(Grey Code)编码；为改善遗传算法的计算复杂性、提高运算效率，提出了浮点数编码、符号编码方法等。为便于利用求解问题的专门知识，便于相关近似算法之间的混合使用，提出了符号编码法；此外还有多参数级联编码和交叉编码方法，近年来，随着生物计算理论研究的兴起，有人提出 DNA 编码法(详见 9.4 节)，并在模糊控制器优化的应用中取得很好的效果。

我们知道，遗传算法中进化过程是建立在编码机制基础上的，编码对于算法的性能如搜索能力和种群多样性等影响很大。就二进制编码和浮点数编码比较而言，一般二进制编码比浮点数编码搜索能力强，但浮点数编码比二进制编码在变异操作上能够保持更好的种群多样性。

下面就常用的二进制编码和浮点数编码的表示机制进行分析和比较。首先讨论一下标准的交叉操作的情况。

1. 浮点数编码

假设* 假定种群中个体数目为 n，x_t^i 表示第 t 代的第 i 个个体，$i \in \{1, 2, \cdots, n\}$。每个个体的基因位数 $L = m$，由 m 个实数构成，个体 $x_t^i \in \mathbf{R}^m$，x_t^i 可以表示 m 维的行向量，即 $x_t^i = (x_t^{i(1)}\ x_t^{i(2)}\ \cdots\ x_t^{i(m)})$。这样第 t 代的种群 $\boldsymbol{X}_t$ 可以表示为 $n \times m$ 的矩阵，$\boldsymbol{X}_t = (x_t^1\ x_t^2\ \cdots\ x_t^n)^{\mathrm{T}}$。初始种群矩阵 $\boldsymbol{X}_0 = (x_0^1\ \ x_0^2\ \ \cdots\ \ x_0^n)^{\mathrm{T}}$ 中没有相同的两行，而且每列中没有相同的元素，即种群中所有个体互异，对任意 $i \neq j$ $(i, j \in \{1, 2, \cdots, n\})$有 $x_0^i \neq x_0^j$；而且个体中没有两个个体的同一基因位是相同的，即对任意 $i \neq j$ $(i, j \in \{1, 2, \cdots, n\}, k \in \{1, 2, \cdots, m\})$有 $x_0^{i(k)} \neq x_0^{j(k)}$。

定理 2.1 在假设*的情况下，设初始种群经交叉操作产生的新个体属于可数集合 D，则集合 D 中的元素的数目 T_D 为

$$T_D = 2(m-1)C_n^2 \tag{2.15}$$

证明 因为种群的个体数目为 n，任意取两个个体进行交叉的可能情况为 C_n^2，又知有 $(m-1)$ 个交叉位置，由假设条件保证交叉后产生的两个新个体是不相同的，因此所有这些可能情况产生的个体构成集合 D，所以定理 2.1 成立。

定理 2.2 设两个互异的个体 x_t^i 和 x_t^j，且 $x_t^{i(k)} - x_t^{j(k)} = \delta$，那么它们交叉后产生的新个体 x'_t 满足

$$\left| x'^{(k)}_t - x_t^{p(k)} \right| = \delta \tag{2.16}$$

或者满足

$$\left| x'^{(k)}_t - x_t^{p(k)} \right| = 0 \tag{2.17}$$

其中 $i \neq j$ $(i, j \in \{1, 2, \cdots, n\}, k \in \{1, 2, \cdots, m\}, p \in \{i, j\})$。

证明 因为交叉后产生的新个体 x'_t 的第 k 个基因位 $x_t'^{(k)} \in \{x_t^{i(k)}, x_t^{j(k)}\}$，所以定理 2.2 显然成立。

定理 2.2 表明了对于任意两个互异的个体，交叉后产生的新个体一定在这两个父个体所构成的超体 $\prod_{k=1}^{m} [\min(x_i^{(k)}, x_j^{(k)}), \max(x_i^{(k)}, x_j^{(k)})]$ 的顶点上。

2. 二进制编码

假设种群中个体数目为 n，x_t^i 表示第 t 代的第 i 个个体，$i \in \{1, 2, \cdots, n\}$。每个个体用 l 位二进制表示。这样每个个体 $x_t^i \in \{IB\}^{ml}$，$IB \in \{0, 1\}$，这样每个个体基因位数目 $L = ml$。个体 x_t^i 可以表示为 ml 维的行向量，即 $x_t^i = [x_t^{i(1)} \cdots x_t^{i(l)} x_t^{i(l+1)} \cdots x_t^{i(2l)} \cdots x_t^{i((m-1)l+1)} \cdots x_t^{i(ml)}]$。第 t 代种群 $\boldsymbol{X}_t$ 可以表示为一个 $n \times ml$ 的矩阵 $\boldsymbol{X}_t = [\boldsymbol{x}_t^1\ \boldsymbol{x}_t^2\ \cdots\ \boldsymbol{x}_t^n]^{\mathrm{T}}$。个体 x_t^i 的第 k 个长度为 l 的二进制码串转化为实数的解码函数 Γ 为

$$\Gamma(x_t^i, k) = u_k + \frac{v_k - u_k}{2^l - 1}\left(\sum_{j=1}^{l} x_t^{i(kl+j)} \times 2^{j-1}\right) \tag{2.18}$$

式中 v_k 和 u_k 分别为第 k 个实数范围的上限和下限。

定理 2.3　设用二进制编码的初始种群 X_0 是假定*的浮点数编码的初始种群转化为相应二进制编码后的种群。设经交叉操作产生的新的个体属于可数集合 B，则集合 B 中元素的数目 T_B 满足：

$$T_B \geqslant 2n_0(n - n_0) + T_D \tag{2.19}$$

这里 n_0 表示种群中第一位二进制码为 0 的个体数目。

证明　当交叉位置为 $l, 2l, \cdots, (m-1)l$ 时，可能产生的不同的新的个体的数目与浮点数编码相同为 T_D，当交叉位置为 1 时，将 n 个个体分为第一位为 0 和第一位为 1 两组，分别记作 B_0 和 B_1，第一位为 0 的个体数目为 n_0，则第一位为 1 的个体数目为 $n - n_0$。那么容易证明从 B_0 中选出一个个体与从 B_1 中选出一个个体交叉产生的新个体的可能情况是 $2n_0(n - n_0)$，而且不与交叉位置 $l, 2l, \cdots, (m-1)l$ 的情况相重复。交叉位置还可以选其他位置，所以 $T_B \geqslant 2n_0(n - n_0) + T_D$ 成立。

定理 2.3 表明只要 $n_0 \neq 0$，或者 $n_0 \neq n$，用二进制编码交叉可产生的遍历搜索空间的不同个体的数目大于用浮点数编码的情况。也就是说，就交叉操作而言，二进制编码的搜索能力比浮点数的搜索能力强。由(2.19)式可以看出种群的个体数目越大，二进制编码的搜索空间的能力比浮点数的搜索能力强就体现得越充分。

定理 2.4　若两个个体 x_t^i，x_t^j，对一给定的 $k \in \{1, 2, \cdots, m\}$，满足

$$\Gamma(x_t^i, k) - \Gamma(x_t^j, k) = \frac{v_k - u_k}{2^l - 1}(2^p - 2^q) \tag{2.20}$$

其中 $p, q \in \{1, 2, \cdots, l-1\}$，$q > p$。则 x_t^i，x_t^j 经交叉后产生的两个新个体 $(x_t^i)'$，$(x_t^j)'$ 统称为 x_t'，满足

$$\Gamma(x_t^i, k) < \Gamma(x_t', k) < \Gamma(x_t^j, k) \tag{2.21}$$

的概率不为 0。(证明从略)

定理 2.5　若两个个体 x_t^i，x_t^j，对一给定的 $k \in \{1, 2, \cdots, m\}$，满足：

$$\Gamma(x_t^i, k) - \Gamma(x_t^j, k) = \frac{v_k - u_k}{2^l - 1}(2^p - 2^q) \tag{2.22}$$

其中 $p, q \in \{1, 2, \cdots, l-1\}$，$q > p$。则 x_t^i，x_t^j 经交叉后产生的两个新个体 $(x_t^i)'$，$(x_t^j)'$，满足

$$\Gamma(x_t^i, k) < \Gamma((x_t^i)', k) \tag{2.23}$$

和

$$\Gamma(x_t^j, k) < \Gamma((x_t^j)', k) \tag{2.24}$$

的概率不为 0(证明从略)。

由定理 2.4 和定理 2.5 可以看出，用二进制编码，交叉操作产生得到的新个体既可能在其对应的二进制数所构成的超体 $\prod_{k=1}^{m}[\min(\Gamma(x_t^i,k),\Gamma(x_t^j,k)),\max(\Gamma(x_t^i,k),\Gamma(x_t^j,k))]$ 的内部，也可能在其外部。而由浮点数编码由定理 2.2 可知，交叉后产生的新个体只能在其父个体构成的超体的顶点上。

通过以上分析，可以得出结论：在使用交叉操作时，二进制编码比浮点数编码产生新个体的可能情况多，而且产生的新个体不受父个体所构成的超体的限制。总之，二进制编码比浮点数编码的搜索能力强，而且随着种群大小的增大，这种差别就越明显。

下面我们分析一下变异操作的情况。

浮点数编码对于个体 x_t^i 的第 p 个基因进行变异操作 ψ_D 定义如下：

$$\psi_D(x_t^i,p)=x_t^{i(p)}+N(0,\delta) \tag{2.25}$$

其中 $N(0,\delta)$ 是均值为 0、方差为 δ 的高斯噪声。可见浮点数编码操作的变异量可以任意地小。这样可以保证只要变异量足够小，产生的新个体可以与父个体充分地接近。

对于二进制编码，对个体的 x_t^i 的第 p 个基因进行变异操作 ψ_B 定义如下：

$$\psi_B(x_t^i,p)=\begin{cases}0, & x_t^{i(p)}=1\\ 1, & x_t^{i(p)}=0\end{cases} \tag{2.26}$$

定理 2.6 对于任意给定的 $i\in\{1,2,\cdots,n\}$，$k\in\{1,2,\cdots,m\}$，二进制编码个体 x_t^i 所对应的第 k 个实数的变异最小量为 $\frac{v_k-u_k}{2^l-1}$。

定理 2.6 说明对于二进制编码，变异的最小量不能任意小，它受到编码长度的限制，l 越大，变异的最小量就越小。这样，即使在最优解附近，由变异操作也可能遍历不到最优解。

定理 2.7 对于任意给定 $i\in\{1,2,\cdots,n\}$，$k\in\{1,2,\cdots,m\}$，二进制编码个体 x_t^i 只变异一位，产生新个体 x'_t，设 $s^k=|\Gamma(x_t^i,k)-\Gamma(x'_t,k)|$，则 s^k 的最大值为：

$$s_{\max}^k=\frac{v_k-u_k}{2^l-1}\times 2^{l-1} \tag{2.27}$$

s^k 的最小值为：

$$s_{\min}^k=\frac{v_k-u_k}{2^l-1} \tag{2.28}$$

定理 2.7 说明了对二进制编码，变异操作不能保证父个体与新个体的充分接近。由定理 2.6 和 2.7 可以得出以下结论：二进制编码的种群稳定性比浮点数编码差。

理论上而言，编码应该适合要解决的问题，而不是简单的描述问题。Balakrishman 等较全面地讨论了不同编码方法的一组特性，针对一类特别的应用，为设计和选择编码方法提供了指南，主要有以下九个特性：

(1) **完全性(completeness)** 原则上，分布在所有问题域的解都可能被构造出来。

(2) **封闭性(closure)** 每个基因编码对应一个可接受的个体，封闭性保证系统从不产生无效的个体。

(3) **紧致性(compactness)** 若两种基因编码 g_1 和 g_2 都被解码成相同的个体，若 g_1 比 g_2 占的空间少，就认为 g_1 比 g_2 紧致。

(4) **可扩展性**(scalability)　对于具体的问题,编码的大小确定了解码的时间,两者存在一定的函数关系,若增加一种表现型,作为基因型的编码大小也作出相应的增加。

(5) **多重性**(multiplicity)　多个基因型解码成一个表现型,即从基因型到相应的表现型空间是多对一的关系,这是基因的多重性。若相同的基因型被解码成不同的表现型,这是表现型多重性。

(6) **个体可塑性**(flexibility)　决定表现型与相应给定基因型是受环境影响的。

(7) **模块性**(modularity)　若表现型的构成中有多个重复的结构,在基因型编码中这种重复是应当避免的。

(8) **冗余性**(redundancy)　冗余性能够提高可靠性和鲁棒性(robustness)。

(9) **复杂性**(complexity)　包括基因型的结构复杂性、解码复杂性,计算时空复杂性(基因解码、适应值、再生等)。

其中满意的特性是:完全性、可测性和复杂性。但以上特性有时是矛盾的。

2.3　适应度函数及其尺度变换

遗传算法在进化搜索中基本不利用外部信息,仅以适应度函数(fitness function)为依据,利用种群中每个个体的适应度值来进行搜索。因此适应度函数的选取至关重要,直接影响到遗传算法的收敛速度以及能否找到最优解。一般而言,适应度函数是由目标函数变换而成的。对目标函数值域的某种映射变换称为适应度的尺度变换(fitness scaling)。

2.3.1　几种常见的适应度函数

适应度函数基本上有以下三种:

① 直接以待求解的目标函数的转化为适应度函数,即:

$$\text{若目标函数为最大化问题}\quad \mathrm{Fit}(f(x)) = f(x) \tag{2.29}$$

$$\text{若目标函数为最小问题}\quad \mathrm{Fit}(f(x)) = -f(x) \tag{2.30}$$

这种适应度函数简单直观,但存在两个问题,其一是可能不满足常用的轮盘赌选择中概率非负的要求;其二是某些代求解的函数在函数值分布上相差很大,由此得到的平均适应度可能不利于体现种群的平均性能,影响算法的性能。

② 若目标函数为最小问题,则

$$\mathrm{Fit}(f(x)) = \begin{cases} c_{\max} - f(x), & f(x) < c_{\max} \\ 0, & \text{其他} \end{cases} \tag{2.31}$$

式中 $c_{\max}$为 $f(x)$的最大值估计。

若目标函数为最大问题,则

$$\mathrm{Fit}(f(x)) = \begin{cases} f(x) - c_{\min}, & f(x) > c_{\min} \\ 0, & \text{其他} \end{cases} \tag{2.32}$$

式中 $c_{\min}$为 $f(x)$的最小值估计。

这种方法是对第一种方法的改进,可以称为"界限构造法",但有时存在界限值预先估计困难、不可能精确的问题。

③ 若目标函数为最小问题

$$\mathrm{Fit}(f(x)) = \frac{1}{1 + c + f(x)} \qquad c \geqslant 0,\ c + f(x) \geqslant 0 \tag{2.33}$$

若目标函数为最大问题

$$\mathrm{Fit}(f(x)) = \frac{1}{1 + c - f(x)} \qquad c \geqslant 0,\ c - f(x) \geqslant 0 \tag{2.34}$$

这种方法与第二种方法类似，c 为目标函数界限的保守估计值。

2.3.2 适应度函数的作用

在选择操作时会出现以下问题：

① 在遗传进化的初期，通常会产生一些超常的个体，若按照比例选择法，这些异常个体因竞争力太突出而控制了选择过程，影响算法的全局优化性能。

② 在遗传进化的后期，即算法接近收敛时，由于种群中个体适应度差异较小时，继续优化的潜能降低，可能获得某个局部最优解。

上述问题，我们通常称为遗传算法的欺骗问题。适应度函数设计不当有可能造成这种问题的出现。

设 n 个个体的适应度函数值 $F_1 \leqslant F_2 \leqslant \cdots \leqslant F_n$，记

$$F_{i+1} - F_i = \Delta_i \qquad (i = 1, 2, \cdots, n-1) \tag{2.35}$$

若 $\Delta = \max\{\Delta_i \mid (i = 1, 2, \cdots, n-2)\}$，且 $\Delta_{n-1} \geqslant \frac{(n-2)(n-1)\Delta}{2}$，则 $F_{n-1} \leqslant \frac{1}{n}\sum_{i=1}^{n} F_i \leqslant F_n$。

若 $\Delta = \max\{\Delta_i \mid (i = 2, 3, \cdots, n-2)\}$，且 $\Delta_1 \geqslant \frac{(n-2)(n-1)\Delta}{2}$，则 $F_1 \leqslant \frac{1}{n}\sum_{i=1}^{n} F_i \leqslant F_2$。

上面两种情况下，适应度函数值的分布对遗传搜索而言是极不合理的，所以适应度函数的设计是遗传算法设计的一个重要方面。

2.3.3 适应度函数的设计

适应度函数设计主要满足以下条件：

(1) **单值、连续、非负、最大化** 这个条件是很容易理解和实现的。

(2) **合理、一致性** 要求适应度值反映对应解的优劣程度，这个条件的达成往往比较难以衡量。

(3) **计算量小** 适应度函数设计应尽可能简单，这样可以减少计算时间和空间上的复杂性，降低计算成本。

(4) **通用性强** 适应度对某类具体问题，应尽可能通用，最好无需使用者改变适应度函数中的参数。从目前而言，这个条件应该是不属于强要求。

在第 6 章以后的章节里，我们将根据遗传算法的实际应用，再来讨论这个问题。

2.3.4 适应度函数的尺度变换

常用的尺度变换方法有以下几种：

1. 线性变换法

假设原适应度函数为 f，变换后的适应度函数为 f'，则线性变换可用下式表示：

$$f' = \alpha * f + \beta \tag{2.36}$$

上式中的系数确定方法有多种，但要满足以下条件：

① 原适应度的平均值要等于定标后的适应度平均值，以保证适应度为平均值的个体在下

一代的期望复制数为 1，即：

$$f'_{\text{avg}} = f_{\text{avg}} \tag{2.37}$$

② 变换后的适应度最大值应等于原适应度平均值的指定倍数，以控制适应度最大的个体在下一代中的复制数。试验表明，指定倍数 c_{mult} 可在 1.0～2.0 范围内。即根据上述条件可确定线性比例的系数：

$$f'_{\max} = c_{\text{mult}} f_{\text{avg}} \tag{2.38}$$

$$\alpha = \frac{(c_{\text{mult}} - 1) f_{\text{avg}}}{f_{\max} - f_{\text{avg}}}, \qquad \beta = \frac{(f_{\max} - c_{\text{mult}} f_{\text{avg}}) f_{\text{avg}}}{f_{\max} - f_{\text{avg}}} \tag{2.39}$$

如图 2.2 所示，线性变换法变换了适应度之间的差距，保持了种群内的多样性，并且计算简便，易于实现。如果种群内某些个体适应度远远低于平均值时，有可能出现变换后适应度值为负的情况，为此，考虑到保证最小适应度值非负的条件，进行如下的变换：

$$\alpha = \frac{f_{\text{avg}}}{f_{\text{avg}} - f_{\min}}, \qquad \beta = \frac{-f_{\min} f_{\text{avg}}}{f_{\text{avg}} - f_{\min}} \tag{2.40}$$

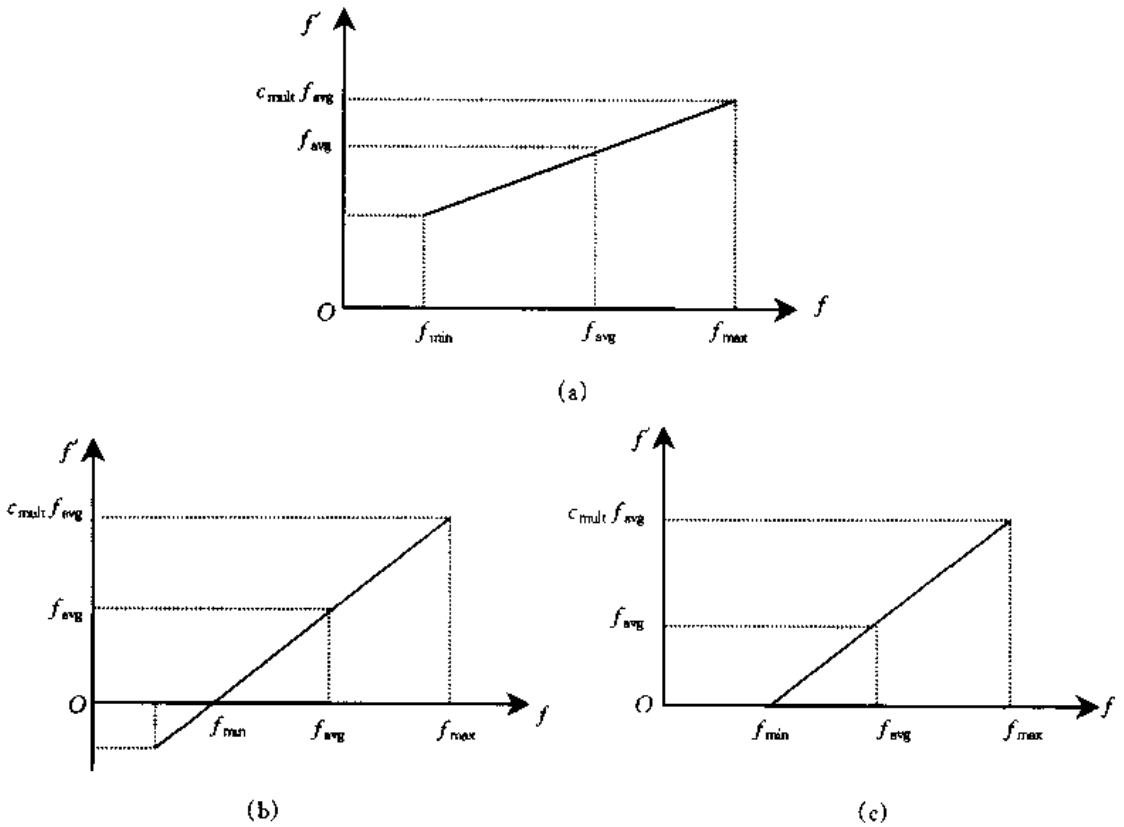

图 2.2　适应度函数的线性调整

(a) 正常情况；(b) 适应度出现负值；(c) 适应度出现负值时的变换

2. 幂函数变换法

变换公式为：

$$f' = f^k \tag{2.41}$$

上式中的幂指数 k 与所求的最优化问题有关，结合一些试验进行一定程度的精细变换才

能获得较好的结果。

3. 指数变换法

变换公式为：
$$f' = e^{-af} \tag{2.42}$$

这种变换方法的基本思想来源于模拟退火过程(simulated annealing, SA)，其中的系数 a 决定了复制的强制性，其值越小，复制的强制就越趋向于那些具有最大适应度的个体。

2.4 遗传操作

在第 1 章中，我们已经熟悉了几个简单遗传操作方法，例如轮盘赌选择、二进制编码的单点交叉和变异。本节将介绍常用的遗传操作方法，并进行比较。

2.4.1 选择(selection)

选择过程的第一步是计算适应度。在被选集中每个个体具有一个选择概率，这个选择概率取决于种群中个体的适应度及其分布。

下面一些概念可以用来比较不同的选择算法：

(1) **选择压力**(selection pressure) 最佳个体选中的概率与平均选中概率的比值。

(2) **偏差**(bias) 个体正规化适应度与其期望再生概率的绝对差值。

(3) **个体扩展**(spread) 单个个体子代个数的范围。

(4) **多样化损失**(loss of diversity) 在选择阶段未选中个体数目占种群的比例。

(5) **选择强度**(selection intensity) 将正规高斯分布应用于选择方法，期望平均适应度。

(6) **选择方差**(selection variance) 将正规高斯分布应用于选择方法，期望种群适应度的方差。

个体选择概率的常用分配方法有以下两种：

(1) **按比例的适应度分配**(proportional fitness assignment) 按比例的适应度分配，可称为选择的蒙特卡罗法，是利用比例于各个个体适应度的概率决定其子孙的遗留可能性。若某个个体 i，其适应度为 f_i，则其被选取的概率表示为

$$P_i = \frac{f_i}{\sum_{i=1}^{M} f_i} \tag{2.43}$$

表 2.2 给出了按比例的适应度计算方法的例子。表中采用的比率 $k=2$，当选择的概率给定后，产生[0, 1]区间的均匀随机数来决定哪个个体参加交配。显然选择概率大的个体，能多次被选中，它的遗传因子就会在种群中扩大。

表 2.2 个体选择概率计算

个体	f	f^2	P_i
1	2.5	6.25	0.18
2	1.0	1.00	0.03
3	3.0	9.00	0.26
4	1.2	1.44	0.04
5	2.1	4.41	0.13
6	0.8	0.64	0.02
7	2.5	6.25	0.18
8	1.3	1.69	0.05
9	0.9	0.81	0.02
10	1.8	3.24	0.09

(2) **基于排序的适应度分配**(rank－based fitness assignment) 在基于排序的适应度分配中，种群按目标值进行排序。适应度仅仅取决于个体在种群中的序位，而不是实际的目

标值。排序方法克服了比例适应度计算的尺度问题，当选择压力太小的情况下，以及选择导致搜索带迅速变窄而产生的过早收敛。因此，再生范围被局限。排序方法引入种群均匀尺度，提供了控制选择压力的简单有效的方法。

排序方法比比例方法表现出更好的鲁棒性，因此，不失为一种好的选择方法。

设定 N 为种群大小，Pos 为个体在种群中的序位，SP 为选择压力，个体的适应度可以计算如下：

线形排序：

$$\mathrm{Fit}(Pos) = 2 - SP + \frac{2(SP-1)(Pos-1)}{N-1} \qquad SP \in [1.0, 2.0] \tag{2.44}$$

非线形排序：

$$\mathrm{Fit}(Pos) = \frac{NX^{Pos-1}}{\sum_{i=1}^{N} X^{i-1}} \tag{2.45}$$

其中 X 是下列多项式方程的根：

$$(SP-1)\cdot X^{N-1} + SP\cdot X^{N-2} + \cdots + SP\cdot X + SP = 0 \tag{2.46}$$
$$SP \in [1.0, N-2.0]$$

选择强度：

$$SelInt_{\mathrm{Rank}} = \frac{SP-1}{\sqrt{\pi}} \tag{2.47}$$

多样化损失：

$$LossDiv_{\mathrm{Rank}} = (SP-1)/4 \tag{2.48}$$

选择方差：

$$SelVar_{\mathrm{Rank}}(SP) = 1 - ((SP-1)^2/\pi) = 1 - SelInt_{\mathrm{Rank}} SP^2 \tag{2.49}$$

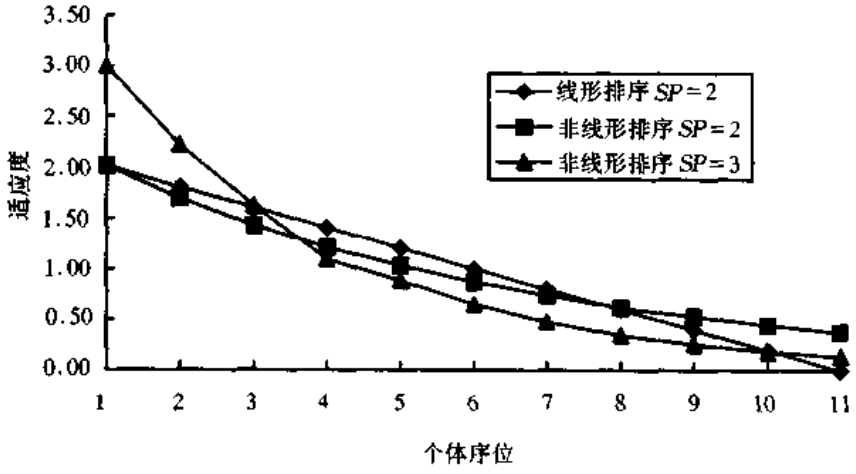

图 2.3　基于排序的适应度计算

关于个体的选择概率计算，Baker 提出了线性排序的计算公式：

$$p_i = \frac{1}{N}\left[\eta^+ - (\eta^+ - \eta^-)\frac{i-1}{N-1}\right] \tag{2.50}$$

上式中 i 为个体排序序号，$1 \leqslant \eta^+ \leqslant 2$，$\eta^- = 2 - \eta^+$

Michalewicz 提出了线性排序的选择概率计算公式：

$$p_i = c(1-c)^{i-1} \tag{2.51}$$

上式中 i 为个体排序序号，c 为排序第 1 的个体的选择概率。

下面介绍几个常用的选择方法：

(1) **轮盘赌选择法**(roulette wheel selection) 这是最简单的一种选择方法。表 2.3 表示了 11 个个体适应度、选择概率和累积概率。为了选择交配个体，需要进行多轮选择。每一轮产生一个[0,1]均匀随机数，将该随机数作为选择指针来确定被选个体。如图 2.4 所示，第 1 轮随机数为 0.81，则第 6 个个体被选中，第 2 轮随机数为 0.32，则第 2 个个体被选中；依此类推，第 3,4,5,6 轮随机数为 0.96,0.01,0.65,0.42，则第 9,1,5,3 个个体依次被选中。这样经过选择产生的交配种群由以下个体组成：1,2,3,5,6,9。

表 2.3 轮盘赌选择法的选择概率计算

个体	1	2	3	4	5	6	7	8	9	10	11
适应度	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	0.1
选择概率	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02	0.0
累积概率	0.18	0.34	0.49	0.62	0.73	0.82	0.89	0.95	0.98	1.00	1.00

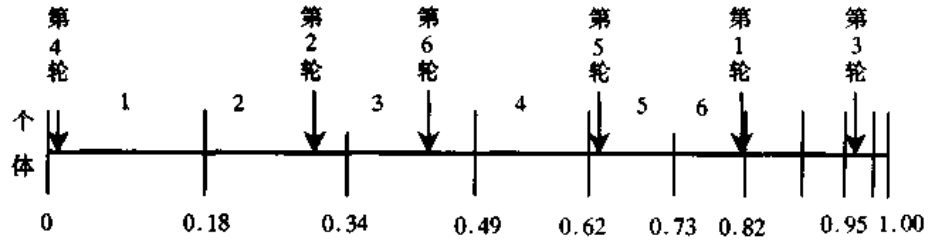

图 2.4 轮盘赌选择法

(2) **随机遍历抽样法**(stochastic universal sampling) 随机遍历抽样法提供了零偏差和最小个体扩展。设定 *npointer* 为需要选择的个体数目，等距离选择个体，选择指针的距离为 1/*npointer*，第一个指针的位置由[0,1/*npointer*]区间的均匀随机数决定。

如图 2.5 所示，需要选择 6 个个体，指针间的距离为 1/6=0.167，第一个指针的随机位置为 0.1，按这种选择方法被选中作为交配集个体为：1,2,3,4,6,8。

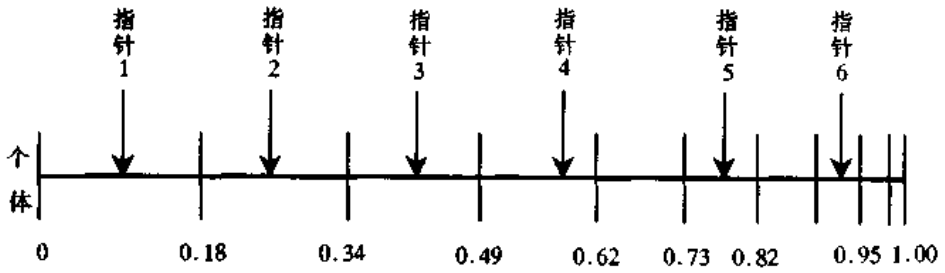

图 2.5 随机遍历抽样法

(3) **局部选择法**(local selection)　在局部选择法中，每个个体处于一个约束环境中，称为局部邻集(而其他选择方法中视整个种群为个体之邻集)，个体仅与其邻近个体产生交互，该邻集的定义由种群的分布结构给出，邻集可被当作潜在的交配伙伴。

首先均匀随机地选择一半交配种群，选择方法可以是随机遍历方法也可以是截断选择方法，然后对每个被选个体定义其局部邻集，在邻集内部选择交配伙伴。邻集的结构可以是以下三种：

(1) **线形邻集**　整环型和半环型，如图 2.6 所示。

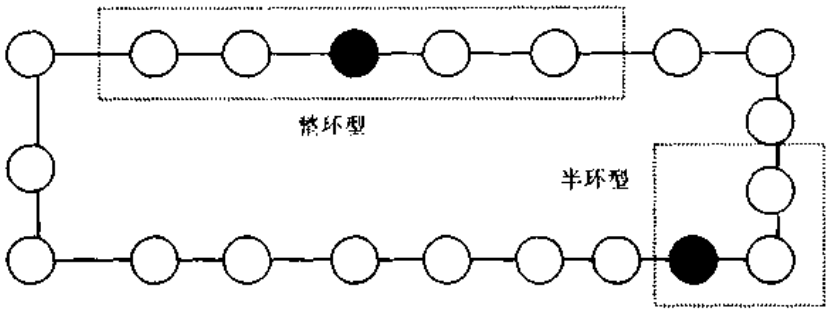

图 2.6　线形邻集(距离为 2)

(2) **两对角邻集**　整对角型和半对角型、整星型和半星型，如图 2.7 所示。

表 2.4 给出了给定结构和不同距离的邻集个体大小。

表 2.4　给定结构和不同距离的邻集个体大小

结　构	距　离	
	1	2
整环型	2	4
半环型	1	2
整对角型	4	8
半对角型	2	4
整星型	8	24
半星型	3	8

种群中的个体之间存在距离，邻集越小，隔离距离越大。但是由于邻集间有重叠，会产生新的变体，这也就保证了所有个体之间的信息交换。邻集的大小决定了这种信息传播的速度，因此也必须在快速传播速度与保持种群多样化之间做出选择。通常求得较好的多样化，可以防止过早收敛到局部最小。对局部选择而言，在一个小邻集里进行优越于在一个大邻集里进行。但在种群范围内提供相互连接关系是必须的。我们推荐采用二维邻集、半星型结构、距离为 1 作为局部选择的方案。如果种群大小较大(超过 100)，可选用较大的距离或其他的二维邻集。

(3) **截断选择法**(truncation selection)　与前面几种自然方式的选择方法比较，截断选择法是一种人工选择方法。它适合于大种群。在截断选择法中，个体按适应度排序。只有最优

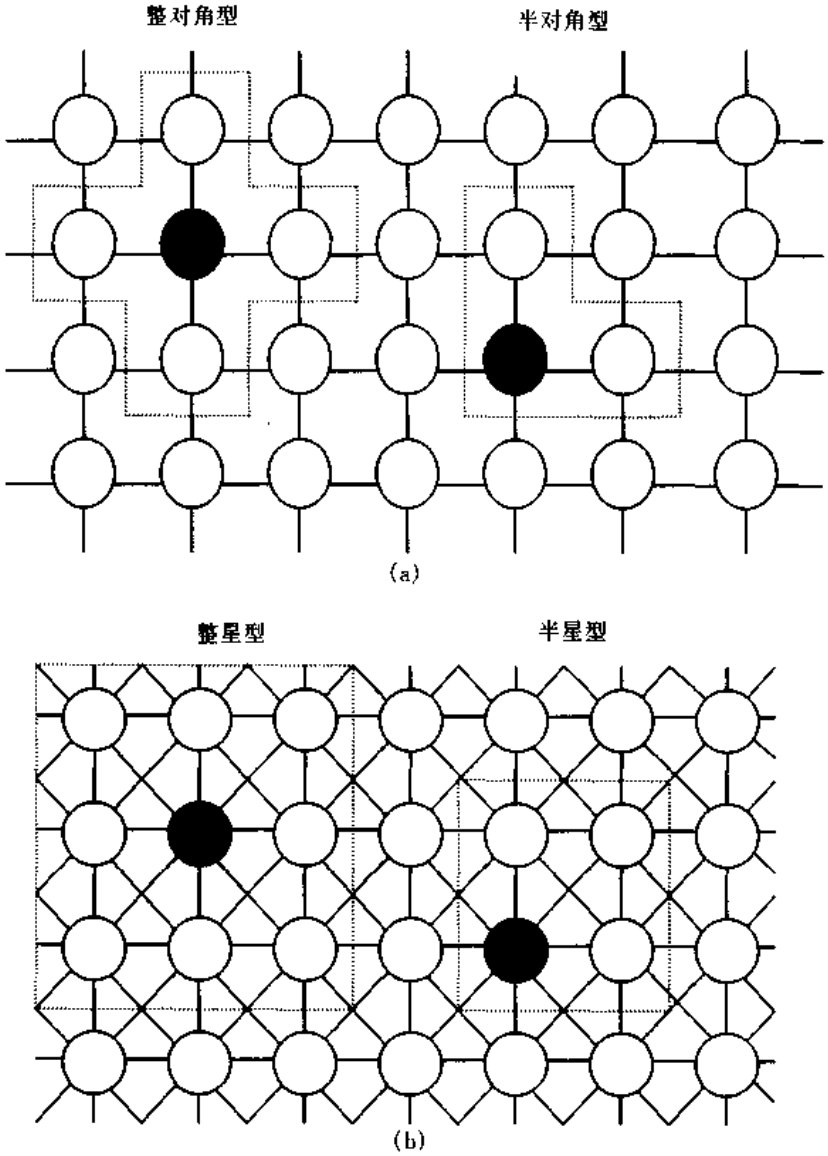

图 2.7 两对角邻集(距离为 1)
(a)整对角形和半对角型; (b)整星型和半星型

秀的个体能够被选作父个体,截断选择的参数叫做截断阀值 $Trunc$。它定义为被选做父个体的百分比,取值范围为 50%－10%。在该阀值之下的个体不能产生子个体。通常选择强度与截断阀值的关系见表 2.5 所示。

表 2.5 选择强度与截断阀值的关系

截断阀值	1%	10%	20%	40%	50%	80%
选择强度	2.66	1.76	1.2	0.97	0.8	0.34

选择强度:
$$SelInt_{Trunc}(Trunc)=\frac{1}{Trunc\ \sqrt{2\pi}e^{-f_c^2/2}} \tag{2.52}$$

多样化损失:
$$LossDiv_{Trunc}(Trunc)=1-Trunc \tag{2.53}$$

选择方差：$SelVar_{Trunc}(Trunc)=1-SelInt_{Trunc}(Trunc)(SelInt_{Trunc}(Trunc)-f_c)$　　(2.54)

上式中 f_c 为下列高斯分布的积分下限：

$$Trunc=\int_{f_c}^{-\infty}\frac{1}{\sqrt{2\pi}}e^{-\frac{f^2}{2}}df \tag{2.55}$$

(4) **锦标赛选择法**(tournament selection)　在锦标赛选择法中，随机地从种群中挑选一定数目($Tour$)个体，然后将最好的个体选做父个体。这个过程重复进行完成个体的选择。锦标赛选择的参数为竞赛规模 $Tour$，其取值范围为$[2, Nind]$。表 2.6 表示了竞赛规模与选择强度之间的关系。

表 2.6　竞赛规模与选择强度之间的关系

竞赛规模	1	2	3	5	10	30
选择强度	0	0.56	0.85	1.15	1.53	2.04

选择强度：　$SelInt_{Tour}(Tour)=\sqrt{2(\log(Tour)-\log\sqrt{4.14\log(Tour)})}$　　(2.56)

多样化损失：　$LossDiv_{Tour}(Tour)=Tour^{-\frac{1}{Tour-1}}-Tour^{-\frac{Tour}{Tour-1}}$　　(2.57)

当 $Tour=5$ 时多样化损失大约为 50%。

选择方差：　$SelVar_{Tour}(Tour)=1-0.096\log(1+7.11(Tour-1))$　　(2.58)

$$SelVar_{Tour}(2)=1-\frac{1}{\pi} \tag{2.59}$$

上述几种选择方法均以适应度为基础进行选择，这就可能在进化过程中导致以下的问题：

① 在种群中出现个别或极少数适应度相当高的个体时，采用这样的选择机制就有可能导致这些个体在种群中迅速繁殖。经过少数几次迭代后占满了种群的位置。这样，遗传算法的求解过程就结束了，也即收敛了。但这样很有可能是收敛到局部最优解，即遗传算法的不成熟收敛，即早熟现象的出现，这是因为搜索的范围很有限。因此一般不希望有个别个体在遗传算法运算的最初几次迭代时就在种群中占据主导地位。

② 当种群中个体适应度彼此非常接近时，这些个体进入配对集的机会相当，而且交配后得到的新个体也不会有多大变化。这样，搜索过程就不能有效地进行，选择机制有可能趋向于纯粹的随机选择，从而是进化过程陷于停顿的状态，难以找到全局最优解。

针对上述问题，可以采用适应度函数尺度变换(本章 2.3 节介绍)的方法来解决。另外，还可以采用以下的几种提高遗传算法性能的选择方法：

(1) **稳态繁殖**(steady state reproduction)　在迭代过程中用部分优质新子个体来更新种群中部分父个体来作为下一代种群。

(2) **没有重串的稳态繁殖**(steady state reproduction without duplicates)　在形成新一代种群时，使其中的种群均不重复。做法是：在将某个个体加入到新的一代种群之前，先检查该个体与种群中现有的个体是否重复，如果重复就舍弃。这种做法会明显改善遗传算法的行为，因为其增大了个体在种群中的分布区域，但增加了计算时间。

不同选择方法的行为是有差别的，基本遗传算法达到收敛的世代数与选择强度成反比，较高的选择强度是很好的选择方法，但太高会导致收敛过快，解的质量差。最小限度的种群大小往往依赖于目标函数的维数和选择强度，而选择强度又与选择参数(如选择压力、截断阈值、竞

赛规模)有关。锦标赛选择法只能赋离散值,线性排序选择法只允许较小区间值的选择强度。截断选择会导致比排序选择和锦标赛选择更高的多样化损失,截断选择倾向于用更好的个体取代较差的个体,因为所有低于适应度阈值之下的个体没有机会被选择,排序选择与锦标赛选择比较相似,但是排序选择往往用在锦标赛选择法因其离散性不能发挥作用的场合。对于同样的选择强度,截断选择的选择方差比排序选择和锦标赛选择小。

2.4.2 交叉/基因重组(crossover/recombination)

基因重组是把两个父个体的部分结构加以替换重组而生成新个体的操作,也称交叉(crossover)。重组的目的是为了能够在下一代产生新的个体,就像人类社会的婚姻过程,通过重组交叉操作,遗传算法的搜索能力得以飞跃地提高。基因重组和交叉是遗传算法获取新优良个体的最重要的手段。

(1) 实值重组

① 离散重组

离散重组在个体之间交换变量的值,考虑如下含有三个变量的个体:

父个体 1　　12　25　5

父个体 2　　23　4　34

子个体的每个变量可按等概率随机地挑选父个体,例如重组之后一个子个体为:

子个体 1　　123　4　5

子个体 2　　12　4　5

图 2.8 表示了离散重组后子个体的可能位置。

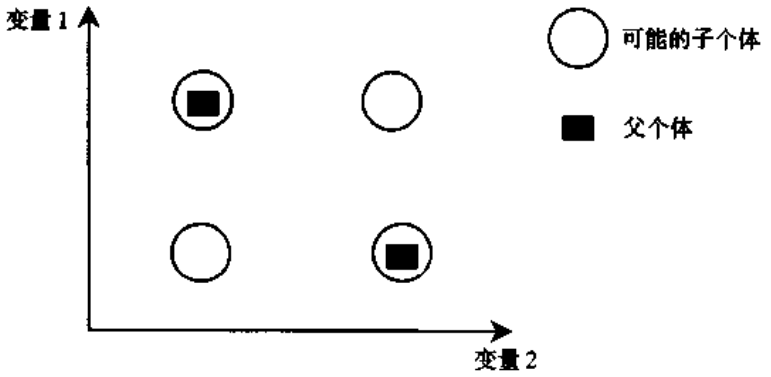

图 2.8　离散重组

② 中间重组

中间重组只能适用于实变量,而非二进制变量,见图 2.9。

子个体的产生按下列公式:

$$\text{子个体} = \text{父个体 1} + \alpha * (\text{父个体 2} - \text{父个体 1}) \tag{2.60}$$

这里 α 是一个比例因子,可由 $[-d, 1+d]$ 上均匀分布随机数产生。对于中间重组 $d=0$;一般选择 $d=0.25$。子代的每个变量的值按上面的表达式计算,对每个变量要选择一个新的 α 值。图 2.10 为按父个体变量值定义的子个体取值范围。考虑含有三个变量的两个个体:

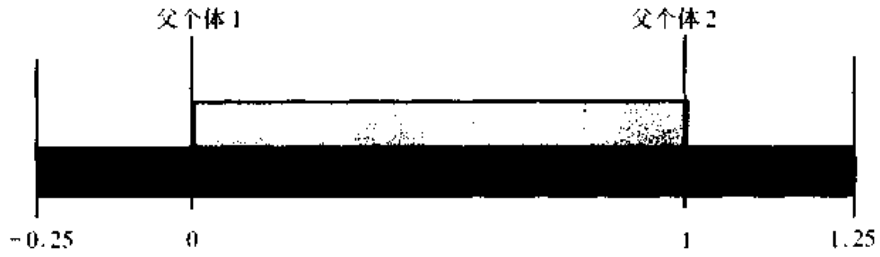

图 2.9　中间重组

父个体 1　　12　25　5
父个体 2　　123　4　34

α 值的样本为：

样本 1　　0.5　1.1　-0.1
样本 2　　0.1　0.8　0.5

计算出新的子个体为：

子个体 1　　67.5　1.9　2.1
子个体 2　　23.1　8.2　19.5

图 2.10 为中间重组后子个体的可能位置。

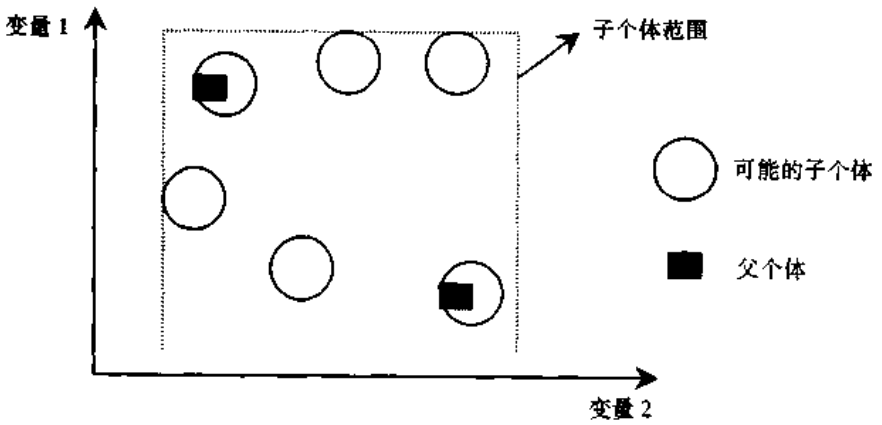

图 2.10　按父个体变量值定义的子个体取值范围

③ 线性重组

线性重组与中间重组比较相似，只是对所有变量只有一个 α 值。

父个体 1　　12　25　5
父个体 2　　123　4　34

α 值的样本为：

样本 1　　0.5
样本 2　　0.1

计算出新的子个体为：

子个体 1　　67.5　14.5　19.5
子个体 2　　23.1　22.9　 7.9

图 2.11 为线性重组后子个体的可能位置。

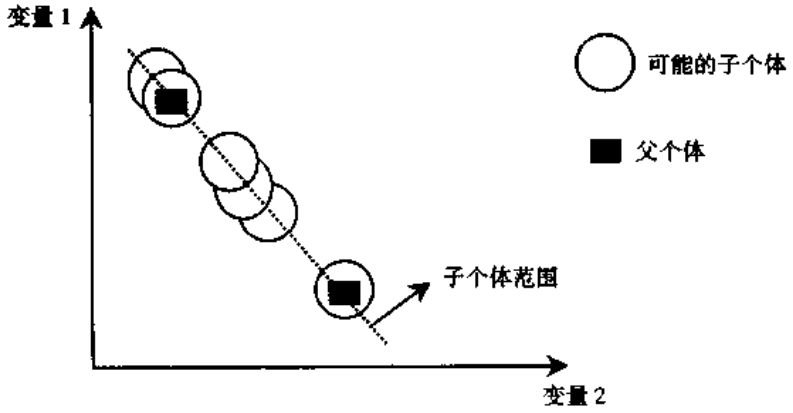

图 2.11　线性重组

(2) **二进制交叉**

① 单点交叉

单点交叉中，交叉点 k 的范围为$[1, Nvar-1]$，$Nvar$ 为个体变量数目，在该点为分界相互交换变量。

考虑如下两个 11 位变量的父个体：

父个体 1　　0　1　1　1　0　0　1　1　0　1　0
父个体 2　　1　0　1　0　1　1　0　0　1　0　1

交叉点的位置为 5，如图 2.12 所示，交叉后生成两个子个体：

子个体 1　　0　1　1　1　0　1　0　0　1　0　1
子个体 2　　1　0　1　0　1　0　1　1　0　1　0

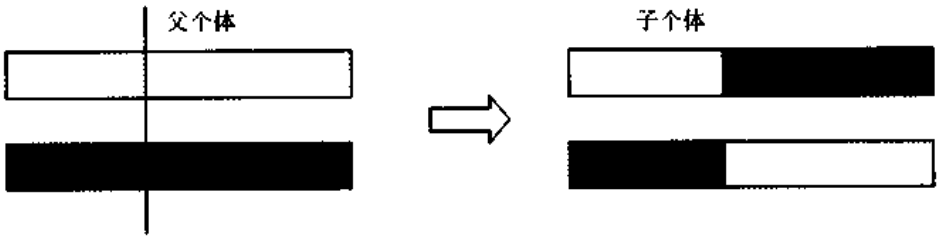

图 2.12　单点交叉

② 多点交叉

对于多点交叉，m 个交叉位置 K_i 可无重复随机地选择，在交叉点之间的变量间续地相互交换，产生两个新的后代，但在第一位变量与第一个交叉点之间的一段不做交换。

考虑如下两个 11 位变量的个体：

父个体 1　　0 1 1 1 0 0 1 1 0 1 0

父个体 2　　1 0 1 0 1 1 0 0 1 0 1

交叉点的位置为：　　2　6　10

如图 2.13 所示，交叉后两个新个体为：

子个体 1　　0 1 1 0 1 1 1 1 0 1 1

子个体 2　　1 0 1 1 0 0 0 0 1 0 0

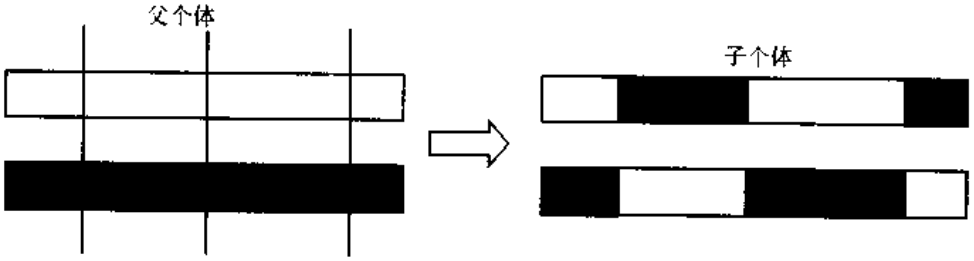

图 2.13　多点交叉

多点交叉的思想源于控制个体特定行为的染色体表示信息的部分无须包含于邻近的子串中，多点交叉的破坏性可以促进解空间的搜索，而不是促进过早地收敛。因此搜索更加健壮。

③ 均匀交叉

单点和多点交叉的定义使得个体在交叉点处分成片段。均匀交叉更加广义化，将每个点都作为潜在的交叉点。随机地产生与个体等长的 0－1 掩码，掩码中的片段表明了哪个父个体向子个体提供变量值。

考虑如下两个 11 位变量的个体：

父个体 1　　0 1 1 1 0 0 1 1 0 1 0

父个体 2　　1 0 1 0 1 1 0 0 1 0 1

掩码样本(1 表示父个体 1 提供变量值，0 表示父个体 2 提供变量值)：

样本 1　　0 1 1 0 0 0 1 1 0 1 0

样本 2　　1 0 0 1 1 1 0 0 1 0 1

交叉后两个新个体为：

子个体 1　　1 1 1 0 1 1 1 1 1 1 1

子个体 2　　0 0 1 1 0 0 0 0 0 0 0

均匀交叉类似于多点交叉，可以减少二进制编码长度与给定参数特殊编码之间的偏差。它的算法与离散重组是等价的。

除了上述交叉方法以外，还有部分匹配交叉(Partially Matched Crossover，PMX)、顺序交叉(Ordered Crossover，OX)、循环交叉(Cycle Crossover，CX)、洗牌交叉(shuffle crossover)、缩小代理交叉(crossover with reduced surrogate)等，前三种交叉方法的描述见 7.1 节。

2.4.3　变异(mutation)

重组之后是子代的变异，子个体变量以很小的概率或步长产生转变，变量转变的概率或步长与维数(即变量的个数)成反比，与种群的大小无关。据研究，对于单峰函数 $1/n$ 是最好的选择，开始时增加变异率，结束时减少变异率可以改善搜索速度。但对于多峰函数变异率的自

适应过程是很有益的选择。变异本身是一种局部随机搜索，与选择/重组算子结合在一起，保证了遗传算法的有效性，使遗传算法具有局部的随机搜索能力；同时使得遗传算法保持种群的多样性，以防止出现非成熟收敛。在变异操作中，变异率不能取得太大，如果变异率大于 0.5，遗传算法就退化为随机搜索，而遗传算法的一些重要的数学特性和搜索能力也不复存在了。

(1) **实值变异**

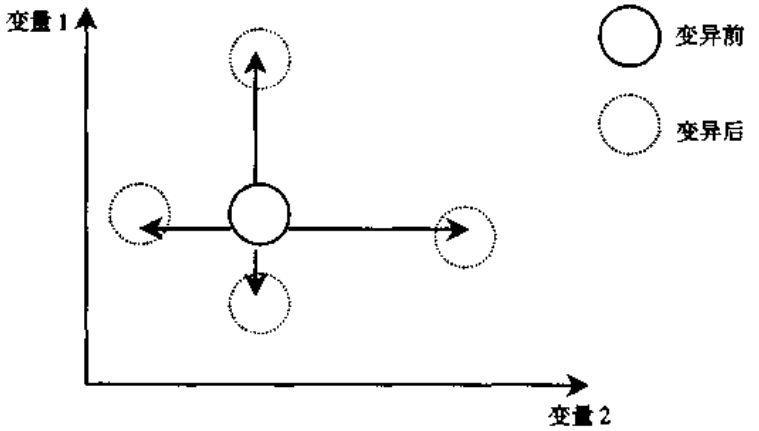

图 2.14 实值变异

变异步长的选择比较困难，最优的步长视具体情况而定，甚至在优化过程中可以改变。通常较小的步长会比较成功，但有时大步长比较快些。建议采用如下的变异算子：

$$X' = X \pm 0.5L\Delta \tag{2.61}$$

上式中，$\Delta = \sum_{i=0}^{m-1} \frac{a(i)}{2^i}$，$a(i)$以概率 $1/m$ 取值 1，以 $1-1/m$ 取值 0，通常 $m=20$；L 为变量的取值范围；X 为变异前变量取值；X'为变异后变量取值。

(2) **二进制变异**

对于二进制编码的个体而言，变异意味着变量的翻转。对于每个个体，变量值的改变是随机的，如下所示，有 11 位变量的个体，第 4 位发生了翻转。

变异前： 01110011010

变异后： 01100011010

上述个体解码成实数，变异的效果依赖于实际的编码方法。除了上述基本位变异法以外，还有其他的变异方法，如换位(displacement)、复制(duplication)、插入(addition)、删除(deletion)等等。

2.5 算法设计与实现

基本遗传算法(Simple Genetic Algorithm, SGA)可定义为一个 8 元组：

$$SGA = (C, E, P_0, M, \Phi, \Gamma, \Psi, T) \tag{2.62}$$

式中， C —— 个体的编码方法，SGA 使用固定长度二进制符号串的编码方法；

E —— 个体的适应度评价函数；

P_0—— 初始群体；

M —— 群体大小，一般取 20～100；

Φ —— 选择算子，*SGA* 使用比例选择算子；

Γ —— 交叉算子，*SGA* 使用单点交叉算子；

Ψ —— 变异算子，*SGA* 使用基本位变异算子；

T —— 算法终止条件，一般终止进化代数为 100～500。

2.5.1 问题的表示

对于一个实际的待优化问题，首先需要将其表示为适合于遗传算法操作的形式。这里以二进制编码为例，它包括以下几个步骤：

① 根据具体问题确定待寻优的参数。

② 对每个参数确定它的变化范围，并用一个二进制数来表示。例如，若参数 a 的变化范围为$[a_{\min}, a_{\max}]$，用 m 位二进制数 b 来表示，则二者之间满足：

$$a = a_{\min} + \frac{b}{2^m - 1}(a_{\max} - a_{\min}) \tag{2.63}$$

这时参数范围的确定应覆盖全部的寻优空间，字长 m 的确定应在满足精度要求情况下尽量取小的 m，以尽量减小遗传算法计算的复杂性。

③ 将所有表示参数的二进制数串接起来组成一个长的二进制字串。该字串的每一位只有 0 或 1 两种取值。该字串即为遗传算法的操作对象。

2.5.2 初始种群的产生

产生初始种群的方法通常有两种。一种是完全随机的方法产生的，它适合于对问题的解无任何先验知识的情况。某些先验知识可转变为必须满足的一组要求，然后在满足这些要求的解中再随机地选取样本。这样选择初始种群可使遗传算法更快地到达最优解。

2.5.3 算法设计与实现

附录Ⅱ.1 给出了基本遗传算法的 C 语言源程序，它是根据 David, E. Goldberg 的 Pascal 源程序改写的，用于 2.1 节的函数优化问题。下面我们讨论基本遗传算法实现方法的几个主要问题。

1. 数据结构与遗传算法参数

基本遗传算法处理的对象主要是个体，因此设计了结构变量 individual 来描述个体信息，其中包括个体的染色体串 chrom，个体适应度 fitness，个体对应的变量 varible，交叉位置 xsite，以及记录父个体编号 parent [2]等。为记录进化历代最佳个体，设计结构变量 bestever，表示最佳个体对应的染色体 chrom，个体适应度 fitness，对应的变量 varible 以及最佳个体产生的代数。根据个体变量定义，设计当前代种群 oldpop 以及新一代种群 newpop 为全局变量。此外，将当前种群中个体最大适应度 max、个体平均适应度 avg、最小适应度 min、个体适应度累计值也定义为全局变量。

基本遗传算法运行参数包括：种群大小 popsize，染色体长度 lchrom，进化最大代数 maxgen，总运行次数 maxruns，交叉率 pcross，变异率 pmutation 等。这些参数在运行开始时由使用者输入，参见函数 initdata()。

```
/* 全局变量 */
struct individual                    /* 个体 */
{
    unsigned    *chrom;              /* 染色体 */
    double      fitness;             /* 个体适应度 */
    double      varible;             /* 个体对应的变量值 */
    int         xsite;               /* 交叉位置 */
    int         parent[2];           /* 父个体 */
    int         *utility;            /* 特定数据指针变量 */
};
struct bestever                      /* 最佳个体 */
{
    unsigned    *chrom;              /* 最佳个体染色体 */
    double      fitness;             /* 最佳个体适应度 */
    double      varible;             /* 最佳个体对应的变量值 */
    int         generation;          /* 最佳个体生成代 */
};
struct individual *oldpop;           /* 当前代种群 */
struct individual *newpop;           /* 新一代种群 */
struct bestever bestfit;             /* 最佳个体 */
double sumfitness;                   /* 种群中个体适应度累计 */
double max;                          /* 种群中个体最大适应度 */
double avg;                          /* 种群中个体平均适应度 */
double min;                          /* 种群中个体最小适应度 */
float  pcross;                       /* 交叉率 */
float  pmutation;                    /* 变异率 */
int    popsize;                      /* 种群大小 */
int    lchrom;                       /* 染色体长度 */
int    chromsize;                    /* 存储一染色体所需字节数 */
int    gen;                          /* 当前世代 */
int    maxgen;                       /* 最大世代数 */
int    run;                          /* 当前运行次数 */
int    maxruns;                      /* 总运行次数 */
```

2. 产生初始种群

为产生初始种群设计的函数为 initpop()。种群中个体的染色体编码的每一位按等概率在 0 与 1 中选择。在产生染色体编码后，对个体进行解码和适应度计算。该算法程序如下：

```
void initpop()  /* 随机初始化种群 */
{
    int j, j1, k, stop;
    unsigned mask = 1;
```

```
    for(j=0; j < popsize; j++)
    {
        for(k=0; k < chromsize; k++)
        {
            oldpop[j].chrom[k]=0;
            if(k==(chromsize-1))
                stop=lchrom -(k*(8*sizeof(unsigned)));
            else
                stop=8*sizeof(unsigned);
            for(j1=1; j1 <=stop; j1++)
            {
                oldpop[j].chrom[k]=oldpop[j].chrom[k]<<1;
                if(flip(0.5))
                    oldpop[j].chrom[k]=oldpop[j].chrom[k]|mask;
            }
        }
            oldpop[j].parent[0]=0;          /* 初始父个体信息 */
            oldpop[j].parent[1]=0;
            oldpop[j].xsite=0;
        objfunc(&(oldpop[j]));          /* 计算初始适应度 */
    }
}
```

解码和适应度计算用 objfunc()函数。这里目标函数用 2.1 式的函数 $f(x)=x\sin(10\pi x)+2$,优化变量区间为[-1,2]。

个体的适应度 *critter.fitness* 计算如下：

$$critter.fitness = critter.varible * \sin(10\pi * critter.varible) + 2$$

上述算法的程序如下：

```
void objfunc(critter)          /* 计算适应度函数值 */
struct individual *critter;
{
    unsigned mask=1;
    unsigned bitpos;
    unsigned tp;
    double pow(), bitpow ;
    int j, k, stop;
    critter->varible=0.0;
    for(k=0; k < chromsize; k++)
    {
        if(k==(chromsize-1))
            stop=lchrom-(k*(8*sizeof(unsigned)));
```

```
        else
            stop = 8 * sizeof(unsigned);
        tp = critter->chrom[k];
        for(j = 0; j < stop; j++)
        {
            bitpos = j + (8 * sizeof(unsigned)) * k;
            if((tp&mask) == 1)
            {
                bitpow = pow(2.0, (double)bitpos);
                critter->varible = critter->varible + bitpow;
            }
            tp = tp>>1;
        }
    }
    critter->varible = -1 + critter->varible * 3/(pow(2.0, (double)lchrom) - 1);
    critter->fitness = critter->varible * sin(critter->varible * 10 * atan(1) * 4) + 2.0;
}
```

3. 遗传操作设计

为轮盘赌选择设计的函数 select(),返回种群中被选择的个体编号。方法是产生一个[0,1]随机数 *pick*,若 $pick < sum = \sum_{i=0}^{i} oldpop[i]/sumfitness$,则第 i 个个体被选中。该算法程序如下:

```
int select()  /* 轮盘赌选择 */
{
    extern float randomperc();
    float sum, pick;
    int i;
    pick = randomperc();
    sum = 0;
    if(sumfitness != 0)
    {
        for(i = 0;(sum < pick)&&(i < popsize); i++)
            sum += oldpop[i].fitness/sumfitness;
    }
    else
        i = rnd(1, popsize);
    return(i - 1);
}
```

为单点交叉操作设计的函数 crossover(),由父个体 parent1 和 parent2 产生子个体 child1

和 child2，若交叉发生处理编码赋值，并返回交叉点位置 jcross；否则不做任何处理，返回 0。方法是，先通过 flip(pcross)函数确定是否发生交叉操作，若发生交叉操作，在[1, lchrom]区间随机确定交叉位置 jcross，child1 继承 parent1 在 jcross 之前的编码和 parent2 在 jcross 之后的编码，child2 继承 parent2 在 jcross 之前的编码和 parent1 在 jcross 之后的编码。编码赋值按染色体的编码位逐一判断处理。该算法程序如下：

```
int crossover(unsigned * parent1, unsigned * parent2, unsigned * child1, unsigned * child2)
/* 由两个父个体交叉产生两个子个体 */
{
    int j, jcross, k;
    unsigned mask, temp;
    if(flip(pcross))
    {
        jcross = rnd(1, (lchrom - 1));        /* 交叉位置在 1 和 l-1 之间 */
        ncross + +;
        for(k = 1; k <= chromsize; k + +)
        {
            if(jcross >= (k * (8 * sizeof(unsigned))))
            {
                child1[k - 1] = parent1[k - 1];
                child2[k - 1] = parent2[k - 1];
            }
            else if((jcross < (k * (8 * sizeof(unsigned))))&&
                (jcross >((k - 1) * (8 * sizeof(unsigned)))))
            {
                mask = 1;
                for(j = 1; j <= (jcross - 1 - ((k - 1) * (8 * sizeof(unsigned)))); j + +)
                {
                    temp = 1;
                    mask = mask<<1;
                    mask = mask|temp;
                }
                child1[k - 1] = (parent1[k - 1]&mask)|(parent2[k - 1]&(~mask));
                child2[k - 1] = (parent1[k - 1]&(~mask))|(parent2[k - 1]&mask);
                }
                else
                {
                    child1[k - 1] = parent2[k - 1];
                    child2[k - 1] = parent1[k - 1];
                }
            }
        }
    }
```

```
    else
    {
        for(k=0; k < chromsize; k++)
        {
            child1[k]=parent1[k];
            child2[k]=parent2[k];
        }
        jcross=0;
    }
    return(jcross);
}
```

为变异操作设计的函数 mutation(),按变异概率 pmutation 确定个体 child 的编码位是否发生操作,若某编码位发生变异,则该编码翻转。该算法程序如下:

```
void mutation(unsigned *child)  /* 变异操作 */
{
    int j, k, stop;
    unsigned mask, temp=1;
    for(k=0; k < chromsize; k++)
    {
        mask=0;
        if(k==(chromsize-1))
            stop=lchrom -(k*(8*sizeof(unsigned)));
        else
            stop=8*sizeof(unsigned);
        for(j=0; j < stop; j++)
        {
            if(flip(pmutation))
            {
                mask=mask|(temp<<j);
                nmutation++;
            }
        }
        child[k]=child[k]^mask;
    }
}
```

4. 世代进化过程的实现

为模拟世代进化过程设计的函数 generation(),以种群的处理对象实现了一个世代内的三种遗传操作。首先在当前种群中用 select()函数选择两个个体,然后对两个个体按交叉概率实行可能的交叉操作和变异操作,然后对个体解码、计算适应度、记录亲子信息数据等,生成新一

代个体。该算法程序如下：

```
void generation()
{
  int mate1, mate2, jcross, j=0;
  preselect();
  /* 选择,交叉,变异 */
  do
    {
      /* 挑选交叉配对 */
      mate1 = select();
      mate2 = select();
      /* 交叉和变异 */
      jcross = crossover(oldpop[mate1].chrom, oldpop[mate2].chrom,
            newpop[j].chrom, newpop[j+1].chrom);
      mutation(newpop[j].chrom);
      mutation(newpop[j+1].chrom);
      /* 解码,计算适应度 */
      objfunc(&(newpop[j]));
      /* 记录亲子关系和交叉位置 */
      newpop[j].parent[0] = mate1 + 1;
      newpop[j].xsite = jcross;
      newpop[j].parent[1] = mate2 + 1;
      objfunc(&(newpop[j+1]));
      newpop[j+1].parent[0] = mate1 + 1;
      newpop[j+1].xsite = jcross;
      newpop[j+1].parent[1] = mate2 + 1;
      j=j + 2;
    }
  while(j <(popsize - 1));
}
```

5. 主程序

主程序设计了多次运行遗传算法的过程，每次运行前由使用者输入不同的参数设置，运行结束时将有关进化过程的统计结果写入输出数据文件中。作为一次的遗传算法运行，首

```
int argc;
char * argv[];
{
    struct individual * temp;
    FILE   * fopen();
    void   title();
    char   * malloc();
        if((outfp = fopen(argv[1],"w")) = = NULL)
        {
            fprintf(stderr,"Cannot open output file %s \ n", argv[1]);
            exit(-1);
        }
    g_init();
    setcolor(9);
    setbkcolor(15);
    title();
    disp_hz16("输入遗传算法执行次数(1 - 5):", 100, 120, 20);
    gscanf(320, 120, 9, 15, 4, "%d", &maxruns);
    for(run = 1; run< = maxruns; run + + )
    {
      initialize();
      for(gen = 0; gen< maxgen; gen + + )
        {
        fprintf(outfp, " \ n 第 %d / %d 次运行: 当前代为 %d, 共 %d 代 \ n",
              run, maxruns, gen, maxgen);
            /* 产生新一代 */
        generation();
            /* 计算新一代种群的适应度统计数据 */
        statistics(newpop);
            /* 输出新一代统计数据 */
        report();
        temp = oldpop;
        oldpop = newpop;
        newpop = temp;
        }
```

每次迭代的时间，一般取 20～100。交叉率的选择决定了交叉操作的频率，频率越高，可以越快地收敛到最有希望的最优解区域，因此一般选取较大的交叉率，但太高的频率也可能导致过早收敛，一般取值 0.4～0.9。变异率的选取一般受种群大小、染色体长度等因素影响，通常选取很小的值，一般取 0.001～0.1。若选取高的变异率，虽然增加样本模式的多样性，但可能会引起不稳定。种群大小及染色体长度越大，变异率选取越小。染色体长度主要决定于问题求解精度的要求，精度越高，染色体长度越长，搜索空间越大，相应地要求种群大小设置大一些。最大进化代数作为一种模拟终止条件，一般视具体问题而定，取 100～500 代。对于具体问题而言，衡量参数设置恰当与否，要依据多次运行的收敛情况和解的质量来判断。如果调整参数难以有效地提高遗传算法的性能，则往往需要借助对基本遗传算法的改进，改进的手段可以是多方面的，如适应度比例调整、引入自适应交叉率和变异率，尝试其他的遗传操作(与基本遗传算法不同的选择、交叉和变异法)，也可以采用一些混合方法等(详见第 4 章内容)。

为了更好地把握基本遗传算法，下面给出 2.1 节优化实例的部分模拟实算统计结果。为简单起见，我们设置了如下的参数：

基本遗传算法参数

种群大小(popsize)	= 30
染色体长度(lchrom)	= 22
最大进化代数(maxgen)	= 150
交叉率(pcross)	= 0.80
变异率(pmutation)	= 0.05

表 2.7、表 2.8 分别列出了初始代到第 1 代、第 7 代到第 8 代的种群进化统计结果。图 2.15显示了历代进化的个体适应度(最大值、最小值和平均值)变化曲线。

表 2.7　模拟计算统计报告(0～1 代)

个体	世代数 0 染色体编码	适应度	父个体	交叉位置	世代数 1 染色体编码	适应度
1	0111010100110001110010	2.041 164	(30,2)	0	1000110010000100100010	1.981 748
2	0000101010011011001001	1.450 069	(30,2)	0	0000101010011011001001	1.450 069

续表 2.7

个体	世代数 0 染色体编码	适应度	父个体	交叉位置	世代数 1 染色体编码	适应度
16	01111000011100000100010	2.003 401	(1,20)	8	00110000001100111110010	2.055 468
17	00100100100110000011110	2.137 394	(25,17)	2	01100100100110010111110	2.382 481
18	01011101001111110011100	1.804 015	(25,17)	2	00110011101100011010111	1.517 991
19	00011000111100111110111	2.200 482	(6,14)	19	00000110111111110110011	2.599 965
20	00000000110011111111011	3.058 461	(6,14)	19	10111110100110111111111	1.562 136
21	01111011100111111101000	1.579 198	(10,27)	9	11110110010100011100101	1.056 378
22	00000111110110001110110	2.041 850	(10,27)	9	00000110010100110010011	1.494 851
23	01011101111100100110010	1.993 267	(3,26)	13	11111000111100110000011	2.476 917
24	10100000110010111000010	1.852 255	(3,26)	13	00100011001110111110100	2.435 961
25	00110111101100001111010	1.986 588	(25,20)	21	00110111101100011110110	1.986 588
26	00100011001110110000011	2.382 383	(25,20)	21	00000000110011110100010	1.959 661
27	00000100010100011000001	1.561 055	(20,13)	13	00000000010010100000110	1.866 818
28	01001100000110001100010	1.983 764	(20,13)	13	11010011000111111111011	3.099 568
29	11111100101000001111110	2.450 350	(20,4)	6	01000010010000100101010	1.445 675
30	10001100100001001000010	1.981 748	(20,4)	6	00010000110010110111011	0.726796

第 1 代统计：

总交叉操作次数 = 14, 总变异操作数 = 30

最小适应度：0.726 796　最大适应度：3.422 902　平均适应度 2.082 725

迄今发现最佳个体 = > 所在代数：1　适应度：3.422 902　染色体：11000111100101100010111

对应的变量值：1.456 889

表 2.8　模拟计算统计报告(8～9 代)

个体	世代数 7 染色体编码	适应度	父个体	交叉位置	世代数 8 染色体编码	适应度
1	11001110110110011111101	3.031 261	(10,7)	3	10100001010100111111011	2.742 680
2	11000001001100011110101	3.003 174	(10,7)	3	10011011001000001101101	2.504 004
3	10001011111001111111110	2.068 714	(27,14)	6	11001101010110001111110	2.443 574
4	11101010100111111111011	3.103 389	(27,14)	6	00000011111110001100101	1.053 324
5	01000011011011111001111	3.803 827	(27,30)	2	11001000100100011100001	2.586 133
6	01010001110101111111011	3.003 873	(27,30)	2	00001101111110001110100	2.443 441
7	10000001010100111111011	2.742 550	(7,21)	18	10000001010100111111100	2.247 591
8	00111111001000011110111	1.445 575	(7,21)	18	11111111110101011111101	3.093 265
9	11001111111010011110111	1.633 465	(26,3)	9	10110110011001111111110	2.072 600
10	10111111001000011110101	2.988 837	(26,3)	9	00001011111110110010111	0.359 490
11	10110010100100111111111	1.384 168	(17,7)	21	10100010111100011010100	2.103 103
12	11100011001100110101101	2.196 195	(17,7)	21	10000001010100111110011	3.091 589
13	01111110110101111111011	3.003 525	(21,18)	12	11110111110101111110111	2.495 166
14	00000001010110001111110	2.443 669	(21,18)	12	11110111101001010101100	1.986 367
15	10011111001000011110111	1.445 461	(22,16)	0	10011010100101011111011	2.455 440
16	01001110100100001111001	2.848 732	(22,16)	0	01001110101100011111001	2.847 207
17	10100010111100110101101	2.210 419	(30,5)	7	00001001011011111001111	3.804 290
18	10110111001000011110111	1.445 004	(30,5)	7	01000010100100000111001	2.668 579
19	01111110010110011111110	2.265 397	(14,21)	8	00000001111101011110100	2.451 121
20	01011011110101111011110	2.449 366	(14,21)	8	11111111010110001111110	2.443 431
21	11111111110101011110100	2.451 027	(11,24)	5	10110100110100001111101	2.961 478
22	10001010100101011111011	2.455 161	(11,24)	5	11011010100100111111111	1.384 760

续表 2.8

个体	世代数 7 染色体编码	适应度	父个体	交叉位置	世代数 8 染色体编码	适应度
23	11101010100111011110100	2.438 146	(20,6)	2	01000001110001111101011	0.886 470
24	11111100110100011111101	2.961 797	(20,6)	2	01001011010101010011111	3.808 676
25	01111111001101011111011	2.487 896	(4,16)	15	11001010100111111111001	2.641 112
26	10110110011100101101101	2.833 003	(4,16)	15	01001110100100011111011	2.165 550
27	11001111111100011101101	3.009 112	(30,1)	0	00001001100110111101000	1.474 243
28	11111010000110101111101	2.695 039	(30,1)	0	11001110111111001111101	3.047 101
29	01001011010100111111011	2.745 186	(4,19)	8	01101010010110010111110	2.384 070
30	00001000100100011111001	2.848 854	(4,19)	8	00111111010011111111011	3.104 397

第 8 代统计：

总交叉操作次数＝94，总变异操作数＝253

最小适应度：0.359 490　最大适应度：3.808 676　平均适应度 2.391 675

迄今发现最佳个体＝＞所在代数：8　适应度：3.808 676　染色体：01001011010101010011111

对应的变量值：1.843 779

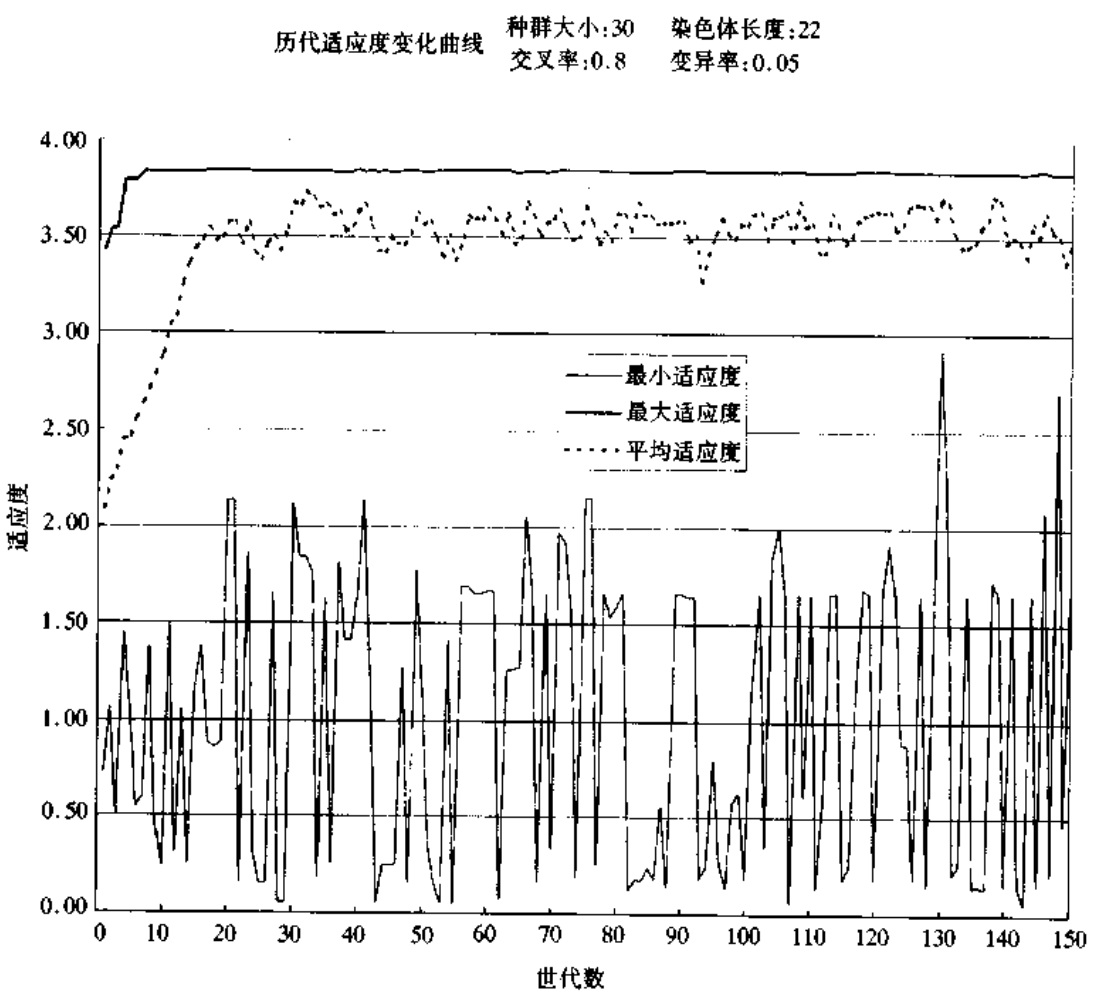

图 2.15　历代适应度进行曲线

参考文献

[1] Goldberg D E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA, Addison - Wisely, 1989

[2] Michalewicz Z. Genetic Algorithms + Data Structures = Evolution Programs. Springer - Verlag, Second, Extended Edition, 1994

[3] Blickle T, Thiele L. A Mathematical Analysis of Tournament Selection. In: Proceedings of the 6th International Conference(ICGA95), Morgan Kaufmann, San Francisco, CA, 1995

[4] Vose M D. Modeling Simple Genetic Algorithms. In: Foundations of Genetic Algorithms II, Morgan Kaufmann Publishers, 1993, 63～73

[5] Blickle T, Thiele L. A Comparison of Selection Schemes Used in the Genetic Algorithms. TIK - Report, Swiss Federal Institute of Technology(ETH), 1995

[6] Muhlenbein H. Evolutionary Algorithms: Theory and Applications. GMD Schloss Birlinhoven, 1995

[7] Syswerda G. Uniform Crossover in the Genetic Algorithms. In: Proceedings of the 3th International Conference(ICGA89), Morgan Kaufmann Publishers, 1989

[8] Whitely D. The GENITOR Algorithm and Selection Pressure: Why Rank - based Allocation of Reproductive Trials is Best. In: Proceedings of the Third International Conference(ICGA89), Morgan Kaufmann Publishers, 1989

[9] Goldberg D E, Deb K. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: Foundations of Genetic Algorithms, Morgan Kaufmann Publishers, 1989, 69～93

[10] Whitely D. Fundamental Principles of Deception in Genetic Search. In: Foundations of Genetic Algorithms, Morgan Kaufmann Publishers, 1989, 221～241

[11] Muhlenbein H. Optimal Interaction of Mutation and Crossover in the Breeder Genetic Algorithm. GMD Schloss Birlinhoven, 1995

[12] Goldberg D E. The Existential Pleasures of Genetic Algorithms. In: Genetic Algorithms in Engineering and Computer Science, Winter G(ed). Wiley, 1995, 23～31

[13] Satyadas, Krishnakumar K. Genetic Algorithm Modules in MATLAB: Design and Implementation Using Software Engineering Practices. In: Genetic Algorithms in Engineering and Computer Science, Winter G (ed). Wiley, 1995, 321～344

[14] Annie S W. Non - Coding DNA and Floating Building Blocks for the Genetic Algorithm. Doctoral Dissertation, Department of Computer Science and Engineering, The University of Michigan, 1996

[15] Manderick, de Weger M. The Genetic Algorithm and the Structure of the Fitness Landscape, . In: Proceedings of the Fourth International Conference on Genetic Algorithms, CA, Morgan Kaufman, 1991, 143～150

[16] 章珂, 刘贵忠. 交叉位置非等概率选取的遗传算法. 信息与控制, 1997, 26, (1): 53～60

[17] 徐洪泽, 陈桂林, 张福恩. 遗传算法单双点交叉方法的比较研究. 哈尔滨工业大学学报, 1999, 30 (2): 63～71

[18] 张晓缋, 方浩等. 遗传算法的编码机制研究. 信息与控制, 1997, 26(2): 134～139

[19] 张晓缋, 戴冠中等. 遗传算法种群多样性的分析研究. 控制理论与应用, 1998, 15(1): 17～22

[20] 辛香非, 朱鉴鑫. 遗传算法的适应度函数研究. 系统工程与电子技术, 1998(11): 58～62

[21] 王秀峰. 实数编码的遗传算法及其在逆变器馈电交流电机中的应用. 自动化学报, 1998, 24(2): 250～253