Chapter 2
Simple Evolutionary Algorithms

Abstract In Chap. 1, we drew a graph for EAs with the statement that EAs are interesting, useful, easy-to-understand, and hot research topics. Starting with Chap. 2, we will demonstrate EAs in a pedagogical way so that you can enjoy the journey around EAs with active reading. We strongly encourage readers to implement their basic EAs in this chapter in one programming environment and improve its search ability through other chapters. Footnotes, exercises, and possible research projects are of great value for an in-depth understanding of the essence of the algorithms.

2.1 Introductory Remarks

Before introducing some standard EAs,[1] we need to standardize some terms used throughout the book.

Classical EAs, including genetic algorithms (GAs), evolution strategy (ES), evolutionary programming (EP), and genetic programming (GP),[2] are all random-based solution space searching metaheuristic algorithms. So the most important thing before discussing a concrete algorithm is how to generate and manipulate random numbers in a programming environment.

We summarize the functions for handling random numbers in MATLAB®, C/C++, and Java in Table 2.1. As you can see, MATLAB® has advantages in generating random numbers in a flexible way. So we suggest using MATLAB® as the programming environment while learning EAs.

Even though MATLAB® has provided a Genetic Algorithm and Direct Search Toolbox, we strongly suggest that readers make their own EA source code from scratch if your purpose in reading this textbook is to really understand how EAs work and maybe improve some famous EAs to some degree.

[1] We sometimes call these standard evolutionary algorithms "simple" algorithms in the sense that they are simple compared to the improvements introduced in Chap. 3. But we need to mention that simple algorithms here do not mean weak performance.

[2] We will discuss the first three algorithms in this chapter and introduce GP in Chap. 10.

Table 2.1 The most common functions related to random numbers

	Distribution	MATLAB®	C/C++	Java
1	Uniform distribution $U(0,1)$	rand	(float)rand()/ RAND_MAX	Math.random
2	Normal distribution $N(0,1)$	randn		nextGaussian
3	Random permutation between 1 and integer n	randperm		
4	Round towards infinity	ceil	ceil	ceil
5	Round towards negative infinity	floor	floor	floor
6	Round towards nearest integer	round		

The density function of a uniform distribution random number in the range $(0,1)$, denoted as $\xi \sim U(0,1)$, is as follows:

$$p(\xi) = \begin{cases} 1 & 0 < \xi < 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Every programming language provides a uniform distribution[3] random function. After generating $\xi \sim U(0,1)$, there are many ways to convert it into other distributions [1, 2]. The most important of these are normal distributions, denoted $\eta \sim N(\mu, \sigma^2)$, whose density function is as follows:

$$p(\eta) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\eta-\mu)^2}{2\sigma^2}} \tag{2.2}$$

where μ is the expectance and $\sigma > 0$ is the standard deviation. It is easy to verify that if $\xi \sim N(0,1)$, then $\eta = (\xi \times \sigma + \mu) \sim N(\mu, \sigma^2)$. So $N(0,1)$ is very critical for generating normal distribution random numbers.

A permutation in the range $[1,n]$ is often used in evolutionary combinatorial optimization. Thus one should be familiar with how to generate permutations and random permutations.

Sometimes we need to generate an integer random number with uniform distribution in the range $[1,n]$, where n is an integer number. We can either combine the round function and $n * rand()$ or select the first cell in the random permutation in the range $[1,n]$.

In EAs, we often say that an operator (such as a crossover or mutation, discussed later) needs to be carried out with probability p. How does one implement such a simple statement in a given programming environment? We give the example in MATLAB ® as follows.[4] These are similar in other environments.

[3] Also called a Gaussian distribution.

[4] We strongly encourage readers to make it clear why this "If" statement could represent that the operator needs to be carried out with probability p.

```
%Operator A is carried out with probability p
        If rand < p
            Operator A
        End
```

where $rand \sim U(0,1)$.

With respect to *selection*, there are two ways to carry it out. We often pick up one solution randomly from current solutions and determine whether it could be selected. We will discuss the selection criteria in detail in later sections of this chapter and in Chap. 3. Here we would just like to determine how to handle the one being picked up. If it is put back, regardless of whether it is selected or not, and has the chance to be picked up again in the next time, we call this *selection with replacement*. If it is discarded, regardless of whether it is selected or not, and will never be picked up again, we call this *selection without replacement*.[5]

Another term that often appears in the EA literature is *norm*, which could be seen as some kind of length measure of vectors. $||\mathbf{u}||$ is vector $\mathbf{u}$'s norm. Suppose $\mathbf{u}$ is a real vector with n variables. The general $p(\geq 1)$ norm for $\mathbf{u}$ is

$$\|\mathbf{u}\|_p = (|u_1|^p + \cdots + |u_n|^p)^{1/p} \tag{2.3}$$

where $|u_j|$ is the absolute value of u_j. If $p = 1$, then Eq. 2.3 is *1-norm*; it is the sum of the absolute values of all cells.

$$\|\mathbf{u}\|_1 = (|u_1| + \cdots + |u_n|) \tag{2.4}$$

If $p = 2$, *2-norm*, it is the Euclidean distance.

$$\|\mathbf{u}\|_2 = \sqrt{u_1^2 + \cdots + u_n^2} \tag{2.5}$$

If $p = \infty$, *∞-norm*.

$$\|\mathbf{u}\|_\infty = \max(|u_1|, \cdots, |u_n|) \tag{2.6}$$

The final concept is *convex function* and *concave function*. A function f is convex if any two points x_1 and x_2 in the definition domain satisfy Eq. 2.7 for any $0 \leq r \leq 1$, which is illustrated in Fig. 2.1.[6]

$$f(rx_1 + (1-r)x_2) \leq rf(x_1) + (1-r)f(x_2) \tag{2.7}$$

For points x_1 and x_2, $rx_1 + (1-r)x_2$ is their *convex combination*. Inverting the inequality defines a concave function.

[5] Sometimes we call this *sampling with replacement* or *sampling without replacement*.

[6] To call a curve convex or concave according to its geometric shape depends on where it is facing. In the definition, we look at the origin.

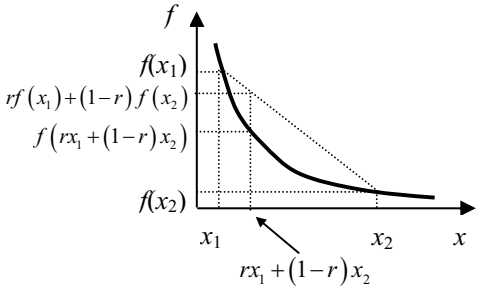

Fig. 2.1 A convex function

2.2 Simple Genetic Algorithm

2.2.1 An Optimization Problem

The basic ideas of GAs were introduced in Chap. 1. In this section, we'd like to introduce the details of implementing GAs with an optimization algorithm. The problem is formulated by Eq. 2.8 and illustrated by Fig. 2.2.

$$\begin{aligned} &\max f(x) = x\sin(10\pi x) + 2.0 \\ &\text{s.t.} \quad -1 \le x \le 2 \end{aligned} \tag{2.8}$$

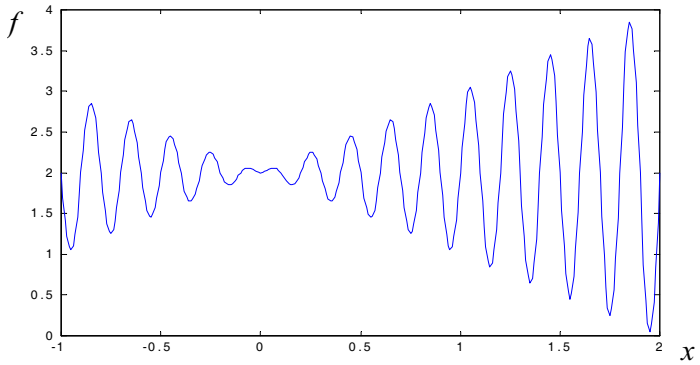

Fig. 2.2 A numerical example of an SGA

The curve of the objective function in Fig. 2.2 constitutes the *solution landscape* in which we are searching for the optimal solution. In this problem, it has many local optima, and, based on Fig. 2.2, the problem may seem difficult. We will demonstrate the strength of *simple genetic algorithm* (SGA) on this problem by the discussion in the following subsections.

In SGA, we maintain many *individuals* in a *population*. The number of individuals in the population is the *population size* (*popsize*). Each individual[7] has two properties: its location (*chromosome*[8] composed of *genes*) and its quality (*fitness value*). After obtaining the quality of all individuals, we use the selection process to generate a *mating pool*. Individuals with higher quality should have a higher probability of being selected into the mating pool so that the good ones will have more chances to breed and the bad ones will not be selected. Individuals in the mating pools are called *parents*. Generally two parents might be selected randomly from the mating pool to generate two *offspring*[9] without replacement and every offspring might undergo small changes to become a new individual. Then the newly generated population replaces the old one and another *generation* starts.

The relationship between the concepts and the operators described above and the principle of Darwinian natural selection theory is listed below.

- Selection $\Longleftrightarrow$ survival of the fittest
- Two parents generate two offspring $\Longleftrightarrow$ crossover or recombination
- Small changes in the location of the offspring $\Longleftrightarrow$ mutation

A good individual has more chances to be selected into the mating pool so that it has more chances to mate than low-quality individuals have. Then the information contained in the good individual has more chances to be preserved and passed onto next generation. Information exchange between two parents and small changes in the offspring promote the search for better individuals. Combining these two factors, the population will become more and more fit until the optimal or near optimal solution has been found, if we are lucky. In this way, GAs could gain favor over traditional gradient-based algorithms. We will discuss these operators step by step.

2.2.2 Representation and Evaluation

We can use a real number, in the range $[-1,2]$, to represent a solution in Eq. 2.8 directly. Many operators can handle real number representation. But we use the *binary code* or *binary representation* here for two reasons. GAs were originally proposed to be binary code to imitate the genetic encoding of natural organisms. On the other hand, binary code is good for pedagogy. A binary chromosome is necessary to represent a solution x in the scale $[-1,2]$. The same holds for the binary representation of real numbers in a computer.

In binary code, we cannot represent a real number completely correctly, so a tradeoff is necessary. A tolerance needs to be defined by the user, which means the errors below the tolerance are extraneous. If we divide the definition domain into

[7] An individual may be understood as an agent.

[8] Sometimes a chromosome is called a *genome*. These two terms have different meanings in genetics and biology, but we disregard them in EAs.

[9] Sometimes they are called *progeny* or *descendants*.

$2^1 = 2$ parts evenly and select the smallest number in the parts to represent any number in the division, we can only represent -1 and 0.5 by 0 and 1 respectively. $2^2 = 4$ divisions make the 00, 01, 10, and 11 represent -1, -0.25, 0.5, and 1.25, respectively. The larger division number we select, the less error there is in representing a real number on binary code. Suppose we use 100 binary codes to represent a real number in the range $[-1,2]$; the maximum error is $3/2^{100} \approx 2.37^{-30}$, which would be satisfactory for most users. In this way,[10] we can represent a real number with any accuracy requirements.

In this problem, we use $l = 12$ binary codes to represent one real number[11] as follows, which constitutes a chromosome to be evolved.

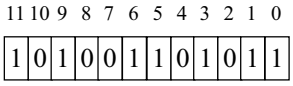

Fig. 2.3 Binary representation of a real number

For 12 binary codes in the chromosome, every part is called a *gene*. A gene has two properties: its value (sometime called *allele*), which is the number in the square, and its location (sometimes called *locus*), which is the number above the square. For the binary representation of a real number with l genes, its counterpart real number is

$$x = \frac{\sum_{i=0}^{l-1} a_i 2^i}{2^l} \times \left(\bar{x} - \underline{x}\right) + \underline{x} \tag{2.9}$$

where a_i is the allele of locus i, $\bar{x}$ and $\underline{x}$ are the upper and lower bounds for the real number, respectively. For the chromosome in Fig. 2.3, its counterpart real number is $x = \left(2^{11} + 2^9 + 2^6 + 2^5 + 2^3 + 2^1 + 2^0/2^{12}\right) \times (2+1) - 1 = 0.9534$. The process of finding one way to represent a solution to the problem is called *representation*. Figure 2.3 illustrates binary *encoding* and Eq. 2.9 represents binary *decoding*.

We can use both 101001101011 and 0.9534 to represent an individual (or a chromosome); the former is called a *genotype* representation of an individual and the lattera *phenotype* representation of an individual.

Every individual has two properties: its location and its quality. The location is the chromosome we described above and the quality is its objective value, which can be evaluated by Eq. 2.8. The objective value of our example individual is $f(x) = 0.9534 \times \sin(10\pi \times 0.9534) + 2.0 = 1.0520$. In GAs, we often use the *fitness value* to evaluate how much the individual fits the problem. So the function used to calculate the fitness value is called the *fitness function*, which is exactly the same as

[10] Even though this might weaken your faith in computers.

[11] What is the maximum error for this representation? Why do we only use $l = 12$? Why not $l = 300$?

the objective function in Eq. 2.8.[12] The process of obtaining a fitness value from a chromosome is called *evaluation*.

2.2.3 Initialization

SGA operates on a group of individuals. Generally the algorithm designers need to define how many individuals will be in the population, which is often represented by a variable *popsize*.[13] We need to generate *popsize* individuals to start the evolving process. This procedure is called *initialization*.

Sometimes we know which part of the definition domain contains better solutions. But in most situations, we do a blind search over the solution landscape.[14] So we could just initialize the population randomly. Another reason for random initialization is that SGA is capable of a global search, which will be illustrated later. In some complicated real-world problems, workable solutions may be in the local optimal solution's *domain of attraction*[15] so that starting from this domain may not be good for a global search. So why not depend on the global search ability of the SGA?

There are many ways to initialize *popsize* individuals randomly. The simplest way might be to generate every individual with uniform distribution. Specifically, for every gene in the chromosome, its allele is 1 with probability 0.5, and *vice versa*.[16]

Another way to generate evenly distributed individuals in the definition domain is to divide the domain into several grids. Initially, a grid is randomly selected and a solution in that grid is in turn randomly selected. We need the encoding procedure to transfer the phenotype to the genotype to get the chromosome.[17] We count the

[12] After reading the Sect. 2.2.4 below, consider why we could use the objective value as the fitness value directly. What is the fitness function in other situations, i.e., with a negative objective value, minimum optimization, etc.? Chap. 3 will discuss this interesting problem.

[13] Defining *popsize* without any *a priori* knowledge about the problem is a very hard job for algorithm designers. So either we need some kind of trial-and-error adjustment or we adopt some information from a current population and change *popsize* according to that information. We will discuss the latter interesting idea in Chap. 3.

[14] The results of other optimization algorithms, currently workable solutions, and uneven sampling on the definition domain according to the preference of users are examples of nonblind initialization.

[15] The domain of attraction means a subset of the definition area. For some search technologies, starting at any point in the domain of attraction will converge to the optimum in that domain even if it is a local optimum. Consider any $x \in [-0.9, -0.8]$ in Fig. 2.2; it will converge to the local optimum $f = 2.8$ with any up-hill algorithm.

[16] Readers unfamiliar with this should review Sect. 2.1.

[17] In our binary representation example, readers may check for methods of transforming a real number into its binary code in the computer basis textbook.

number of individuals generated from a grid. The larger the number for a grid is, the less opportunity the grid will have to generate new ones.[18]

After the initial chromosomes have been generated, their fitness values are calculated using a decoding process (Eq. 2.9) and the objective function (Eq. 2.8). It bears mentioning again that an individual is comprised of a chromosome and a fitness value.

For the problem illustrated by Eq. 2.8 and Fig. 2.2, we set $popsize = 10$ and obtain the ten randomly generated initial individuals, illustrated by Fig. 2.4.

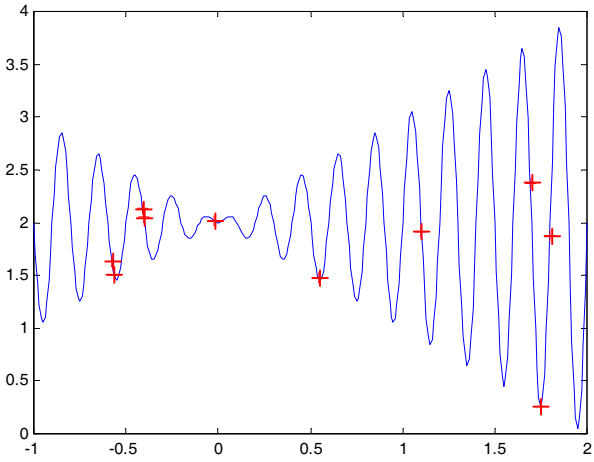

Fig. 2.4 Initial population of an SGA

The crosses in Fig. 2.4 are the initial individuals. As can be seen, the performance of the initial population is not impressive at all.

2.2.4 Selection

After initialization, the SGA enters the main loop. It starts with a *selection process*, which imitates natural selection by granting fitter individuals higher opportunity to breed, and ends with two *variation operators*,[19] crossover and mutation, which imitate natural reproduction by exchanging genes of parents to generate offspring.

In programming, we need to open another memory to reserve the individuals selected to breed. This memory is called the *mating pool*.

[18] Consider how to implement this idea in your program.

[19] These operators are also called *reproductive operators*.

There are many ways to embody the idea of natural selection. We will discuss one here and introduce others in Chap. 3. Individual i[20] in the current population has a fitness value f_i . According to natural selection, fitter individuals have more advantages in breeding. So we could define the *relative fitness value* of individual i as follows:

$$p_i = \frac{f_i}{\sum_{i=1}^{popsize} f_i} \tag{2.10}$$

It is easy to verify that $\sum_{i=1}^{popsize} p_i = 1$. Thus the relative fitness value can also be seen as the *probability of being selected* for individual i, i.e., p_i could be seen as the probability of being selected as one candidate in the mating pool for individual i.[21]

How can we implement this in a programming environment? We can do the selection as in roulette way.[22] The difference between real roulette and our selection is that the holes of the roulette wheel in our selection have different sizes. The larger p_i is, the larger hole individual i has. So the ball could drop into the hole with higher probability. The size of the hole for i is propositional to p_i, which is illustrated by Fig. 2.5. Instead of digging a hole on the wheel, we use sectors with different central angles to represent individuals. The central angle of individual i is $2\pi p_i$. The thick arrow in Fig. 2.5 represents the ball. We want the arrow to start rotating clockwise and stop randomly with a central angle $2\pi \cdot rand$, where $rand \sim U(0,1)$. If the arrow stops at one sector, the corresponding individual is selected into the mating pool. It is clear that fitter individuals have more chances to be selected.

How is the roulette wheel implemented in a programming environment? We could simulate the rotation process by an accumulated process. After obtaining *rand*, we know where the arrow will be. We memorize the individual we have already passed during the rotation process, accumulate the probabilities, and compare it to *rand*. Whenever we find an individual that is satisfied that $\sum_{i=1}^{k} p_i < rand < \sum_{i=1}^{k+1} p_i$, we know that the arrow stops in sector i[23] and select individual i into the mating pool. Then the arrow returns to the original location, just like in Fig. 2.5. We can do the above procedure *popsize* times until there are *popsize* individuals in the mating pool. This selection procedure is called *roulette wheel selection* (RWS).

In this way, some individuals in the population will be selected more than once and some will never be selected. The probability of being selected for individual i is its relative fitness. It is also necessary to mention that fitter individuals are not to be selected by RWS if they are unlucky enough never to have the arrow stop at their sector *popsize* times. We call this phenomenon *selection bias*. Many studies have

[20] Sometimes we also use ind_i to represent individual i.

[21] This sentence has the implicated meaning that we want to select the candidates in the mating pool in a serial way. We will introduce a parallel way in Chap. 3.

[22] Roulette is a gambling game in which a ball is dropped onto a wheel with numbered holes in it while the wheel is spinning round.

[23] How can we make such as statement?

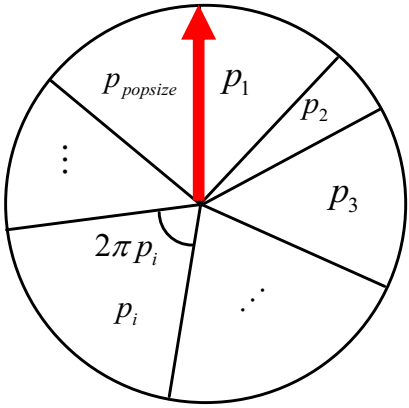

Fig. 2.5 Roulette wheel selection

been done to minimize the selection bias, and we will introduce some of them in later chapters.

Another consideration about RWS is that the problem needs to be maximum and all the objective values need to be greater than zero so that we can use objective values as fitness values and take RWS as the selection process directly.[24]

2.2.5 Variation Operators

There are many *variation operators* to change information in individuals in the mating pool. If information exchange, i.e., gene exchange, is done between two or more individuals[25], this variation operator is called *crossover* or *recombination*. If the genes of one individual changes on its own, this variation operator is called *mutation*. We will introduce single-point crossover and bit-flip mutation here.

There are two ways to select two individuals in the mating pool to determine whether or not to cross over them. One is to shuffle the mating pool randomly and assign individuals 1 and 2 without replacement to be a crossover pair, 3 and 4 without replacement to be another pair, etc. The other is to generate a random integer permutation, per, between $[1, popsize]$. $per(i) = j$ means the ith element in the permutation is the jth individual in the mating pool. Then we assign $ind_{per(1)}$ and $ind_{per(2)}$ without replacement as the first crossover pair, $ind_{per(3)}$ and $ind_{per(4)}$ without replacement as the second crossover pair, etc.

[24] Why do we need two such requirements for RWS to handle objective values directly?

[25] We will give examples of multiparent crossover in Chap. 3.

Generally, we will assign the probability of crossover p_c, called the *crossover rate*, to control the possibility of performing a crossover.[26]

For two individuals selected to cross over, we assign a point between 1 and $l-1$ randomly, where l is the length of the chromosome. This means generating a random integer in the range $[1, l-1]$. The genes after the point are changed between parents and the resulting chromosomes are offspring. We call this operator a *single-point crossover*. Figure 2.6 illustrates this.

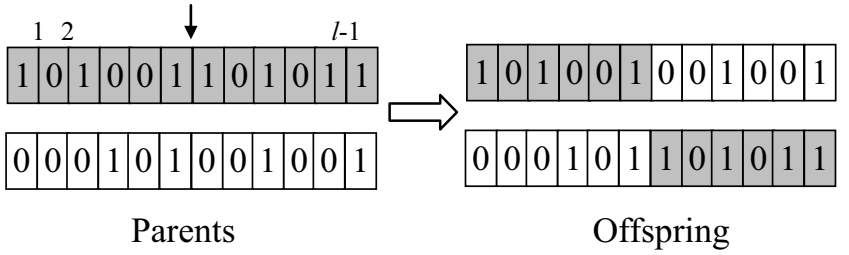

Fig. 2.6 Single-point crossover

As can be seen from Fig. 2.6, two new individuals are generated by crossover, which is generally seen as the major exploration mechanism of SGA.

If two parents do not perform a crossover according to probability p_c, their offspring are themselves.

Now we discuss mutation. There are also two ways to implement mutation. One way is to open another memory with size *popsize* to store the results of crossover, and mutation is carried out in that memory. The other way is to mutate the offspring of crossover directly. We use the latter way.

For every gene in an individual, we mutate it with probability p_m, called the *mutation rate*.[27] Provided gene j needs to be mutated, we make a bit-flip change for gene j, i.e., 1 to 0 or 0 to 1. We call this operator a *bit-flip mutation*. Figure 2.7 illustrates the bit-flip mutation. The individual after mutation is called the *mutant*.

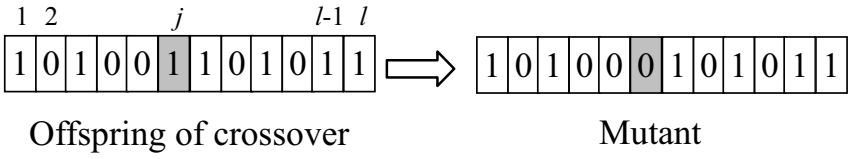

Fig. 2.7 Bit-flip mutation for gene j of the offspring

[26] How do we implement the statement "Individual i and individual j cross over with probability p_c"?

[27] How do we implement the statement "Gene j mutates with probability p_m"?

As can be seen from Fig. 2.7, small changes are introduced into a chromosome by bit-flip mutation,[28] which could be seen as one local search method and is generally considered a minor exploring mechanism of SGA. If every gene of an offspring does not mutate according to probability p_m, the mutant is the offspring itself.

After the variation operators, a new population with *popsize* individuals is generated. We need to evaluate the fitness for every individual and then replace the old population with the new one.[29] The process of replacing the current population with the new population is called *replacement*.

Although we call the results of crossover offspring and the results of mutation mutants, offspring is also used to represent the results after performing all variable operators.

2.2.6 Simple Genetic Algorithm Infrastructure

The selection, crossover, mutation, and replacement discussed above constitute one *generation* or *iteration* of an SGA. Then the evolving process continues to the next generation. As mentioned above, selection grants fitter individuals greater odds opportunity of propagating their high-quality genes, and crossover and mutation explore the solution landscape. We could have a rational expectation that the population will become better and better. But when do we stop? Here we just assign the maximum generation number *maxgen*.[30] If the generation number exceeds *maxgen*, the SGA stops and the individuals in the final population are the results.

So the infrastructure, *solution process*, of an SGA can be illustrated as follows:

Solution Process of SGA

Phase 1: Initialization.

Step 1.1: Assign the parameters for SGA, such as p_c, p_m, *popsize*, *maxgen*, etc.

Step 1.2: Generate *popsize* uniformly distributed individuals randomly to form the initial population and evaluate their fitness values. $gen = 0$.

Phase 2: Main loop. Repeat the following steps until $gen > maxgen$.

Step 2.1: Select *popsize* individuals from current population using RWS to generate the mating pool.

[28] How large a perturbation will the bit-flip mutation introduce in a chromosome? Does something not seem right? We will discuss this question in Chap. 3.

[29] It seems cruel and unreasonable because perhaps we want to keep the good ones in the current population. How do we solve this problem? We will discuss it in Chap. 3. Sect. 2.3 also gives some hints.

[30] We will discuss other more flexible techniques to stop EAs in Chap. 3.

Step 2.2: Repeat the following operations until a new population with *popsize* individuals has been generated. Select two individuals from the mating pool randomly without replacement to perform single-point crossover with probability p_c, and perform bit-flip mutation for every gene of the offspring with probability p_m. Then insert the mutant into the new population.

Step 2.3: Evaluate the fitness value for every new individual in the new population.

Step 2.4: Replace the current population with the new population. $gen = gen + 1$.

Phase 3: Submitting the final *popsize* individuals as the results of the SGA.

For the problem described by Eq. 2.8 and Fig. 2.2, we use an SGA with $popsize = 10$, $maxgen = 10$, $p_c = 0.8$, and $p_m = 0.01$[31] and obtain the results illustrated in Fig. 2.8. The cross in Fig. 2.8 represents the final individuals.

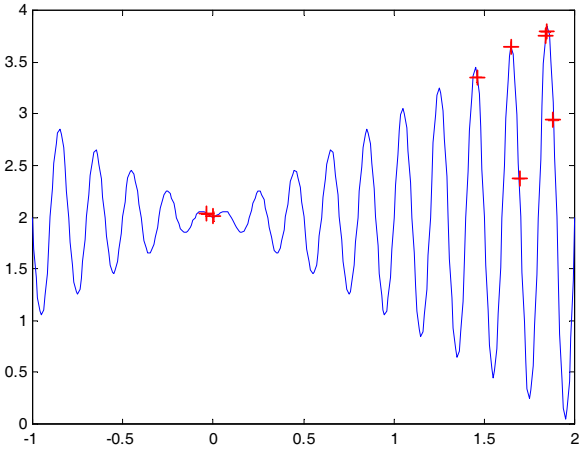

Fig. 2.8 Final population of SGA

As can be seen from comparing Figs. 2.2 and 2.8, an SGA can find a point very close to the global optimal solution in $10 \times 10 = 100$ samples over a solution landscape without any requirement of gradient information for such a not-so-easy problem, which could illustrate that SGA, with the help of selection, crossover, and mutation, is an effective search method.

To demonstrate the evolving process of SGA, we can draw a graph with the horizontal axis representing generation, *gen*, and the vertical axis representing the best fitness values in one generation, f_{best}. For our implementation, the graph is

[31] Why do we assign such a small p_m?

Fig. 2.9. The global optimal solution is $f(1.85) = 3.85$ and the SGA finds 3.8 at generation 9.

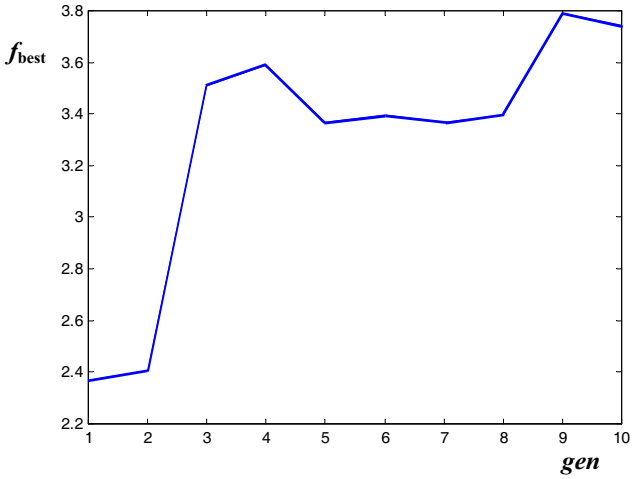

Fig. 2.9 The evolving process of the best fitness value

As can be seen from Fig. 2.9, the randomly generated initial best fitness value is close to 2.4 and it evolves to about 3.8 with only nine generations, which could demonstrate the global search and local fine-tuning ability of the SGA. It is necessary to mention two considerations for Fig. 2.9. The first thing is that connecting the best values of different generations with direct lines is meaningless because the points on the lines do not have any meaning. But we'd like to use this form to emphasize that EAs are evolutionary processes, even though they are implemented in a discrete environment. The second thing is that after implementing and running an SGA one time, drawing the results as in Fig. 2.8 and the evolving process as in Fig. 2.9 is far from thoroughly evaluating the global and local search ability of an algorithm. We will discuss this important issue, statistical performance evaluation, in Chap. 3.

Figure 2.10 illustrates the number of papers indexed by the SCI on GAs).[32]

From Fig. 2.10 we can say that GA is becoming more and popular in recent days.

[32] TS = ("genetic algorithm" OR "genetic algorithms"). The SCI index "TS" is for the search topic in the title, the keywords, and the abstract.

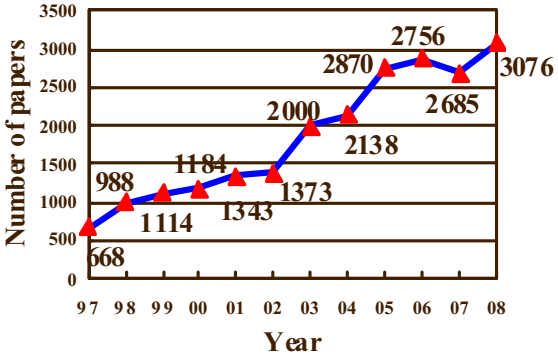

Fig. 2.10 Number of papers on GAs indexed by SCI

2.3 Evolution Strategy and Evolutionary Programming

2.3.1 Evolution Strategy

In designing an ES, Rechenberg and Schwefel use real numbers to represent alleles of genes and make normal distribution mutation the most important exploration technique over a solution landscape.

Let us take the simple example described by Eq. 2.8 and Fig. 2.2 again to illustrate the procedure of ES.

Suppose we have μ individuals in the current population. For every individual,[33] its chromosome is x, which is a real number in the range $[-1,2]$. We first randomly select two of them, x_1 and x_2, to do the crossover and generate one offspring x as

$$x = \frac{x_1 + x_2}{2} \tag{2.11}$$

This crossover operator is called an *intermediate crossover*.[34] There are other options for crossover operators in ES. Because crossover is considered a minor exploring tool in ES by Rechenberg and Schwefel, we do not discuss it here. Real code crossover operators will be discussed in Chap. 3.

We complete the crossover λ times to generate λ offspring (λ is often larger than μ[35]). For every offspring, we want to give a normal distribution disturbance on every variable (the example has only one variable).

For a normal distribution $N\left(\xi,\sigma^2\right)$ with mean ξ and standard deviation σ, it can be generated by $N\left(\xi,\sigma^2\right) = \xi + \sigma N(0,1)$, where $N(0,1)$ is a standard normally distributed random number with mean 0 and standard deviation 1. In MATLAB®,

[33] Without any preference for fitter individuals.

[34] Why?

[35] So in the previous crossover, we need to sample parents with replacement.

the *randn* function can generate $N(0,1)$ directly. For other programming environments, readers should refer to textbooks on probability for generating normally distributed random numbers from uniformly distributed random numbers [1].

In ES, we often want to make changes for every variable based on its current value. So we only use σ to represent the scale of mutation in ES.[36] For the ith variable x_i of one offspring, execute

$$x_i^{'} = x_i + N_i\left(0,\sigma^2\right) = x_i + \sigma N_i(0,1) \tag{2.12}$$

where $x_i^{'}$ is the mutant of x_i.[37] By using Eq. 2.12 for every gene of every offspring, we can get new individuals. This mutation operator is called a *normal mutation*. Then their fitness values can be calculated using the objective function. The standard deviation σ might be the same for all variables and it also might be different for every variable, depending on the algorithm's designer.

Then we combine μ current individuals and λ new individuals and then pick the μ best ones according to their fitness values to form new population. So the solution process of ES can be illustrated as follows:

Solution Process of $(\mu+\lambda)$*-ES*

Phase 1: Initialization.

Step 1.1: Assign the parameters for ES, such as λ, μ, and σ.

Step 1.2: Generate μ uniformly distributed individuals randomly to form the initial population and evaluate their fitness values. $gen = 0$.

Phase 2: Main loop. Repeat the following steps until $gen > maxgen$.

Step 2.1: Repeat the following operations until a new population with λ individuals has been generated. Randomly select two individuals with replacement to perform crossover, such as intermediate crossover (Eq. 2.11). Then perform a mutation (Eq. 2.12) for every gene of the offspring. Then insert the mutant into the new population.

Step 2.2: Calculate the fitness value for every new individual in the new population.

Step 2.3: Combine μ current and λ new individuals and pick the μ best ones to form a new population. $gen = gen + 1$.

Phase 3: Submitting the final μ individuals as the results of ES.

ES differs considerably from SGA and we will leave it to the reader to discover the differences. Here we need to point out that difference in the replacement procedure. A new population will certainly replace the old one in SGA, but the new population

[36] What is the logical relationship between this sentence and the previous one?

[37] Why do Rechenberg and Schwefel choose a normal distribution instead of a uniform distribution? What is their initial design idea behind $N(0,1)$?

is selected from the union of λ new individuals and μ current individuals in ES. Obviously, the replacement mechanism in ES conserves the best individual. Apart from replacement, the coolest part of ES is perhaps its self-adaptive control of standard deviation σ , i.e., coding σ into the chromosomes. This will be discussed in Chap. 3.

In the above solution process, μ current individuals and λ new individuals are combined and the μ best ones form a new population. Thus we call it $(\mu+\lambda)$-ES. In another type, (μ, λ)-ES, μ current individuals are used to generate λ new individuals and the μ best ones among the λ new individuals form the new population.[38] There are also other types like $(1+1)$-ES, $(1+\lambda)$-ES, and $(1, \lambda)$-ES.

2.3.2 Evolutionary Programming

Fogel used EP to solve the learning problem and used a finite state machine to represent the chromosome, which causes some difficulties for solving optimization problems with EP. In the 1990s, many researchers developed EP into an optimization field and formed many types of EP. The most cited EP is listed as the following solution process, where real numbers are used to represent variables.

Solution Process of One Type of EP Implementation

Phase 1: Initialization.

Step 1.1: Assign the parameters for EP, such as λ, μ, and σ.

Step 1.2: Generate μ uniformly distributed individuals randomly to form the initial population and evaluate their fitness values. $gen = 0$.

Phase 2: Main loop. Repeat the following steps until $gen > maxgen$.

Step 2.1: Repeat the following operations until a new population with μ individuals has been generated. Perform a mutation (Eq. 2.12) for every gene of the individual to generate a new one.

Step 2.2: Calculate the fitness value for every new individual.

Step 2.3: Combine μ current and μ new individuals and pick the μ best ones to form a new population. $gen = gen + 1$.

Phase 3: Submitting the final μ individuals as the results of EP.

Thus the above listed implementation of EP can be regarded as a $(\mu+\mu)$-ES without crossover.

[38] SGA can be regarded as a $(popsize, popsize)$-ES if we only consider the replacement procedure. Here we would like to raise an interesting but important question: Is $(\mu+\lambda)$-ES always better than (μ, λ)-ES in optimization? The answer is no! In Chap. 3, we will introduce the dilemma of exploration and exploitation and discuss the famous No Free Lunch Theorem. This question may be revisited after reading these materials.

Figure 2.11 illustrates the number of papers indexed by the SCI on ES and EP.[39]

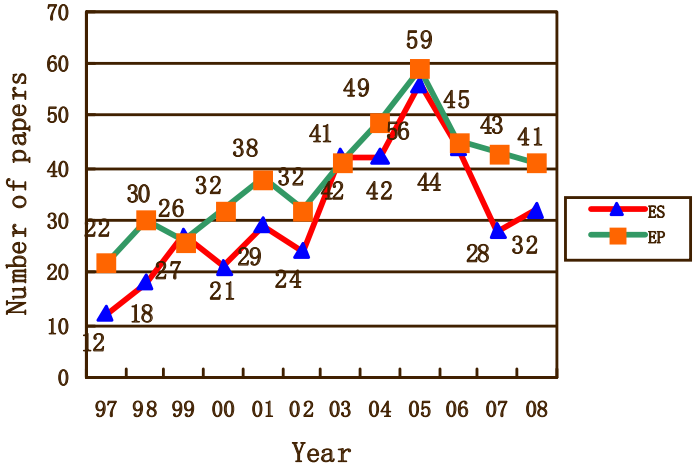

Fig. 2.11 Number of papers indexed by SCI on ES and EP

From Fig. 2.11 we can say that ES and EP attract a similar level of attention.

2.4 Direction-based Search

All the aforesaid algorithms are random-based, i.e., generating initial points, exploring new points, selecting better points, etc. In this section, we will first introduce a deterministic search method, simplex search, that can explore and exploit the solution space without the requirement for gradient information. Then two stochastic direction-based search methods, scatter search and differential evolution, are introduced. All of these algorithms use a specific direction, unlike SGA or ES discussed above, for generating new solutions. Thus we call them *direction-based search* methods.

2.4.1 Deterministic Direction-based Search

Methods that do not require gradient information to perform a search and sequentially explore the solution space are called *direct search methods*. There are many effective direct search methods, such as *simplex search*, *pattern search*, etc. All

[39] TS = ("evolution strategies") and TS = ("evolutionary programming"). The SCI index "TS" is for the search topic in the title, the keywords, and the abstract.

of them are based on the following philosophy. The methods maintain a group of points. They utilize some sort of deterministic exploration methods to search the objective space and almost always utilize a greedy method to update the maintained points.

Some of them use direction information, which might be the improvement directions, to search the objective space. Thus it might be very useful to embed these directions into one's evolutionary algorithm as either a local search method or an exploration operator.

2.4.1.1 Simplex Search

Nelder and Mead introduced the most famous deterministic direction-based search method, the *simplex search*, in 1965 [3]. Thus sometimes the simplex search is referred as the Nelder–Mead method. Do not confuse it with the simplex methods used in linear programming. But these algorithms use the same concept of simplex, which is a polytope with $n+1$ vertices in n dimensions: a line segment in one dimension, a triangle in 2-D, a tetrahedron in 3-D space, and so forth.[40]

In multidimensional spaces, the subtraction of two vectors means a new vector starting at one vector and ending at the other, like $(\mathbf{x}_2 - \mathbf{x}_1)$ in Fig. 2.12. Vectors in the space could be moved with their length and direction freely. So we often refer to the subtraction of two vectors as a direction.

The addition of two vectors can be implemented in a triangular way, moving the start of one vector to the end of the other to form an addition vector, like $\mathbf{x}_3 + (\mathbf{x}_2 - \mathbf{x}_1)$ in Fig. 2.12. We often refer to the addition of two vectors as a point.

The expression $\mathbf{x}_3 + (\mathbf{x}_2 - \mathbf{x}_1)$ can be regarded as the destination of a moving point that starts at $\mathbf{x}_3$ and has a length and direction of $(\mathbf{x}_2 - \mathbf{x}_1)$.

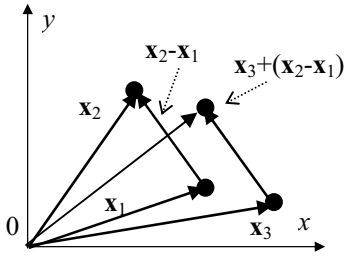

Fig. 2.12 Graphical meaning of the subtraction between two vectors

In a direct search, generally we cannot obtain the gradient information so that the idea of a step in the negative gradient direction for a minimum problem is impractical. But if the objectives of a group of solutions are available, we then know which one is the best one (suppose it is **B**). For any other solution (suppose it is

[40] A simplex can be considered the simplest set of points to make an effective search.

C), its improvement direction is unambiguously $\mathbf{C} \rightarrow \mathbf{B}$.[41] This intuitive idea is the foundation of many powerful direction-based search methods and EAs.

Now let us discuss the simplex search in the 2-D minimum optimization situation. There are three maintained points that are the vertices of a triangle, each with an objective value. Let us rank and name them by Eq. 2.13, which means the first point (B) will *always* be the best one, the second point (G) will *always* be the middle one, and the third point (W) will *always* be the worst one.

$$f(\mathbf{x}_\mathrm{B}) < f(\mathbf{x}_\mathrm{G}) < f(\mathbf{x}_\mathrm{W}) \tag{2.13}$$

Point B is the best one in the simplex and point W is the worst one, which means moving from W to B is a good search direction. Also moving from W to G is a good search direction. So why not combine these two considerations and move from W toward the centroid, gravity center, of B and G for a further step?

The gravity center of B and G is M. We think the direction from W to M is the optimizing direction.

$$\mathbf{x}_\mathrm{M} = \frac{\mathbf{x}_\mathrm{G} + \mathbf{x}_\mathrm{B}}{2} \tag{2.14}$$

We start from point W and go in the good search direction (W→M) and extend a further step to point R, which satisfies

$$\mathbf{x}_\mathrm{R} = \mathbf{x}_\mathrm{M} + (\mathbf{x}_\mathrm{M} - \mathbf{x}_\mathrm{W}) \tag{2.15}$$

The search discussed above is called a *reflection* procedure. W is reflected with respect to M, which means a moving point starts at M and has a length and direction of $(\mathbf{x}_\mathrm{M} - \mathbf{x}_\mathrm{W})$. Figure 2.13 illustrates the situation. Now B, G, and R constitute the new simplex.

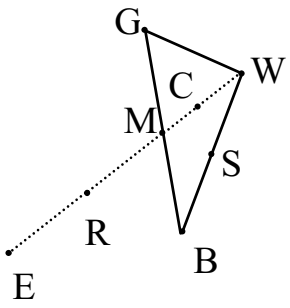

Fig. 2.13 Simplex search

If $f(\mathbf{x}_\mathrm{R}) < f(\mathbf{x}_\mathrm{B})$, then the reflection improves the best points thus far and proves our guess that the direction (W→M) is good. So why not extend it more? This is

[41] Perhaps this direction is not the best or the fastest direction, but it is a workable one because **B** is better than **C**.

the *expansion* procedure, which means that we want to expand the simplex B-G-R to B-G-E. That is, we search point E by

$$\mathbf{x}_{\mathrm{E}} = \mathbf{x}_{\mathrm{M}} + 2\left(\mathbf{x}_{\mathrm{M}} - \mathbf{x}_{\mathrm{W}}\right) \tag{2.16}$$

If $f(\mathbf{x}_{\mathrm{E}}) < f(\mathbf{x}_{\mathrm{B}})$, then we are fully satisfied and make E, B, and G the current simplex. If not, then the expansion procedure is not good enough, but we still have good point R. So R, B, and G constitute the current simplex.

If $f(\mathbf{x}_{\mathrm{B}}) < f(\mathbf{x}_{\mathrm{R}}) < f(\mathbf{x}_{\mathrm{G}})$, which means the reflection is acceptable. So B, R, G constitutes the current simplex.

If $f(\mathbf{x}_{\mathrm{G}}) < f(\mathbf{x}_{\mathrm{R}})$, then the reflection does not find good points. But we still believe that the direction (W→M) is correct. The too-large step is the cause of the failure. So we shorten the step and make a *contraction* procedure, which means we want to contract the simplex B-G-R to B-G-C. That is, we search point C by

$$\mathbf{x}_{\mathrm{C}} = \mathbf{x}_{\mathrm{W}} + 0.5\left(\mathbf{x}_{\mathrm{M}} - \mathbf{x}_{\mathrm{W}}\right) \tag{2.17}$$

If $f(\mathbf{x}_{\mathrm{C}}) < f(\mathbf{x}_{\mathrm{W}})$, then we accept the results and make B, G, and C the current simplex.

If $f(\mathbf{x}_{\mathrm{W}}) < f(\mathbf{x}_{\mathrm{C}})$, this weakens our idea that the direction (W→M) is correct. We reject it and do a *shrink* procedure, which means we want to shrink the simplex based on point B. Then B, M, and S constitute the new simplex.

$$\mathbf{x}_{\mathrm{S}} = \frac{\mathbf{x}_{\mathrm{W}} + \mathbf{x}_{\mathrm{B}}}{2} \tag{2.18}$$

For every new simplex, we need to assign B, G, W according to their objective values. Then the simplex search repeats reflection, expansion, contraction, and shrink again and again in a very efficient and deterministic way. Vertices of the simplex will move toward the optimal point (perhaps the local optimal solution) and the simplex will become smaller and smaller. Stop criteria could be made based on the time of function evaluation, the length of the edge, the improving rate of B, etc.

Dennis and Torczon modified the standard simplex search by a different reflection procedure [4]. Based on that, MATLAB® contains a direct search toolbox [5].

The simplex search is a group-based deterministic search method capable of exploring the objective space very fast, but sometimes becoming trapped in the local optimal point. Thus many EAs use simplex as a local search method after mutation, which can combine the global search ability of EAs and the local search ability of the simplex search. In Chap. 3, we will discuss it again as part of memetic algorithms.

2.4.2 Random Direction-based Search

2.4.2.1 Scatter Search

Although the simplex search is effective, it is more like a technique than an algorithm when facing real-world complex problems. Glover, Laguna, and Marti include the elitism mechanism in the simplex search and suggest an ES-like algorithm: the *scatter search* [6, 7].

The basic idea of the scatter search is the same as that of the simplex search. Given a group of points, the algorithm somehow finds new points, accepts the better ones, and discards the worse ones.

The scatter search has four main steps, illustrated by Fig. 2.14. The *reference set* (RS) contains b "best" solutions, b_1 of which are good with respect to their objective value ($RefSet_1$) and b_2 of which are good with respect to diversity (far away from $RefSet_1$ points) ($RefSet_2$) ($b = b_1 + b_2$).

The initialization procedure of a scatter search randomly generates solutions in such a way that the more individuals are generated in one area, the less opportunity this area will have to generate new ones.[42] In this way, the initial solutions of the scatter search can maintain maximum diversity. After the initialization procedure, the scatter search makes use of the improvement procedure, the simplex search, to make the initial solutions better. After that, $RefSet_1$ is selected from the improvement results according to the objective quality, and $RefSet_2$ is selected from the improvement results according to the smallest distance to $RefSet_1$ of the remaining improved individuals (the larger the better). Then the main loop starts. We use RS to generate subsets. The solutions in the subsets are combined in various ways to get *Psize* new solutions. Then the solutions are improved by some local search approaches (such as simplex search) to become better solutions. Finally, the improved solutions will replace some solutions of RS if they are good with respect to objective quality or diversity. The main loop is illustrated in Fig. 2.14.

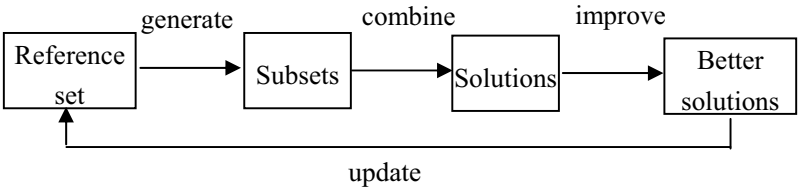

Fig. 2.14 Scatter search

There are four types of subsets to be generated in a scatter search:

1. All two-element subsets containing all pairwise combinations of the b reference set solutions.

[42] Readers are encouraged to reread the second method for the initialization of SGA.

2. Three-element subsets derived from two-element subsets by adding the best solution not in this subset (measured by objective value).
3. Four-element subsets derived from three-element subsets by adding the best solution not in this subset (measured by objective value).
4. Subsets containing the best i elements (measured by objective value), for $i = 5$ to b.

In this way, subsets regard the objective value as the most important factor but still contain the diversity factor.

There are many types of combinations for generating new solutions from subsets. Let us give an example for a two-element subset: $\mathbf{x}_1$ and $\mathbf{x}_2$. We can first find a vector starting at $\mathbf{x}_1$ and pointing to $\mathbf{x}_2$ as $\mathbf{d} = \frac{\mathbf{x}_2 - \mathbf{x}_1}{2}$(the length of the vector is half the distance between $\mathbf{x}_1$ and $\mathbf{x}_2$). Glover, Laguna, and Marti suggested three types of re-combination: (1) $\mathbf{x}_{\text{new}} = \mathbf{x}_1 - r\mathbf{d}$, (2) $\mathbf{x}_{\text{new}} = \mathbf{x}_1 + r\mathbf{d}$, and (3) $\mathbf{x}_{\text{new}} = \mathbf{x}_2 + r\mathbf{d}$,[43] where $r \sim U(0,1)$. Every subset can generate several new solutions according to the composition of the subset.[44]

- Both $\mathbf{x}_1$ and $\mathbf{x}_2$ belong to $RefSet_1$, which means that they are all good solutions. Four new solutions are generated by types 1 and 3 once and type 2 twice.
- Only one of $\mathbf{x}_1$ and $\mathbf{x}_2$ belongs to $RefSet_1$. Three new solutions are generated by types 1, 2, and 3 once.
- Neither $\mathbf{x}_1$ nor $\mathbf{x}_2$ belongs to $RefSet_1$, which means that they are all uncrowded solutions. Two new solutions are generated by type 2 once and type 1 or 3 once.

As has been stated, a simplex search is used by Glover, Laguna, and Marti to improve the new solutions.

If an improved solution's objective value is better than that of the worst one in $RefSet_1$, it will replace the worst one without replacement (delete it from the set of improved solutions). If an improved solution's distance to the closest RS solutions is larger than that of most crowded solutions in $RefSet_2$, it will replace the most crowded one without replacement.

If RS does not change in the updating procedure and the stop criterion has not been satisfied, then the initialization procedure will be started to construct a new $RefSet_2$.

Glover, Laguna, and Marti suggested that $Psize = \max(100, 5 * b)$. We can consider the scatter search as a special kind of $(b + Psize)$-ES. But in the updating (replacement) phase, the objective value is not the only criterion.

2.4.2.2 Differential Evolution

Differential evolution (DE) is also a kind of direction-based search, suggested by Storn and Price in 1997 [8, 9]. DE also maintains a population with Np individ-

[43] Readers may find out the geometric explanation for these formulae using Fig. 2.12.

[44] What is the intuitive idea of Glover, Laguna, and Marti for generating new solutions? This is the origin of the name "scatter."

uals and has mutation, crossover operators, and a selection process. So DE could be regarded as a real parameter coded version of GA.[45] We suppose there are n dimensions in the decision space so that individual i of DE can be expressed as

$$\mathbf{x}_i = (x_{i,1}, x_{i,2}, \cdots, x_{i,n}) \tag{2.19}$$

where $x_{i,j} \in R,\ i = 1,2,\cdots,Np,\ j = 1,2,\cdots,n$.

The main difference between DE, SGA, and ES lies in the mutation operator, where the direction is used to make the perturbation from the current individual.

Let us start with individual i. Unlike the random step size of mutation along each dimension of ES, DE uses the directional information from the current population. The standard mutation operator of DE needs three randomly selected different individuals from the current population ($\mathbf{r}_1 \neq \mathbf{r}_2 \neq \mathbf{r}_3 \neq \mathbf{x}_i$) for each individual to form a simplex like triangle. For $\mathbf{x}_i$, its mutants $\mathbf{v}_i$ is[46]

$$\mathbf{v}_i = \mathbf{r}_1 + F(\mathbf{r}_2 - \mathbf{r}_3) \tag{2.20}$$

where F is a positive real number (seldom larger than one) that controls the strength of the direction. Differential $(\mathbf{r}_2 - \mathbf{r}_3)$ forms a direction that is the origin of DE and the reason it is a direction-based search. Equation 2.20 means the mutant of $\mathbf{x}_i$ starts at $\mathbf{r}_1$ and has direction and length of $F(\mathbf{r}_2 - \mathbf{r}_3)$.

After mutation, DE utilizes a uniform crossover operator[47] to combine the information of the parents $\mathbf{x}_i$ and $\mathbf{v}_i$ into the offspring $\mathbf{u}_i$. We need to have at least one variable of $\mathbf{v}_i$, so a random integer number j_{rand} in the range $[1,n]$ should be generated before crossover. The offspring $\mathbf{u}_i$ is generated by Eq. 2.21:

$$u_{i,j} = \begin{cases} v_{i,j} & rand \leq Cr \text{ or } j = j_{\text{rand}} \\ x_{i,j} & \text{otherwise} \end{cases} \tag{2.21}$$

where $Cr \in [0,1]$ is a user-defined parameter controlling the effect of crossover, $rand \sim U(0,1)$. Equation 2.21 means that the offspring $\mathbf{u}_i$ has the possibility of Cr to select variables from mutant $\mathbf{v}_i$ and ensures at least the j_{rand}th variable will be picked from $\mathbf{v}_i$.

Then $\mathbf{u}_i$, called a *trial vector*, competes with $\mathbf{x}_i$, called a *target vector*, for survival in the next generation in a steady way,[48] which means

$$\mathbf{x}_i(t+1) = \begin{cases} \mathbf{u}_i(t) & f(\mathbf{u}_i(t)) \leq f(\mathbf{x}_i(t)) \\ \mathbf{x}_i(t) & \text{otherwise} \end{cases} \tag{2.22}$$

Mutation, crossover, and selection will be carried out for Np individuals in one generation. The initialization and the stop criteria might be the same with SGA.

---

[45] So we generally do not differentiate "individuals" between "points," "solutions," and "vectors."

[46] What is the geometric explanation of Eq. 2.20?

[47] Uniform crossover will be discussed in detail in Chap. 3.

[48] DE treats every individual with mutation, crossover, and a replacement procedure, which is a steady state EA, unlike the generational approach in SGA. These two terms will be discussed in Chap. 3.

Even though standard DE utilizes a random direction to mutate individuals, the direction may point to the promising area during the evolving process. Apart from that, more effectively assigning $\mathbf{r}_1 \neq \mathbf{r}_2 \neq \mathbf{r}_3 \neq \mathbf{x}_i$ may promote evolution toward the designated area, which will be discussed in later chapters. DE is a kind of simple but powerful EA that has been attracting increasing research interests. Figure 2.15 illustrates the number of papers indexed by the SCI on DE.[49] Mathematica® has already added DE to its numerical optimizer package.

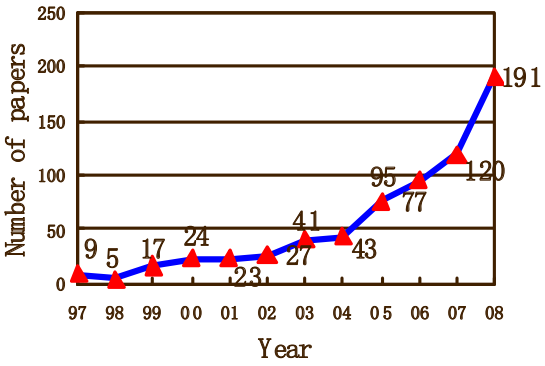

Fig. 2.15 Number of papers indexed by SCI on DE

All the algorithms introduced in this section may be regarded as EAs because they all maintain a group of individuals and have some kind of selection scheme and variation operators.

2.5 Summary

After providing the necessary mathematical and programming backgrounds in Sect. 2.1, we introduced SGA in detail using a not-so-easy problem to demonstrate the strength of EAs, and we explained the implementation key points for programming EAs. After that, five algorithms, including ES, EP, simplex search, scatter search, and DE, were introduced. We do not want readers to remember every step of these algorithms. But the intuitive ideas of these algorithms are of utmost importance because they might be the effective search techniques of your algorithms.

Observing Fig. 2.9 carefully, you might notice that SGA can improve the quality of the best fitness value very fast, and there is a saturation like character thereafter. It is almost always correct with respect to every EA that about 80% of its initial improvements from randomly generated initial individuals are done in about 20%

[49] TS = (“differential evolution”). The SCI index “TS” is for the search topic in the title, the keywords, and the abstract.

of its evolving process and about 20% of its final improvements are made in about 80% of its evolving process.[50] In Chap. 3, we will introduce many ways to improve the final search efficiency.

Good metaheuristics, such as GA, ES, EP, scatter search, DE, etc., need at least three mechanisms to accomplish the search requirement satisfactorily.

- Global search mechanism. Good algorithms need a global search mechanism to find the domain of attraction of global optimum.
- Convergence mechanism. Good algorithms need a convergence mechanism to promote the evolution of individuals toward the best ones.
- Up-hill mechanism for minimum optimization problems.[51] Good algorithms need an up-hill mechanism to accept no-so-good individuals so that the population can escape the domain of attraction of the local optimum where the current best individual resides.

For SGA, its global search mechanism is randomly generated initial individuals and crossover and mutation operators; its convergence mechanism is RWS; its up-hill mechanism is also RWS and maintaining a group of individuals in the population. Readers are strongly suggested to find out these mechanisms for every EA we introduce hereafter.

Before concluding this chapter, we need to mention two viewpoints. The prerequisite for designing an effective algorithm for real-world problems is to grasp the innovative elements of standard EAs. The only way to grasp, instead of memorizing them, is to read actively. Apart from that, if an powerful algorithm takes advantage of various effective search techniques introduced in this chapter and the later ones, it is sometimes hard to call it a GA, ES, EP, DE, or other specific algorithm.

You need to grasp the programming details of SGA and understand the intuitive ideas behind other introduced EAs.

Suggestions for Further Reading

This chapter deals with the basics of EAs, so the suggested reading list only includes chapters of the textbooks.

For SGA, ES, and EP, we encourage interested readers to read Chaps. 2–5 of Eiben and Smith's textbook [10] and Chaps. 8–10 of Bäck *et al.*'s textbook [11]. Haupt and Haupt also give good introductions for binary GA in Chap. 2 of their second version of the textbook published in 2004 [12].

For DE, we recommend Chap. 2 of Price *et al.*'s textbook [9] and Chap. 1 of Feoktistov's textbook [13].

We introduced several EAs in this chapter. Readers interested in their taxonomy are encouraged to read paper by Calégari *et al.* [14].

---

[50] This is an example of the $80/20$ rule.

[51] Or down-hill mechanism for maximum optimization problem.

Exercises and Potential Research Projects

2.1. Implement an SGA from scratch in any programming environment to repeat the solution process of the problem illustrated by Eq. 2.8. We insist on encouraging readers to implement their SGA without the help of any source codes to promote an in-depth understanding of EAs. MATLAB® is suggested for implementing the SGA. Comparisons between different parameter settings are encouraged.

2.2. In RWS, is the selection with replacement or without replacement?

2.3. Why do we need to define the crossover rate p_c and the mutation rate p_m? What will happen if $p_c = p_c = 1$ in an SGA?

2.4. Summarize the difference between SGA, ES, and EP in a table.

2.5. What are the meanings of $(1+1)$-ES, $(1+\lambda)$-ES, and $(1, \lambda)$-ES?

2.6. Find out the global search mechanism, the convergence mechanism, and the up-hill mechanism of ES.

2.7. Find out the global search mechanism, the convergence mechanism, and the up-hill mechanism of scatter search.

2.8. Why do we use the centroid of B and G as the reflection center in simplex search? Is there any other method? For example, if we think B should have twice as much influence on the reflection center as G, then what is the expression for M?

2.9. Read [4] and summarize its simplex search on a single sheet of paper.

References

1. Papoulis A, Pillai S (2002) Probability, random variables and stochastic processes, 4th edn. McGraw Hill Higher Education
2. Ross S (2009) A first course in probability, 8th edn. Prentice Hall
3. Nelder J, Mead R (1965) A simplex method for function minimization. Comput J 7(4):308–313
4. Dennis JE, Jr, Torczon V (1991) Direct search methods on parallel machines. SIAM J Optim 1:448–474
5. MathWorks T (2009) Genetic algorithm and direct search toolbox 2 user's guide. Tech. rep., The MathWorks, Natick, MA
6. Glover F, Laguna M, Marti R (2003) Advances in evolutionary computation: theory and applications, chap. Scatter search, 519–537. Springer, Berlin Heidelberg New York
7. Laguna M, Marti R (2003) Scatter search: methodology and implementations in C. Springer, Berlin Heidelberg New York
8. Storn R, Price K (1997) Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. J Glob Optim 11(4):341–359
9. Price KV, Storn RM, Lampinen JA (2005) Differential evolution: a practical approach to global optimization. Springer, Berlin Heidelberg New York

10. Eiben AE, Smith JE (2003) Introduction to evolutionary computing. Springer, Berlin Heidelberg New York
11. Bäck T, Fogel D, Michalewicz Z (2000) Evolutionary computation 1: basic algorithms and operators. Taylor & Francis, London, UK
12. Haupt RL, Haupt SE (2004) Practical genetic algorithms, 2nd edn. Wiley, New York
13. Feoktistov V (2006) Differential evolution: in search of solutions. Springer, Berlin Heidelberg New York
14. Calégari P, Coray G, Hertz A *et al* (1999) A taxonomy of evolutionary algorithms in combinatorial optimization. J Heurist 5(2):145–158

http://www.springer.com/978-1-84996-128-8

Introduction to Evolutionary Algorithms

Yu, X.; Gen, M.

2010, XVI, 422 p. 168 illus., Hardcover
ISBN: 978-1-84996-128-8