3 Evolutionary Computing

Evolutionary Computing is the collective name for a range of problem-solving techniques based on principles of biological evolution, such as natural selection and genetic inheritance. These techniques are being increasingly widely applied to a variety of problems, ranging from practical applications in industry and commerce to leading-edge scientific research.

In computer science, evolutionary computation is a subfield of artificial intelligence (more particularly computational intelligence) that involves combinatorial optimization problems. Evolutionary computation uses iterative progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. Such processes are often inspired by biological mechanisms of evolution. As evolution can produce highly optimised processes and networks, it has many applications in computer science.

Problem solution using evolutionary algorithms is shown in Figure 35 (Pohlheim 2006).

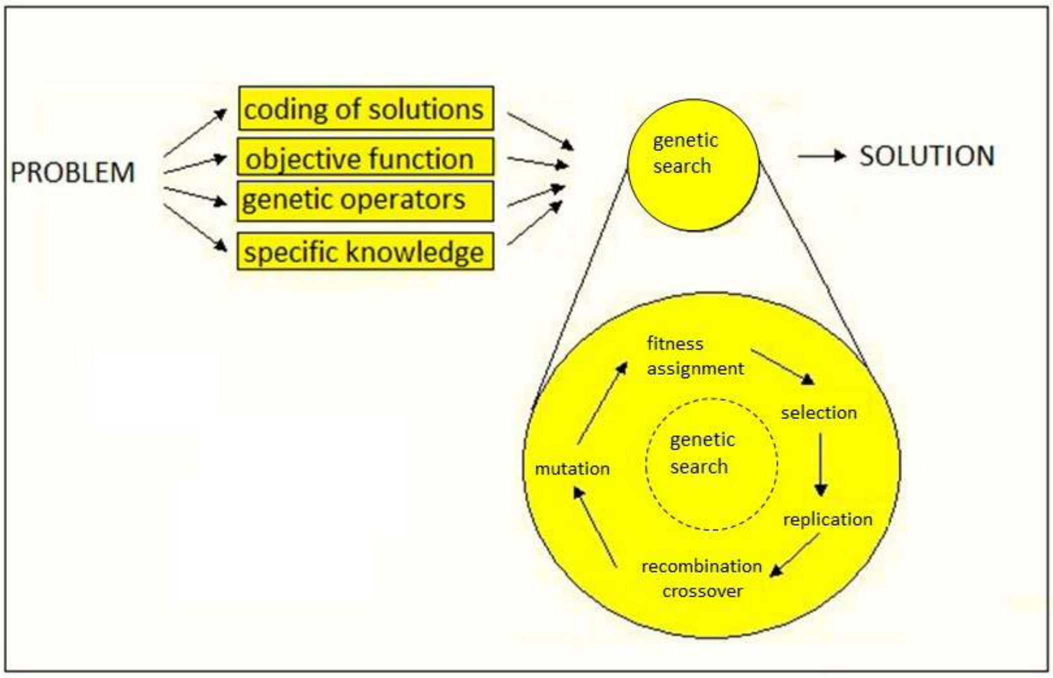


Figure 35: Problem solution using evolutionary algorithms (adapted from http://jpmc.sourceforge.net)

3.1 Evolutionary algorithms

Different main schools of evolutionary algorithms have been evolved during the last 50 years: genetic algorithms, mainly developed in the USA by J.H. Holland (Holland 1975), evolutionary strategies, developed in Germany by I. Rechenberg (Rechenberg 1973) and H.-P. Schwefel (Schwefel 1981), and evolutionary programming (Fogel, Owens, and Walsh 1966). Each of these constitutes represents a different approach, however, they are inspired by the same principles of natural evolution. A good introductory survey can be found in (Fogel 1994).

Evolutionary algorithms are stochastic search methods that mimic the metaphor of natural biological evolution. Evolutionary algorithms operate on a population of potential solutions applying the principle of survival of the fittest to produce better and better approximations to a solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation. Evolutionary algorithms model natural processes, such as selection, recombination, mutation, migration, locality and neighbourhood. Figure 36 shows the structure of a simple evolutionary algorithm. Evolutionary algorithms work on populations of individuals instead of single solutions. In this way the search is performed in a parallel manner.

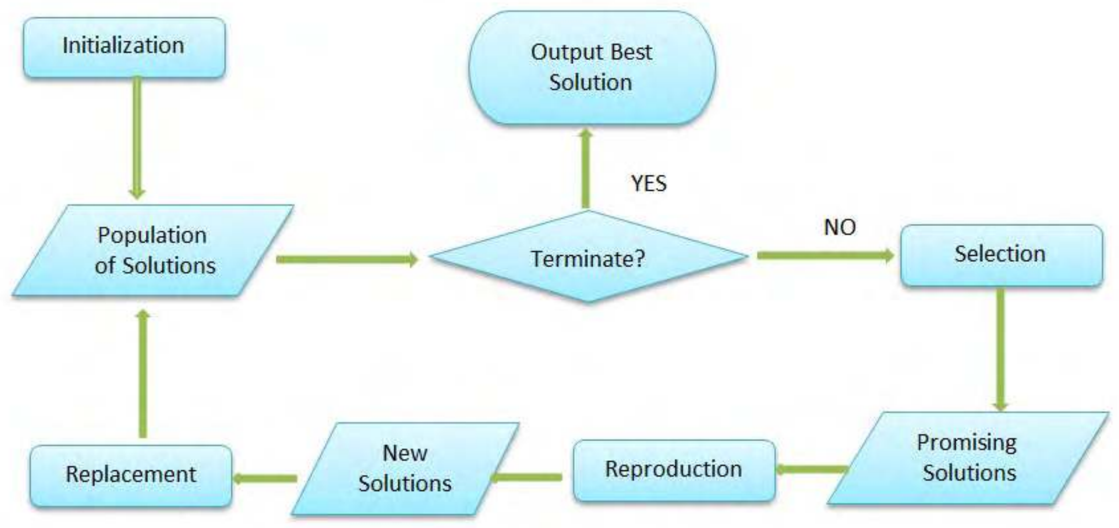


Figure 36: Structure of a single population evolutionary algorithm (adapted from www.sciencedirect.com)

At the beginning of the computation a number of individuals (the population) are randomly initialized. The objective function is then evaluated for these individuals. The first/initial generation is produced. If the optimization criteria are not met, the creation of a new generation starts. Individuals are selected according to their fitness for the production of offspring. Parents are recombined to produce offspring. All offspring will be mutated with a certain probability. The fitness of the offspring is then computed. The offspring are inserted into the population replacing the parents, producing a new generation. This cycle is performed until the optimization criteria are reached.

From the above discussion, it can be seen that evolutionary algorithms differ substantially from more traditional search and optimization methods. The most significant differences are (Pohlheim 2006):

- Evolutionary algorithms search a population of points in parallel, not just a single point.
- Evolutionary algorithms do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the directions of search.
- Evolutionary algorithms use probabilistic transition rules, not deterministic ones.
- Evolutionary algorithms are generally more straightforward to apply, because no restrictions for the definition of the objective function exist.
- Evolutionary algorithms can provide a number of potential solutions to a given problem. The final choice is left to the user. (Thus, in cases where the particular problem does not have one individual solution, for example a family of pareto-optimal solutions, as in the case of multi-objective optimization and scheduling problems, then the evolutionary algorithm is potentially useful for identifying these alternative solutions simultaneously.)

3.1.1 Selection

In selection the offspring producing individuals are chosen. The first step is fitness assignment. Each individual in the selection pool receives a reproduction probability depending on the own objective value and the objective value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards. Throughout the chapter some terms are used for comparing the different selection schemes. The definitions of these terms follow (Baker 1987).

selective pressure: probability of the best individual being selected compared to the average probability of selection of all individuals

bias: absolute difference between an individual's normalized fitness and its expected probability of reproduction

spread: range of possible values for the number of offspring of an individual

loss of diversity: proportion of individuals of a population that is not selected during the selection phase *selection intensity:* expected average fitness value of the population after applying a selection method to the normalized Gaussian distribution

selection variance: expected variance of the fitness distribution of the population after applying a selection method to the normalized Gaussian distribution

Roulette wheel selection

The simplest selection scheme is roulette-wheel selection, also called stochastic sampling with replacement (Baker 1987). This is a stochastic algorithm and involves the following technique: The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness.

Example

Table 2 shows the selection probability for 11 individuals. Individual 1 is the most fit individual and occupies the largest interval, whereas individual 10 as the second least fit individual has the smallest interval on the line (see Figure 37). Individual 11, the least fit interval, has a fitness value of 0 and get no chance for reproduction.

Number of individual	1	2	3	4	5	6	7	8	9	10	11
fitness value	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	0.0
selection probability	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02	0.0

Table 2: Selection probability and fitness value

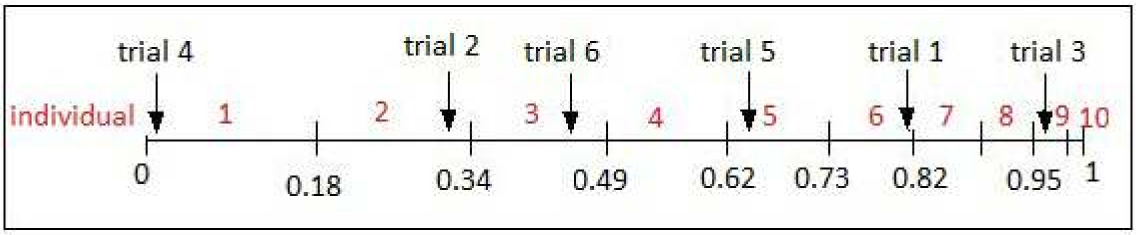


Figure 37: Roulette-wheel selection (adapted from http://www.geatbx.com/)

For selecting, the mating population the appropriate number of uniformly distributed random numbers (uniform distributed between 0.0 and 1.0) is independently generated.

Sample of 6 random numbers: 0.81, 0.32, 0.96, 0.01, 0.65, 0.42.

Figure 37 shows the selection process of the individuals for the example in Table 2 together with the above sample trials.

After selection the mating population consists of the individuals: 1, 2, 3, 5, 6, 9.

The roulette-wheel selection algorithm provides a zero bias but does not guarantee minimum spread.

Stochastic universal sampling

Stochastic universal sampling (Baker 1987) provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection. Here equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider *NPointer* the number of individuals to be selected, then the distance between the pointers are 1/*NPointer* and the position of the first pointer is given by a randomly generated number in the range [0, 1/*NPointer*].

Example (cont.)

For 6 individuals to be selected, the distance between pointers is 1/6 = 0.167. Figure 38 shows the selection for the above example.

Sample of 1 random number in the range [0, 0.167]: 0.1.

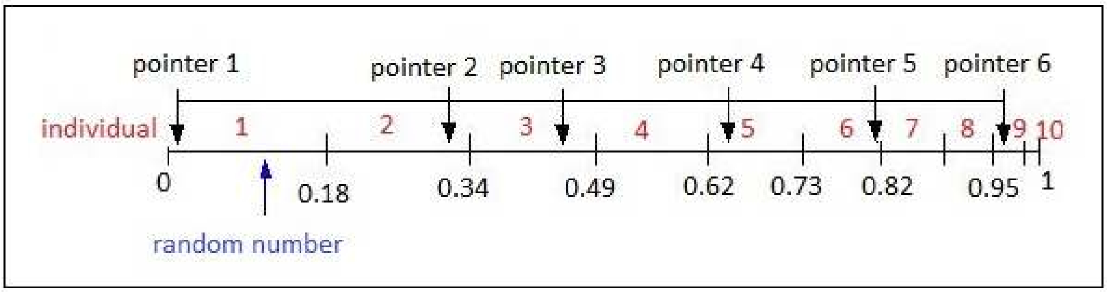


Figure 38: Stochastic universal sampling (adapted from http://www.geatbx.com/)

After selection the mating population consists of the individuals: 1, 2, 3, 4, 6, 8.

Stochastic universal sampling ensures a selection of offspring which is closer to what is deserved then roulette wheel selection.

Local selection

In local selection every individual resides inside a constrained environment called the local neighbourhood. (In the other selection methods the whole population or subpopulation is the selection pool or neighbourhood.) Individuals interact only with individuals inside this region. The neighbourhood is defined by the structure in which the population is distributed. The neighbourhood can be seen as the group of potential mating partners.

The first step is the selection of the first half of the mating population uniform at random (or using one of the other mentioned selection algorithms, for example, stochastic universal sampling or tournament selection). Now a local neighbourhood is defined for every selected individual. Inside this neighbourhood the mating partner is selected (best, fitness proportional, or uniform at random).

The structure of the neighbourhood can be:

- *linear*
  full ring, half ring (see Figure 39)

- *two-dimensional*
  full cross, half cross (see Figure 40, left)
  full star, half star (see Figure 40, right)

- *three-dimensional* and more complex with any combination of the above structures.

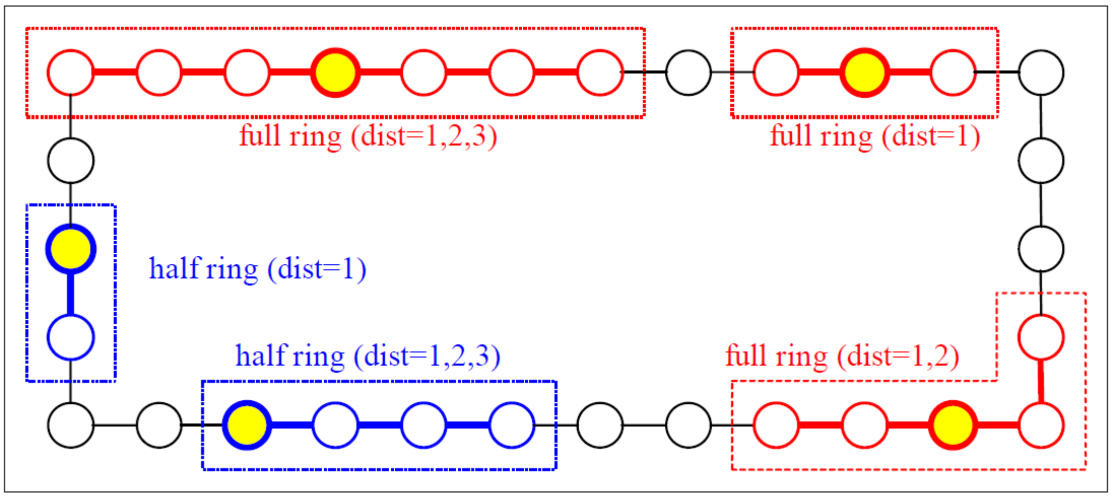


Figure 39: Linear neighbourhood: full and half ring (adapted from http://www.geatbx.com/)

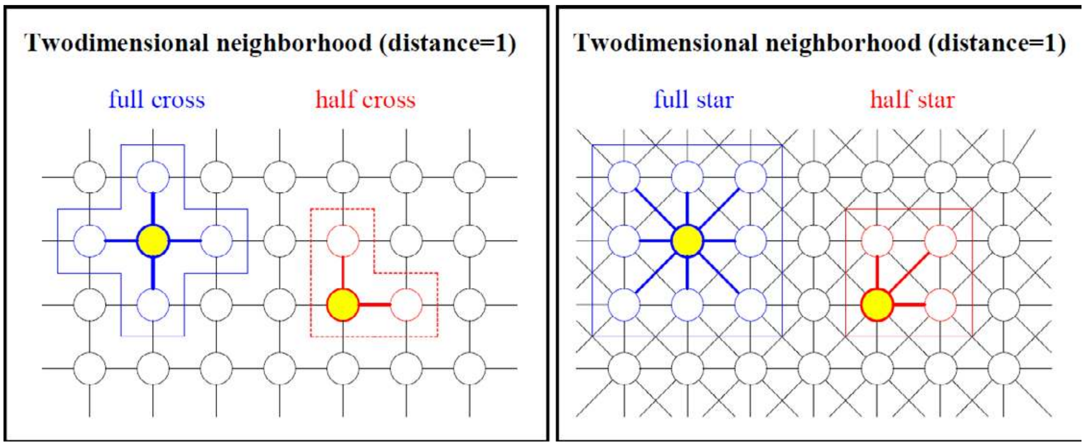

Twodimensional neighborhood (distance=1)
full cross
half cross
Twodimensional neighborhood (distance=1)
full star
half star

Figure 40: Two-dimensional neighbourhood; left: full and half cross, right: full and half star (adapted from http://www.geatbx.com/)

The distance between possible neighbours together with the structure determines the size of the neighbourhood. Between individuals of a population an "isolation by distance" exists. The smaller the neighbourhood, the bigger the isolation distances. However, because of overlapping neighbourhoods, propagation of new variants takes place. This assures the exchange of information between all individuals. The size of the neighbourhood determines the speed of propagation of information between the individuals of a population, thus deciding between rapid propagation and maintenance of a high diversity/variability in the population. A higher variability is often desired, thus preventing problems such as premature convergence to a local minimum. Local selection in a small neighbourhood performed better than local selection in a bigger neighbourhood. Nevertheless, the interconnection of the whole population must still be provided. Two-dimensional neighbourhood with structure half star using a distance of 1 is recommended for local selection. However, if the population is bigger (>100 individuals) a greater distance and/or another two-dimensional neighbourhood should be used (Pohlheim 2006).

Tournament selection

In tournament selection (Goldberg and Deb 1991) a number *Tour* of individuals is chosen randomly from the population and the best individual from this group is selected as parent. This process is repeated as often as individuals must be chosen. These selected parents produce uniform at random offspring. The parameter for tournament selection is the tournament size *Tour*. *Tour* takes values ranging from 2 to *Nind* (number of individuals in population). Table 3 and Figure 41 show the relation between the tournament size and selection intensity (Blickle 1995).

tournament size	1	2	3	5	10	30
selection intensity	0	0.56	0.85	1.15	1.53	2.04

Table 3: Relation between tournament size and selection intensity

In (Blickle 1995) an analysis of tournament selection can be found.

Selection intensity

$$SelInt_{Turnier}(Tour) \approx \sqrt{2 \cdot \left(\ln(Tour) - \ln\left(\sqrt{4.14 \cdot \ln(Tour)}\right)\right)}$$

Loss of diversity

$$LossDiv_{Turnier}(Tour) = Tour^{\frac{-1}{Tour-1}} - Tour^{\frac{-Tour}{Tour-1}}$$

(About 50% of the population are lost at tournament size *Tour*=5).

Selection variance

$$SelVar_{Turnier}(Tour) \approx \frac{0.918}{\ln(1.186 + 1.328 \cdot Tour)}$$

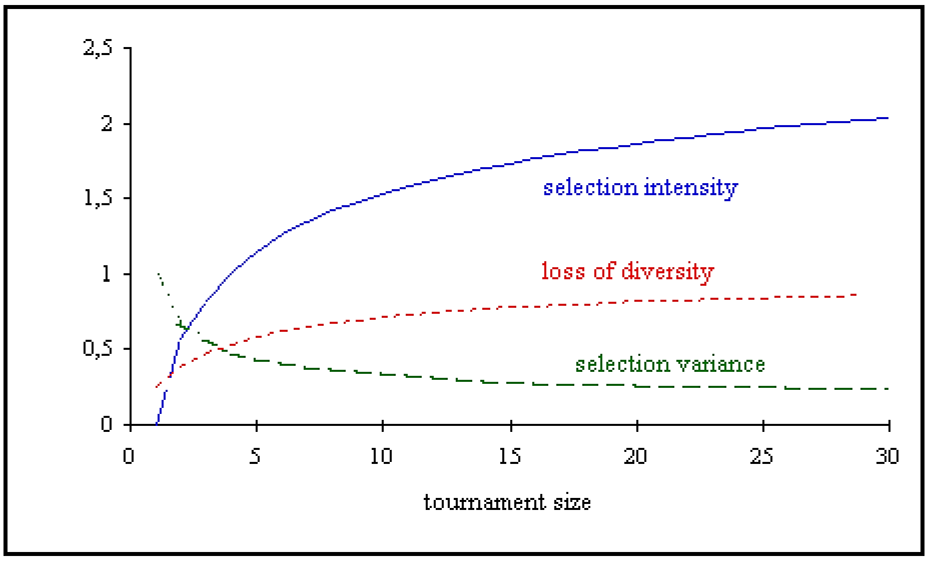


Figure 41: Properties of tournament selection (adapted from http://www.geatbx.com/)

3.1.2 Recombination

Recombination produces new individuals in combining the information contained in two or more parents (parents – mating population). This is done by combining the variable values of the parents. Depending on the representation of the variables different methods must be used.

The methods for binary valued variables constitute special cases of the discrete recombination. These methods can be applied to integer valued and real valued variables as well.

Discrete recombination – All representations

Discrete recombination (Mühlenbein and Schlierkamp-Voosen 1993) performs an exchange of variable values between individuals. For each position, the parent who contributes its variable to the offspring is chosen randomly with equal probability.

$$Var_i^0 = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i) \quad i \in (1, 2, \cdots, Nvar),$$
$$a_i \in \{0,1\} \text{ uniform at random, } a_i \text{ for each } i \text{ new defined}$$

Discrete recombination generates corners of the hypercube defined by the parents. Figure 42 shows the geometric effect of discrete recombination.

Example

Consider the following two individuals with 3 variables each (3 dimensions), which will also be used to illustrate the other types of recombination for real valued variables:

individual 1 12 25 5
individual 2 123 4 34

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability:

sample 1 2 2 1
sample 2 1 2 1

After recombination the new individuals are created:

offspring 1 123 4 5
offspring 2 12 4 5

Discrete recombination can be used with any kind of variables (binary, integer, real or symbols).

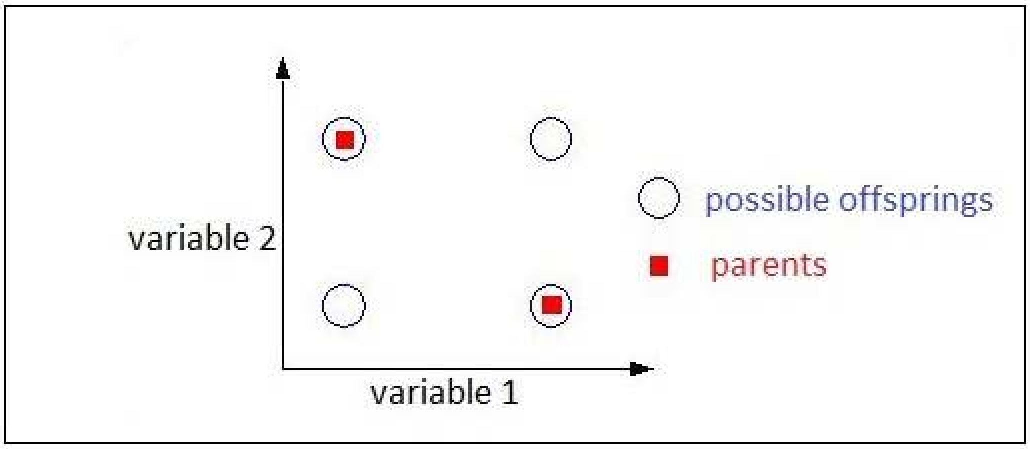


Figure 42: Possible positions of the offspring after discrete recombination (adapted from http://www.geatbx.com/)

Intermediate recombination – Real valued recombination

Intermediate recombination (Mühlenbein and Schlierkamp-Voosen 1993) is a method only applicable to real variables (and not binary variables). Here, the variable values of the offspring are chosen somewhere around and between the variable values of the parents.

Offspring are produced according to the rule:

$$Var_i^0 = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i) \quad i \in (1, 2, \cdots, Nvar),$$
$$a_i \in [-d, 1+d] \text{ uniform at random, } d = 0.25, \quad a_i \text{ for each } i \text{ new}$$

where *a* is a scaling factor chosen uniformly at random over an interval [-*d*, 1+*d*] for **each** variable anew.

The value of the parameter *d* defines the size of the area for possible offspring. A value of *d* = 0 defines the area for offspring the same size as the area spanned by the parents. This method is called (standard) intermediate recombination. Because most variables of the offspring are not generated on the border of the possible area, the area for the variables shrinks over the generations. This shrinkage occurs just by using (standard) intermediate recombination. This effect can be prevented by using a larger value for *d*. A value of *d* = 0.25 ensures (statistically), that the variable area of the offspring is the same as the variable area spanned by the variables of the parents. See Figure 43 for a picture of the area of the variable range of the offspring defined by the variables of the parents.

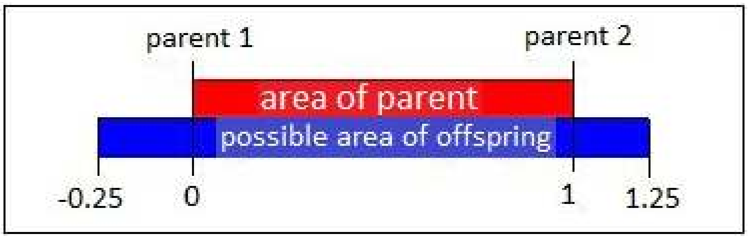


Figure 43: Area for variable value of offspring compared to parents in intermediate recombination (adapted from http://www.geatbx.com/)

Example

Consider the following two individuals with 3 variables each:

individual1	12	25	5
individual 2	123	4	34

The chosen *a* for this example are:

sample 1	0.5	1.1	-0.1
sample 2	0.1	0.8	0.5

The new individuals are calculated as:

offspring 1	67.5	1.9	2.1
offspring 2	23.1	8.2	19.5

Intermediate recombination is capable of producing any point within a hypercube slightly larger than that defined by the parents. Figure 44 shows the possible area of offspring after intermediate recombination.

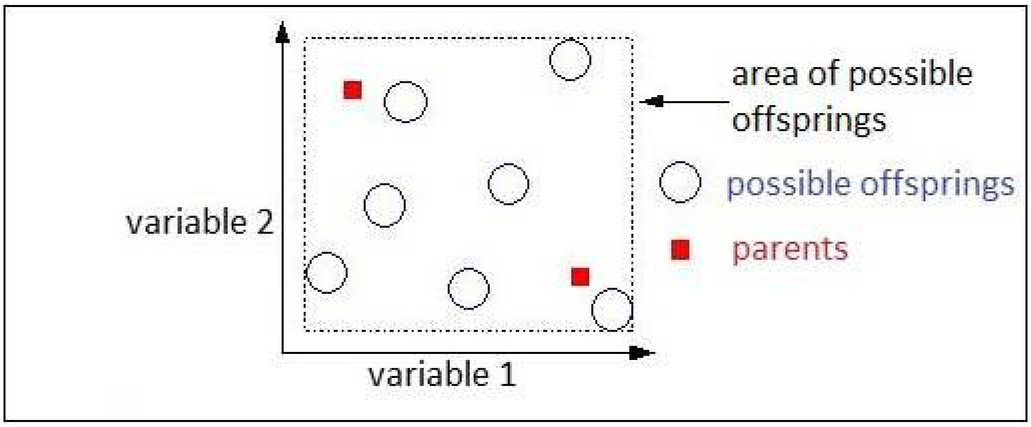


Figure 44: Possible area of the offspring after intermediate recombination (adapted from http://www.geatbx.com/)

Line recombination – Real valued recombination

Line recombination (Mühlenbein and Schlierkamp-Voosen 1993) is similar to intermediate recombination, except that only one value of *a* for all variables is used. The same *a* is used for **all** variables:

$$Var_i^0 = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i) \quad i \in (1, 2, \cdots, Nvar),$$
$$a_i \in [-d, 1 + d] \text{ uniform at random, } d = 0.25, \quad a_i \text{ for all } i \text{ identical}$$

For the value of *d* the statements given for intermediate recombination are applicable.

Example

Consider the following two individuals with 3 variables each:

individual 1	12	25	5
individual 2	123	4	34

The chosen *a* for this example are:

sample 1	0.5
sample 2	0.1

The new individuals are calculated as:

offspring 1	67.5	14.5	19.5
offspring 2	23.1	22.9	7.9

Line recombination can generate any point on the line defined by the parents. Figure 45 shows the possible positions of the offspring after line recombination.

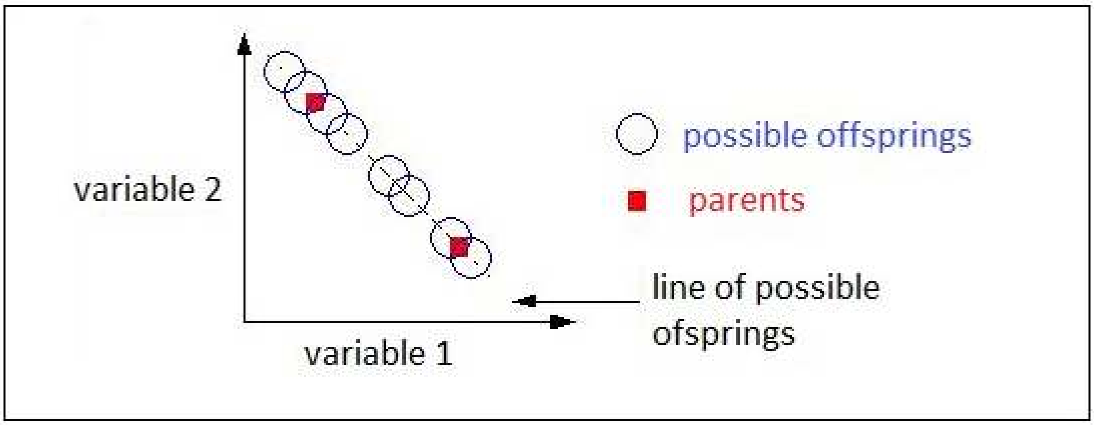


Figure 45: Possible positions of the offspring after line recombination (adapted from http://www.geatbx.com/)

3.1.3 Binary valued recombination (crossover)

Recombination produces new individuals in combining the information contained in two or more parents (parents – mating population). This is done by combining the variable values of the parents. Depending on the representation of the variables different methods must be used.

During the recombination of binary variables only parts of the individuals are exchanged between the individuals. Depending on the number of parts, the individuals are divided before the exchange of variables (the number of cross points). The number of cross points distinguishes the methods.

Single-point / double point / multi-point crossover

In single-point crossover one crossover position *k*_[1,2,…,*Nvar*-1], *Nvar*: number of variables of an individual, is selected uniformly at random and the variables exchanged between the individuals about this point, then two new offspring are produced. Figure 46 illustrates this process.

Example

Consider the following two individuals with 11 binary variables each:

individual 1 0 1 1 1 0 0 1 1 0 1 0
individual 2 1 0 1 0 1 1 0 0 1 0 1

The chosen crossover positions are:

crossover position: 5

After crossover the new individuals are created:

offspring 1 0 1 1 1 0| 1 0 0 1 0 1
offspring 2 1 0 1 0 1| 0 1 1 0 1 0

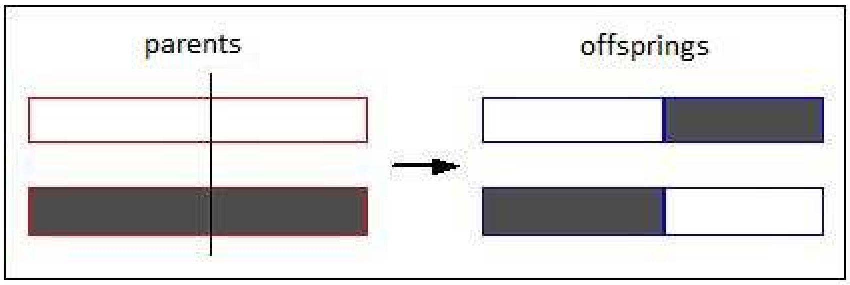


Figure 46: Single-point crossover (adapted from http://www.geatbx.com/)

In double-point crossover two crossover positions are selected uniformly at random and the variables exchanged between the individuals between these points, then two new offsprings are produced. Single-point and double-point crossover are special cases of the general method multi-point crossover.

For multi-point crossover, *m* crossover positions k_i_[1,2,…,*Nvar*-1], *i*=1:*m*, *Nvar*: number of variables of an individual are chosen at random with no duplicates and sorted into ascending order. Then, the variables between successive crossover points are exchanged between two parents to produce two new offsprings. The section between the first variable and the first crossover point is not exchanged between individuals. Figure 47 illustrates this process.

Example

Consider the following two individuals with 11 binary variables each:

individual 1	0 1 1 1 0 0 1 1 0 1 0
individual 2	1 0 1 0 1 1 0 0 1 0 1

The chosen crossover positions are:

cross pos. (m=3):	2	6	10

After crossover the new individuals are created:

offspring 1	0 1| 1 0 1 1| 0 1 1 1| 1
offspring 2	1 0| 1 1 0 0| 0 0 1 0| 0

Line recombination can generate any point on the line defined by the parents. Figure 47 shows the possible positions of the offspring after line recombination.

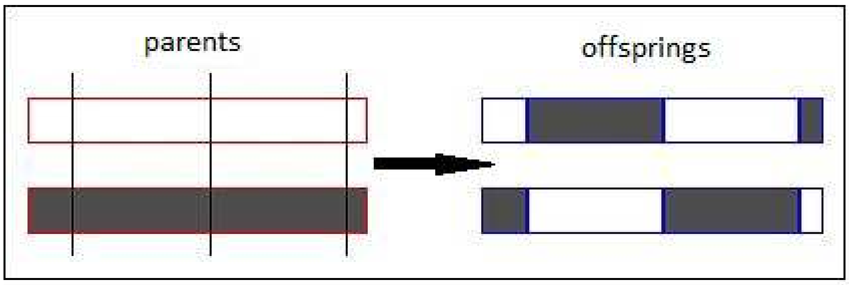


Figure 47: Multi-point crossover (adapted from http://www.geatbx.com/)

The idea behind multi-point, and indeed many of the variations on the crossover operator, is that parts of the chromosome representation that contribute most to the performance of a particular individual may not necessarily be contained in adjacent substrings (Booker 1987). Further, the disruptive nature of multi-point crossover appears to encourage the exploration of the search space, rather than favouring the convergence to highly fit individuals early in the search, thus making the search more robust (Spears and De Jong 1991).

Uniform crossover

Single and multi-point crossover defines cross points as places between loci where an individual can be split. Uniform crossover (Syswerda 1989) generalizes this scheme to make every locus a potential crossover point. A crossover mask, the same length as the individual structure is created at random and the parity of the bits in the mask indicate which parent will supply the offspring with which bits. This method is identical to discrete recombination.

Example

Consider the following two individuals with 11 binary variables each:

individual 1	0 1 1 1 0 0 1 1 0 1 0
individual 2	1 0 1 0 1 1 0 0 1 0 1

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability. Here, the offspring 1 is produced by taking the bit from parent 1 if the corresponding mask bit is 1 or the bit from parent 2 if the corresponding mask bit is 0. Offspring 2 is created using the inverse of the mask, usually.

sample 1	0 1 1 0 0 0 1 1 0 1 0
sample 2	1 0 0 1 1 1 0 0 1 0 1

After crossover the new individuals are created:

offspring 1	1 1 1 0 1 1 1 1 1 1 1
offspring 2	0 0 1 1 0 0 0 0 0 0 0

Uniform crossover, like multi-point crossover, has been claimed to reduce the bias associated with the length of the binary representation used and the particular coding for a given parameter set. This helps to overcome the bias in single-point crossover towards short substrings without requiring precise understanding of the significance of the individual bits in the individuals' representation. (Spears and De Jong 1991) demonstrated how uniform crossover may be parametrized by applying a probability to the swapping of bits. This extra parameter can be used to control the amount of disruption during recombination without introducing a bias towards the length of the representation used.

3.1.3 Mutation

By mutation individuals are randomly altered. These variations (mutation steps) are mostly small. They will be applied to the variables of the individuals with a low probability (mutation probability or mutation rate). Normally, offspring are mutated after being created by recombination.

Real valued mutation

Mutation of real variables means that randomly created values are added to the variables with a low probability. Thus, the probability of mutating a variable (mutation rate) and the size of the changes for each mutated variable (mutation step) must be defined.

The probability of mutating a variable is inversely proportional to the number of variables (dimensions). The more dimensions one individual has, the smaller is the mutation probability. Different papers reported results for the optimal mutation rate. (Mühlenbein and Schlierkamp-Voosen 1993) writes that a mutation rate of $1/n$ (n: number of variables of an individual) produced good results for a wide variety of test functions. It means that per mutation only one variable per individual is changed/mutated. Thus, the mutation rate is independent of the size of the population. Similar results are reported in (Bäck 1993) and (Bäck 1996) for a binary valued representation. For unimodal functions a mutation rate of $1/n$ was the best choice. An increase in the mutation rate at the beginning connected with a decrease in the mutation rate to $1/n$ at the end gave only an insignificant acceleration of the search. The given recommendations for the mutation rate are only correct for separable functions. However, most real world functions are not fully separable. For these functions no recommendations for the mutation rate can be given. As long as nothing else is known, a mutation rate of $1/n$ is suggested as well.

The size of the mutation step is usually difficult to choose. The optimal step size depends on the problem considered and may even vary during the optimization process. It is known, that small steps (small mutation steps) are often successful, especially when the individual is already well adapted. However, larger changes (large mutation steps) can produce good results much quicker. Thus, a good mutation operator should often produce small step sizes with a high probability and large step sizes with a low probability.

In (Mühlenbein and Schlierkamp-Voosen 1993) and (Mühlenbein 1994) such an operator is proposed (mutation operator of the Breeder Genetic Algorithm):

$$
\begin{aligned}
&Var_i^{Mut} = Var_i + s_i \cdot r_i a_i \quad i \in \{1,2,\cdots,n\} \text{ uniform at random,}\\
&s_i \in \{-1,1\} \text{ uniform at random,}\\
&r_i = r \cdot domain_i, \, r : \text{mutation range}(\text{standard} : 10\%),\\
&a_i 2^{-u \cdot k}, u \in [0,1] \text{ uniform at random, } k : \text{mutation precision.}
\end{aligned}
$$

This mutation algorithm is able to generate most points in the hypercube defined by the variables of the individual and range of the mutation (the range of mutation is given by the value of the parameter *r* and the domain of the variables). Most mutated individuals will be generated near the individual before mutation. Only some mutated individuals will be far away from the not mutated individual. That means, the probability of small step-sizes is greater than that the probability of bigger steps. Figure 48 tries to give an impression of the mutation results of this mutation operator.

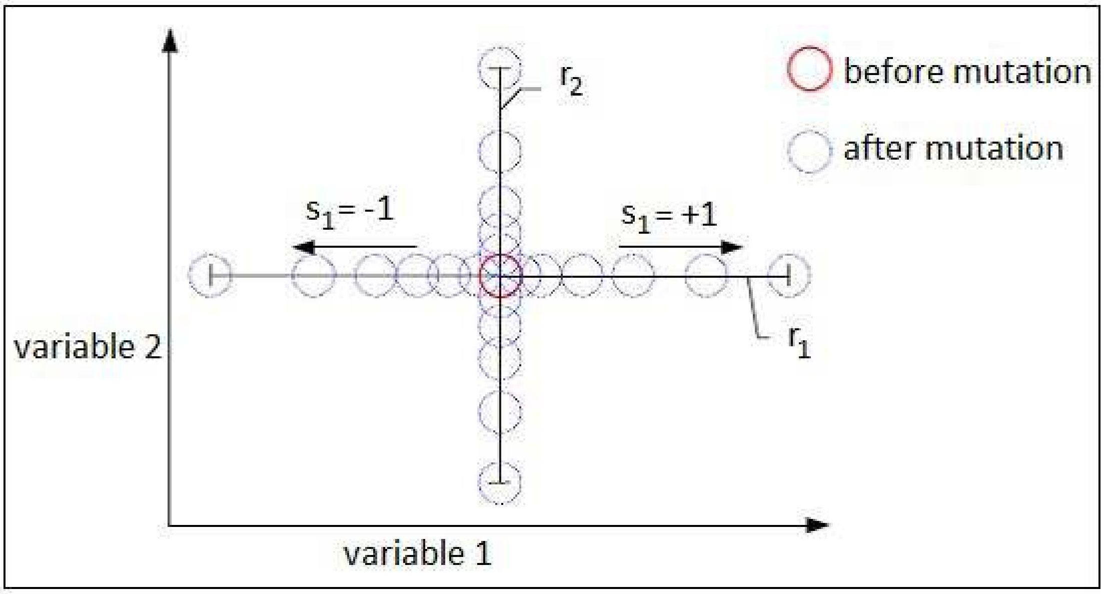


Figure 48: Effect of mutation of real variables in two dimensions (adapted from http://www.geatbx.com/)

The parameter *k* (mutation precision) defines indirectly the minimal step size possible and the distribution of mutation steps inside the mutation range. The smallest relative mutation step size is 2^{-k}, the largest $2^0 = 1$. Thus, the mutation steps are created inside the area $[r, r{\cdot}2^{-k}]$ (*r*: mutation range). With a mutation precision of $k = 16$, the smallest mutation step possible is $r{\cdot}2^{-16}$. Thus, when the variables of an individual are so close to the optimum, a further improvement is not possible. This can be circumvented by decreasing the mutation range (restart of the evolutionary run or use of multiple strategies)

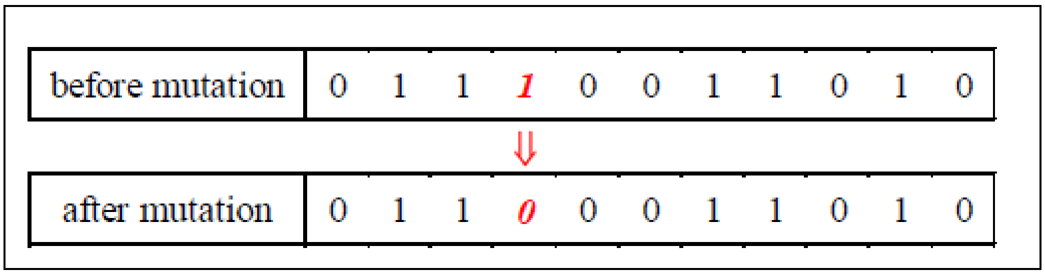
Typical values for the parameters of the mutation operator are the following:

mutation precision *k*:	$k \in \{4, 5, \ldots, 20\}$
mutation range *r*:	$r \in [0.1, 10^{-6}]$

By changing these parameters, very different search strategies can be defined.

Binary mutation

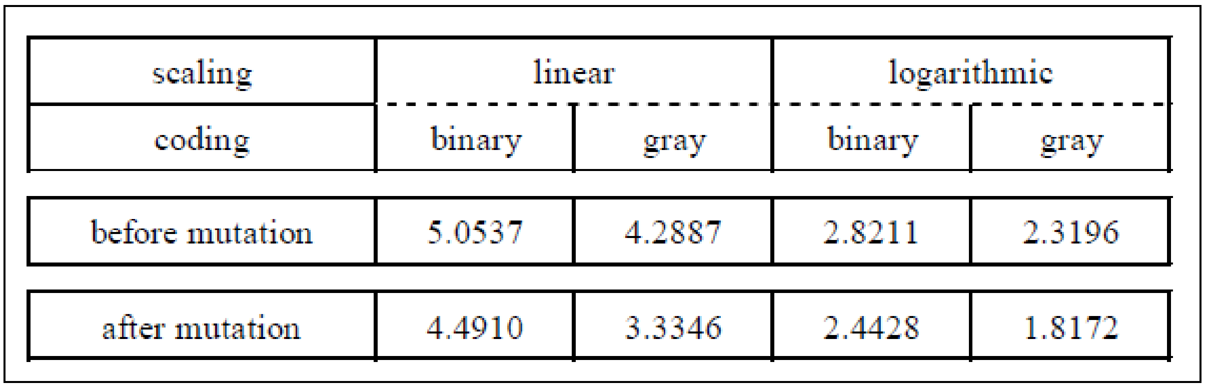
For binary valued individuals mutation means the flipping of variable values, because every variable has only two states. Thus, the size of the mutation step is always 1. For every individual the variable value to change is chosen (mostly uniform at random). Figure 49 shows an example of a binary mutation for an individual with 11 variables, where variable 4 is mutated.



before mutation	0	1	1	*1*	0	0	1	1	0	1	0
				⇓							
after mutation	0	1	1	*0*	0	0	1	1	0	1	0

Figure 49: Individual before and after binary mutation (adapted from http://www.geatbx.com/)

Assuming that the above individual decodes a real number in the bounds [1, 10], the effect of the mutation depends on the actual coding. Figure 50 shows the different numbers of the individual before and after mutation for binary/grey and arithmetic/logarithmic coding.



scaling	linear		logarithmic	
coding	binary	gray	binary	gray
before mutation	5.0537	4.2887	2.8211	2.3196
after mutation	4.4910	3.3346	2.4428	1.8172

Figure 50: Result of the binary mutation (adapted from http://www.geatbx.com/)

However, there is no longer a reason to decode real variables into binary variables. The advantages of mutation operators for real variables were shown in some publications, e.g. (Michalewicz 1994) and (Davis 1991).

3.2 Genetic algorithms

A genetic algorithm is a type of a searching algorithm. It searches a solution space for an optimal solution to a problem. The key characteristic of the genetic algorithm is how the searching is done. The algorithm creates a "population" of possible solutions to the problem and lets them "evolve" over multiple generations to find better and better solutions. The generic form of the genetic algorithm is shown in Figure 51. The items in bold in the algorithm are defined here.

```
1. Create a population of random candidate solutions named pop.
2. Until the algorithm termination conditions are met, do the following (each iteration is called a generation):
   a) Create an empty population named new-pop.
   b) While new-pop is not full, do the following:
      1) Select two individuals at random from pop so that individuals which are more fit are more likely to
         be selected.
      2) Cross-over the two individuals to produce two new individuals.
   c) Let each individual in new-pop have a random chance to mutate.
   d) Replace pop with new-pop.
3. Select the individual from pop with the highest fitness as the solution to the problem.
```

Figure 51: The Genetic Algorithm

The **population** consists of the collection of candidate solutions that we are considering during the course of the algorithm. Over the generations of the algorithm, new members are "born" into the population, while others "die" out of the population. A single solution in the population is referred to as an **individual**. The **fitness** of an individual is a measure of how "good" is the solution represented by the individual. The better solution has a higher fitness value – obviously, this is dependent on the problem to be solved. The **selection** process is analogous to the survival of the fittest in the natural world. Individuals are selected for "breeding" (or **cross-over**) based upon their fitness values. The crossover occurs by mingling two solutions together to produce two new individuals. During each generation, there is a small chance for each individual to **mutate**.

To use a genetic algorithm, there are several questions that need to be answered:

- How is an individual represented?
- How is an individual's fitness calculated?
- How are individuals selected for breeding?
- How are individuals crossed-over?
- How are individuals mutated?
- What is the size of the population?
- What are the "termination conditions"?

Most of these questions have problem specific answers. The last two, however, can be discussed in a more general way.

The size of the population is highly variable. The population should be as large as possible. The limiting factor is, of course, the running time of the algorithm. The larger population means more time consuming calculation.

The algorithm in Figure 51 has a very vague end point – the meaning of "until the termination conditions are met" is not immediately obvious. The reason for this is that there is no one way to end the algorithm. The simplest approach is to run the search for a set number of generations – the longer. Another approach is to end the algorithm after a certain number of generations pass with no improvement of the fitness of the best individual in the population. There are other possibilities as well. Since most of the other questions are dependent upon the search problem, we will look at two example problems that can be solved using genetic algorithms: finding a mathematical function's maximum and the travelling salesman problem.

3.2.1 Function maximization

Example (Thede 2004) One application for a genetic algorithm is to find values for a collection of variables that will maximize a particular function of those variables. While this type of problem could be solved otherwise, it is useful as an example of the operation of genetic algorithms. For this example, let's assume that we are trying to determine such variables that produce the maximum value for this function:

$$f(w, x, y, z) = w^3 + x^2 - y^2 - z^2 + 2yz - 3wx + wz - xy + 2$$

This could probably be solved using multivariable calculus, but it is a good simple example of the use of genetic algorithms. To use the genetic algorithm, we need to answer the questions listed in the previous section.

How is an individual represented?

What information is needed to have a "solution" of the maximization problem? It is clear that we need only values: *w, x, y*, and *z*. Assuming that we have values for these four variables, we have a candidate solution for our problem.

The question is how to represent these four values. A simple way to do this is to use an array of four values (integers or floating point numbers). However, for genetic algorithms it is usually better to have a larger individual – this way, variations can be done in a more subtle way. The research shows (Holland 1975) that representation of individuals using bit strings offers the best performance. We can simply choose a size in bits for each variable, and then concatenate the four values together into a single bit string.

For example, we will choose to represent each variable as a four-bit integer, making our entire individual a 16-bit string. Thus, an individual such as

1101 0110 0111 1100

represents a solution where $w = 13$, $x = 6$, $y = 7$, and $z = 12$.

How is an individual's fitness calculated?

Next, we consider how to determine the fitness of each individual. There is generally a differentiation between the *fitness* and *evaluation* functions. The evaluation function is a function that returns an absolute measure of the individual. The fitness function is a function that measures the value of the individual relative to the rest of the population.

In our example, an obvious evaluation function would be to simply calculate the value of *f* for the given variables. For example, assume we have a population of 4 individuals:

1010 1110 1000 0011
0110 1001 1111 0110
0111 0110 1110 1011
0001 0110 1000 0000

The first individual represents $w = 10$, $x = 14$, $y = 8$, and $z = 3$, for an *f* value of 671. The values for the entire population can be seen in the following table:

Individual	***w***	***x***	***y***	***z***	***f***
1010111010000011	10	14	8	3	671
0110100111110110	6	9	15	6	-43
0111011011101011	7	6	14	11	239
0001011010000000	1	6	8	0	-91

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK
Click on the ad to read more

The fitness function can be chosen from many options. For example, the individuals could be listed in order from the lowest to the highest evaluation function values, and an ordinal ranking applied. OR The fitness function could be the individual's evaluation value divided by the average evaluation value. Looking at both of these approaches would give us something like this:

Individual	***evaluation***	***ordinal***	***averaging***
1010111010000011	671	4	2.62
0110100111110110	-43	2	0.19
0111011011101011	239	3	0.81
0001011010000000	-91	1	0.03

The key is that the fitness of an individual should represent the value of the individual relative to the rest of the population, so that the best individual has the highest fitness.

How are individuals selected for breeding?

The key to the selection process is that it should be probabilistically weighted so that higher fitness individuals have a higher probability of being selected. Other than these specifications, the method of selection is open to interpretation.

One possibility is to use the ordinal method for the fitness function, then calculate a probability of selection that is equal to the individual's fitness value divided by the total fitness of all the individuals. In the example above, that would give the first individual a 40% chance of being selected, the second a 20% chance, the third a 30% chance, and the fourth a 10% chance. It gives better individuals more chances to be selected.

A similar approach could be used with the average fitness calculation. This would give the first individual a 72% chance, the second a 5% chance, the third a 22% chance, and the fourth a 1% chance. This method makes the probability more dependent on the relative evaluation functions of each individual.

How are individuals crossed-over?

Once we have selected a pair of individuals, they are "bred" – or in genetic algorithm language, they are *crossed-over*. Typically two children are created from each set of parents. One method for performing the cross-over is described here, but there are other approaches. Two locations are randomly chosen within the individual. These define corresponding substrings in each individual. The substrings are swapped between two parent individuals, creating two new children. For example, let's look at our four individuals again:

1010 1110 1000 0011
0110 1001 1111 0110
0111 0110 1110 1011
0001 0110 1000 0000

Let's assume that the first and third individuals are chosen for cross-over. Keep in mind that the selection process is random. The fourth and fourteenth bits are randomly selected to define the substring to be swapped, so the cross-over looks like this:

1010111010000011   *101**01110100000**11*   *1011011011101011*
→   →
0111011011101011   *011**10110111010**11*   *0110111010000011*

Thus, two new individuals are created. We should create new individuals until we replace the entire population – in our example, we need one more cross-over operators. Assume that the first and fourth individuals are selected this time. Note that an individual may be selected multiple times for breeding, while other individuals might never be selected. Further assume that the eleventh and sixteenth bits are randomly selected for the cross-over point. We could apply the second cross-over like this:

1010111010000011   *1010111010000011*   *1010111010000000*
→   →
0001011010000000   *0001011010000000*   *0001011010000011*

The second generation of the population is the following:

1011 0110 1110 1011
0110 1110 1000 0011
1010 1110 1000 0000
0001 0110 1000 0011

How are individuals mutated?

Finally, we need to allow individuals to mutate. When using bit strings, the easiest way to implement the mutation is to allow every single bit in every individual a chance to mutate. This chance should be very small, since we don't want to have individuals changing dramatically due to mutation. Setting the percentage so, that roughly one bit per individual has a chance to change on average.

The mutation will consist of having the bit "flip": *1* changes to *0* and *0* changes to *1*. In our example, assume that the bold and italicized bits have been chosen for mutation:

1011011011101011 → *1011011011101011*
*01101**1**1010000011* → *01101**0**1010000011*
*10101110100**0**0000* → *10101110100**1**0000*
*0**0**0101101000000**1**1* → *0**1**0101101000000**0**1*

Wrapping Up

Finally, let's look at the next population:

Individual	*w*	*x*	*y*	*z*	*f*
1011011011101011	11	6	14	11	1,045
0110101010000011	6	10	8	3	51
1010111010010000	10	14	9	0	571
0101011010000001	5	6	8	1	-19

The average evaluation value is 412, versus an average of 194 for the previous generation. Clearly, this is a constructed example, but the exciting thing about genetic algorithms is that this sort of improvement actually does occur in practice.

3.2.2 The travelling salesman problem

Example (Thede 2004)

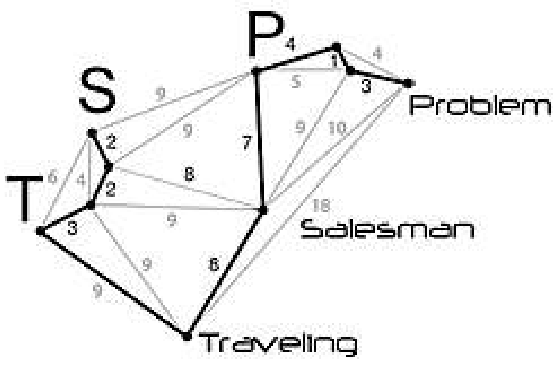


Figure 52: Traveling Salesman Problem (adapted from http://www.amdusers.com/wiki/tiki-index.php?page=TSP/)

Now let's look at a less contrived example. Genetic algorithms can be used to solve the traveling salesman problem (TSP), see Figure 52. For those who are unfamiliar with this problem, it can be stated in two ways. Informally, there is a traveling salesman who services some number of cities, including his home city. He needs to travel on a trip such that he starts in his home city, visits every other city exactly once, and returns home. He wants to set up the trip so that it costs him the least amount of money possible. The more formal way of stating the problem casts it as a graph problem. Given a weighted graph with *N* vertices, find the lowest cost path from some city *v* that visits every other node exactly once and returns to *v*. For a more thorough discussion of TSP, see (Garey and Johnson 1979).

The problem with TSP is that it is an NP-complete problem. The only known way to find the answer is to list every possible route and find the one with the lowest cost. Since there are a total of (*N* – 1)! routes, this quickly becomes intractable for large *N*. There are approximation algorithms that run in a reasonable time and produce reasonable results – a genetic algorithm is one of them.

Individual Representation and Fitness

Our first step is to decide on a representation for an individual candidate solution, or *tour*. The bit string model is not very useful because cross-overs and mutations should produce a tour that is invalid. Remember that every city has to occur in the tour exactly once except the home city.

The only real choice for representing the individual is a vector or array of cities, most likely stored as integers. The costs of travel between cities should be provided. Using a vector of integers causes some problems with cross-over and mutation, as we'll see in the next section.

For example, the following file defines a TSP with four cities:

0 2 6 3
2 0 9 7
6 9 0 8
3 7 8 0

This file shows that the cost of travel from city 0 to city 2 is 6, while the cost from city 3 to city 1 is 7. Then we could represent an individual as a vector of five cities:

[*0 2 1 3 0*]

We also need to be careful when calculating fitness. The clear choice for the evaluation function is the cost of the tour. Remember that a good tour is one whose cost is low, so we need to calculate fitness so that a low cost tour – it corresponds to high fitness individual (path cost).

Cross-over and Mutation

When performing cross-over and mutation, we need to make sure that we produce valid tours. This means modifying the cross-over and mutation process. We can still use the same basic idea, however, choose a substring of the vector of cities at random for cross-over, and choose a single point in the vector at random for mutation. The mechanics are a bit different.

For cross-over, rather than simply swapping the substrings (which could easily result in an invalid tour), we will instead keep the same cities, but change their order to match the other parent of the cross-over. For example, assume we have the following two individuals, with the third and sixth cities chosen for the cross-over substring

7 3 ***6 1 0 2*** *5 4*
5 4 ***1 7 2 3*** *6 0*

Rather than swapping the cities, which would result in duplications of cities within each tour, we keep the cities in each tour the same, but we reorder the cities to match their order in the other parent. In the above example, we would replace the "**6 1 0 2**" section in the first parent with "**1 2 6 0**", because that is the order in which those four cities appear in the second parent. Similarly, the "**1 7 2 3**" section in the second parent is replaced with "**7 3 1 2**", and we get offspring

7 3 ***1 2 6 0*** *5 4*
5 4 ***7 3 1 2*** *6 0*

This allows the concept of the cross-over to remain – that each parent contributes to the construction of new individuals – while guaranteeing that a valid tour is created.

There are a number of approaches for mutation. The simplest is that whenever a city is chosen as the location of a mutation, it is simply swapped with the next city in the tour. For example, if the 6 is chosen for mutation in this tour:

7 3 1 2 ***6 0*** *5 4*

we would get this tour after the mutation occurs:

7 3 1 2 ***0 6*** *5 4*

Other options for mutation include selecting two cities at random and swapping them, or selecting an entire substring at random and reversing it, but the concept remains the same – making a relatively small change to an individual.

3.3 Genetic programming

Genetic programming is much more powerful than genetic algorithms. The output of the genetic algorithm is a quantity, while the output of the genetic programming is a computer program. In essence, this is the beginning of computer programs that program themselves. Genetic programming works best for several types of problems. The first type is where there is no ideal solution, (for example, a program that drives a car). There is no one solution to driving a car. Some solutions drive safely at the expense of time, while others drive fast at a high safety risk. Therefore, driving a car consists of making compromises of speed versus safety, as well as many other variables. In this case genetic programming will find a solution that attempts to compromise and be the most efficient solution from a large list of variables. Furthermore, genetic programming is useful in finding solutions where the variables are constantly changing. In the previous car example, the program will find one solution for a smooth concrete highway, while it will find a totally different solution for a rough unpaved road.

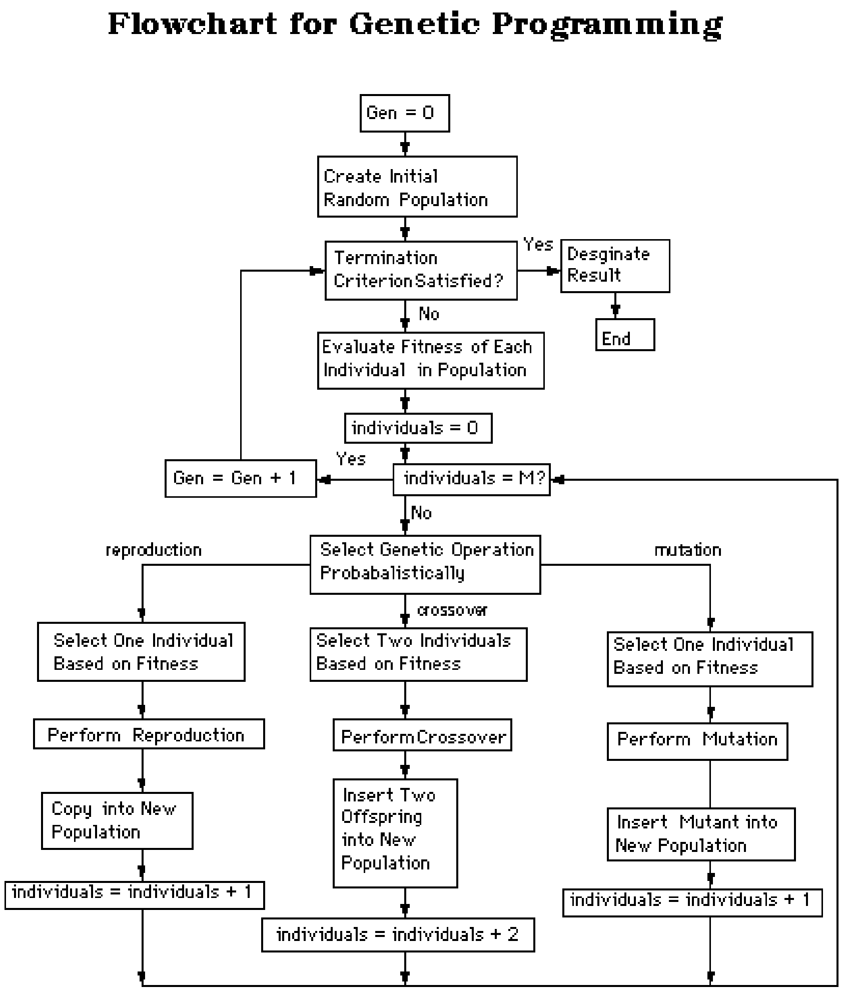

Flowchart for Genetic Programming
Gen = 0
Create Initial Random Population
Termination Criterion Satisfied?
Yes
Desginate Result
End
No
Evaluate Fitness of Each Individual in Population
individuals = 0
Yes
Gen = Gen + 1
individuals = M?
No
reproduction
Select Genetic Operation Probabalistically
mutation
crossover
Select One Individual Based on Fitness
Select Two Individuals Based on Fitness
Select One Individual Based on Fitness
Perform Reproduction
Perform Crossover
Perform Mutation
Copy into New Population
Insert Two Offspring into New Population
Insert Mutant into New Population
individuals = individuals + 1
individuals = individuals + 2
individuals = individuals + 1

Figure 53: Result of the binary mutation (adapted from http://www.geneticprogramming.com)

The main difference between genetic programming and genetic algorithms is the representation of the solution. Genetic programming creates computer programs in the scheme computer languages as the solution. Genetic algorithms create a string of numbers that represent the solution. Genetic programming uses four steps to solve problems:

1. Generate an initial population of random compositions of the functions and terminals of the problem (computer programs).
2. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
3. Create a new population of computer programs.
   1) Copy the best existing programs
   2) Create new computer programs by mutation.
   3) Create new computer programs by crossover (sexual reproduction).
4. The best computer program that appeared in any generation, the best-so-far solution, is designated as the result of genetic programming (Koza 1992).

The flowchart for Genetic Programming (GP) is shown in Figure 53.

The most difficult and most important concept of genetic programming is the *fitness function*. The fitness function determines how well a program is able to solve the problem. It varies greatly from one type of program to the next. For example, if one were to create a genetic program to set the time of a clock, the fitness function would simply be the amount of time that the clock is wrong. Unfortunately, few problems have such an easy fitness function; most cases require a slight modification of the problem in order to find the fitness.

The terminal and function sets are also important components of genetic programming. The terminal and function sets are the alphabet of the programs to be made. The terminal set consists of the variables and constants of the programs. Functions are several mathematical functions, such as addition, subtraction, division, multiplication and other more complex functions.

3.3.1 Operators of genetic programming

Crossover Operator

Two primary operations exist for modifying structures in genetic programming. The most important one is the crossover operation. In the crossover operation, two solutions are combined to form two new solutions or offspring. The parents are chosen from the population by a function of the fitness of the solutions. Three methods exist for selecting the solutions for the crossover operation.

The first method uses probability based on the fitness of the solution. If $f(s_i(t))$ is the fitness of the solution S_i and

$$F = \sum_{i=1}^{M} f(s_i(t))$$

F is the total sum of all the members of the population M. The probability to solution S_i is copied to the next generation is (Koza 1992) then:

$$p_i = \frac{f(s_i(t))}{\sum_{i=1}^{M} f(s_i(t))}$$

Another method for selecting the solution to be copied is tournament selection. Typically the genetic program chooses two solutions random. The solution with the higher fitness will win. This method simulates biological mating patterns in which, two members of the same sex compete to mate with a third one of a different sex. Finally, the third method is done by rank. In rank selection, selection is based on the rank, (not the numerical value) of the fitness values of the solutions of the population (Koza 1992).

The creation of offsprings from the crossover operation is accomplished by deleting the crossover fragment of the first parent and then inserting the crossover fragment of the second parent. The second offspring is produced in a symmetric manner. For example consider the two S-expressions in Figure 54, written in a modified scheme programming language and represented in a tree.

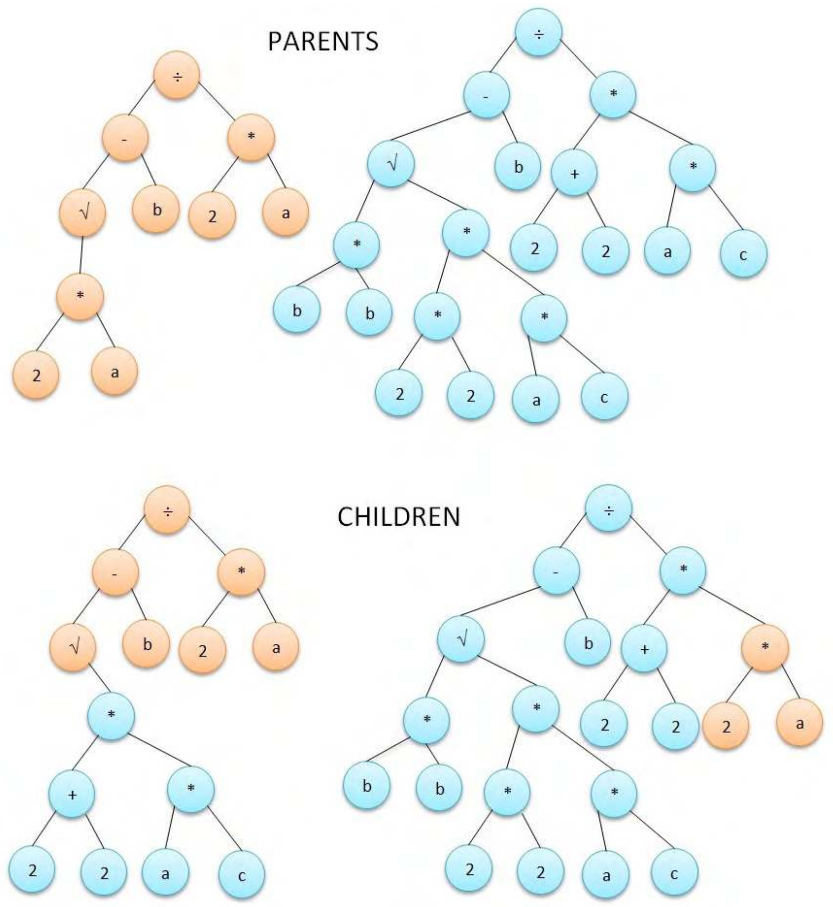


Figure 54: Crossover operation for genetic programming. The bold selections on both parents are swapped to create the offspring or children. The child on the right is the parse tree representation for the quadratic equation. (adapted from http://www.geneticprogramming.com)

An important improvement that genetic programming displays over genetic algorithms is its ability to create two new solutions from the same solution. In Figure 55 the same parent is used twice to create two new children. This figure illustrates one of the main advantages of genetic programming over genetic algorithms. In genetic programming identical parents can yield different offspring, while in genetic algorithms identical parents would yield identical offspring. The bold selections indicate the subtrees to be swapped.

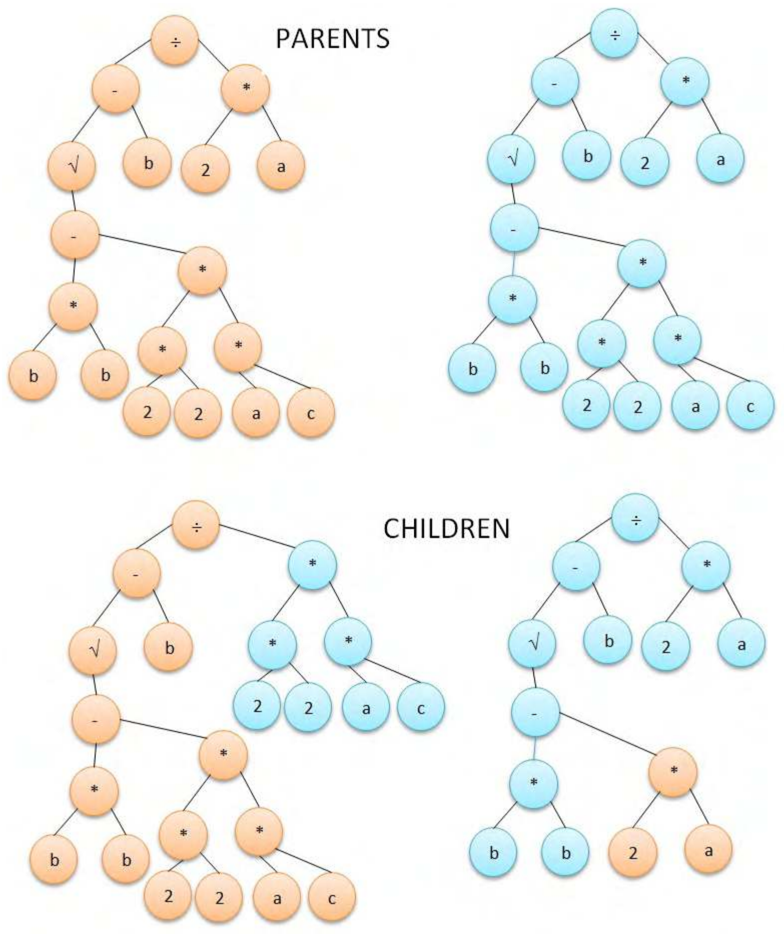


Figure 55: Crossover operation for identical parents. (adapted from http://www.geneticprogramming.com)

Mutation Operator

Mutation is another important feature of genetic programming. Two types of mutations are possible. In the first kind a function can only replace a function or a terminal can only replace a terminal. In the second kind an entire subtree can replace another subtree. Figure 56 explains the concept of mutation. Genetic programming uses two different types of mutations. The top parse tree is the original agent. The bottom left parse tree illustrates a mutation of a single terminal (2) for another single terminal (a). It also illustrates a mutation of a single function (-) for another single function (+). The parse tree on the bottom right illustrates a replacement of a subtree by another subtree.

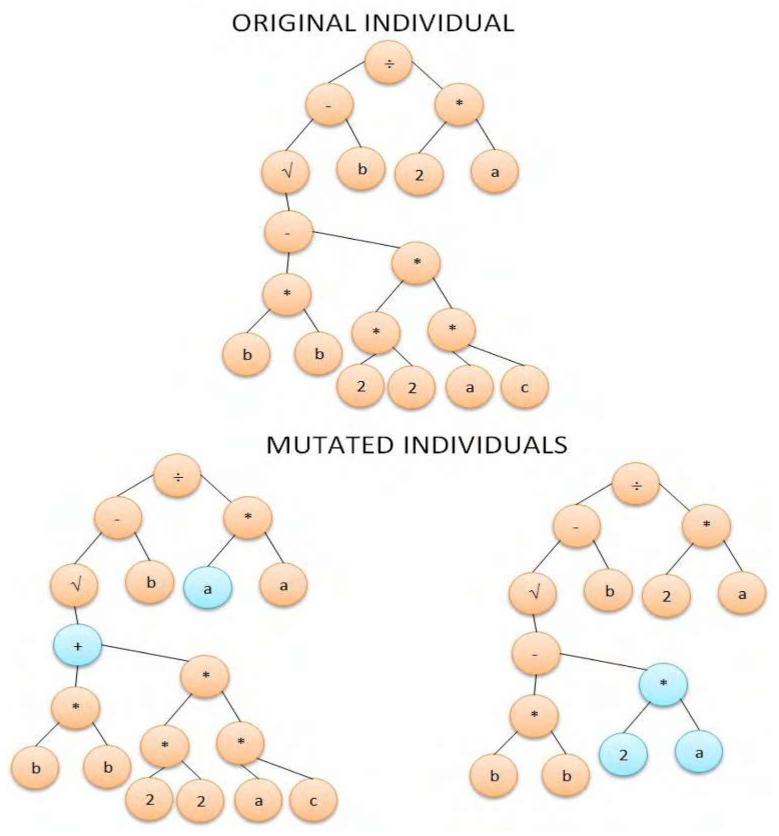


Figure 56: Mutation operation. (adapted from http://www.geneticprogramming.com)

3.3.2 Applications of genetic programming

Genetic programming can be used for example in the following task solving:

Gun Firing Program. A more complicated example consists of training a genetic program to fire a gun to hit a moving target. The fitness function is the distance that the bullet is off from the target. The program has to learn to take into account a number of variables, such as the wind velocity, the type of gun used, the distance to the target, the height of the target, the velocity and acceleration of the target. This problem represents the type of problem for which genetic programs are best. It is a simple fitness function with a large number of variables.

Water Sprinkler System. Consider a program to control the flow of water through a system of water sprinklers. The fitness function is the correct amount of water evenly distributed over the surface. Unfortunately, there is no one variable encompassing this measurement. Thus, the problem must be modified to find a numerical fitness. One possible solution is placing water-collecting measuring devices at certain intervals on the surface. The fitness could then be the standard deviation in water level from all the measuring devices. Another possible fitness measure could be the difference between the lowest measured water level and the ideal amount of water; however, this number would not account in any way the water marks at other measuring devices, which may not be at the ideal mark.

Maze Solving Program. If one were to create a program to find the solution to a maze, first, the program would have to be trained with several known mazes. The ideal solution from the start to the finish of the maze would be described by a path of dots. The fitness in this case would be the number of dots the program is able to find. In order to prevent the program from wandering around the maze too long, a time limit is implemented along with the fitness function.

3.4 Differential evolution

Differential Evolution (DE) is a population-based optimization method that works on real-number-coded individuals (Storn and Price 1997). DE is quite robust, fast, and effective, with global optimization ability. It does not require the objective function to be differentiable, and it works well even with noisy and time-dependent objective functions. Differential Evolution is a parallel direct search method which utilizes *NP* parameter vectors $x_{i,G}$, $i = 0, 1, 2, \ldots, NP\text{-}1$ as a population for each generation *G*. *NP* does not change during the minimization process. The initial population is chosen randomly if nothing is known about the system. As a rule, we will assume a uniform probability distribution for all random decisions unless otherwise stated. In case a preliminary solution is available, the initial population is often generated by adding normally distributed random deviations to the nominal solution $x_{nom,0}$. The crucial idea behind DE is a scheme for generating trial parameter vectors. DE generates new parameter vectors by adding a weighted difference vector between two population members to a third member. If the resulting vector yields a lower objective function value than a predetermined population member, the newly generated vector will replace the vector with which it was compared in the following generation. The comparison vector can but need not be part of the generation process mentioned above. In addition the best parameter vector $x_{best,G}$ is evaluated for every generation *G* in order to keep track of the progress that is made during the minimization process.

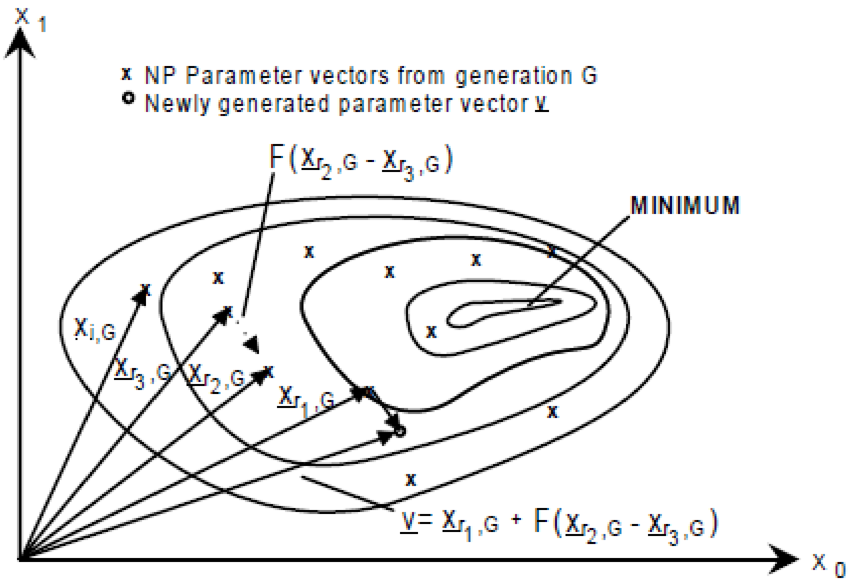

x NP Parameter vectors from generation G
o Newly generated parameter vector v
$F(x_{r_2,G} - x_{r_3,G})$
MINIMUM
$x_{i,G}$
$x_{r_3,G}$
$x_{r_2,G}$
$x_{r_1,G}$
$v = x_{r_1,G} + F(x_{r_2,G} - x_{r_3,G})$
x_1
x_0

Figure 57: Two-dimensional example of an objective function showing its contour lines and the process for generating v in scheme DE1. The weighted difference vector of two arbitrarily chosen vectors is added to a third vector to yield the vector *v*.

It is extracting distance and direction information from the population to generate random deviations results in an adaptive scheme with excellent convergence properties. Several variants of DE have been tried, the two most promising of which are subsequently presented in greater detail.

Scheme DE1

The first variant of DE works as follows: for each vector $x_{i,G}$, $i = 0, 1, 2, \dots, NP\text{-}1$, a trial vector v is generated according to

$$v = x_{r_1,G} + F \cdot \left(x_{r_2,G} - x_{r_3,G}\right),$$

with $r_1, r_2, r_3 \in [0, NP\text{-}1]$, integer and mutually different, and $F > 0$. Integers r_1, r_2 and r_3 are chosen randomly from the interval $[0, NP\text{-}1]$ and are different from the running index i. F is a real and constant factor which controls the amplification of the differential variation $\left(x_{r_2,G} - x_{r_3,G}\right)$. Figure 57 (Storn and Price 1997) shows a two-dimensional example that illustrates the different vectors that are used in DE1.

In order to increase the diversity of the parameter vectors, the vector

$$u = (u_0, u_1, \cdots, u_{D-1})^T$$

with $u_j = \begin{cases} v_j & \text{for } j = \langle n \rangle_D, \langle n+1 \rangle_D, \cdots, \langle n+L-1 \rangle_D \\ (x_{i,G})_j & \text{for all other } \; j \in [0, D-1] \end{cases}$

is formed where the acute brackets $\langle \, \rangle_D$ denote the modulo function with modulus *D*.

Equations yield a certain sequence of the vector elements of *u* to be identical to the elements of *v*, the other elements of *u* acquire the original values of $x_{i,G}$. Choosing a subgroup of parameters for mutation is similar to a process known as crossover in Genetic Algorithms. This idea is illustrated in Figure 58 (Storn and Price 1997) for $D = 7$, $n = 2$ and $L = 3$. The starting index *n* is a randomly chosen integer from the interval [0, *D*-1]. The integer *L*, which denotes the number of parameters that are going to be exchanged, is drawn from the interval [1, *D*]. The algorithm which determines L works according to the following lines of pseudo code where *rand*( ) is supposed to generate a random number $\in$ [0,1):

```
L = 0;
do {
L = L + 1;
}while(rand()< CR) AND (L < D));
```

Hence the probability $\Pr(L >= \nu) = (CR)^{\nu-1}$, $\nu > 0$. $CR \in [0,1]$ is the crossover probability and constitutes a control variable for the DE1-scheme. The random decisions for both *n* and *L* are made anew for each trial vector *v*.

In order to decide whether the new vector u shall become a population member of generation *G*+1, it will be compared to $x_{i,G}$. If vector *u* yields a smaller objective function value, than $x_{i,G}$, $x_{i,G+1}$ is set to *u*, otherwise the old value $x_{i,G}$ is retained.

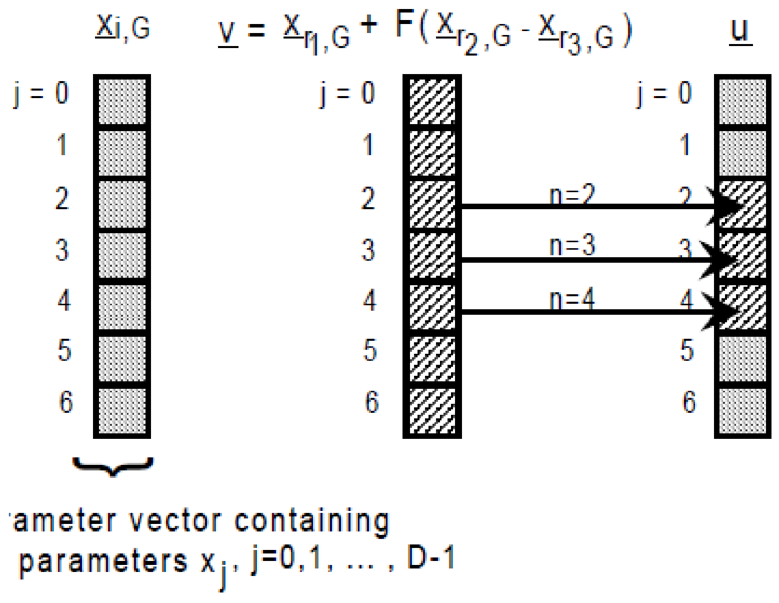


Figure 58: Illustration of the crossover process for *D*=7, *n*=2 and *L*=3.

Scheme DE2

Basically, scheme DE2 works in the same way as DE1 but generates the vector v according to

$$v = x_{r_1,G} + \lambda \cdot (x_{best,G} - x_{i,G}) + F \cdot (x_{r_2,G} - x_{r_3,G}),$$

introducing an additional control variable λ. The idea behind λ is to provide a means to enhance the greediness of the scheme by incorporating the current best vector $x_{best,G}$. This feature can be useful for objective functions where the global minimum is relatively easy to find. Figure 59 (Storn and Price 1997) illustrates the vector-generation process defined by the previous equation. The construction of u from v and $x_{i,G}$ as well as the decision process are identical to DE1.

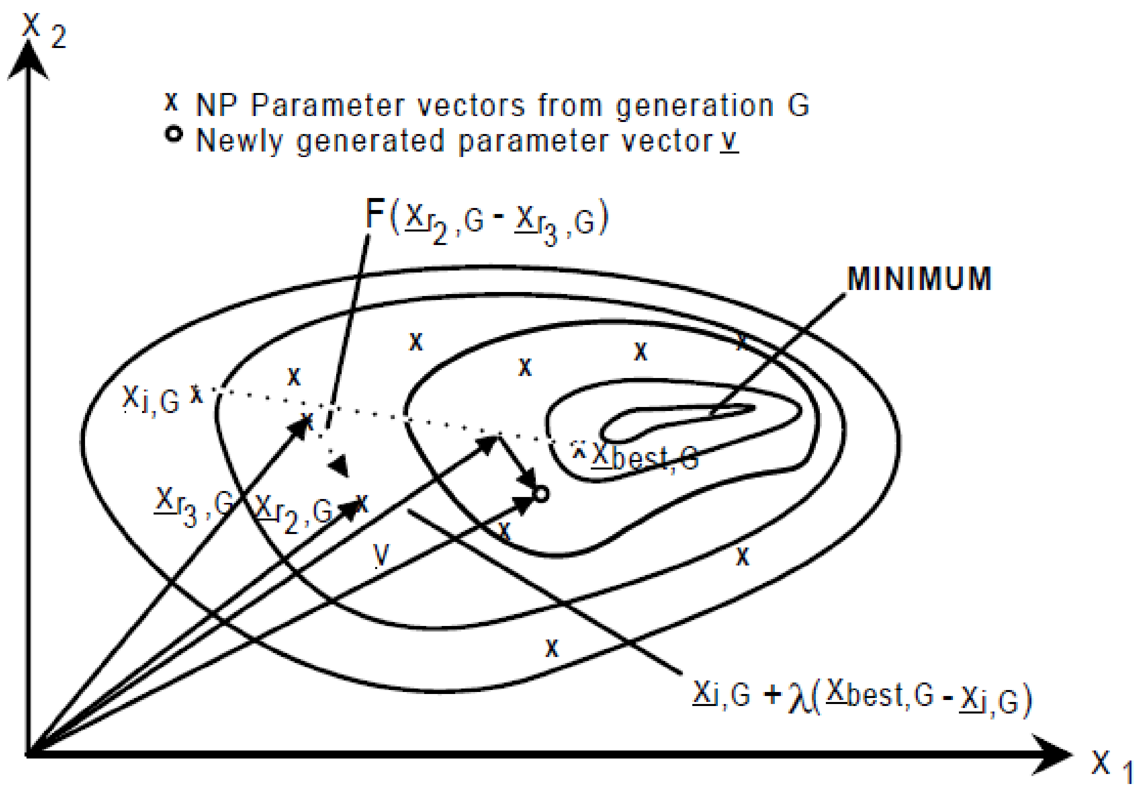

X 2
x NP Parameter vectors from generation G
o Newly generated parameter vector v
F(Xr2,G - Xr3,G)
MINIMUM
Xi,G
Xr3,G
Xr2,G
V
Xbest,G
Xi,G + λ(Xbest,G - Xi,G)
X 1

Figure 59: Two dimensional example of an objective function showing its contour lines and the process for generating v in scheme DE2.

Canonical DE

A schematic of the canonical DE strategy is given in Figure 60 (Price 1999). There are essentially five sections to the code depicted in figure 60 (Price 1999).

Section *1* describes the input to the heuristic. *D* is the size of the problem, *Gmax* is the maximum number of generations, *NP* is the total number of solutions, *F* is the scaling factor of the solution and *CR* is the factor for crossover. *F* and *CR* together make the internal tuning parameters for the heuristic.

Section *2* in figure 60 outlines the initialization of the heuristic. Each solution $x_{i,j,G}$=0 is created randomly between the two bounds $x^{(lo)}$ and $x^{(hi)}$. The parameter *j* represents the index to the values within the solution and parameter *i* indexes the solutions within the population. So, to illustrate, $x_{4,2,0}$ represents the fourth value of the second solution at the initial generation.

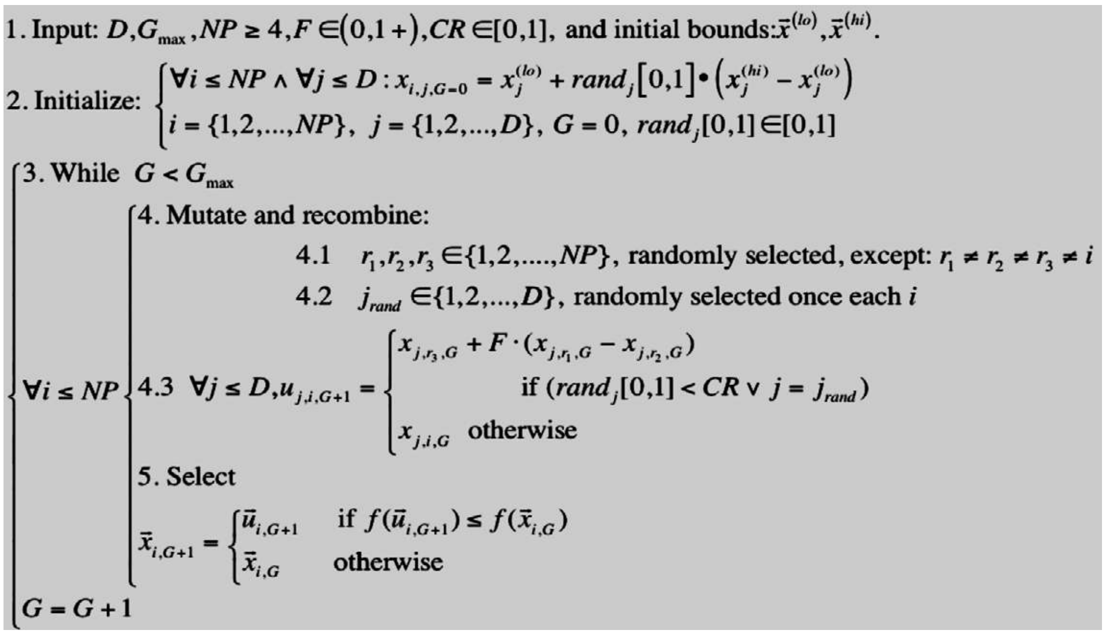
After initialization, the population is subjected to repeated iterations in section *3*.

Section *4* describes the conversion routines of DE. Initially, three random numbers r_1, r_2, r_3 are selected, unique to each other and to the current indexed solution *i* in the population in 4.1. Henceforth, a new index j_{rand} is selected in the solution, j_{rand} points to the value being modified in the solution as given in 4.2. In 4.3, two solutions, $x_{j,r1,G}$ and $x_{j,r2,G}$ are selected through the index r_1 and r_2 and their values subtracted. This value is then multiplied by *F*, the predefined scaling factor. This is added to the value indexed by r_3. However, this solution is not arbitrarily accepted in the solution. A new random number is generated, and if this random number is less than the value of *CR*, then the new value replaces the old value in the current solution. The fitness of the resulting solution, referred to as a perturbed vector $u_{j,i,G}$, is then compared with the fitness of $x_{j,i,G}$. If the fitness of $u_{j,i,G}$ is greater than the fitness of $x_{j,i,G}$, then $x_{j,i,G}$ is replaced with $u_{j,i,G}$; otherwise, $x_{j,i,G}$ remains in the population as $x_{j,i,G+1}$. Hence, the competition is only between the new *child* solution and its *parent* solution.

The description of some of the commonly used basic DE strategies is presented in Table 4. The description of all 10 basic strategies is described in (Price 1999) or (Storn. and Price 1997). These strategies differ in the way of calculating the perturbed vector $u_{j,i,G}$. Scaling vector *F* is replaced with a randomly generated vector F_{Rand} in *DELocalToBest* strategy and with a randomly generated vector $F_{NormRand}$ with normal distribution in strategies *DEBest1JIte*r and *DERand1DIter*.

```
1. Input: $D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, and initial bounds: $\bar{x}^{(lo)}, \bar{x}^{(hi)}$.
2. Initialize: $\begin{cases} \forall i \leq NP \wedge \forall j \leq D : x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \{1,2,...,NP\},\ j = \{1,2,...,D\},\ G = 0,\ rand_j[0,1] \in [0,1] \end{cases}$
3. While $G < G_{\max}$
    $\forall i \leq NP$
        4. Mutate and recombine:
            4.1  $r_1, r_2, r_3 \in \{1,2,....,NP\}$, randomly selected, except: $r_1 \neq r_2 \neq r_3 \neq i$
            4.2  $j_{rand} \in \{1,2,...,D\}$, randomly selected once each $i$
            4.3  $\forall j \leq D, u_{j,i,G+1} = \begin{cases} x_{j,r_3,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G}) \\ \quad \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i,G} \quad \text{otherwise} \end{cases}$
        5. Select
            $\bar{x}_{i,G+1} = \begin{cases} \bar{u}_{i,G+1} & \text{if } f(\bar{u}_{i,G+1}) \leq f(\bar{x}_{i,G}) \\ \bar{x}_{i,G} & \text{otherwise} \end{cases}$
    $G = G + 1$
```

Figure 60: Canonical DE Schematic.



```
1. Input: $D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, and initial bounds: $\vec{x}^{(lo)}, \vec{x}^{(hi)}$.
2. Initialize: $\begin{cases} \forall i \leq NP \wedge \forall j \leq D : x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \{1,2,\ldots,NP\},\ j = \{1,2,\ldots,D\},\ G = 0,\ rand_j[0,1] \in [0,1] \end{cases}$
3. While $G < G_{\max}$
    $\forall i \leq NP$
        4. Mutate and recombine:
            4.1  $r_1, r_2, r_3 \in \{1,2,\ldots,NP\}$, randomly selected, except: $r_1 \neq r_2 \neq r_3 \neq i$
            4.2  $j_{rand} \in \{1,2,\ldots,D\}$, randomly selected once each $i$
            4.3  $\forall j \leq D, u_{j,i,G+1} = \begin{cases} x_{j,r_3,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G}) & \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i,G} & \text{otherwise} \end{cases}$
        5. Select
            $\vec{x}_{i,G+1} = \begin{cases} \vec{u}_{i,G+1} & \text{if } f(\vec{u}_{i,G+1}) \leq f(\vec{x}_{i,G}) \\ \vec{x}_{i,G} & \text{otherwise} \end{cases}$
    $G = G + 1$
```

Table 4: Description of selected DE Strategies

Alcatel·Lucent
www.alcatel-lucent.com/careers
What if you could build your future and create the future?
One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".