4. Genetic vector field estimation

Vector field estimation is an optimisation problem. The search space presented by a vector field is large and contains a large number of sub-optimal solutions. Classical blockmatchers handle this by examining the vector field one block at a time. Vector field coherence or smoothness is obtained using schemes which results in spatial convergence effects in the vector field.

The genetic algorithm for vector field estimation presented in this chapter does not optimise vectors but it optimises vector fields. The objective, as with classic blockmatchers, is the minimisation of a matching criterion but the strategy used in this minimisation results in smooth vector fields without imposing spatial convergence artefacts..

4.1 Genetic algorithms

Genetic algorithms, commonly abbreviated as GAs, form a class of optimisation algorithms with robust performance on large search spaces with large numbers of local sub-optima. Large search spaces preclude full enumeration: the examination of all possible solutions. An abundance op local sub-optima prevents hill climbing techniques such as Powell's method or the conjugate gradient technique [Press91] from being effective. An example of a function with a large number of sub-optima is the function known as *f6* [Davis91] in genetic algorithm literature. This function is plotted in Figure 4.1.

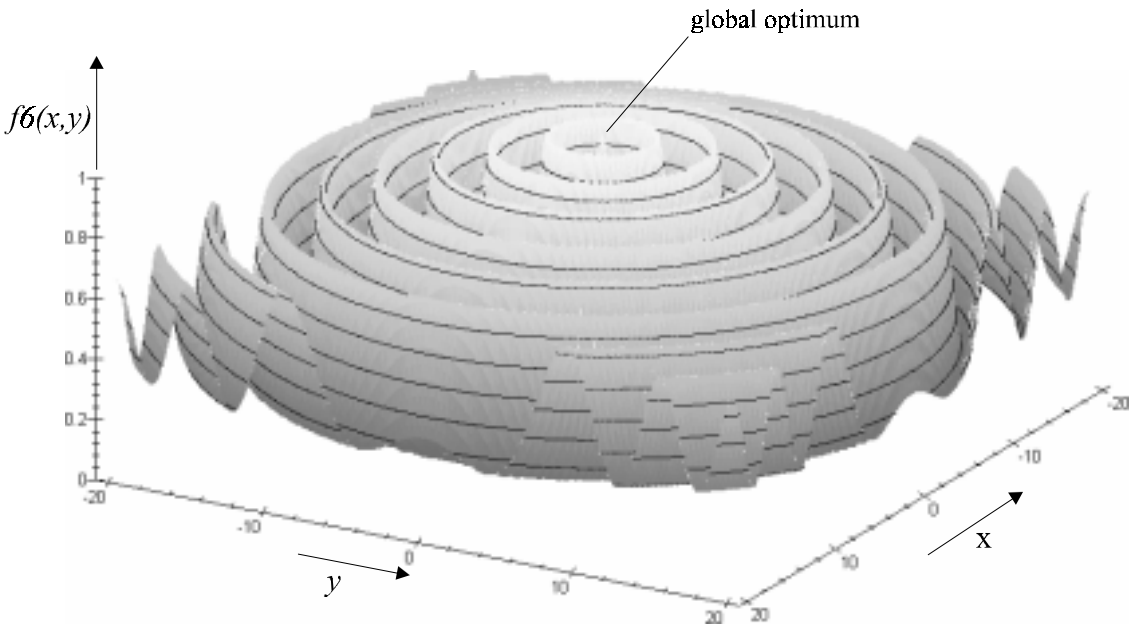

Figure 4.1: Plot of f6.

The mathematical description of *f6* is given by:

$$f\,6(x,y) = 0.5 - \frac{\left(\sin\left(\sqrt{x^2+y^2}\right)\right)^2 - 0.5}{\left(1+0.001(x^2+y^2)\right)^2} \quad . \tag{4.1}$$

The optimum of *f*6 is in this case chosen to be the maximum of *f*6. This *f*6 is a simple function of only two real numbers for which the optimum can be easily found using mathematical insights but which is difficult for numerical optimisation techniques because of the multitude of local optima. And this is with a search space which consists of only two numbers.

Many real world optimisation problems have much larger search spaces and also feature local sub-optima.

It is with this kind of problem in mind that genetic algorithms were conceived [Holl75]. They were inspired and are described along the lines of the metaphor of natural evolution. Whilst this metaphor is useful in describing, talking and even thinking about these techniques it is important to note that, as with all metaphors, it shouldn't be taken too far. The link between genetic algorithms and biology/genetics is no stronger than the link between neural networks and the human brain or the link between simulated annealing and actual physical cooling processes. The connection may have played some part in initial inspiration and it is certainly used in description of the various algorithms but that is the full extent of it.

One of the most important aspects of genetic algorithms is that they work not on one solution at a time but on a set of solutions. In keeping with the genetic metaphor each solution is encoded as a chromosome. At any one time the present set of chromosomes is called the population. The basic genetic algorithm can now be described as follows:

- Optimisation becomes evolution: as one generation of the population changes to the next generation the strongest chromosomes survive by reproducing into the next generation.
- The criterion being optimised becomes the fitness of a chromosome (or of it's decoded solution). The "objective" of evolution is to improve the overall fitness of the population.
- Natural selection is reflected by the fact that the chance which chromosomes have of being selected for reproduction into the next generation is proportional to their fitness.
- Reproduction is where evolution takes place. Parts of the parent chromosomes are exchanged, mixed, in the production of child chromosomes. Additionally random mutation may introduce chromosomal patterns which are present in neither parent chromosome.

- Evolution has no memory. In producing a new fitter population for the next generation is has available to it only the chromosomal material present in the current generation of the population.

A block diagram of the basic genetic algorithm is given in Figure 4.2.

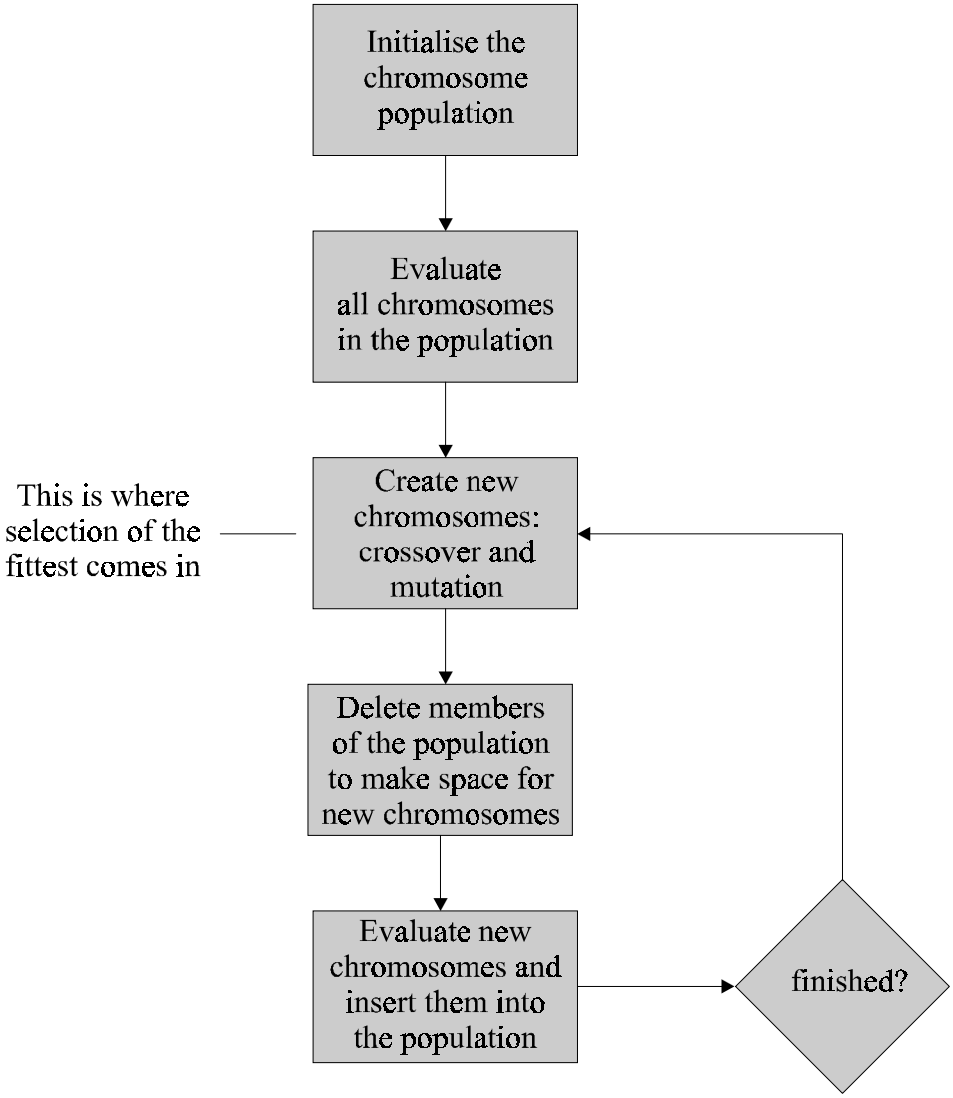

Figure 4.2: Block diagram of basic GA.

This block diagram gives a high level description of the basic genetic algorithm. In the following sections the various components of the genetic algorithm will be discussed in more detail.

4.1.1 Chromosome encoding and decoding

The method of chromosome encoding is one of the most important links between the genetic algorithm and the optimisation problem at hand. The data structure which is used to describe a solution is at least as important as the algorithm which is used to search for an optimal solution.

Basic genetic algorithms use binary bit strings as encoding technique. For example the two real numbers which represent *x* and *y* in *f6* can be encoded as a string of 64 bits (two single precision floating point numbers). Or they could be encoded into a string of 22 bits where decoding consists of splitting the 22 bit string into two 11 bit integers, subtracting and offset value from each and subsequently multiplying by a suitable scaling factor as illustrated in Figure 4.3.

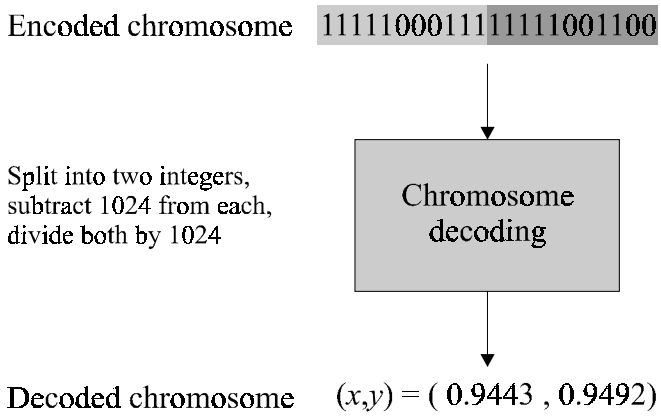

Figure 4.3: Example chromosome decoding.

As all data structures are internally represented as binary bit strings in digital computers this makes a genetic algorithm which operates on pure bit strings (without any knowledge of what these bit strings represent) a kind of universal optimisation engine. In fact this is exactly what it is. Proper bit string based genetic algorithms work on a very large variety of problems. They may not be optimally efficient, in fact they probably won't be, but they will find a good solution. They are robust and flexible.

Better, faster, results can be obtained by using encoding techniques which are closer to the problem at hand as discussed in Section 4.2.

4.1.2 Fitness and evaluation

Having defined what a solution looks like it is also important to define an index for what a good solution is. In genetic algorithms the index which defines the goodness of a solution is called fitness while the function which determines this fitness is referred to as the fitness function. A function which generates a value, from a decoded chromosome, which must be somehow manipulated before being used as a fitness value is referred to as an intermediate function.

While the nature of the fitness function for a particular problem can be quite obvious it always warrants proper attention. If, for example, the problem at hand is the optimisation of *f6* then *f6* itself (the intermediate function) can function as the fitness function. Low *f6* values indicate undesirable chromosomes while high *f6* values indicate desirable ones.

The chance which chromosomes have of being chosen for reproduction into the next generation is proportional to their fitness. So if we have two chromosomes in the *f6* problem, one with a fitness of 0.25 and one with a fitness of 0.50 then the second one will have twice the chance of being selected for reproduction that the first one has.

If the algorithm has run for a number of generations and the worst chromosome, in the population, A, evaluates to 0.8760 and the best chromosome, B, evaluates to 0.9971 then B will have only 1.14 times the chance of being selected that A has. Faster convergence may well be obtained if a higher proportional reproduction chance is somehow assigned to B.

This aspect can become even more pronounced if a trivial change is made to the (intermediate) function. For example a problem very similar to the maximisation of *f6* is the maximisation of elevated *f6* or $f6_e$:

$$f6_e(x,y) = f6(x,y) + 999 = 999.5 - \frac{\left(\sin\left(\sqrt{x^2+y^2}\right)\right)^2 - 0.5}{\left(1+0.001(x^2+y^2)\right)^2} \quad . \tag{4.2}$$

Now for $f6_e$ chromosome A gets a fitness value of 999.8760 and chromosome B has a fitness of 999.9971. The chance which B stands of being selected for reproduction is now only 1.0001 times higher than A's chance of being selected for reproduction. In fact the best possible solution has a fitness value (1000) which is only 1.001 times better than the worst fitness value (999).

There are two approaches to handling this problem. The first is known as windowing and is illustrated in Figure 4.4. The minimum intermediate function evaluation for the population is determined. The fitness is then taken to be the

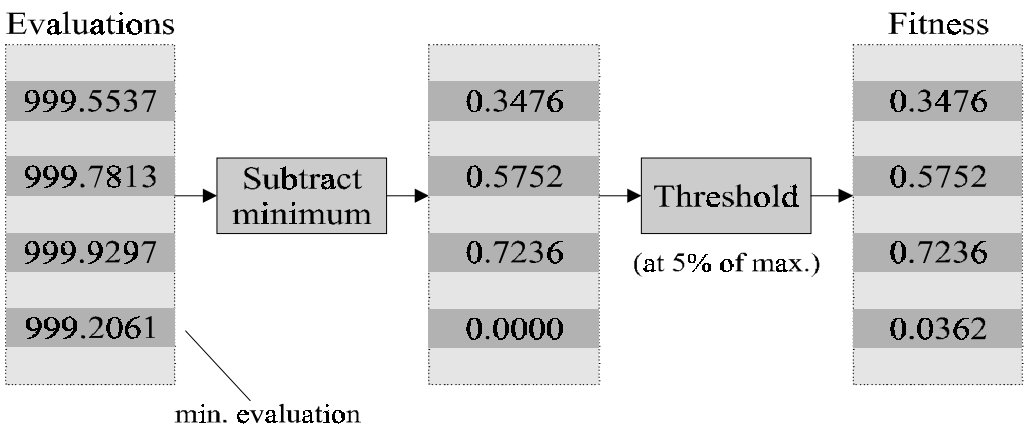

Figure 4.4: Example of fitness windowing.

difference between a chromosome's intermediate function evaluation and this minimum value. Differences which fall below a certain threshold can be assigned a default offset fitness. This is to avoid the lowest evaluating chromosomes in the population having zero or near zero chance of being selected for reproduction. A

convenient choice for this offset fitness is a set fraction of the difference between the maximum and the minimum evaluation. In the example of Figure 4.4 5% of the maximum difference (0.05x0.7236≈0.0362) is used as threshold and offset. The highest evaluating chromosome has 20 times the chance of being selected for reproduction that the lowest evaluating chromosome has.

The second technique for deriving fitness from intermediate function evaluations is referred to as linear normalisation in genetic algorithm literature [Davis91]. The chromosomes are ordered in descending order of intermediate function evaluation value and subsequently numbered in ascending order. Thus assuming the population to have N_{pop} members and i to be the numbering index then the chromosome with the largest function evaluation is numbered i=0, the largest but 1 is numbered i=1, the largest but 2 is numbered i=2 and the smallest is numbered $i=N_{pop}-1$. The fitness of a chromosome which comes i^{th} in this list is then given by:

$$fitness_i = a_{linnorm} \cdot i + b_{linnorm} \tag{4.3}$$

if we wish the chromosome with the largest plain function evaluation to have a fitness G times as great as the chromosome with the smallest plain function evaluation $a_{linnorm}$ and $b_{linnorm}$ would have to be chosen such that:

$$a_{linnorm} = \frac{b_{linnorm}(1-G)}{\mathrm{N}_{\mathrm{pop}} - 1} \quad G > 1, \quad b_{linnorm} > 0. \tag{4.4}$$

An example is given in Figure 4.5. The same population of four with the same

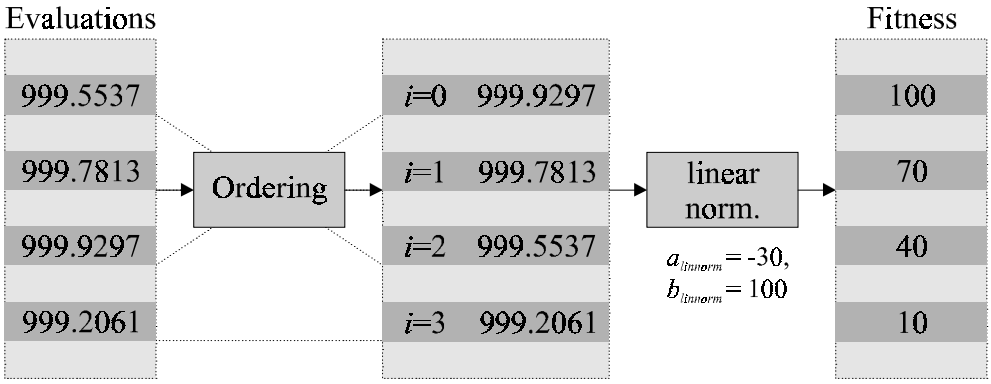

Figure 4.5: Example of linear normalisation for fitness.

intermediate function evaluations is used. G was chosen as 10 and $b_{linnorm}$ was chosen as 100 so that the fitness of the chromosome with the largest raw function evaluation becomes 100 whilst the fitness of the chromosome with the smallest raw function evaluation becomes 10.

Both windowing and linear normalisation can handle the presence of a super individual within the population. A super individual is a chromosome with an intermediate function evaluation which is much larger than the raw function evaluation of any other chromosome within the population. The thresholding parameter in windowing and the $a_{linnorm}$ and $b_{linnorm}$ parameters in linear normalisation allow the proportion between the largest and the smallest fitness value to be set. This is illustrated in Figure 4.6 where a single set of plain evaluations is converted to fitness values using both windowing and linear

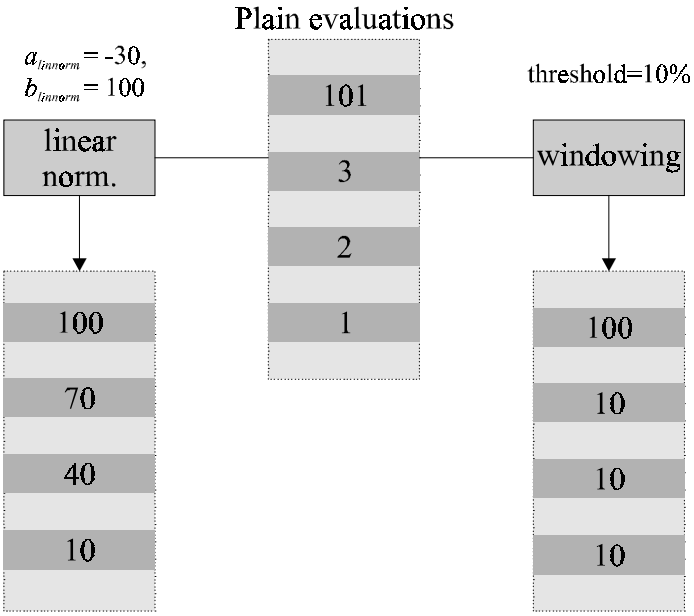

Figure 4.6: Comparison of windowing and linear normalisation.

normalisation. Both result in the fitness of the chromosome with the largest intermediate evaluation being 10 times as large as the fitness of the chromosome with the smallest intermediate evaluation even though a factor of 101 separates their intermediate function evaluations.

The advantage of linear normalisation over windowing becomes apparent when there are groups of chromosomes with very similar intermediate function values. A common situation with chromosome populations generated by genetic algorithms. Windowing assigns these chromosomes very close fitness values (in Figure 4.6 the effect is exaggerated because the thresholding make the low plain function evaluations effectively identical) while linear normalisation assigns uniformly rising fitness thus giving “better” chromosomes a higher chance at reproduction even if their raw function evaluation value differs only slightly from the plain function evaluation value of the next best chromosome.

4.1.3 Parent selection

The function of the parent selection mechanism within a genetic algorithm is to favour chromosomes with relatively high fitness values by (positively) biasing their

chance of being selected for reproduction. The most common way of handling this is the so called roulette wheel selection mechanism. This roulette wheel selection mechanism is illustrated in Figure 4.7.

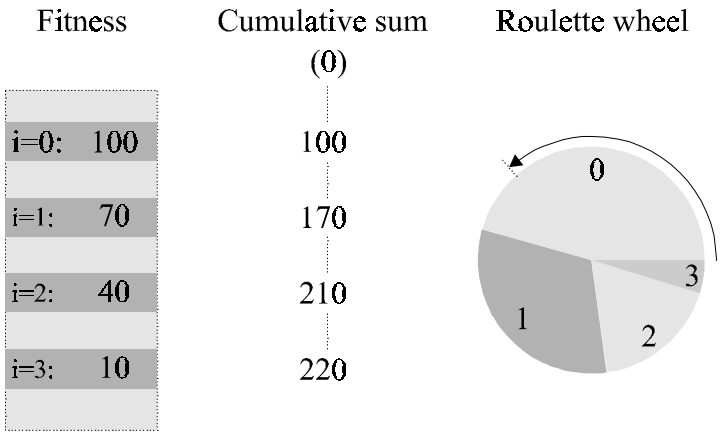

Figure 4.7: Roulette wheel selection.

For each chromosome in the population the cumulative sum of fitness values is determined. Thus if the N_{pop} chromosomes are numbered i=0..N_{pop}-1 and the fitness value of each chromosome is given by F_i then the cumulative sum CS_i for chromosome i is given by:

$$CS_i = \sum_{j=0}^{i} F_j . \tag{4.5}$$

The sum of fitnesses of all chromosomes in the population is:

$$PS = \sum_{j=0}^{i} F_j = CS_{N_{pop}-1} . \tag{4.6}$$

With all the CS values having been calculated two steps remain in picking a chromosome from the population. A random integer r has to be generated according to a uniform distribution:

$$p(r = R) = \frac{1}{PS} \quad \text{for} \quad 0 < R \leq PS \quad \text{and} \quad p(r = R) = 0 \quad \text{elsewhere.} \tag{4.7}$$

Finally the value chromosome i_c is selected by setting i_c equal to the first value of I for which $CS_i \geq r$:

$$i_c = \min\{\, i \mid CS_i \geq r \,\}. \tag{4.8}$$

Taking the example of Figure 4.7 a random number sequence of {65, 10, 63, 174, 9, 192, 35, 124, 70, 159} would result in the following i_c: {0, 0, 0, 2, 0, 2, 0, 1, 0, 1}.

The reason that this method of selection is referred to as roulette wheel selection is because the picking of a random r and subsequent determination of the appropriate i_c is the probabilistic equivalent of tossing a ball into the mathematical roulette wheel of Figure 4.7 seeing where it stops comes to rest.

4.1.4 Reproduction operators

The actual improvement of a population of chromosomes (the evolution) is driven by the reproduction operators of the genetic algorithm. In the basic genetic algorithm there are two reproduction operators: mutation and crossover.

The mutation operators takes a single chromosome as input and changes part of it. The function of this operator is to introduce (parts of) chromosomes into the population that were not there previously. These parts of chromosomes are known as schemata. For example a three bit chromosome '110' has eight possible schemata.

In the black-box genetic algorithm the mutation operator operates on chromosomes which are encoded as bit strings. The mutation operator runs down all the bits in a chromosome in sequence. Each bit is inverted with a probability p_m. The process is illustrated in Figure 4.8 where the example bit string has a length of 22 bits (it is in fact the same bit string as in Figure 4.3). A series of 22 random numbers which is uniformly distributed on the interval [0,1> is generated. For this example p_m=0.10. Those random numbers in the sequence of 22 which fall below p_m correspond to a bit change. In the instance illustrated in this example there are 4 such bit positions while on average there will off course be 2.2 bits inverted. Typical values for p_m in a black box genetic algorithm would be at least an order of magnitude lower than in the example of Figure 4.8, i.e. they would range from 0.001 to 0.01.

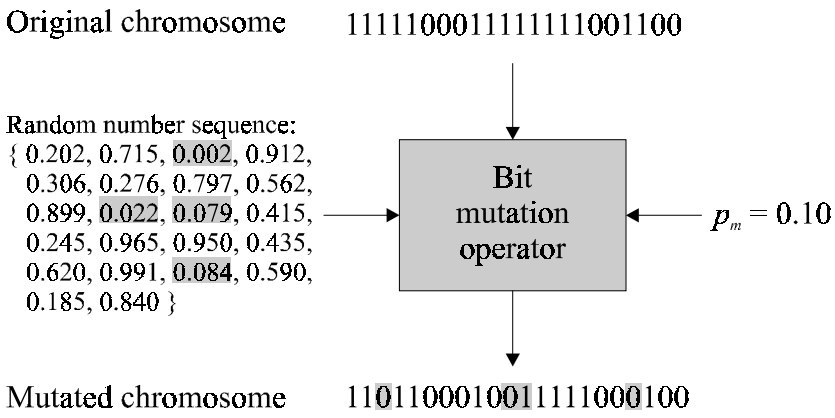

Figure 4.8: Example of bit mutation.

The second operator which is present in all genetic algorithms is the crossover operator. The crossover operator takes two chromosomes as input and produces two so called child chromosomes as output. What the crossover operator does is to take a certain amount genetic of material, according to some schema, from the two parent chromosomes and exchange it between these two parent chromosomes.

The elementary form of bit string crossover is known as one point crossover and is illustrated in Figure 4.9. A position along the chromosome bit string is randomly chosen. All bits behind this position are exchanged and all bits in front of the position are left unchanged. The result is two child chromosomes.

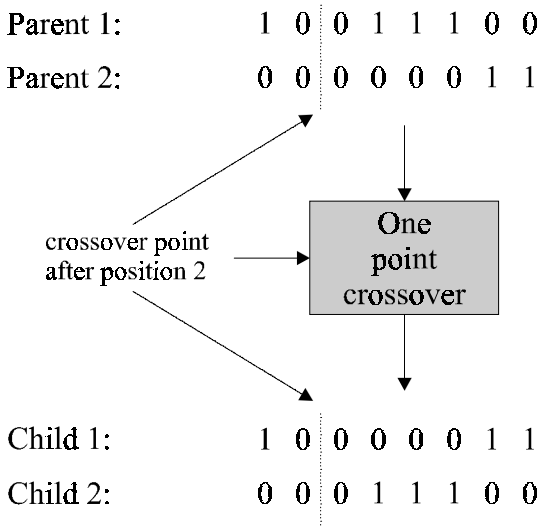

Figure 4.9: Example of one point crossover.

Note that exchanging bits after the random position is completely equivalent to exchanging material before the chosen position and leaving unchanged bits behind this position. The same child chromosomes result only in a different order.

The purpose of crossover is to propagate successful genetic material (schemata that result in high fitness) through the population much faster than (random) mutation could achieve this by itself.

Crossover operators are the most characteristic feature of genetic algorithms. There are other optimisation algorithms which operate on sets of solutions and perform some kind of mutation, for example the rearrangement function in simulated annealing [Press91] but crossover is particular to genetic algorithms in at least as much that genetic algorithms without some kind of crossover operator are not really genetic algorithms at all.

Crossover and mutation acting on a population provide a kind of intrinsic parallelism. The genetic algorithm manipulates a number of schemata in parallel. Mutation introduces diverseness (new schemata). Crossover reproduces and combines successful schemata, schemata which resulted in relatively high fitness, through the population.

In the black box genetic algorithm one point crossover and bit mutation are commonly combined into a single one point crossover and mutate operator. Two parent chromosomes are subjected to crossover and the resulting child chromosomes are first subjected to mutation before being considered for re-insertion into the population.