CHAPTER 1

GENETIC ALGORITHMS

1.1. Introduction

Numerical mathematics has been developing steadily over the last century and especially over the last 50 years when computers became readily available. Like many other sciences numerical mathematics has been getting much inspiration from nature. Advancements in our understanding of the brain, inspired numerical mathematicians to recreate some of the structures of this biological computer. These attempts resulted in Neural Networks which today have wide applications in many areas of science and engineering. Optimization algorithms, which are a sizeable subfield of numerical mathematics have also borrowed heavily from nature. Some examples are Ant Colony Optimization and Particle Swarm Optimization. The timeline in Figure 1.1 shows some of the milestones in optimization.

Biological evolution can be seen as an optimization algorithm that adjusts organisms to adapt perfectly to their environment. The idea of an algorithm which imitates the principal of natural evolution was first introduced by Holland (1962). Shortly after other groups independently started using similar approaches (e.g. Rechenberg, 1973). The entire field of evolutionary algorithms has since split up into many subfields (e.g. genetic programming, evolutionary programming, evolution strategy, etc.). Out of these subfields, the GAs are

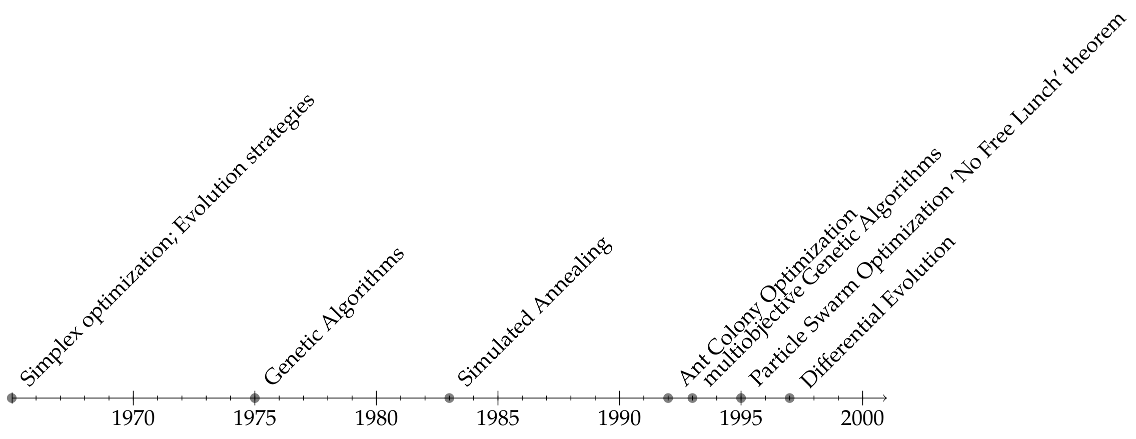

Figure 1.1 Timeline of milestones in numerical optimization

the most widely known and have been applied to many different problems.

In general, optimization algorithms have two seemingly conflicting goals: exploiting good leads (optimum seeking) while still exploring the parameter space sufficiently. Simple algorithms like Hillclimbing (randomly selecting a point in the neighbourhood of the current point and then picking the more optimal point as the new basis) will exploit good leads but will neglect to explore the search space often leading to the convergence on local optima. Random searches, on the other hand, are excellent at exploring the search space but will fail to quickly converge on an optimal solution (and are not guaranteed to converge at all). Which algorithm to use, naturally depends on the type of parameter space. Spaces with independent variables can be solved relatively simply by employing optimum seeking methods. Random search algorithms are suited to parameter spaces with highly correlated variables. One should note that the better an algorithm is optimized for a specific search space the more poorly it performs on other problems. This constitutes the so called 'No Free Lunch' theorem (Wolpert & Macready, 1997) , which also states that across the space of all problems, all algorithms perform equally well. GAs seem to strike a balance between exploration and exploiting the current best solution. Consequently they can be applied to a wide range of different parameter spaces. In addition, they have many options and fine tuning parameters so that one can adjust them to individual problems.

1.2. Genetic Algorithms

GAs are a stochastic search technique that tries to find solutions in n-dimensional search spaces. In most optimization algorithm we have a function $f(\vec{x}) = s$, where $\vec{x}$ is are the input parameters and s is a solution scalar (sometimes referred to as figure-of-merit). The goal is to find the input parameters that optimize s (finding a maximum or minimal value of s). In some optimization algorithms, however, this function $f(\vec{x})$ does not immediatley produce a scalar solution s but instead produces a solution vector $\vec{y}$ (e.g. curve fitting where input coefficients produce a number of output points). This gives two option, first we can find a function $g(\vec{y}) = s$ that maps the solution to an optimizable scalar and proceed as above, or we can try to optimize the individual components of $\vec{y}$ simultatneously. The latter option is called multi-objective optimization and is a vast field of research, but outside the scope of this work. From now on we will consider that we have a function $f(\vec{x}) = \vec{y}$ and a function $g(\vec{y}) = s$.

GAs have terms for the described functions and parameters that borrow heavily from evolutionary science. The individual input parameters of $\vec{x}$ are often referred to as genes. The input vector $\vec{x}$ is called individual or sometimes genome. We refer to representation in parameter space as genotype ($\vec{x}$) and the representation in solution space as phenotype ($\vec{y}$) of a solution. This is similar to biology where the input $\vec{x}$ can be thought of as the DNA sequence. The phenotype in biology however does not resemble a vector (strictly speaking). Mathematically speaking it is possible to have multiple genotypes map to one phenotype, however each genotype only maps to one phenotype. One should also note that in many, if not most, optimization problems there exists no phenotype as the function $f(\vec{x})$ maps directly to the optimizable scalar s. Finally, the function that results in the figure-of-merit ($f(\vec{x}) = s$ or $g(\vec{y}) = s$) is called the fitness function which results in the fitness s.

In general, GAs maintain a pool of individuals called a population or generation. The general idea is that each new generation is created out the old generation. Owing to the special processes of the GA each new generation will consist of individuals that are on average closer to the optimal solution than the last generation. Following the notation of Michalewicz (1994) we introduce the population $P(t)$ with the individuals $\{p_1^t, \dots, p_n^t\}$, where t denotes the iteration (or generation number). Each individual (p_i^t) is a data structure consisting of a vector $\vec{x}_i$ and its corresponding fitness scalar s_i. When we speak of evaluating p_i^t we mean that we use $g(f(\vec{x}_i)) = s_i$ to determine the fitness. A new population (or generation) $P(t+1)$ is formed by choosing, in the *select step*, the more fit individuals. Some or all of the new population undergo transformations in the *recombine step*. These transformations are called genetic operators. We define unary transformations, which create new individuals by small changes in single individuals called mutations. Higher order transformations called crossovers combine the traits of multiple individuals to form a next generation individual. After the new population has been created in the *recombine step*, it is evaluated (perform the computation $g(f(\vec{x}_i)) = s_i$) and the *select step* begins anew.

This procedure is repeated until some termination condition is reached (see Algorithm 1.2.1). One way is to wait until best individual in a generation or the whole generation has reached a certain threshold fitness. Another way is to set a limit on the number of generations.

Algorithm 1.2.1: Genetic Algorithm()

```
procedure Genetic Algorithm()
  $t \leftarrow 0$
  Initialize($P(t)$)
  Evaluate($P(t)$)
  while (not termination condition)
     do { $t \leftarrow t + 1$
          $P(t) \leftarrow$ Select($P(t-1)$)
          Recombine($P(t)$)
          Evaluate($P(t)$)
```

In order to apply a GA to an optimisation problem, the following are needed:

- a genetic representation of the search space (e.g. a vector)
- a function (or a chain of functions) that can calculate a fitness for a genetic representation
- transformations that create a new population/generation out of selected members of the old population/generation
- a method of creating an initial population

If the problem fulfills all these requirements one can start constructing a GA. This involves multiple steps the first of which is choosing a suitable genetic representation for each solution in the parameter space.

There are two main ways to represent a genome, binary encoding and value encoding (sometimes called gray encoding). Binary encoding was the form of encoding used in early genetic algorithms, the advantage being that the same GA can be easily adapter for many problems. In one-dimensional problems, for example, value encoding only offers one gene, whereas binary encoding, depending on the requested precision of the value, offers multiple genes. This becomes obvious in the one-dimensional minimization example: $f(\vec{x}) = (x_0 - 3.141)^2$. The solution vector that minimizes the problem in value encoding is $\vec{x} = (3.141)$ using IEEE 754 floating point encoding the optimal vector is $\vec{x} = (0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1)$. There are however many problems with binary encoding. The so called *hamming cliff* describes the problem that a simple bit-flip at one high encoding bit (ocurring in the *recombination step* using mutation or crossover) can dramatically change the encoded value (e.g. Chakraborty & Janikow, 2003). This can improve covering of search space but also can hinder the code from converging. In addition, when using binary encoding for many input variables the genomes can get incredibly long and GAs have been shown to perform poorly for very long genomes. Value encoding often is a natural way to encode the parameters of a problem. In contrast to binary encoding the genetic operators are often much more problem specific. It seems that for the moment value encoding is the preferred method in many cases (e.g. Janikow & Michalewicz, 1991; Wright, 1991; Goldberg, 1990). The 'No Free Lunch' theorem proves that there is no one optimal encoding, but the optimal encoding is different for each problem.

The fitness function maps the phenotype ($\vec{y}$) to a scalar and is one of the requirements for optimization. It is often hard to define one number that describes how optimal a solution is for any given problem. For example it is not possible to map desirable traits of a car to one number and one might have prioritize which traits to optimize at the cost of others. A sensible fitness function that maps from the multi-dimensional phenotype to a scalar is sometimes impossible to construct. As described above these cases need to be treated under multi-objective optimization schemes. Multi-objective optimizations are a vast field and outside the scope of this work.

Many GAs employ a so called fitness scaling operation in every generation to the fitness function. This scaling often helps the GA to follow promising leads but also explore the parameter space. For example, it sometimes happens, especially in early generations, that a fitness function values a small fraction of individuals extremely highly when compared to the rest of the solutions. These individuals with a near optimal value in very few genes might have a much higher fitness than an individual with a subpar value for these few genes, but is near optimal values at all other genes. The subsequent populations will then be reigned by the genes of these few individuals which prohibits the GA to explore the search space. In GA terms individuals with extremely high fitness based on very few near optimal genes are called *superindividual*s. They can drastically reduce the genetic breadth and often cause the GA to fail. One way to overcome this problem is scaling the fitness of all individuals after they have been computed. There are other steps which can be undertaken in the *select step* described later. The scaling is often a simple algebraic transformation, like linear or exponential scaling. A specific example can be found in Section **??**. In addition to fitness scaling, there exists the possibilty to just use the rank of the individuals as a fitness measure. In the case of ranking, we order the individuals according to their fitness value and assign them monotonically incrementing values (see Figure 1.3). In summary, fitness functions and scaling are a very crucial part of a successful algorithm. For a description of different scaling methods please refer to Kreinovich et al. (1993).

Superindividuals can not only be avoided with fitness scaling but also by initial population choice. The most basic quantity to consider when choosing the initial population is that of population size. Generally the population size remains the same over the course of a GA. The population size should be chosen in relation to the size and complexity of the parameter space. For example, a small population size and a large search space can lead the GA to find local optima rather than the global optimum. In this work, we have chosen a population which is roughly 15 times bigger than the number of input parameters. After having chosen the population size the most basic method of selecting the initial population is to draw individuals uniformly and randomly from the entire search space. One might however know a probability distribution for the parameter space and can draw randomly weighted by the distribution (e.g. when trying to find find parameters for a random star, we can rule a $20\,M_\odot$white dwarfs with some likelihood). An initial population that is closer to predicted optimal values will converge faster, but won't explore the parameter space that well.

Once we have evaluated the fitness for each member of the individual population the next step is the *selection step*. There are many different approaches for selecting individuals from the current generation to create the next generation. Before selecting individuals we can a make coarse selection on the entire population. One selection that is often performed is elitism in which a fraction of the fittest individuals is selected to advance unaltered to the next generation. Another possibility is to discard a fraction of the least fit individuals. The gene combinations of these individuals then won't be used in the upcoming *recombination step*. After this first coarse selection on the population the remaining members form the so called mating population.

We then start with the recombination step acting only on the current mating population. The first action in the recombination step is the selection of two or more individuals from the mating population and add them to a mating pool. A mating pool is a collection of individuals whose genes will be combined to form one or more individuals of the next generation. In all our next examples we will assume a mating-pool with only two slots (similar to two parents in biological reproduction). There is a multitude of options for selecting members from the mating population and adding them to a mating pool (**?**, for an overview see). The most widely used of the selection algorithms is the roulette wheel selection (RWS). In Figure 1.2 we see that individuals are assigned a wedge of the wheel. The area of the wedge is according to the individual's fitness. The wheel is then spun and slows down until the wheel comes to a halt. The selection chevron on the left points then to the chosen individual. In RWS fitter individuals are more likely to be chosen than individuals with poorer fitness. In Figure 1.3 we show how rank scaling of the fitnesses can alleviate the problem of early *superindividuals*. In addition to RWS there is *tournament selection* where we randomly select two individuals and compare those. The fitter of those two individuals is selected.

There are multiple steps for creating a new population from the current mating population. In the previous paragraph we described how to select individuals and place them in a mating pool. The individuals in the mating pool are also often referred to as parents. Similar to the term parent we refer to the newly created individuals as children. The reader should notice that the same individual can be in the mating pool twice (unlike with biological parents)! We create one or multiple children from this mating pool and place them in the next generation. The current mating pool is disbanded and a new one is

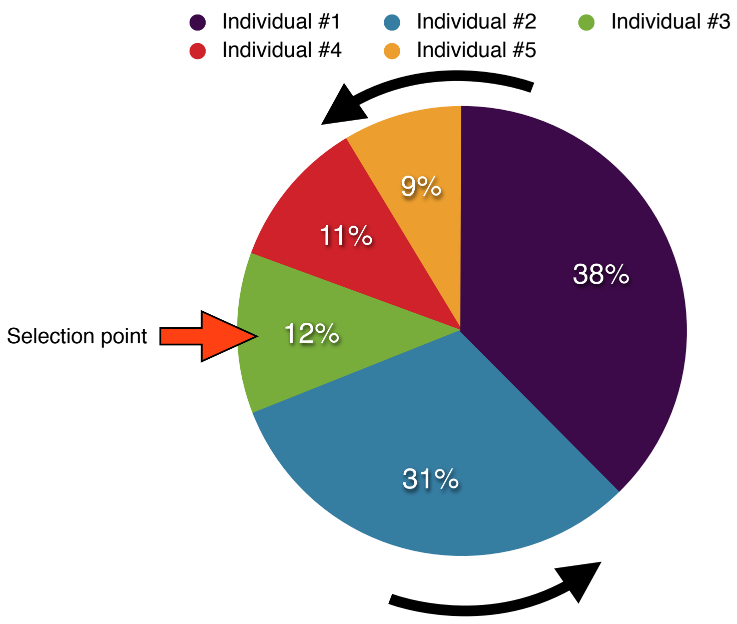

Figure 1.2 The individual fitnesses are assigned proportional fractions on the roulette wheel. The wheel is then spun and will slowly decelerate and stop at some point. Individuals with a higher fitness have a higher chance of being chosen with this method.

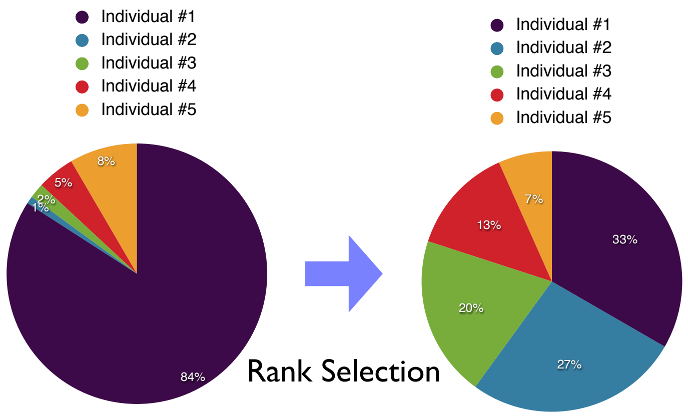

Figure 1.3 We assign new fitness values to individuals before assigning them probabilities on the roulette wheel. The fitnesses are assigned by location in an ordered list. The least fit individual gets assigned the value 1 the fittest individual the number n, where n is the population size. This is can be viewed as a special case of fitness scaling. After the new fitnesses have been chosen we use normal roulette wheel selection to select for the mating pool in the population.

formed. These steps are repeated until the new population has the same number as the old population (minus the number of individuals that advanced to the new population through elitism). There are two main processes to create a new individual from a mating pool: crossover and mutation. The simplest form of a crossover is the single-point crossover (see Figure 1.4). A random integer $r \in [1, N-1]$, where N describes the number of genes in a genome, is selected. The new individual is created out of the first r genes from the first parent and the last $N-r$ genes from the second parent (see Figure 1.4). Now it is trivial to create a second child (using the same random number r) from the mating pool by just switching first and second parent. Two-point crossover is very similar to single-point crossover. In two-point crossover Two random numbers are selected and the crossover occurs at these places. Multi-point crossover (see Figure 1.4) is essentially just an extension of two-point crossover. In addition, there is uniform crossover in the case that each gene is selected randomly with equal chance from either parent. Finally, arithmetic crossovers use a function to calculate the new gene from each of the parent genes. For a value encoding this function could be the *mean* function. For a binary encoding the function could be the *and* operator. One can mix arithmetic and Multi-point crossovers. For example we select the r genes from the first parent and create the last $N-r$ genes by taking the mean of first and second parent's genes. After the new child or children have been created through crossovers they are subjected to the mutation operator. There is a chance for each individual gene (often chosen to be less than 5%) that it is altered. For bit encoding this altering is a simple bit inversions. There are many options for value encoding. For example, one can add or multiply with a random number. Once this step is complete the children are added to the new population.

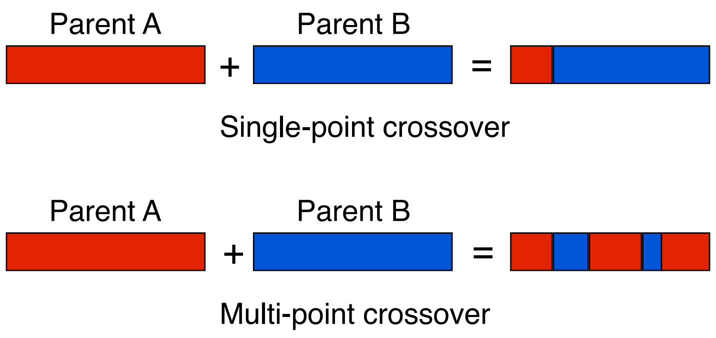

Figure 1.4 In the single crossover a random point in the Individual is chosen. Before this point the genes are taken from the first parent and after that the we use the genes from the second parent. Using the same random number it easily allows for the creation of two children by reversing the roles of first and second parent. The multi-point crossover employs multiple places in the genome where the crossover happens.

Once the new generation is created by crossovers and mutations we start the iterative process anew. First we calculate the fitness of all individuals, then scale the resulting fitness, create a mating population and select individuals with, for example, RWS to create children which forms the next generation. This iterative process is run until we have reached a certain number of generations or some individuals have reached some fitness threshold.

1.3. Convergence in Genetic Algorithms

A key problem with many GA-implementations is the premature convergence on a local optimum. The more complex the search space and the more interlinked the parameters are, the more likely it is that traditional search routines will fail. GAs are inherently better at bypassing local optima but are in no way immune to this problem. A feature that separates GAs from traditional optimization algorithms is that they will never fully converge. The algorithm will get close to the optimum but due to continued mutation of the individuals the GA will in most cases not reach an optimal value. To alleviate this problem some authors suggest switching to a different algorithm when close to the optimum solution, whereas others suggest changing the mutation rate over time (see Rudolph, 1994, and references therein).

one of the unsolved problems is determining a mathematical description for the GAs convergence. The predominate schemata theory explains only a subset of the intrinsic complexity of GAs.

1.4. Genetic Algorithm Theory

The schemata theory first described by Holland (1975) is one of the accepted theoretical interpretations of GAs. There is some criticism and it is known that this theory only explains part of the complexity that are inherent to GAs (see Whitley, 1994, and references therein). We will describe the basic concepts of schemata using an example in binary encoding (notation adapted from Goldberg, 1989). A schemata or similarity template is formed by adding an extra letter to the binary alphabet denoted by $*$. Using the ternary alphabet $0, 1, *$ we can now describe a pattern matching schemata were the $*$ symbol can be thought of as *don't care*-symbol. The schemata $0, 1, 0, *$ for example, matches both the string $0, 1, 0, 0$ and $0, 1, 0, 1$. The order of a schemata is defined as how many places are not filled by the $*$-symbol. For example, the given example is a third order schemata. Schematas provide a powerful way to describe similarities in a set of individuals. A whole population with many individuals therefore samples a range of different schematas . Essentially low-order and above-average schemata receive exponentially increasing trials in subsequent generations of a GA. Michalewicz (1994) describe the workings of a GA in the following way: "A genetic algorithm seeks near optimal performance through the juxtaposition of low-order, high-performance schemata, called building blocks". This schemata theory is the standard explanation for GAs, there are however some examples that violate the implications that stem from this schemata theory (see chapter 3 of Michalewicz, 1994, for some examples).

1.5. A Simple Example

We will illustrate the use of a GA on a simple astrophysical problem. The task at hand is to fit an observed spectrum with a synthetic spectrum. The input parameters for this synthetic spectrum are T_{eff}, $\log g$, [Fe/H], α-enhancement, $v_{\mathrm{rad}}$and v_{rot}. The simplest genetic representation of this is the vector $\vec{x} = (T_{\mathrm{eff}}, \log g, [Fe/H], \alpha, v_{\mathrm{rad}}, v_{\mathrm{rot}})$. $\vec{x}$ is the *genotype* of the individual. The resulting synthetic spectrum is the *phenotype*.

For this relatively small number of genes a population size of 75 should suffice. The first step is drawing an initial population. We will draw uniformly randomly from the search space: $T_{\mathrm{eff}} \in [2000, 9000]$, $\log g \in [0, 5]$, $[Fe/H] \in [-5, 1]$, $\alpha \in [0, 0.4]$, $v_{\mathrm{rad}} \in [-100, 100]$ and $v_{\mathrm{rot}} \in [0, 200]$. We compute the synthetic spectrum for each individual. The fitness of each individual is the inverse of the root-mean-square of the residuals between the observed and the synthetic spectrum. In the select step we will first advance 10 % of the fittest individuals to the next population unaltered (*elitism*). For the next population to be complete we need $75 - 8 = 67$ individuals which are created through mating. We select two individuals through RWS and place them in the mating pool. A single crossover point is randomly selected and the child is created. Before being placed in the new population the mutation operator is applied but has a very small chance to mutate any of the child's genes (in this case we choose 2%). This mating step (see Figure **??**) is repeated 67 times. The new population now consists of the 8 fittest individuals of the old population and 67 new individuals created by mating. We will then start again to compute the synthetic spectrum and the resulting fitness for each individual of the new population. This loop is continued until one individual or a whole population has reached a predefined convergence criterium.

1.6. Conclusion

GAs with their inherently parallel nature are very useful in the increasingly parallel computing. They are relatively easy to implement and can be used for a variety of problems. In many applications GA quickly deliver impressive results, but require a lot of time for fine-tuning to be able to sufficiently handle the problem. GAs also offer a plethora for each of the individual steps (encoding type, selection type, etc.), which can confuse users that are not very familiar with the consequence for each choice. To alleviate this problem in our work we have sought the help of experts in GAs and hope to find the best set of options for our GA. In summary, although there are some drawbacks (no guaranteed convergence, many choices for implementation, etc.) we find that we progressed much quicker with GAs than with other algorithms.

BIBLIOGRAPHY

Chakraborty, U. K., & Janikow, C. Z. 2003, Information Sciences, 156, 253 , evolutionary Computation (Link)

Goldberg, D. E. 1989, Genetic Algorithms in Search, Optimization, and Machine Learning, 1st edn. (Addison-Wesley Professional) (Link)

—. 1990, Complex Systems, 5, 139

Holland, J. H. 1962, J. ACM, 9, 297 (Link)

—. 1975, Adaptation in Natural and Artificial Systems (Ann Arbor, MI, USA: University of Michigan Press)

Janikow, C. Z., & Michalewicz, Z. 1991, in Proc. of the 4th International Conference on Genetic Algorithms, ed. R. K. Belew & L. B. Booker (Morgan Kaufmann), 151–157

Kreinovich, V., Quintana, C., & Fuentes, O. 1993, Cybernetics and Systems: an International Journal, 24, 9

Michalewicz, Z. 1994, Genetic algorithms + data structures = evolution programs (2nd, extended ed.) (New York, NY, USA: Springer-Verlag New York, Inc.)

Rechenberg, I. 1973, Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Problemata No. 15 (Stuttgart-Bad Cannstatt: Frommann-Holzboog)

Rudolph, G. 1994, Neural Networks, IEEE Transactions on, 5, 96 (Link)

Whitley, D. 1994, Statistics and Computing, 4, 65

Wolpert, D., & Macready, W. G. 1997, IEEE Trans. Evolutionary Computation, 1, 67

Wright, A. H. 1991, in Foundations of Genetic Algorithms (Morgan Kaufmann), 205–218