Chapter 2
Genetic Algorithms

2.1 Introduction

GENETIC ALGORITHMS are heuristic search approaches that are applicable to a wide range of optimization problems. This flexibility makes them attractive for many optimization problems in practice. Evolution is the basis of GENETIC ALGORITHMS. The current variety and success of species is a good reason for believing in the power of evolution. Species are able to adapt to their environment. They have developed to complex structures that allow the survival in different kinds of environments. Mating and getting offspring to evolve belong to the main principles of the success of evolution. These are good reasons for adapting evolutionary principles to solving optimization problems.

In this chapter we will introduce the foundations of GENETIC ALGORITHMS. Starting with an introduction to the basic GENETIC ALGORITHM with populations, we will introduce the most important genetic operators step by step, which are crossover, mutation, and selection. Further, we will discuss genotype-phenotype mapping, common termination conditions, and give a short excursus to experimental analysis.

2.2 Basic Genetic Algorithm

The classic GENETIC ALGORITHM is based on a set of candidate solutions that represent a solution to the optimization problem we want to solve. A solution is a potential candidate for an optimum of the optimization problem. Its representation plays an important role, as the representation determines the choice of the genetic operators. Representations are usually lists of values and are more generally based on sets of symbols. If they are continuous, they are called vectors, if they consist of bits, they are called bit strings. In case of combinatorial problems the solutions often consist of symbols that appear in a list. An example is the representation of a tour in case of the traveling salesman problem. Genetic operators produce new solutions in the chosen representation and allow the walk in solution space. The coding of the solution as representation, which is subject to the evolutionary process, is called genotype or chromosome.

Algorithm 1 shows the pseudocode of the basic GENETIC ALGORITHM, which can serve as blueprint for many related approaches. At the beginning, a set of solutions, which is denoted as population, is initialized. This initialization is recommended to randomly cover the whole solution space or to model and incorporate expert knowledge. The representation determines the initialization process. For bit string representations a random combination of zeros and ones is reasonable, for example the initial random chromosome 1001001001 as a typical bit string of length 10. The main generational loop of the GENETIC ALGORITHM generates new offspring candidate solutions with crossover and mutation until the population is complete.

Algorithm 1 Basic GENETIC ALGORITHM

```
1: initialize population
2: repeat
3:    repeat
4:       crossover
5:       mutation
6:       phenotype mapping
7:       fitness computation
8:    until population complete
9:    selection of parental population
10: until termination condition
```

2.3 Crossover

Crossover is an operator that allows the combination of the genetic material of two or more solutions [97]. In nature most species have two parents. Some exceptions do not know different sexes and therefore only have one parent. In GENETIC ALGORITHMS we can even extend the crossover operators to more than two parents. The first step in nature is the selection of a potential mate partner. Many species spend a lot of resources on selection processes, but also on the choice of a potential partner and on strategies to attract partners. In particular, males spend many resources on impressing females. After the selection of a partner, pairing is the next natural step. From a biological perspective, two partners of the same species combine their genetic material and inherit it to their offspring.

Crossover operators in GENETIC ALGORITHMS implement a mechanism that mixes the genetic material of the parents. A famous one for bit string representation is n-point crossover. It splits up two solution at n positions and alternately assembles them to a new one (Fig. 2.1). For example, if 0010110010 is the first parent and 1111010111 is the second one, one-point crossover would randomly choose a position, let us assume 4, and generate the two offspring candidate solutions 0010-

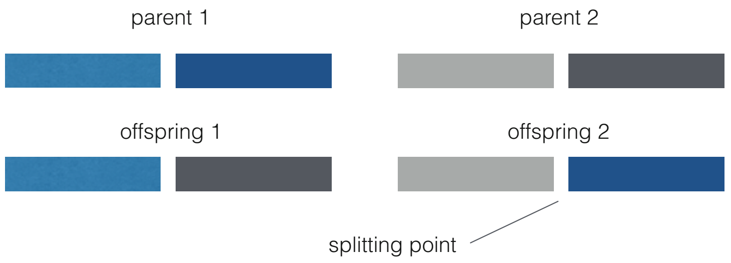

Fig. 2.1 Illustration of one-point crossover that splits up the genome of two solutions at an arbitrary point (here in the *middle*) and re-assembles them to get two novel solutions

010111 and 1111-110010. The motivation for such an operator is that both strings might represent successful parts of solutions that when combined even outperform their parents. This operator can easily be extended to more points, where the solutions are split up and reassembled alternately.

For continuous representations, the crossover operators are oriented to numerical operations. Arithmetic crossover, also known as intermediate crossover, computes the arithmetic mean of all parental solutions component-wise. For example, for the two parents $(1, 4, 2)$ and $(3, 2, 3)$ the offspring solution is $(2, 3, 2.5)$. This crossover operator can be extended to more than two parents. Dominant crossover successively chooses each component from one of the parental solutions. Uniform crossover by Syswerda [97] uses a fix mixing ratio like 0.5 to randomly choose a bit from either of the parents. The question comes up, which of the parental solutions take part in the generation of new solutions. Many GENETIC ALGORITHMS simplify this step and randomly choose the parents for the crossover operation with uniform distribution.

2.4 Mutation

The second protagonist in GENETIC ALGORITHMS is mutation. Mutation operators change a solution by disturbing them. Mutation is based on random changes. The strength of this disturbance is called mutation rate. In continuous solution spaces the mutation rate is also known as step size.

There are three main requirements for mutation operators. The first condition is reachability. Each point in solution space must be reachable from an arbitrary point in solution space. An example that may complicate the fulfillment of this condition is the existence of constraints that shrink the whole solution space to a feasible subset. There must be a minimum chance to reach every part of the solution space. Otherwise, the chance is not positive that the optimum can be found. Not every mutation operator can guarantee this condition, for example decoder approaches have difficulties covering the whole solution space.

The second good design principle of mutation operators is unbiasedness. The mutation operator should not induce a drift of the search to a particular direction, at least in unconstrained solution spaces without plateaus. In case of constrained solution spaces bias can be advantageous, which has been shown in [50, 62]. Also the idea of novelty search that tries to search in parts of the solution space that are unexplored yet, see Chap. 4, induces a bias on the mutation operator.

The third design principle for mutation operators is scalability. Each mutation operator should offer the degree of freedom that its strength is adaptable. This is usually possible for mutation operators that are based on a probability distribution. For example, for the Gaussian mutation that is based on the Gaussian distribution the standard deviation can scale the randomly drawn samples in the whole solution space. The implementation of the mutation operators depends on the employed representation. For bit strings bit flip mutation is usually used. Bit flip mutation flips a zero bit to a one bit and vice versa with a defined probability, which plays the role of the mutation rate. It is usually chosen according to the length of the representation. If N is the length of the bit string, each bit is flipped with mutation rate $1/N$. In Chap. 7 we will present a runtime analysis that is based on bit flip mutation. If the representation is a list or string of arbitrary elements, mutation randomly chooses a replacement for each element. This mutation operator is known as random resetting. Let $[5, 7, -3, 2]$ be the chromosome with integer values that come from the interval $[-10, 10]$, then random resetting decides for each component, if it is replaced. If the component is replaced, it randomly chooses a new value from the interval. For example, the result can be $[8, -2, -5, 6]$.

For continuous representations, Gaussian mutation is the most popular operator. Most processes in nature follow a Gaussian distribution, see Fig. 2.2. This is a reasonable assumption for the distribution of successful solutions.

A vector of Gaussian noise is added to a continuous solution vector [5]. If $\mathbf{x}$ is the offspring solution that has been generated with crossover,

$$\mathbf{x}' = \mathbf{x} + \sigma \cdot \mathcal{N}(0, 1) \tag{2.1}$$

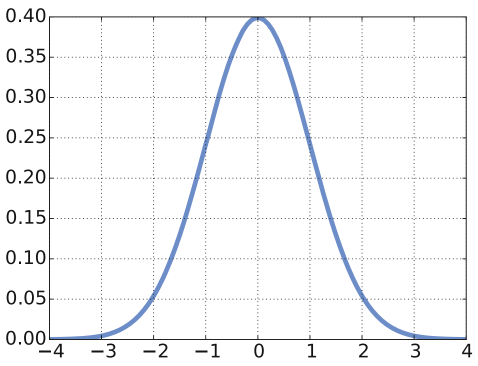

Fig. 2.2 The Gaussian distribution is basis of the Gaussian mutation operator adding noise to each component of the chromosome

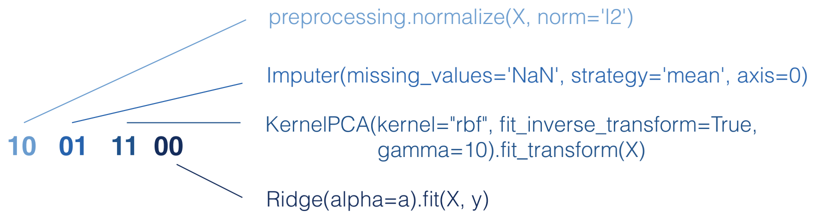
10 01 11 00
preprocessing.normalize(X, norm='l2')
Imputer(missing_values='NaN', strategy='mean', axis=0)
KernelPCA(kernel="rbf", fit_inverse_transform=True, gamma=10).fit_transform(X)
Ridge(alpha=a).fit(X, y)

Fig. 2.3 Example of genotype-phenotype mapping for a machine learning pipeline. The bit string encodes a pipeline of normalization, imputation, dimensionality reduction, and regression

is the Gaussian mutation with $\mathcal{N}(0, 1)$ as notation for a vector of Gaussian-based noise. Variable σ is the mutation rate that scales the strengths of the noise added. The Gaussian distribution is maximal at the origin. Hence, with the highest probability the solution is not changed or only slightly. The Gaussian mutation is an excellent example for a mutation operator that fulfills all mentioned conditions. With σ it is arbitrarily scalable. Moreover, with a scalable σ, all regions in continuous solution spaces will be reachable. Due to the symmetry of the Gaussian distribution, it does not prefer any direction and is hence driftless.

2.5 Genotype-Phenotype Mapping

After crossover and mutation, the new offspring population has to be evaluated. Each candidate solution has to be evaluated with regard to its ability to solve the optimization problem. Depending on the representation a mapping of the chromosome, the genotype, to the actual solution, which is denoted as phenotype, is necessary. This genotype-phenotype mapping should avoid introducing a bias. For example, a biased mapping could map the majority of the genotype space to a small set of phenotypes. The genotype-phenotype mapping is not always required. For example, in continuous optimization, the genotype is the solution itself. But many other evolutionary modeling processes require this mapping.

An example is the evolution of machine learning pipelines. Each step in a machine learning pipeline can be coded as binary part in a genome, see Fig. 2.3 for a mapping from a bit string to `Python` commands from `sklearn`. For example, 10 at the beginning of the bit string causes a normalization preprocessing step while 00 at the end calls ridge regression. Such a mapping from genotypes to phenotypes is an essential part of the GENETIC ALGORITHM modeling process.

2.6 Fitness

In the fitness computation step the phenotype of a solution is evaluated on a fitness function. The fitness function measures the quality of the solutions the GENETIC ALGORITHM has generated. The design of the fitness function is part of the modeling process of the whole optimization approach. The practitioner can have an influence on design choices of the fitness function and thus guide the search. For example, the fitness of infeasible solutions can be deteriorated like in the case of penalty functions, see Chap. 5. In case of multiple objectives that have to be optimized at the same time, the fitness function values of each single objective can be aggregated, for example by computing the weighted sum. This technique and further strategies to handle multiple objective functions at the same time are discussed in Chap. 6. An important aspect is a fair evaluation of the quality of a solution. It sounds simple to postulate that a worse solution should employ a worse fitness function value, but a closer look is often necessary. Should a solution that is very close to the global optimum, but constrained have a worse fitness value than a bad solution that is feasible? And should a solution that is close to the optimum of the first objective in multi-objective optimization, but far away from the optimum of a second objective, which is much less important, get a worse fitness function value than a solution that is less close to the first optimum but much closer to the second one? To summarize, the choice of the penalty for infeasible solutions and the choice of appropriate weights in multi-objective optimization are important design objectives.

Most approaches aim at minimizing the number of fitness function calls. The performance of a GENETIC ALGORITHM in solving a problem is usually measured in terms of the number of required fitness function evaluations until the optimum is found or approximated with a desired accuracy. Minimizing the number of fitness function calls is very important, if a call is expensive, for example, if a construction element has to be generated for each evaluation. Fitness function calls may also require a long time, for example, if a simulation model has to be run to evaluate the parameters generated with the GENETIC ALGORITHM. The machine learning pipeline that is evolved with a GENETIC ALGORITHM is a good example for a comparatively long fitness evaluation. For each evaluation the machine learning pipeline has to be trained on the data set. To avoid overfitting it is required to repeat the training multiple times with cross-validation, which additionally takes time. Finally, the accuracy of the prediction model has to be evaluated on a test set in order to get a precision score that can be used as fitness function value.

2.7 Selection

To allow convergence towards optimal solutions, the best offspring solutions have to be selected to be parents in the new parental population. A surplus of offspring solutions is generated and the best are selected to achieve a progress towards the optimum. This selection process is based on the fitness values in the population. In case of minimization problems low fitness values are preferred and vice versa in case of maximization problems. Minimization problems can easily be transformed into maximization problems with negation. Of course, this also works for transforming maximization problems into minimization problems.

Elitist selection operators select the best solutions of the offspring solutions as parents. Comma selection selects the μ best solutions from λ offspring solutions.

Plus selection selects the μ best solutions from λ offspring solutions and the μ old parents that led to their creation.

Many selection algorithms are based on randomness. Roulette wheel also known as fitness proportional selection selects parental solutions randomly with uniform distribution. The probability for being selected depends on the fitness of a solution. For this sake, the relative fitness of solutions normalized with the sum of all fitness values in a population, usually by division. This fraction of fitness can be understood as probability for a solution of being selected. The advantage of fitness-proportional selection operators is that each solution has a positive probability of being selected.

In case of comma selection good parents can be forgot. Also the randomness of fitness proportional selection allows forgetting of the best solutions. Although this might sound contra-productive for the optimization process at first, forgetting may be a reasonable strategy to overcome local optima. Another famous selection operator is tournament selection, where a set of solutions is selected randomly and within this competition subset, the best solutions are finally selected as new parents. The second step can be implemented with fitness proportional selection as typical example. Tournament selection offers a positive probability for each solution to survive, even if it has worse fitness values than other solutions.

When using selection as mechanism to choose the parents of the new generation, it is called survival selection. The selection operator determines, which solutions survive and which solutions die. This perspective directly implements Darwin's principle of survival of the fittest. But the introduced selection operators can also be employed for mating selection that is part of the crossover operators. Mating selection is a strategy to decide, which parents take part in the crossover process. It makes sense to consider other criteria for mating selection than for survival selection.

2.8 Termination

The termination condition defines, when the main evolutionary loop terminates. Often, the GENETIC ALGORITHM runs for a predefined number of generations. This can be reasonable in various experimental settings. Time and cost of fitness function evaluations may restrict the length of the optimization process. A further useful termination condition is convergence of the optimization process. When approximating the optimum, the progress of fitness function improvements may decrease significantly. If no significant process is observed, the evolutionary process stops. For example, when approximating the optima of continuous optimization problems, the definition of stagnation as repeated fitness difference lower than 10^{-8} in multiple successive generations is reasonable. Of course, stagnation can only mean that the search might have got stuck in local optima, hence missing the global one. Restart strategies, see Chap. 4, are approaches that avoid getting stuck in the same local optima. If the GENETIC ALGORITHM repeatedly approximates the same area in solution space although starting from different areas, the chance is high that the local optimum is a large attractor, and a better local optimum is unlikely to find. It can also be that this local optimum is the global one.

2.9 Experiments

The experiment has been the main analytic tool since the beginning of GENETIC ALGORITHM research. Hence, carefully conducted experiments have an important part to play. The first task before the experimental analysis is the formulation of a research question. As GENETIC ALGORITHM experiments have a stochastic outcome, a temptation of some researchers might be to bias the results by selecting only the best runs. However, a fair comparison shows all experiments, although one might feel that at least one run was bad luck. It may not have reached the optimum and thus may be disturbing the presentation of the average runs. To be statistically sound, at least 25 repetitions are usually required, 50 or 100 is also a recommendable choice. More runs are often not necessary, more than 1000 repetitions can already be bad as unlikely outliers might occur. In the extreme case of outstandingly expensive optimization runs 15, 10, or even 5 runs can be a necessary compromise.

Table 2.1 shows the experimental comparison between two GENETIC ALGORITHMS, a normal $(1+1)$-GENETIC ALGORITHM and a $(1+1)$-GENETIC ALGORITHM with nearest neighbor meta-model (MM-GA) on the two continuous benchmark functions Sphere and Rosenbrock [89]. The concept of a GENETIC ALGORITHM with meta-model, also called fitness function surrogate, will be introduced in Chap. 8. The experiments have been repeated 25 times. The results show the means and the standard deviations of all runs in terms of fitness function values after 5000 iterations. Iterations correspond to fitness function evaluations in case of a $(1+1)$-GENETIC ALGORITHM.

The results show that the GENETIC ALGORITHM with meta-model achieves lower fitness function values. The optimum of both benchmark functions lies at the origin with a fitness value of 0.0. The question comes up, if the algorithm that achieves a better fitness function value in average is really better in practice. Is the result resilient from a statistical perspective? An answer to this question is only possible,

Table 2.1 Experimental comparison of two GENETIC ALGORITHMS, one with, the other without fitness function meta-model on the Sphere function and on Rosenbrock, from [55]

Problem		(1+1)-GA		MM-GA		Wilcoxon
	d	Mean	Dev	Mean	Dev	p-value
Sphere	2	2.067e-173	0.0	2.003e-287	0.0	0.0076
	10	1.039e-53	1.800e-53	1.511e-62	2.618e-62	0.0076
Rosenbrock	2	0.260	0.447	8.091e-06	7.809e-06	0.0076
	10	0.519	0.301	2.143	2.783	0.313

if we perform a statistical test. It tells us, if the comparison between two algorithms is statistically valid.

The question for an appropriate statistical test is comparatively easy to answer. The famous student T-test is not applicable, since the outcome of GENETIC ALGORITHM experiments is not Gaussian distributed. But the Gaussian distribution is a necessary prerequisite of the T-test, which examines, if two sets of observations come from the same distribution. An appropriate test for GENETIC ALGORITHMS is the Wilcoxon rank sum test [104]. It does not make assumptions on the distributions of the data. Instead, the Wilcoxon test sorts the outcomes of both sets of experimental observations and performs an analysis solely based on the ranks of this sorting. A small Wilcoxon value of under 0.05 proofs statistical relevance. Coming back to Table 2.1, the results show that the GENETIC ALGORITHM with meta-model and a superior fitness is performing significantly better than its competitor, the simple GENETIC ALGORITHM.

2.10 Summary

GENETIC ALGORITHMS are successful optimization approaches that allow optimization in difficult solution spaces. In particular, if no derivatives are available and if the fitness landscape suffers from ill-conditioned parts, GENETIC ALGORITHMS are reasonable problem solvers. In this chapter we summarized the foundations of GENETIC ALGORITHMS. They are based on populations of solutions that approximate optima in the course of iterations. Genetic operators change the solutions. Crossover operators combine the genomes of two or more solutions. Mutation adds randomness to solutions and should be scalable, drift-less, and reach each location in solution space. The genotype or chromosome of a solution is mapped to a phenotype, the real solution, before it can be evaluated on the fitness function. The latter has to be carefully designed as it has a crucial impact on the search direction. Selection chooses the best solutions in a population for survival. These solutions are the parents of the following generation. With the introduced concepts at hand, we are already able to implement simple GENETIC ALGORITHMS. The next chapters will introduce useful mechanisms and extensions for GENETIC ALGORITHMS, which tweak their performance and make them applicable to a broad spectrum of problems.