Leonardo Vanneschi
NOVA Information Management School (NOVA IMS)
Universidade Nova de Lisboa
E-mail: lvanneschi@novaims.unl.pt

Computational Intelligence for Optimization

Document extracted from the book:
"Lectures in Intelligent Systems"
To appear

Springer

Contents

Chapter 1
Introduction

Computer science is the science that deals with the treatment of information by means of automatic procedures. It has multiple objectives, among which studying computation at a logical level and the several possible practical techniques for its implementation and application in automated electronic systems, the computers. One of the focal points of computer science is *problem solving*, i.e. the act of defining a problem, and developing and implementing a solution. In this context, a *problem* can be intended as a task that has to be fulfilled automatically, and solving a problem typically implies the design and development of an *algorithm*. An algorithm is a process, or method, that solves a problem by means of a finite set of well-defined and non-ambiguous actions, each of which can be mechanically executed in a finite amount of time. At the bottom of the process stands implementation, that consists in the development of a computer *program*, i.e. the encoding of an algorithm in a programming language, that can be directly compiled and/or executed on a computer. A problem can typically be expressed by means of a set of data, called *inputs*, that the algorithm, and consequently the program that implements it, elaborate. The execution of the program on a computer finally produces the solution to the problem, that can consists in the value of a variable or set of variables, or any other type of information we were looking for, called *output*.

Example 1.1. Arguably, one of the simplest problems that can be solved using a computer is the following one:

Find the maximum of a finite sequence of numbers.

An algorithm to solve this problem can be described by the following actions:

1. Define a numeric variable called *max*;
2. Assign to *max* the value of the first number in the sequence;
3. For each value x in the sequence:

   3.1. if x is larger than the value currently contained in variable *max*, then change the value of *max*, so that it becomes equal to x;

4. return the value of variable *max*;

Such an algorithm can be executed manually for small sequences, but it clearly needs automatic computation for large sequences. In order to execute the algorithm automatically on a computer, it needs to be encoded in a programming language. For instance, a possible program that implements the previous algorithm can be represented by the following Java method:

```
public static int findMax(int[] inputSequence) {
   int max = inputSequence[0];
   for (int i = 1; i < inputSequence.length; i++) {
      if (inputSequence[i] > max)
         max = inputSequence[i];
   }
   return max;
}
```

In this example, the input (i.e. the data that characterize the problem) is the given sequence of numbers and the output (i.e. the sought for result) is the a value, the maximum of the sequence. Encoding an algorithm into a computer program typically implies a number of implementative choices, like for instance, in this case, the data types of the numeric values contained in the sequence (integer numbers in the implementation given above), the way of representing the sequence of numbers (an array of integer values, in this case), and the way of receiving inputs and returning outputs (parameter passage of Java methods and `return` Java statement, respectively, in this example).

From this simple example, it should be clear that, in the context of Computer Science, a problem is generally not dependent on the particular data, or instance, used, but it is instead a much more general concept, that actually *abstracts* from the particular instance data used. The above problem could, in fact, be reformulated as "Develop a method to find the maximum of *any* finite sequence of numbers", and this is exactly what the reported algorithm and program do. In particular, one should remark that the Java method `findMax` can be used for finding the maximum of *any* array of integer values, independently of its length (provided that it is not longer than the longest sequence that can fit in the memory of the computer we are using) and of the particular values it contains.

Furthermore, it should not be hard to convince oneself that, apart from some implementative choices like the ones discussed above, coding an algorithm into a computer program is broadly a mechanic task. In fact, each particular action, or group of actions, contained in an algorithm has an immediate correspondance in one or more instructions of a programming language. On the other hand, developing an algorithm able to solve a problem is, typically, a very *creative* task, for which no formal rules exist in general. In *traditional computer science*, the design of an algorithm, that actually corresponds to the development of the "strategy" to solve the problem, is completely *handmade* and demanded to human beings.

1.1 Motivation for Intelligent Systems

Intelligent systems can be broadly defined as systems that have the ability of solving problems autonomously, i.e. with reduced, or even no intervention from humans. The motivation for the need of intelligent systems is straightforward: the traditional computational method described previously, where the design of an algorithm is demanded to a human being, in many cases, *fails*! This can happen, accordingly to the cases, because it is extremely hard, or even impossible, for a human being to conceive an algorithm for a problem, or because the algorithms that we are able to design for particular problems are so inefficient that it is impossible to execute their implementations in a reasonable amount of time, even on the most powerful existing computers. As a first, simple, example, one may think, for instance, of the following problem: assume that you are the owner of a store, and a person enters your store. The problem consists in giving a reliable answer to the following question:

Will that person be a good client?

Or, in other words: will that person buy enough products sold in your store, so that you will achieve a business? Remembering that an algorithm is a precise sequence of formal actions that allow us to solve a general problem (in other words, the algorithm should potentially work for *any* person entering the store), we challenge the reader to design an algorithm for this problem. We believe that we can state, with a high degree of confidence, that such a design is impossible for a human being.

Let us now consider another example: given a set of photographs picturing human faces, answer the following question:

Who is the person that is pictured in this photo?

In this case, a human being may possibly imagine an algorithm, mainly based on somatic characteristics of the face, like the color of the hair, the color of the eyes, the distance between the eyes, the position, shape and size of nose and mouth, etc. However, it is clear that it is possible to present to a human being two photos picturing the faces of two different persons with absolutely identical somatic characteristics, and, even in those conditions, the human brain is able to distinguish between the two pictured persons. No handmade algorithm will probably ever have the discernment ability that is typical of the human brain, and so none of them will probably ever solve this problem is a satisfactory way. Indeed, practically the totality of the systems of face recognition existing nowadays heavily rely on intelligent systems, instead of traditional computational methods.

Let us consider a third example. The problem consists in answering the following question:

What is the right mix of molecules that can be used to create a new drug?

For instance, we may be interested in creating a new medicine with some particular characteristics, or able to cure a new disease. In this cases, it is possible, and arguably even “easy”, for a human being, to conceive an algorithm. For instance, the algorithm may consist in trying, one by one, all the possible combinations of

molecules that exist on Earth and, for each one of them, test them, to see if they obtain the sought for effect. This algorithm can, in principle, perfectly solve the problem, but it has a major flaw: the number of molecules existing in one cubic centimeter is estimated as being approximately equal to 2.7×10^{19}. Even without dwelling with the calculations to estimate the number of the possible combinations of existing molecules, it should not be hard to convince oneself that testing all these combinations in acceptable time is purely utopian.

As a fourth example, one may for instance imagine the following problem:

> *Write the program able to drive a robot in a 3D space, so that the robot is able to efficiently accomplish complex tasks, independently from the characteristics of the particular space*

To fix ideas, one may, for instace, consider the practical case of self-driving vehicles. Also in this case, it is not hard to convince oneself that the number of possible situations which the robot may have to deal with is so high that any handcrafted algorithm is probably destined to fail. In fact, modern driving vehicles are heavily based on intelligent systems.

The previous list of examples could possibly continue indefinitely. For all these problems, either it is impossible for a human being to conceive an algorithm, or the algorithms that can be developed are not executable in practice. As a conclusion, generally speaking, none of these problems can be solved using the traditional computational method. At the same time, with the proliferation of data to which we are assisting nowadays, those types of problems are becoming more and more numerous and, thanks to the improvements of technology, we are every day more and more interested in solving those problems.

So, what to do? The idea is to design computational systems that have the ability to autonomously *learn* to solve problems. This is the objective of the area of study, partcularly popular and active nowadays, called *Machine Learning*. The idea of building machines able to learn is not new, and can be broadly dated back to the studies of Samuel in the late 50s [Samuel, 1959], but it has been renovated and deepened by Mitchell in the 90s [Mitchell, 1997]. According to Mitchell, Machine Learning is:

> *"the study of algorithms able to automatically improve by means of experience"*

This definition is significant for at least two reasons: first of all, Mitchell puts the accent on algorithms, actually raising the bar on the objective of overcoming the limitations of traditional computing, that entirely demands the design of algorithms to humans. Secondly, Mitchell defines the concept of learning as improvement by means of experience. In this way, Mitchell gives a clear idea of the general functioning of a learning computational system: it has to be based on an iterative process in which, at each iteration, either a new entire solution, or a piece of an existing solution is proposed, based on the experience gathered in the previous iterations. Focal on all this process is, once again, the idea that solutions to complex problems should be built with very limited, or even no human intervention. The only task of humans

now is to give the system a very precise description of the problem, and no hint on how to solve it, while the solution should be completely generated, in an automatic way, by the computational system.

1.2 Intelligent Systems and Bio-Inspired Algorithms

Many techniques that can be considered "intelligent" exist nowadays, and a significant part of those techniques draw inspiration by the idea that learning is a characteristic that is typical of living beings. So, those techniques are inspired by the biology of some living beings, with the objetive of, in some way, capturing and inheriting their ability to learn. When this happens, we talk about *bio-inspired* intelligent systems, or simply bio-inspired algorithms. The most known of them are:

- *Artificial Neural Networks*, that are inspired by the structure of the human brain, and try to simulate, although in an elementary way, its functioning.
- *Evolutionary Algorithms*, that are inspired by the Theory of Evolution of Darwin [Darwin, 1859], and have the objective of evolving solutions to complex problems using processes that, although simplified, are similar to the ones that have carried on the evolution of the species.
- *Swarm Intelligence*, that draws inspiration from the collective intelligence coming from the colaboration of animals in groups, like for instance ant colonies, bird flocking, hawks hunting, animal herding, bacterial growth, fish schooling and microbial intelligence.
- *Fuzzy Systems*, that simulate the ability of human beings of dealing with concepts as uncertainty or imprecise information, an ability that is crucial for learning general behaviors.
- *Local Search Systems*, whose biological inspiration can arguably be imagined as coming from their ability to simulate the sense of orientation of living beings.

All these approaches share one important characteristic: they have the ability of returning an approximated, or imprecise, solution to problems whenever they are not able to find a perfect one. This characteristic is typical of the set of techniques that are identified with the name *Soft Computing* [Tettamanzi and Tomassini, 2001]. This feature can be very important in many situations for at least the following reasons: first of all, problems exist that are so complex that the idea of finding a perfect solution is purely utopian. In those cases, obviously, it is better to have an approximated solution than not having any solution at all. Secondly, there are cases in which an approximated solution can be as good as a perfect one. One may, for instance, imagine a problem consisting in driving a robot in a room, following a path from a prefixed starting point A to a prefixed destination B, avoiding obstacles that are positioned in the room. In this context, it is straightforward that not a particular path is required, but any path that allows the robot to go from A to B avoiding obstacles is exactly as good as all the others that have this characteristic. Last but not least, having the ability of returning approximated solutions can be crucial for

improving the generalization ability of a system, given that a perfect solution may be affacted by *overfitting*, a focal concept of Machine Learning.

This document introduces various bio-inspired intelligent systems. In particular, we will discuss two types of local search algorithms, like Hill Climbing and Simulated Annealing, two types of evolutionary algorithms, like Genetic Algorithms and Genetic Programming and one swarm intelligence algorithm, Particle Swarm Optimization.

1.3 Applications of Intelligent Systems

Intelligent systems are used nowadays in a number of applications, both in Industry and Academia, that is so huge that listing them all is inconceivable. However, in this section, we try to give an incomplete and limited list of possible applications, just to give the reader an idea of the abundance and breadth of possible areas where intelligent systems are successfully employed.

- *Robotics*. Optimization of paths and movements of robots, treatment of signals coming from sensors and other sources, and many other related applications are nowadays a reality, and they have lead, for instance, to the development of self-driving vehicles or autonomous appliances.
- *Engineering*. The automatic synthesis of integrated electronic circuits and the management of mechanical systems, like appliances, trains or assistance devices are only few examples of the numerous engineering applications that are nowadays strongly based on intelligent systems.
- *Biology*. From the study of DNA and human genome, to the modelling of complex biological system like gene regulatory networks or viruses, and many other applications, intelligent systems have played a central role, in the last decades, in the developments of Biology.
- *Chemistry*. The automatization of the drug discovery process, aimed at designing, developing and commercializing new medicines and the study and the optimization of the 3D structure of molecules and their properties are only few examples of the numerous applications of chemistry in which intelligent systems have been applied successfully.
- *Medicine*. The analysis and mining of the vast amounts of data, like radiomics, genomics, epigenomics, proteomics, together with clinical data and many others are nowadays analyzed by intelligent systems with the objective of developing predictive models for diagnosis, prognosis and therapy, supporting decision making in Medicine.
- *Economics*. Simulation and prediction of stock market trends, risk management, decision support in companies and organizations and pricing are only some of the numerous applications in which intelligent systems are applied in Economics nowadays.

- *Marketing*. Optimization of marketing campains, placement of goods in supermarket shelves and design of marketing strategies are performed successfully using intelligent systems in several companies and organizations.
- *Security*. Traffic crontrol, face recognition, identity checking and tracking are tasks that are automatized and optimized by means of intelligent systems in many situations nowadays.
- *Computational Security*. Crittography, identification of malware and protection of computational systems, from large to small scale, by means of intelligent systems are a reality.
- *Society*. Predicticing over-indebtment of families, school performance of children or inclination towards gambling addiction, drugs addiction, or other types of addiction is currently performed by intelligent systems in several situations, facilitating early intervention and social support.
- *Pervasive Computing*. Computers are everywhere nowadays; it is the case, for instance, of houses equipped with domotics, smart appliances and interactive facilities, where intelligent systems obviously play a crucial role.

1.4 Objectives and Limits of This Course

The list of possible applications of intelligent systems presented in the previous section could continue indefinitely. The conclusion is straightforward: intelligent systems are general purpose methods, that can be used with success in practically all existing application areas. Under this perspective, it is clear that it would be presumptuous, and ultimately impossible, to try to discuss all possible applications in one University course. Instead, in this course, we will not enter into the details of any application, except for some examples in some specific cases. The course focuses on computational methods, with the objective of providing studehts with the necessary methodological tools and competences, that can hopefully allow them to tackle complex applications in the future.

The course has two ambitions: first, remaining general enough to be independent from the application, but at the same time concrete enough to promote the future use of intelligent systems in any of them. Second, disseminating in the students the message that intelligent systems are still a young area, where many improvements are still possible, and often necessary. Research in this area is, today more than ever, in demand, and this course wants to be an incentive for young, enthusiastic and dynamic minds to express and develop new ideas, for actively contributing to the field.

Of course, being general (i.e. independent from applications) does not have to mean being abstract or insubstantial. The numerous examples contained in this document, as well as the necessary practical classes that are part of the course, are the means to maintain the appropriate level of concreteness, in a discipline that is naturally at the fronteer of research. However, it is our convinction that, in order to successfully apply intelligent computational methods at an industial level, a deep

expertise is needed, that involves not only a “final user” knowledge of the needed technicalities to apply those methods, but also a deep understanding and possession of their inner functioning and dynamics. This is the type of knowledge that this course has the ambition of transferring.

This document will not present any programming language or environment for intelligent systems, for at least two reasons: first, for reasons of space: the document would otherwise become unbearably long. Second, because the most popular technologies and programming environments evolve and change with a high velocity, and any of them will shortly become updated and will be replaced by others.

Chapter 2
Optimization Problems and Local Search

This chapter introduces optimization problems, one of the largest classes of complex tasks where intelligent systems have become a reality, in the last few years. Then, this chapter tackles the first type of algorithms that can be used to approach optimization problem: local search algorithms. Generally speaking, local search algorithms function by "moving" from solution to solution in the space of candidate solutions, by applying *local* changes, until a satisfactory solution is found or a time bound is elapsed.

2.1 Introduction to Optimization

Optimization [Antoniou and Lu, 2007, Kochenderfer and Wheeler, 2019] is a field of study aimed at developing methods, strategies and algorithms for solving complex optimization problems. In its most general sense, the objective of an optimization problem is to find the best solution(s) to a problem in a (huge) set of possible alternative solutions. Generally speaking, an optimization problem can be approached if we are in possess of at least two pieces of information: we need to know all the possible solutions, or at least to recognize whether an object is a possible solution or not, and we need to know, or at least to be able to measure, the quality of each one of the solutions, in such a way that each one of them can be compared to the others. Furthermore, a general hypothesis of optimization is that the set of all possible solutions is so large that it is impossible to enumerate all of them, looking for the best one(s). And this is why "intelligent" algorithms are generally in demand for solving optimization problems. Just to fix ideas, let us consider the following examples of optimization problems:

- Example 1: given a tridimensional space characterized by a set of paths where a robot can move, find the path that allows the robot to minimize the number of hits of obstacles during its motion;

- Example 2: given a set of photographs of human faces, find the one that "resembles" more a given target face.

Both these examples can be considered optimization problems, because for both of them we know the solutions, or at least we are able to recognize them, and we are able to assess the quality of all the solutions. For instance: in Example 1, solutions are all the possible paths contained in the tridimensional space at hand, and in order to asses the quality of each one of them, all we have to do is to allow the robot to move along the path, and count the number of obstacles that are hit: the lower this number, the better the quality. Concerning Example 2, the solutions are pictures of human faces and in order to quantify their quality, all we have to do is to define a measure of similarity between pictures: the largest the similarity with the target face, the better the quality.

Several optimization problems are NP-complete [Garey and Johnson, 1990]. This means that optimal solutions cannot be obtained in a "reasonable" amount of time by means of classical, deterministic algorithms (and generally, the time required to solve this type of problem is exponential with the size of the data). This is the main motivation for using Computational Intelligence methods to solve this kind of problems. Optimization problems can usually be specified by means of a set of *instances* of the problem, where:

Definition 2.1. An instance of an optimization problem is a pair:

$$(S, f)$$

where:

- S is the set of all possible solutions, also called solutions space, or search space;
- f is a function, defined on all elements of S, that associates a real number to each one of them:

  $$f : S \rightarrow \mathbb{R}$$

  f quantifies the quality of the solutions in S and it is called cost function, or fitness function.

Before continuing, it is important to understand the difference between an optimization problem and an instance of an optimization problem. An instance of an optimization problem is a specification of the problem itself, given by the formal definition of S and f. Defining a particular instance for a given problem is usually a process that forces us to make choices and also to give an interpretation of the problem (an instance of an optimization problem, in fact, cannot be ambiguous). Let us consider again the optimization problem of Example 2, consisting in finding, in a given set of photographs of human faces, the one that "resembles" more a given target face. Let us assume that also the target face is represented in a picture, and let t be that picture. Let $\{\phi_1, \phi_2, ..., \phi_n\}$ be the set of photographs of human faces, among which we have to chose the most similar to t. For that problem, we can imagine, at least, the existance of the following instances:

- Instance 1:
  - $S = \{P_i,\ i = 1,2,...,n \mid P_i$ is the matrix of pixels composing picture $\phi_i\}$;
  - $\forall i = 1,2,...,n : f(P_i) =$ pixel-to-pixel distance between P_i and the target image t.
- Instance 2: let us assume the existance of an algorithm $\mathscr{A}$ such that, given a photograph ϕ representing a face, returns a set of *features* of ϕ; just to fix ideas, one could for instance imagine somatic features, like for instance the position of the nose, of the eyes and of the mouth, the color of the eyes, the hair color, etc.
  - $S = \{\mathscr{A}(\phi_i),\ i = 1,2,...,n\}$;
  - $\forall i = 1,2,...,n : f(P_i) =$ feature-based distance between $\mathscr{A}(\phi_i)$ and $\mathscr{A}(t)$.

Several things can be observed from the previous examples: first of all, several instances can be defined for the same problem; secondly, the possible instances can even be extremely different between in each other, both in terms of the representation of the solutions in S and in terms of the fitness function f; third, while an optimization problem can be defined in a rather generic way (for instance, if the set of pictures is small enough, it is even possible to solve the problem manually, using a concept of "similarity" between faces that is subjective, and not measurable), an instance of an optimization problem has to be defined formally, and this is a necessary step so that it can be solved using a computer; fourth, to define an instance, we have to interpret and formalize the problem, and this usually implies a set of choices; last but not least, as we will see in the continuation of this document, the algorithms that we will study may have very different performance on different instances of the same problem. So, the choices that we make when we define an instance of an optimization problem are very important, because they may have a direct impact on the functioning of the algorithm. For instance, considering the previous example, it is well known that a distance between image features is more likely to correspond to our idea of "similarity" between the pictures than a pixel-to-pixel distance. For this reason, an algorithm solving Instance 2 will probably return better results than an algorithm solving Instance 1.

An instance of an optimization problem can identify a maximization or a minimization problem.

Definition 2.2. A *minimization problem* consists in finding a solution $o \in S$ such that:

$$f(o) \leq f(i),\ \forall i \in S$$

while a *maximization problem* consists in finding a solution $o \in S$ such that:

$$f(o) \geq f(i),\ \forall i \in S$$

In both cases, the sought for solution o is called *global optimum* or *globally optimal* solution, $f(o)$ or f_o is called optimal fitness and the notation S_o is used to indicate the set of the global optima contained in S (remark that, given that in general more solutions can have the same fitness value, global optima are generally not unique).

In the area of optimization, it is typical to talk about several types of problems. In particular, it is frequent to talk about *combinatorial* optimization problems. With this terminology, it is customary to identify a subset of optimization problems where the search space S, although typically huge, is finite. It is not infrequent to also find in the literature the term combinatorial optimization problems associated to optimization problems in which the feasible solutions can be expressed using concepts from combinatorics (such as sets, subsets, combinations or permutations) and/or graph theory (such as vertices, edges, cliques, paths, cycles or cuts). The term "combinatorial" can be understood as a combination of steps chosen from a series/set of possible steps, which will allow us to arrive at the optimum result. Other typical cases of optimization problems are *discrete* and *continuous* optimization problems. These terms are used, once again, to identify the search space S, distinguishing the case in which it is a discrete, or a continuous set, respectively. The definition of optimization problem given so far is general enough to include all these variants.

2.2 Examples of Optimization Problems

Before undertaking the study of optimization algorithms, we present some examples of optimization problems, along with instance definitions.

Example 2.1. (*Knapsack Problem*). Given a set of *n objects*, each one with a known *weight* and *value*, and a knapsack with a predefined maximum *capacity* k, the objective of this optimization problem is to fill the knapsack with objects with the largest possible total value, such that the total weight of these objects does not surpass the knapsack's capacity. A possible instance for this problem could be defined as follows; let $\{1,2,...,n\}$ be the available objects and, for each $i = 1,2,...,n$, let $weight(i)$ be the weight of object i and $value(i)$ be its value. Assuming that, for each $i = 1,2,...,n$, $value(i)$ and $weight(i)$ are positive numbers:

- The search space S can be defined as the set of all possible strings of bits of length equal to n (i.e. the number of available objects);
- Given a solution $\mathbf{z} = \{z_1, z_2, ..., z_n\}$, where for each $i = 1,2,...,n, z_i \in \{0,1\}$, the fitness of $\mathbf{z}$ can be defined as:

$$f(\mathbf{z}) = \begin{cases} \sum\limits_{z_i=1} value(i) & \text{if } \sum\limits_{z_i=1} weight(i) \leq k \\ -1 & \text{otherwise} \end{cases} \tag{2.1}$$

  With this fitness function, the problem is a maximization one: the higher the fitness, the better the solution.

Using this representation, the solutions represent the possible selections of the n available objects. In fact, when a bit z_i is equal to 1, this can be interpreted as the object i being carried inside the knapsack. Analogously, if a bit z_i is equal to 0, then the corresponding object i is not carried inside the knapsack. In other words, with

this representation, we are creating two groups, or *clusters* of objects: the ones that are carried inside the knapsack, corresponding to a bit equal to 1, and the ones that are not carried, corresponding to a bit equal to 0. The fitness function distinguishes between admissible solutions, i.e. solutions for which the total weight of the carried objects is not larger than the knapsack's capacity, and not admissible ones. For the admissible solutions, the fitness is equal to the sum of the values of the objects carried in the knapsack, while for the non-admissible solutions it is equal to a negative constant (for instance -1, as in the example). In this way, knowing that all the values are positive, each admissible solution will have a better fitness than any non-admissible solution. The fact that all non-admissible solutions have the same fitness may be a problem, particularly when these solutions are numerous. For this reason, an improved version of Equation (2.1) may be:

$$f(\mathbf{z}) = \begin{cases} \sum\limits_{z_i=1} value(i) & \text{if } \sum\limits_{z_i=1} weight(i) \leq k \\ -\sum\limits_{z_i=1} weight(i) & \text{otherwise} \end{cases} \tag{2.2}$$

In this way, we are identifying a *gradient* also in the area of the non-admissible solutions, which can be useful in some cases for the algorithms that will be studied in the continuation of this document. With the definition given in Equation (2.2), the fitness of non-admissible solutions becomes better and better as the total weight of the carried objects gets closer to the threshold.

Let us now consider a numeric case, characterized by the following data:

- let the number of available objects be $n = 10$;
- let the knapsack's capacity be $k = 165$;
- let the weights of the objects be: 23, 31, 29, 44, 53, 38, 63, 85, 89, 82;
- let the values of the objects be: 92, 57, 49, 68, 60, 43, 67, 84, 87, 72.

Let us now consider solution:

$$\mathbf{z} = 1111010000$$

This object represents the case in which the 1^{st}, 2^{nd}, 3^{rd}, 4^{th} and 6^{th} objects are carried in the knapsack, while the others are not. The total weights of the carried objects is:

$$\sum_{z_i=1} weight(i) = 23+31+29+44+38 = 165$$

Given that the total weight of the carried objects is identical to the knapsack's capacity, the solution $\mathbf{z}$ is admissible, so its fitness is:

$$f(\mathbf{z}) = \sum_{z_i=1} value(i) = 92+57+49+68+43 = 309$$

An exhaustive analysis of the search space was done for this particular numeric case, and it showed that this solution represents a global optimum, in other words it is not

possible to find another combination of 10 bits corresponding to a better selection of the objects than this one.

The Knapsack Problem has a large number of real-life applications. Just as an example, one may consider the case of a set of investments that can be made on the stock market, for each of which we know the cost and the expected profit. The objective, in that case, would be to select the subsets of investments that allow us to maximize the expected profit, inside a total cost that is not larger than our predefined budget.

The Knapsack Problem is a well known and widely studied optimization problem. The reader interested in deepening her knowledge on this problem is referred, for instance, to [Martello and Toth, 1990].

Example 2.2. (*Travelling Salesperson Problem*). Given a set of cities and distances between each pair of cities, the objective of the Travelling Salesperson Problem (TSP) is to find the shortest possible route that visits each city, returning to the origin city. More specifically, n cities are given and the pairwise distances of all the cities are known. For instance, they can be given in a $n \times n$ matrix D, where the element of indexes p and q ($D_{p,q}$) denotes the distance between the p^{th} and the q^{th} cities. The matrix is, of course, symmetric; in other words $D_{p,q} = D_{q,p}$ for any pair of cities p and q. A cycle is a closed walk that visits each city exactly once. The problem consists in finding the cycle of minimal length. A *permutation* of the n cities could be indicated as:

$$\pi = \{k, \pi(k), \pi^2(k), ..., \pi^{n-1}(k)\}$$

where $k = 1, 2, ..., n$ denotes a city. For each k, $\pi(k)$ denotes the successor of city k, i.e. the city that is visited right after k, and, by definition:

$$\pi^t(k) = \underbrace{\pi(\pi...(\pi(k)))}_{t \text{ times}}$$

in other words $\pi^t(k)$ denotes the city that is visited t steps after that k was visited. A *cycle* is a permutation π such that the following properties are respected:

- $\pi^\ell(k) \neq k$, if $\ell = 1, 2, ..., n-2$;
- $\pi^{n-1}(k) = k$

Given this formalization, an instance of the TSP can be defined as:

- $S = \{\pi \mid \pi \text{ is a cycle on the } n \text{ given cities}\}$
- Given any solution $\pi \in S$, the fitness of π can be defined as:

$$f(\pi) = \sum_{i=1}^{n} D_{i,\pi(i)}$$

$f(\pi)$ returns the total length of cycle π.

Let us now apply these concepts to a numeric case. For simplicity, let the number of cities be $n = 4$, and let the matrix of the pairwise distances be:

$$D = \begin{array}{|c|c|c|c|}\hline \times & 7 & 2 & 3 \\ \hline \times & \times & 4 & 1 \\ \hline \times & \times & \times & 8 \\ \hline \times & \times & \times & \times \\ \hline \end{array}$$

This matrix represents the graph shown in Figure 2.1.

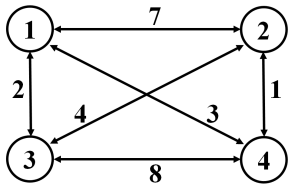


Fig. 2.1 Graph representation of the instance of the TSP discussed in Example 2.2.

Let us now consider solution $\pi_1 = \{1,2,3,4,1\}$ and let us calculate its fitness. We have:

$$f(\pi_1) = D_{1,2} + D_{2,3} + D_{3,4} + D_{4,1} = 7 + 4 + 8 + 3 = 22$$

Let us now consider solution $\pi_2 = \{1,4,2,3,1\}$ and let us calculate its fitness. We have:

$$f(\pi_2) = D_{1,4} + D_{4,2} + D_{2,3} + D_{3,1} = 3 + 1 + 4 + 2 = 10$$

The TSP has many possible real-life applications, particularly in the field of logistics and transportation. One may think, for instance, of optimizing the itinerary of a person delivering pizza to a set of houses, starting from, and returning to, the same location, that may be the pizzeria. At the same time, it is not difficult to imagine how many concepts of the TSP can be applied to the optimization of a bus itinerary or even to airtraffic optimization.

The TSP is a well known and widely studied optimization problem. The reader interested in deepening her knowledge on this problem is referred, for instance, to [Applegate et al., 2007].

2.3 No Free Lunch Theorem

In the continuation of this document, we will study several algorithms to solve optimization problems. These algorithms will generally be called optimization algo-

rithms or, more particularly, *computational intelligence* optimization algorithms, with the objective of distinguishing them from classical, deterministic optimization algorithms. Although very different between each other, these algorithms all share a common structure: they are all iterative algorithms that, at each iteration, return a solution to the problem. In other words, given an instance of an optimization problem (S, f), an execution of an optimization algorithm can be identified by a sequence, or vector, of solutions, each one being returned at termination of an iteration. Let this vector be $\mathbf{b} = \{s_1, s_2, ..., s_m\}$, where, for all $i = 1, 2, ..., m$, $s_i \in S$. Given that, by definition, the fitness function f must be defined on all the elements of S, for each solution s_i in $\mathbf{b}$, its fitness value $f_i = f(s_i)$ can be calculated. So, instead of using $\mathbf{b}$, one may identify an execution of an optimization algorithm by means of the vector:

$$\mathbf{c} = \{f_1, f_2, ..., f_m\}$$

$\mathbf{c}$ is the vector of the fitness values of the solutions returned at each generation by an optimization algorithm. Let $\mathscr{A}$ be an optimization algorithm, and let us now consider the following conditional probability:

$$P(\mathbf{c} \mid f, m, \mathscr{A})$$

This is the conditional probability that algorithm $\mathscr{A}$, using f as a fitness function, yields exactly vector $\mathbf{c}$ as the sequence of the fitness values of the solutions returned in the first m iterations of its execution. If we think carefully, this conditional probability can be imagined as a generalization of the concept of *performance* of an algorithm. A particular case, in fact, is if for any $i = 1, 2, ..., m$, $f_i = f_o$, in other words if the sequence of solutions returned by $\mathscr{A}$ in its first m iterations contains a globally optimal solution. In that case, this conditional probability can be interpreted as the probability of algorithm $\mathscr{A}$ of finding a global optimum in its first m iterations, using f as a fitness function. Given that the objective of an optimization problem is to find a globally optimal solution, we can say that, in this particular case, this conditional probability corresponds to our interpretation of the performance of an algorithm, intended as its ability of finding a global optimum. In simple terms, we could informally say: the higher the probability of finding a global optimum, the better the performance of the algorithm.

Using this notion, we are now ready to enunciate one of the most general and fundamental results in the field of optimization:

Theorem 2.1. (**No Free Lunch Theorem**) [Wolpert and Macready, 1997].
Given any sequence of fitness values $\mathbf{c} = \{f_1, f_2, ..., f_m\}$ *and any pair of optimization algorithms* $\mathscr{A}_1$ *and* $\mathscr{A}_2$*, we have:*

$$\sum_f P(\mathbf{c} \mid f, m, \mathscr{A}_1) = \sum_f P(\mathbf{c} \mid f, m, \mathscr{A}_2) \tag{2.3}$$

A formal proof of the No Free Lunch Theorem is beyond the scope of this document; the interested reader is referred to [Wolpert and Macready, 1997] for a proof

and a wide discussion of this very important result. Here, we are more interested in discussing the intuitive meaning, and the important consequences, of the No Free Lunch Theorem. In order to understand what this theorem is saying to us, we first need to have an intuition of the meaning of summing up $P(\mathbf{c} \mid f, m, \mathscr{A})$ over all possible fitness values f (remark that this is what is happening on both sides of Equation (2.3): the summations run over all possible fitness functions). Informally, we can interpret it as a sum of the performance of an algorithm *over all existing optimization problems*. To convince oneself about it, consider a numerable space of solutions S[1]. In this situation, for each possible optimization problem, we could rename the existing feasible solutions into solution$_1$, solution$_2$, solution$_3$,... In this way, we could imagine that all optimization problems have the same set of solutions. Given that, as we will understand when we will study some optimization algorithms, the fitness function is useful only to compare solutions between each other (so that the best one can be identified), we can also imagine that, once solutions have been sorted from the worst to the best, the fitness values are modified into $1, 2, 3, \ldots$ (the case of solutions with identical fitness values are also taken into account in [Wolpert and Macready, 1997]). In this way, what makes the difference between one problem and another is the assignment of the fitness values to the solutions, or, if we imagine solutions to always keep the same order, the sorting of the possible fitness values. In this interpretation, a fitness function identify a possible sorting of fitness values and all possible sortings identify all the possible problems. Under this perspective, a new, and more informal formulation of the No Free Lunch Theorem could be given as follows:

Theorem 2.2. (No Free Lunch Theorem, informal statement). *Given any pair of optimization algorithms $\mathscr{A}_1$ and $\mathscr{A}_2$, $\mathscr{A}_1$ and $\mathscr{A}_2$ have identical average performance, calculated on all existing optimization problems.*

In other words, it cannot exist an algorithm (that we could call a "super"-algorithm) that performs better than all the others on all possible existing optimization problems, and if a set of problems exists on which an algorithm $\mathscr{A}_1$ outperforms an algorithm $\mathscr{A}_2$, another set of problems is likely to exist in which $\mathscr{A}_2$ outperforms $\mathscr{A}_1$.

This fact has an interesting consequence: every time that we are faced with a particular otimization problem, we have no formal/automatic method to decide what is the best algorithm to solve it. Indeed, if such a method existed, it would be the super-algorithm that would contradict the No Free Lunch Theorem. In other words, the choice of an appropriate algorithm to solve a particular problem can only be a heuristic and informal process, typically based on our experience as problem solvers, on our knowledge on the dynamics of the different algorithms, and/or on a set of experiments, aimed at comparing different algorithms. Under this perspective, the No Free Lunch Theorem incentivates the study of many different algorithms, and motivates and paves the way for the existance of several software environments, in which several different Computational Intelligence

[1] The validity of the No Free Lunch Theorem for continuous optimization, i.e. when S is infinite and not numerable, was questioned in [Auger and Teytaud, 2010], and so that case will not be discussed in this section.

techiques are implemented, and a comparison between them is made particularly easy and automatic. Only two of the numerous existing software environments with these characteristics are: Weka [Hall et al., 2009] (implemented in Java) and Scikit-learn [Pedregosa et al., 2011] (implemented in Python).

We conclude this section by drawing the attention of the reader on a singular, and possibly counterintuitive, fact. In the beginning of this section, we have defined the concept of optimization algorithm as an iterative algorithm, able to return a solution at the end of each iteration. This definition is general, and it applies regardless of the principle that is used to generate the next solution, which is what distinguishes the different algorithms between each other. The definition is so general that even *random search*, i.e. a rather "naive" algorithm that returns a random solution at each iteration, respects it. Thus, also random search can be considered an optimization algorithm, and so the No Free Lunch Theorem applies also to it. In other words, if averaged on all existing problems, random search performs exactly as all the other optimization algorithms, including the ones that are considered as more sophisticated or "intelligent", and a set of problems exists on which random search outperforms them. Although surprising, this fact is true, and in the continuation of this chapter, problems on which random search outperforms all the other algorithms will be studied. Of course, those problems have partcular characteristics that make them rather different from real-life applications, where, instead, more sophisticated or "intelligent" methods than random search are often the most appropriate choice.

2.4 Hill Climbing

As studied in the previous section, one of the consequences of the No Free Lunch Theorem is that the choice of an appropriate algorithm to solve a problem can only be a heuristic process, in which our knowledge of the functioning and dynamics of many different algorithms may play a crucial role. Thus, it makes sense to study several optimization algorithms of different nature. This study begins in this section, in which one of the simplest and most naive Computational Intelligence algorithms is presented: *Hill Climbing* [Aarts and Korst, 1989, Russell and Norvig, 2009].

Hill Climbing is possibly the most natural and immediate technique to try to solve an optimization problem, and it consists in an attempt to improve fitness in a stepwise refinement way, by means of the concept of *neighborhood*. The process is so simple, that it can be informally told in few words: let us assume that we are able, for each solution i belonging to the space of solutions S, to generate a subset $N(i)$ of S, where $N(i)$ can be interpreted as the set of "neighbor" solutions of i ($N(i)$ is also called the neighborhood of i). The Hill Climbing starts with an initial (typically random) solution i, that is made the current solution, and tries to improve it. More in particular, it chooses one solution j from $N(i)$ (for instance, it can be the solution with the best fitness in $N(i)$), and, if the fitness of j is better than the fitness of i, turns j into the new current solution i. The process is iterated until the neighborhood

of the current solution i does not contain any better solution than i. At that point, the algorithm terminates, returning i as the final result.

This process depends on the fundamental concept of neighborhood structure N, which is discussed here, before the functioning of the Hill Climbing is presented with more rigour and details.

Definition 2.3. (Neighborhood Structure) Let (S, f) be an instance of an optimization problem. A neighborhood structure is a mapping:

$$N : S \to 2^S$$

that associates to each solution $i \in S$ a subset of S, that we indicate as $N(i)$, and that we call the neighborhood of i.

Each solution $j \in N(i)$ is called a neighbor of i and in general we assume that, for any pair of solutions $i, j \in S$, $j \in N(i)$ if and only if $i \in N(j)$. Furthermore, we assume the existance of a precise algorithm $\mathscr{A}$ that allows, given a solution i to generate its neighborhood $N(i)$ and we assume that, for each solution $i \in S$, $\mathscr{A}$ applied to i terminates and returns a set containing at least one admissible solution $j \in S$.

In general, there are neither restrictions nor rules for defining a neighborhood structure, and any mapping respecting the above properties is, in general, acceptable. However, it is customary to associate the definition of a neighborhood structure either to an operator of transformation of the solutions, or to a measure of distance between solutions. So, given a solution $i \in S$, we define the neighborhood of i in one of the following ways:

- $N(i) = \{j \in S \mid j = op(i)\}$, for a given operator op.
- $N(i) = \{j \in S \mid d(i, j) \leq k\}$, for a given distance metric d and prefixed constant k.

According to the first definition, the neighborhood of a solution i is the set of solutions that can be obtained by applying an operator op to i. According to the second one, once a distance metric has been chosen, the neighborhood of a solution i is the set of solutions that have a distance to i smaller or equal than a given prefixed constant k. As we will see in the next examples, these two definitions often coincide, being generally possible to define a distance *corresponding* to an operator, and vice versa. In practice, if we apply one of these two definitions, the neighbors of a solutions i often end up for being solutions that are structurally rather similar to i.

Example 2.3. Let us assume that the feasible solutions are strings of bits of a prefixed length, like in the knapsack problem that we have studied in Example 2.1. For instance, one possible neighborhood could be defined in such a way that two solutions are neighbors if and only if they differ by one bit in a corresponding position. For instance, given a solution:

$$i = 1011011$$

the solution:

$$j = 1011001$$

is a neighbor of i, because all the bits of j are, position by position, identical to the corresponding bits of i, except for the bit in the 6^{th} position of the string, that is different. On the other hand, solution:

$$h = 1000101$$

is *not* a neighbor of i, in fact the bits in the 3^{rd}, 4^{th}, 5^{th} and 6^{th} position of h are different from the corresponding bits of i, and so the number of different bits in corresponding positions is larger than one.

Let us now see how it is possible to define such a neighborhood by means of an operator and by means of a distance. The operator can simply work as follows: choose a position in the string and flip the bit contained in that position, leaving the other bits unchanged. The distance can be Hamming distance [Norouzi et al., 2012] (defined as the number of different bits in corresponding positions), and $k = 1$.

Example 2.4. Let us now assume that fleasible solutions are all the permutations of integer numbers in a given range, and let us assume that, as in the TSP studied in Example 2.2, the first and last values in the string are fix and unchangeable. A possible neighborhood could be obtained by exchanging two values in a solution. For instance, given solution:

$$i = 1234567891$$

a possible neighbor of i could be:

$$j = 1237564891$$

because all the characters in j are identical to the corresponding characters in i, except for 4 and 7, that have been exchanged. The reader is invited to notice that, in order to define this neighborhood, it was natural to use the concept of operator: the operator that swaps two characters, at any pair of positions.

Given an instance of an optimization problem, neighborhoods are generally not unique. On the other hand, a vast number of possible neighborhoods could be chosen. Just as a matter of example, exactly for the same problem studied in this example, another possible neighborhood could be obtained using an operator that selects two characters at any pair of positions and exchanges the order of all the characters in between. Using this new definition of neighborhood, a neighbor of individual i could be:

$$h = 1265437891$$

since all characters in h are like the corresponding ones in i except for the characters that appear from the 3^{rd} to the 6^{th} positions, that appear in opposite order.

Given an instance of an optimization problem (S, f), and a neighborhood structure N, the pseudo-code of the Hill Cimbing is reported in Algorithm 1. In that algorithm, i represents the variable storing the current solution at each step. The different steps of the algorithm can be explained as follows:

- In Step 1, an initial solution i_{start} is generated to allow the process to begin. If we have some information about the problem, like for instance that solutions with

Algorithm 1: Pseudo-code of the Hill Climbing for an instance of an optimization problem (S, f) and a neighborhood structure N.

1. Initialize a feasible solution i_{start} from the search space S (typically at random);
2. $i := i_{start}$; // Let the current solution i be equal to i_{start}
3. **repeat**
   - 3.1. Generate a solution j from $N(i)$;
   - 3.2. **if** ($f(j)$ is better or equal than $f(i)$) **then**
     - $i := j$; // Let j become the new current solution i
   - **end**

   until $\forall j \in N(i) : f(j)$ is worse than $f(i)$;
4. **return** i

good fitness should have determinate characteristics, that information has to be used in this step. But the typical situation is that we do not have this type of information. So, in general, the initial solution is randomly generated[2];

- In Step 2, the variable i, used to store the current solution at each step of the algorithm, is initialized by assigning i_{start} to it;

- Step 3 contains the main cycle of the algorithm, in which we try to improve the fitness of the current solution in a stepwhise manner:

  – Step 3.1 consists in the generation of a neighbor j of the current solution i. This can be obtained using the transformation operator that defines the neighborhood structure N. Several different variants of the Hill Climbing can exist, corresponding to different possible choices, but the most frequent choice is that j is the best neighbor (in terms of fitness) of i;

  – Step 3.2 consists in a comparison of the fitness of j with the fitness of i. In case the fitness of j is better or equal than the one of i, j becomes the new current solution. In case of minimization problem, $f(j)$ is better or equal than $f(i)$ if $f(j) \leq f(i)$, while in case of maximization problems $f(j)$ is better or equal than $f(i)$ if $f(j) \geq f(i)$. Remark that a variant of the Hill Climbing algorithm also exists in which the current solution is *not* replaced in case the generated

[2] One important point to understand is that, in general, given that we know how the space of solutions S is defined, we are also able to generate a random solution. For instance, consider the case of Example 2.3, where S is the set of all possible strings of bits of a given prefixed length n. Generating a random solution, in that case, can be simply done by flipping a coin n times, inserting a 0 in case of tail and a 1 in case of head, or vice versa.

neighbor has identical fitness to the current solution. In that case, that we call *strict Hill Climbing*, the term "better or equal" in the pseudo-code should be replaced with "better".

The cycle terminates when all neighbors of the current solution i are worse in fitness than i. In the case of strict Hill Climbing, the cycle terminates also in case the best neighbor of i has a fitness that is identical to the one of i.

- Step 4 is executed when the main cycle of the algorithm has terminated, and it consists in returning the current solution i as the final result.

Before presenting an example that should clarify the functioning of the Hill Climbing, it is important to study the following definition.

Definition 2.4. (*Local Optimum*). Let (S, f) be an instance of an optimization problem, and let N be a neighborhood structure. A solution $\bar{i} \in S$ is called a local optimum with respect to N if $\bar{i}$ has a fitness that is better or equal than all the other solutions in its neighborhood. In other words, in case of minimization problems, $\bar{i}$ is a local optimum with respect to N if:

$$f(\bar{i}) \leq f(j), \forall j \in N(\bar{i})$$

and in case of maximization problems, $\bar{i}$ is a local optimum with respect to N if:

$$f(\bar{i}) \geq f(j), \forall j \in N(\bar{i})$$

The reader is invited to notice that, by its very definition, the solution returned by the Hill Climbing is always a local optimum. In fact, the termination condition of the algorithm is exactly corresponding to the definition of local optimum. Furthermore, it should be noticed that *a global optimum is also a local optimum*. In fact, being a global optimum the best solution among all the feasible ones, it is also the best of its neighborhood. We conclude that the Hill Climbing *may* return a global optimum, however we have no guarantee that this will happen. For a given optimization problem instance, the quality of the solution returned by the Hill Climbing mainly depends on the initial solution i_{start}, that is typically generated at random, and by the particular employed neighborhood structure, that is a choice we must make when solving the problem.

Example 2.5. (*Execution of the Hill Climbing on a Numeric Case*). Let (S, f) be an instance of an optimization problem, where:

- $S = \{i \mid i \in \mathbb{N} \ \& \ 0 \leq \mathrm{i} \leq 15\}$;
- $\forall i \in S : f(i) =$ number of bits equal to 1 in the binary code of i (maximization).

Furthermore, consider the following neighborhood structure:

$$\forall i, j \in S : j \in N(i) \iff |j - i| = 1$$

In this "toy" case of study, the search space is composed by only 16 feasible solutions: the natural numbers included between 0 and 15. As a fitness function, we use the number of 1 in the binary code. Give that the problem was defined as a maximization one, it is straightforward to understand that the global optimum for this problem is represented by solution 15, given that it is the only natural number included between 0 and 15 that contains 4 bits equal to 1 (the binary code of 15 is, in fact, 1111). Finally, the neighborhood structure represents, so to say, the "intuitive" neighborhood of natural numbers: for instance, 4 is a neighbor of 3 and 5, 9 is neighbor of 8 and 10, and so on and so forth.

Let us, now, simulate the execution of the Hill Climbing on this problem. The first step consists in the random generation of an initial solution. Let us assume that our random number generator allowed us to obtain 5 as the initial solution. So, $i = 5$ is the first current solution. Given that the binary code of 5 is 101, the fitness of this solution is 2 (two bits are equal to 1 in the binary code). At this point, the Hill Climbing is supposed to generate a solution from the neighborhood of 5. Let us assume that, as it is usual, the solution chosen by the Hill Climbing is the best in the neighborhood. So, the neighborhood of the current solution is:

$$N(5) = \{4, 6\}$$

Given that the fitness of 4 is 1 and the fitness of 6 is 2, the generated neighbor of 5 is $j = 6$. The fitness of j is now compared to the fitness of i and, consistently with the pseudo-code of Algorithm 1, given that the two fitness values are identical, the current solution is updated. The new current solution is: $i = 6$. The algorithm now iterates the process by analysing the neighborhood of 6:

$$N(6) = \{5, 7\}$$

Given that the fitness of 5 is 2 and the fitness of 7 is 3, the best neighbor is $j = 7$. Since the fitness of j is better than the fitness of i, the current solution is updated again. The new current solution is $i = 7$. The algorithm iterates again, analysing now the neighborhood of 7:

$$N(7) = \{6, 8\}$$

Given that the fitness of 6 is 2 and the fitness of 8 is 1, the best neighbor is $j = 6$. But since the fitness of the best neighbor j is worse than the fitness of i, it is straightforward to infer that all the solutions in the neighborhood of 7 are worse than 7. Consequently, the algorithm terminates, and 7 is returned as the final solution. It is worth pointing out that 7 is a local optimum, but it is not the global optimum for this problem, since the global optimum, as previously mentioned, is 15.

Before terminating this section, it is worth discussing the pros and cons of the Hill Climbing: advantages of the Hill Climbing are that it is very simple, and easy to specify, implement and use. A further advantage of this algorithm consists in its flexibility: in fact, it is rather simple to change the configuration of the problem, the neighborhood structure, or even only the initial solution, and run the algorithm

again, obtaining a different result. The main disadvantage of Hill Climbing is, of course, that it always returns a local optimum, with no guarantee that it corresponds to a global optimum. This is a very important flaw, since a local optimum can even be a very poor solution. So, methods to overcome this disadvantage deserve to be studied. In order to increase the probability of the Hill Climbing to return solutions of better quality, one may imagine the following approaches:

- run the Hill Climbing multiple times, each time using a different initial solution (eventually, all these independent executions can be run in parallel, to save computational time);
- use a more complex neighborhood structure, so that we are able to explore a larger portion of the search space at each iteration;

Unfortunately, both these stragies are, in general, destined to fail. Concerning the first idea, in fact, in real problems the number of local optima can be so high that each agent of a parallel Hill Climbing may end up trapped in a different local optimum. On the other hand, although extending the neighborhood may effectively improve exploration ability, in general, in order to significantly increase our confidence that the algorithm returns a global optimum, the neighborhood should become so large to become unmanageable. Indeed, one of the most accepted approaches for improving the Hill Climbing consists in accepting, with a given limited probability, a worsening in the fitness of the current solution. This is the idea that is at the basis of the Simulated Annealing, that will be studied in Section 2.6. But before studying the Simulated Annealing, the fundamental concept of Fitness Landscape is presented in Section 2.5.

2.5 Fitness Landscapes

Let us take back the optimization problem instance studied in Example 2.5, with the neighborhood considered in the example, and let us now perform the following exercise: let us draw a plot in which in the horizontal direction we arrange all the solutions in the search space, *sorted consistently with the used neighborhood structure*, and on the vertical axis we put fitness. In the particular case of Example 2.5, sorting the solutions consistently with the neighborhood structure is straightforward: we just need to arrange them using the habitual ordering of natural numbers. The obtained graphic is shown in Figure 2.2. Such a plot is called *fitness landscape*, given that it visually roughly reminds of a landscape, with peaks, valleys, plateaux, etc. The behavior of the Hill Climbing can be imagined as a "walk" on this landscape, where each single movement is represented by the passage from one current solution to the next. Given that, at every step of the algorithm, the current solution can only be a neighbor of the previous current solution, and given that, in the fitness landscape, neighbor solutions are actually physically "neighbors" in the horizontal direction, no "jumps" are allowed in the landscape. For instance, in the case of Example 2.5, the Hill Climbing started its "motion" at abscissa 5 (the randomly generated ini-

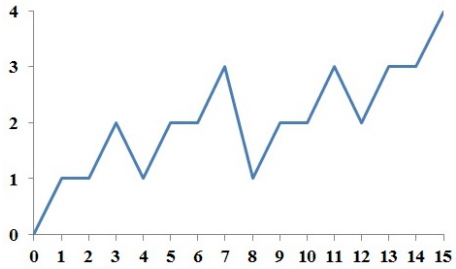


Fig. 2.2 Fitness landscape for the optimization problem instance and neighborhood studied in Example 2.5.

tial solution was 5), then "moved" to 6, and finally moved to 7, and then stopped, returning 7 as a final solution.

As we can see from this simple example, the behaviour of the Hill Climbing can be imagined as the one of a "mountaineer", devoted to "climb" the fitness landscape, and that stops everytime it reaches a top (peak), being it, or not, the highest one in the landscape. A Fitness Landscape [Stadler, 2002, Pitzer and Affenzeller, 2012] is a classical way of visualizing the relationship between the syntactic structures of the solutions and their fitness. The concept is inherited from Biology, and it can be defined as follows:

Definition 2.5. (*Fitness Landscape*). Given an instance of an optimization problem (S, f) and a neighborhood structure N, a Fitness Landscape (FL) is a plot in which all the solutions in S are represented on the horizontal direction, sorted consistently with N, and, for each solution $i \in S$, the fitness value $f(i)$ is reported on the vertical direction. A FL is completely identified by the triple:

$$(S, f, N)$$

A FL gives a visual intuition on the difficulty, or simplicity, with which a problem instance can be solved using a configuration of an optimization algorithm. In particular, we can imagine the existence of at least the following cases:

- a "smooth" landscape, with one, or very few, peaks;
- a "rugged" landscape, with several different steep peaks.

The former scenario typically corresponds to an easy problem, that can often be solved by many algorithms, including Hill Climbing. The latter case generally corresponds to a hard problem, that is difficult to solve not only by the Hill Climbing, but also by any of the other existing algorithms, that often get stuck in one of the numerous local optima. Besides these two cases, one could also mention neutral landscapes, i.e. FLs in which a large number of neighbors have the same, or approximately the same, fitness values. This scenario corresponds to the presence of plateaux on the landscape. Although no gradient can be identified in flat portions of

the landscape, the usefulness of the presence of neutrality in FLs is still controversial. Smooth, rugged and neutral FLs, and their implications on the performance of optimization algorithms are discussed in [Vassilev et al., 2003].

Although the concept of FL can be very useful to understand the difficulty of a problem, it is generally impossible to draw a FL (even though significant steps forward are proposed in [McCandlish, 2011]), at least for the following reasons:

- the vast magnitude of the search space;
- the large dimensionality of the neighborhood.

The former point makes it generally impossible to arrange all the feasible solutions on the horizontal direction of a plot. The latter one turns the plot into a multi-dimensional one, which makes it hard to draw it. The difficulty represented by the latter point can be mitigated by representing the FL in the following way: let d be the distance that is associated with the used neighborhood structure (see the second definition of neighborhood defined at page 19); a FL can be represented using a graph, where each vertex represents a solution, and it is labelled with the fitness of the solution it represents, and each edge joining a solution i_1 to a solution i_2 is labelled with the value of the distance $d(i_1, i_2)$. An execution of the Hill Climbing defines a walk on this graph. To make the representation more "visual", one may imagine to "lean" the graph on a horizontal plane, arranging the solutions in such a way that the distance of each pair of solutions i_1 and i_2 on the plane is directly proportional to $d(i_1, i_2)$, and fitness could be given by a projection of each vertex in the third dimension. This 3-dimensional plot can give a visual idea of the ruggedness of a landscape even in presence of high-dimensional neighborhoods, but still it cannot be drawn in general, due to the vast magnitude of the search space. Nevertheless, in many real cases, it is possible to imagine the shape of the FL, for instance starting from some points of known fitness, and this can be useful to obtain information about the ability of an algorithm to find a global optimum. Let us now consider some examples of fitness landscape, drawing its shape whenever possible, and trying to imagine it otherwise.

Example 2.6. Let us take back Example 2.5, and let us extend the number of solutions in the search space, by incrementing the upper bound of the natural numbers up to 1023 in the definition of S. In other words, we have the following optimization problem instance:

- $S = \{i \mid i \in \mathbb{N} \ \& \ 0 \leq \mathrm{i} \leq 1023\}$;
- $\forall i \in S : f(i) =$ number of bits equal to 1 in the binary code of i (maximization).

And the following neighborhood structure:

$$\forall i, j \in S : j \in N(i) \iff |j - i| = 1$$

Analogously to the case of Example 2.5, it is easy to see that the global optimum is represented by solution 1023, which can be represented by a chain of 10 bits, each of which equal to 1, while all other numbers between 0 and 1022 can be represented

using 10 bits, but all of them contain at least one bit equal to 0. Given that the neighborhood is bi-dimensional, we can still draw the FL using a bi-dimensional plot as in Figure 2.3. As we can see, this landscape is very rugged. The reader is

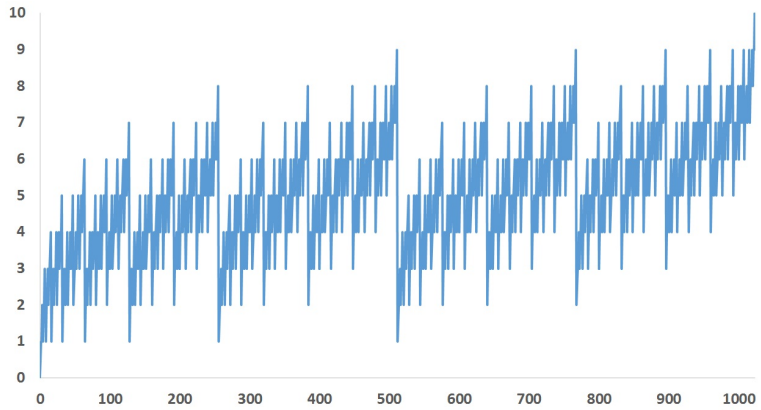


Fig. 2.3 Fitness landscape for the optimization problem instance and neighborhood studied in Example 2.6.

invited to implement the Hill Climbing and try to use it to solve this problem. It will be quick to observe that very often the Hill Climbing will not be able to return the global optimum.

Example 2.7. Let us now consider the following optimization problem instance:

- $S = \{i \mid i \in \mathbb{N} \ \& \ 0 \leq \mathrm{i} \leq 1023\}$;
- $\forall i \in S : f(i) = i^2$ (maximization).

And the following neighborhood structure:

$$\forall i, j \in S : j \in N(i) \iff |j - i| = 1$$

As we can see, the only difference between this case and the one studied in Example 2.6 consists in a different fitness function. The FL, in this case, is represented in Figure 2.4. The landscape is clearly smooth, with no local optimum, except for the unique global optimum, represented, again, by solution 1023. This corresponds to the typical configuration of a problem that is easy to solve. The interested reader is invited, once again, to implement this simple problem and try to solve it with the Hill Climbing. It will be immediate to observe that the Hill Climbing will always be able to return the global optimum.

Example 2.8. Let us now consider the following optimization problem instance:

- $S = \{i \mid i \in \mathbb{N} \ \& \ 0 \leq \mathrm{i} \leq 1023\}$;
- $\forall i \in S : f(i) =$ number of bits equal to 1 in the binary code of i (maximization).

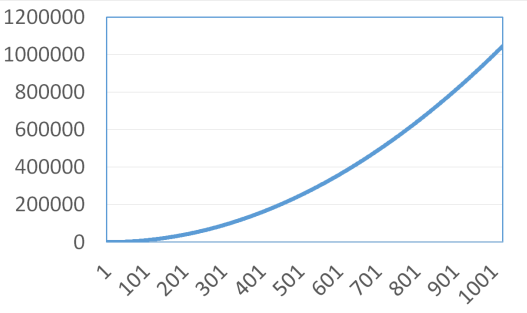


Fig. 2.4 Fitness landscape for the optimization problem instance and neighborhood studied in Example 2.7.

And the following neighborhood structure:

$$\forall i, j \in S : j \in N(i) \iff \text{the Hamming distance between } i \text{ and } j \text{ is equal to } 1.$$

As we can see, the only difference between this case and the one studied in Example 2.6 consists in a different neighborhood structure. This time, two solutions are neighbor if they differ by just one bit. As an attentive reader will agree, given that all solutions in S can be represented by strings of 10 bits, each solution has now 10 neighbors. In other words, the neighborhood, in this case, is 10-dimensional. This makes it very hard to draw the fitness landscape. Nevertheless, it should not be too hard to imagine what its shape should look like: if a solution is not the global optimum (that, once again, is represented by solution 1023, whose binary code is 1111111111), it will have at least one bit equal to 0 in its binary code. And thus, changing that 0 into a 1, we will be able to obtain a better neighbor. Consequently, all the solutions in S, except for the global optimum, have at least one neighbor that is better than them. We conclude that, in this case, the FL is smooth, with no local optima, except for the unique global optimum. Once again, this can be easily confirmed in practice, by implementing a Hill Climbing to solve this problem. The reader will observe that, in a worst case of 10 steps, the Hill Climbing will always be able to return the global optimum.

From Examples 2.6, 2.7 and 2.8, it is possible to understand that a variation of any of the three elements defining a FL, i.e. S, f and N, may completely change the shape of the FL, and thus the ability of an algorithm to find the global optimum.

We conclude this section with a last example, that should represent cause for reflection about the No Free Lunch Theorem (Theorem 2.1).

Example 2.9. Let us consider the bi-dimensional FL shown in Figure 2.5, and let the problem be a maximization one. Let us also assume that $B \ll D$, or, in other words, let us assume that the probability that a random number drawn with uniform

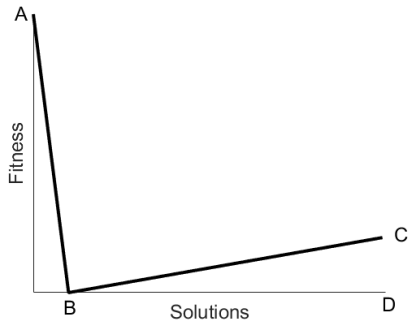


Fig. 2.5 Graphical representation of a deceptive fitness landscape.

distribution in $[0,D]$ is smaller or equal than B is practically equal to zero. In such a situation, it should not be difficult to convince oneself that the Hill Climbing has a very poor performance. In fact, with high probability the random initial solution will correspond to an abscissa in $(B,D]$, and given that the algorithm tends to improve fitness at every step, the solution returned by the Hill Climbing will often be the one corresponding to abscissa D. This solution is a local optimum, with a fitness equal to C, which is qualitatively much worse than the fitness A of the global optimum. But the fact of returning a solution of poor quality is only one of the flaws that the Hill Climbing has on this type of problem: if we consider the distance d that is associated with the used neighborhood structure (see the second definition of neighborhood defined at page 19), the returned solution is even the one that is further away from the global optimum according to d. In simple terms, the Hill Climbing is returning a solution of poor quality, that is also very different from the global optimum.

These types of problems are called *deceptive* problems, and they are characterized by the fact that, most of the times, a steady attempt of improving fitness leads the algorithm towards local optima. In other words, the fitness function is misleading, in the sense that it tends to conduct the search towards poor quality solutions. Even though problems that are deceptive in each part of the search space are hard to find, it is not infrequent in real-life applications to have significant portions of the search space that are deceptive. The existance of problems of this type pushes us to the following reflections:

- On deceptive problems, Random Search (i.e. an optimization algorithm that returns a random solution at each iteration) generally outperforms Hill Cimbing. This is a corroboration of the validity of the No Free Lunch Theorem (Theorem 2.1), and of the fact that this theorem holds also for an apparently very naive algorithm such as Random Search. Indeed, surprisingly as it may seem, in presence of deceptive or partially deceptive problems, Random Search can be a reasonable strategy.

- Always blindly following fitness, in the steady attempt of improving it, can be a very losing strategy for an optimization algorithm. This is what Hill Climbing does, and this is one of the reasons why Hill Climbing is not one of the most effective optimization algorithms for real-life problems. Actually, as already previously mentioned, one of the most useful strategies to improve the Hill Climbing is to release the algorithm from the idea of always improving fitness. In Simulated Annealing, that is the algorithm we study in the next section, in some cases the fitness of the current solution can worsen. This corresponds to the possibility of letting an algorithm go downhill in a FL during its exploration.

Looking again at a case such as the one represented in Figure 2.2, it should not be hard to convince oneself that, if the current solution is any of the solutions corresponding to an abscissa smaller than 12, the only possibility that an agent has of getting to the global optimum (abscissa 15) is accepting some downhill steps during the exploration. Thus, the idea behind the Simulated Annealing seems reasonable and promising.

2.6 Simulated Annealing

Simulated Annealing [Kirkpatrick et al., 1983, Černý, 1985, Aarts and Korst, 1989] extends the Hill Climbing, taking inspiration from a metallurgy and materials science heat treatment, called annealing [Vlack, 2008]. Annealing is a process that allow us to obtain materials in solid state, with the lowest possible level of energy. It alters the physical and sometimes chemical properties of the material, and it is used to increase its ductility and reduce its hardness, making it more workable. It involves heating the material above its recrystallization temperature, maintaining a suitable temperature for a suitable amount of time, and then allow slow cooling.

In simple terms, the annealing process can be summarized as follows: it begins with the material in a solid state i, with energy E_i. Then some chemical bonds are modified, so as to obtain a new solid state j, with energy E_j. At this point, the new current state of the material is probabilistically chosen between i and j and the process is iterated until the material stabilizes in a given state. The choice of accepting i or j as the new current state is based on the respective energy values E_i and E_j. In particular, the probability of accepting j is given by $P(\text{accept } j)$, defined in Equation (2.4), while the probability of maintaining i as the current state is $1 - P(\text{accept } j)$. $P(\text{accept } j)$ is defined as:

$$P(\text{accept } j) = \begin{cases} 1 & \text{if } E_j \leq E_i \\ e^{\frac{-|E_j - E_i|}{k_B T}} & \text{otherwise} \end{cases} \tag{2.4}$$

where T is the temperature and k_B is the Boltzmann constant [Fischer, 2019].

The process described so far is a particular case of the Metropolis algorithm [Chib and Greenberg, 1995] and it was shown to be effective in finding the

solid state of a material with the lowest level of energy. Its similarities with the Hill Climbing are visible, in the sense that we talk of a current state, similarly to how in the Hill Climbing we talk of a current solution, and we try to update the current state by applying some transformations that may roughly remind the application of an operator to obtain a neighbor. Energy in the annealing corresponds to fitness for optimization algorithms, and it is supposed to be minimized. The macroscopic difference with the Hill Climbing is clearly that the energy of the current state can increase with some probability. The Simulated Annealing extends the Hill Climbing to also envisage the case of a temporal worsening in the fitness of the current solution. This worsening will be accepted with a given probability, that is inspired by Equation (2.4).

Given an instance of an optimization problem (S, f) and a current solution $i \in S$, the probability of accepting a new solution $j \in S$ as the new current solution is given by:

$$P(\text{accept } j) = \begin{cases} 1 & \text{if } f(j) \text{ is better or equal than } f(i) \\ e^{\frac{-|f(j)-f(i)|}{c}} & \text{otherwise} \end{cases} \tag{2.5}$$

where, as usual, in case of minimization problems $f(j)$ is better or equal than $f(i)$ if $f(j) \leq f(i)$, while for maximization problems $f(j)$ is better or equal than $f(i)$ if $f(j) \geq f(i)$. Compared to Equation (2.4), in Equation (2.5) the concept of energy of a material was replaced by the fitness of a solution. Furthermore, the term $k_B\, T$ was replaced by a *positive* number c, that will be called *control parameter* of the Simulated Annealing and, as we will see, plays an important role in the dynamics of the algorithm.

Once established these similarities between the Simulated Annealing and the Metropolis algorithm used for the annealing, the Simulated Annealing can be seen as an iteration of the Metropolis algorithm, using decreasing values of the control parameter parameter c. The idea of beginning the execution with high values of c, and steadily decreasing c during the execution of the algorithm can be motivated if we observe Figure 2.6, reporting the graphical representation of function $\phi(c) = e^{-\frac{k}{c}}$, where k is a positive constant ($k = 2$ was used in figure). This plot can be used to study the variation of the probability of the Simulated Annealing to accept a worse solution than the current one, as c is variated. As we can observe, if we use a "large" value of c in the early phase of the execution of the algorithm, in this phase we will have a rather "large" probability of accepting fitness worsenings. Also, if we assume that c is steadily decreased during the execution, we can clearly see that this is equivalent to a steady decrease in the probability of accepting a worse solution than the current one. Finally, if we assume that the algorithm works by decreasing c in such a way that c tends towards zero, without ever arriving at zero[3], it

[3] If the value of c is equal to zero and j is a worse solution than i, then $P(\text{accept } j)$ in Equation (2.5) returns an error, due to a denominator equal to zero. This case is avoided, by avoiding c to ever become equal to zero during the execution of the Simulated Annealing.

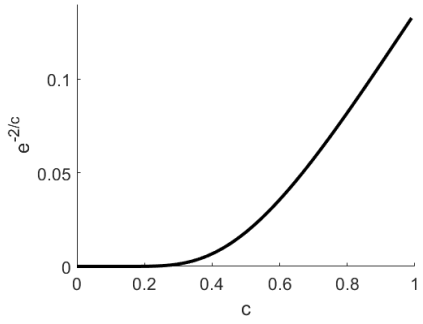


Fig. 2.6 Graphical representation of the function $\phi(c) = e^{-\frac{2}{c}}$, used to understand the contribution of the control parameter c in the probability of accepting a solution with worse fitness, compared to the current one.

is easy to understand that this corresponds to a probability of accepting a worsening in fitness that tends towards zero.

From all this, we infer that beginning with high values of c, and steadily decreasing it, in such a way that c tends towards zero without ever reaching zero, is equivalent to beginning the execution of the algorithm in a situation in which the probability of worsening the fitness of the current solution is high, and then steadily decreasing this probability, tending towards a situation that resembles the one of the Hill Climbing, i.e. where the probability of accepting fitness worsenings is very unlikely. The rationale beyond this idea is, in simple terms, that in the beginning of its execution the algorithm may be in a difficult area of the search space, characterized by the presence of several local optima. In such a situation, going downhill with reasonably high probability can be useful to overstep some hills. On the other hand, as long as the execution proceeds, we are possibly approaching one of the highest hills, hopefully the basin of attraction of a global optimum. In that case, we do not have any interest in going downhill. On the other hand, in such a situation, the most effective behavior is climbing up the hill as fast as possible, as the Hill Climbing would do.

The pseudo-code of the Simulated Annealing is reported in Algorithm 2. The algorithm has the objective of navigating the search space by iteratively updating the current solution i. This is done by executing several *transitions*, where a transition of the Simulated Annealing is, by definition, a sequence characterized by the generation of a neighbor j of the current solution i, followed by the decision whether to accept or not j as the new current solution, a decision that may depend on the current value of the control parameter c. The algorithm is characterized by two nested loops. The idea of having two loops is that, for each value of the control parameter c, we should give the algorithm a number L of "attempts", before c is modified. Several considerations concerning Algorithm 2 are made in the next paragraphs.

1. *External Loop and Termination Condition* (point 4). The external loop has a termination condition, that actually corresponds to the stopping criterion of the al-

Algorithm 2: Pseudo-code of the Simulated Annealing for an instance of an optimization problem (S, f) and a neighborhood structure N.

1. Initialize a feasible solution i_{start} from the search space S (typically at random);
2. $i := i_{start}$; // Let the current solution i be equal to i_{start}
3. Initialize L and c;
   // L is the number of iterations of the internal loop, c is the control parameter
4. **repeat until** *termination condition*
   - 4.1. **repeat** L **times**
     - 4.1.1. Generate a solution j from $N(i)$;
     - 4.1.2. **if** ($f(j)$ is better or equal than $f(i)$) **then**
       - $i := j$; // Let j become the new current solution
     - **else if** $\left(\text{Rand}[0,1) < e^{\frac{-|f(j)-f(i)|}{c}}\right)$ **then**
       - $i := j$; // Let j become the new current solution
     - **end**
   - 4.2. Update c;
   - 4.3. Update L;
   - **end**
5. **return** the solution with best fitness encoutered so far;

gorithm, that is usually verified when one of the following two situations happens:

- a "satisfactory" solution has been found, or
- a prefixed maximum number of iterations was executed.

Concerning the first point: let us assume that the optimal fitness f_o is known[4]. In such a situation, we could define an admissible deviation ε_f from that fitness value. By "satisfactory" solution, here, it is meant a solution x, whose fitness value f_x is at a distance smaller or equal than ε_f to f_o. Concerning the second point: the prefixed maximum number of iterations can be considered a parameter of the algorithm, to be set before beginning the execution. In case f_o is not known, the second point remains the only termination condition.

[4] The reader should remark that knowing the optimal fitness is rather usual in optimization problems, where, instead, the objective is finding a solution with such a fitness. To convince oneself about this, one may consider Example 1 at page 9. In that case, the optimal fitness is equal to zero (no obstacles hit by the robot). Finding a path that allows the robot to hit zero obstacles is the objective of the problem.

2. *Internal Loop* (point 4.1). The internal loop of the algorithm is executed for L iterations. The most general situation (the one reported in Algorithm 2), is that the value of L is modified at each iteration of the external loop, but it can be kept as a constant during the whole execution, eliminating point 4.3., and making L a further parameter to be set beforehand.

3. *Generation of the neighbor j* (point 4.1.1). Although the algorithm is general, and any strategy to choose the neighbor can be used, in the case of Simulated Annealing it is customary to consider a *random* neighbor of the current solution. In order to convince oneself about the appropriateness of this choice, the reader is invited to have a look back at the fitness landscape represented in Figure 2.2. Let us assume that the current solution is 7. This solution has two neighbors: 6 and 8. The fitness of 6 is equal to 2, while the fitness of 8 is equal to 1. So, if our choice was to choose the best neighbor, as it is customary for the Hill Climbing, in such a situation the generated neighbor would be 6. On the other hand, the reader should recognize that, if the current solution is 7, the only hope that the algorithm has to get to the global optimum (that is 15) is to accept 8 as the next solution. Such a situation can be generalized, saying that, in some situations, always choosing the best neighbor of the current solution may jeopardize our chances of getting to a global optimum. Besides this, it is also straightforward to understand that, in particular in presence of large neighborhoods, generating a random neighbor is much faster than finding the best neighbor, which implies an evaluation of all the solutions in the neighborhood.

4. *Probabilistic Acceptance of a Worse Solution* (else branch of point 4.1.2). If the chosen neighbor j has a worse fitness than the current solution i, the event of accepting j as the new current solution is probabilistic. Its implementation in Algorithm 2 uses the primitive $\mathrm{Rand}[0,1)$, which returns a random number between 0 and 1, drawn with uniform distribution[5]. The reader is invited to reflect on the portion of pseudo-code:

$$\textbf{if } \left(\mathrm{Rand}[0,1) < e^{\frac{-|f(j)-f(i)|}{c}}\right) \textbf{ then } i := j;$$

This can be interpreted as the "translation", in implementative terms, of the sentence:

j is accepted as the new current solution with a probability given by $e^{\frac{-|f(j)-f(i)|}{c}}$

5. *Update of the Control Parameter c* (point 4.2). As previously discussed, c should be decremented at each iteration, in such a way that it steadily tends towards zero, but without ever being equal to zero. In this way, the algorithm should be able to

[5] Practically all existing programming languages have such a predefined primitive. For instance, in Java, one may use the method `random()` of the class `Math`.

climb over several hills, basins of attraction of local optima, in the first phase of the execution, while it should "resemble" the Hill Climbing in a second phase, when the basin of attraction of a global optimum has hopefully been reached. Any way of updating c that respects these principles can generally be used. A simple example example is to divide the value of c by a constant that is larger than 1. As we will understand later, it is generally a good idea to have a slow decrementation of the value of c. In order to obtain this, for instance, c could be divided by a constant that is "slightly" larger than 1.

Let us now study the functioning of the Simulated Annealing on a simple numeric example.

Example 2.10. Let us take back the optimization problem instance (S, f) and neighborhood structure N of Example 2.5, i.e.:

- $S = \{i \mid i \in \mathbb{N} \ \& \ 0 \leq \mathrm{i} \leq 15\}$;
- $\forall i \in S : f(i) =$ number of bits equal to 1 in the binary code of i (maximization);
- $\forall i, j \in S : j \in N(i) \iff |j - i| = 1$

As we did in Example 2.5, let us assume that the initial random solution is $i = 5$. Given that the binary code of 5 is 101, the fitness of 5 is equal to 2 (since the binary code has two bits equal to 1). Let us also assume that the generation of the neighbor j is random and that $j = 6$. Given that the binary code of 6 is 110, also the fitness of 6 is equal to 2. Although a "strict" version of the Simulated Annealing exists, the most common version (and the one reported in Algorithm 2) envisages a replacement of the current solution when the generated neighbor has a fitness that is identical to the current solution. Thus, the new solution is now $i = 6$, and the algorithm is iterated.

Let us assume, now, that the random neighbor of 6 generated by the algorithm is $j = 7$. Given that the binary code of 7 is 111, the fitness of j is equal to 3, i.e. better than the fitness of i. In such a situation, without any further computation, j is accepted as the new current solution. So, now the current solution is $i = 7$.

Let us assume that the generated neighbor of 7 is $j = 8$. The binary code of 8 is 1000, so the fitness of 8 is equal to 1. We are now in a situation in which the generated neighbor has a worse fitness than the current solution. In such a situation, we can accept or not j as the new current solution, with a certain probability, given by:

$$P(\text{accept } 8) = e^{\frac{-|f(8)-f(7)|}{c}}$$

Let us assume, just for simplicity, that in this moment of the execution of the algorithm the value of the control parameter c is equal to 1. We have:

$$P(\text{accept } 8) = e^{\frac{-|1-3|}{1}} = e^{-2} \approx 0.13$$

In other words, we have a probability approximately equal to 13% of accepting 8 as the new current solution. Just to give the reader an informal understanding of the usual dynamics of the Simulated Annealing, it is worth pointing out that this

must be considered a significantly *large* probability of accepting the new solution. In fact, over 10 independent attempts in the same situation, the solution should on average be accepted at least once. Although it can be useful in some circumstances, accepting a fitness worsening is generally a rather rare event. When the Simulated Annealing encounters a local optimum, it is typical to remain stuck on it for several iterations before being able to climb over it and begin to explore new regions of the search space.

Contrarily to the Hill Climbing, the Simulated Annealing has the ability of trespassing local optima, while still maintaining some positive characteristics of the Hill Climbing, such as simplicity and generality. The convergence speed of the algorithm depends on several factors, among which:

- the initial value of the control parameter c;
- the speed at which c is decreased;
- the number of iterations L in which the same value of c is maintained.

Setting these parameters in an appropriate way is generally a hard task, depending on the characteristics and complexity of the problem. Nevertheless, some heuristics can be given, after having studied some theoretical properties of the algorithm.

2.6.1 Theory of Simulated Annealing

The objective of this section is to study the asymptotic convergence behavior of the Simulated Annealing. The final result will be presented and commented in Theorem 2.3. But, as a stepping stone to that result, we first introduce Definition 2.6 and Lemma 2.1.

Definition 2.6. Given an instance of an optimization problem (S, f) and a neighborhood structure N, we say that N is a *completely interconnected* neighborhood if and only if for each pair of solutions $i, j \in S$, a sequence $\ell_0, \ell_1, ..., \ell_p$ exists such that:

- $\forall k = 0, 1, ..., p : \ell_k \in S$;
- $\forall k = 1, 2, ..., p : \ell_k \in N(\ell_{k-1})$;
- $\ell_0 = i$;
- $\ell_p = j$.

In informal terms, a neighborhood structure is said completely interconnected if given any pair of solutions i and j it is always possible to obtain j starting from i by means of a sequence of solutions, that are neighbors two-by-two. The fact of using a completely interconnected neighborhood structure is a necessary condition for Lemma 2.1 and Theorem 2.3 to hold.

Lemma 2.1. *Let (S, f) be an instance of a minimization problem, on which the Simulated Annealing is executed using a completely interconnected neighborhood structure. After a "sufficiently large" number of transitions performed using constant c as a control parameter, the Simulated Annealing stabilizes on a solution $i \in S$ with a probability equal to:*

$$P\{X = i\} = q_i(c) = \frac{1}{N_0(c)} \; e^{-\frac{f(i)}{c}}$$

where:

$$N_0(c) = \sum_{j \in S} e^{-\frac{f(j)}{c}}$$

$P\{X = i\} = q_i(c)$ is called *stationary probability*, or *equilibrium distribution* of the Simulated Annealing, for control parameter c. Lemma 2.1 is not proven in this document. The reader interested in a proof of this Lemma, based on Markov Chains, is referred to [Aarts and Korst, 1989]. Lemma 2.1 was enunciated for minimization problems; an analogous result also holds for maximization problems, but will not be discussed in this document. With Theorem 2.3, we are now interested in understanding on what solution(s) the Simulated Annealing will stabilize, after a large number of iterations, when c is variated.

Theorem 2.3. (Theorem of Asymptotic Convergence of Simulated Annealing). *Let (S, f) be an instance of a minimization problem, on which the Simulated Annealing is executed using a completely interconnected neighborhood structure. Assuming that the Simulated Annealing is executed by steadily decreasing the value of the control parameter c, in such a way that c tends towards zero, without ever being equal to zero, we have:*

$$\lim_{c \to 0} q_i(c) \; = \; \frac{1}{|S_{opt}|} \; \chi_{(S_{opt})}(i)$$

where:

- *S_{opt} is the set of all the globally optimal solutions in the serach space[6];*
- *$\chi_{(S_{opt})} : S \to \{0, 1\}$ is a function defined for all the solutions i in the search space, such that:*

$$\chi_{(S_{opt})}(i) = \begin{cases} 1 & \text{if } i \in S_{opt} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

[6] It is worth reminding that globally optimal solutions are not necessarily unique, in fact several solutions can have the same fitness. So, all the solutions that have an optimal fitness are global optima. For this reason, in general, we talk of a set of globally optimal solutions.

Proof. For the sake of simplicity, in this proof we will use a different notation for the exponential function: from now and until the end of the proof, for each argument x, e^x will be represented using the notation $exp(x)$. From Lemma 2.1, and applying the limit for c tending towards infinity, we directly obtain:

$$\lim_{c\to 0} q_i(c) = \lim_{c\to 0} \frac{exp(-\frac{f(i)}{c})}{\sum_{j\in S} exp(-\frac{f(j)}{c})} \tag{2.7}$$

Let f_{opt} be the optimal (in this case, minimum) fitness value. Multiplying numerator and denominator of the right part of Equation (2.7) by $exp(\frac{f_{opt}}{c})$, we obtain:

$$\lim_{c\to 0} q_i(c) = \lim_{c\to 0} \frac{exp(\frac{f_{opt}}{c}) \cdot exp(-\frac{f(i)}{c})}{exp(\frac{f_{opt}}{c}) \cdot \sum_{j\in S} exp(-\frac{f(j)}{c})}$$

from which, it is possible to immediately derive:

$$\lim_{c\to 0} q_i(c) = \lim_{c\to 0} \frac{exp(\frac{f_{opt}-f(i)}{c})}{\sum_{j\in S} exp(\frac{f_{opt}-f(j)}{c})} \tag{2.8}$$

Now, in order to complete the proof, we have to use a property, according to which:

$$\forall a \leq 0 : \lim_{x\to 0} exp(\frac{a}{x}) = \begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

The interested reader is referred to [Rudin, 1986] for a proof and discussion of this property.

Let us isolate the numerator in the right hand of Equation (2.8):

$$\lim_{c\to 0} exp(\frac{f_{opt}-f(i)}{c}) \tag{2.10}$$

As we can observe, Equation (2.10) is rather similar to Equation (2.9): both of them express the limit of an exponential function, in both cases the argument of the exponential is a fraction, and in both cases the denominator of this fraction is the quantity tending towards zero in the limit. Furthermore, given that we are considering a minimization problem, by definition of f_{opt} we have: $\forall i \in S : f_{opt} \leq f(i)$. So, the numerator of the fraction, i.e. $f_{opt} - f_i$ is a quantity that is smaller or equal than zero. We conclude that Equation (2.9) can be used to obtain the result of Equation (2.10): that result will be equal to 1 when $f_{opt} - f_i$ is equal to zero, and equal to zero otherwise. But if $f_{opt} - f_i = 0$, then $f(i) = f_{opt}$, which means that i is a globally

optimal solution. In other terms, Equation (2.10) is equal to 1 if $i \in S_{opt}$, and equal to zero otherwise. But this is exactly the definition of $\chi_{(S_{opt})}(i)$ given in Equation (2.6). We conclude that:

$$\lim_{c \to 0} \; exp(\frac{f_{opt} - f(i)}{c}) \; = \; \chi_{(S_{opt})}(i) \tag{2.11}$$

Let us now isolate the denominator in the right hand of Equation (2.8):

$$\lim_{c \to 0} \; \sum_{j \in S} exp(\frac{f_{opt} - f(j)}{c}) \tag{2.12}$$

Applying exactly the same reasoning used previously, we have that, for each $j \in S$, $\lim_{c \to 0} \; exp(\frac{f_{opt} - f(j)}{c})$ is equal to 1 if $j \in S_{opt}$ and equal to zero otherwise. But given that this quantity is summed up for each $j \in S$, the result of the summation is clearly equal to the number of globally optimal solutions in S. In other terms:

$$\lim_{c \to 0} \; \sum_{j \in S} exp(\frac{f_{opt} - f(j)}{c}) \; = \; |S_{opt}| \tag{2.13}$$

Now, replacing Equation (2.11) and Equation (2.13) into Equation (2.8), we obtain:

$$\lim_{c \to 0} q_i(c) \; = \; \frac{1}{|S_{opt}|} \; \chi_{(S_{opt})}(i) \tag{2.14}$$

But Equation (2.14) is identical to the thesis of the theorem, which allows us to terminate this proof.

□

Theorem 2.3 was enunciated for minimization problems; an analogous result also holds for maximization problems, but will not be studied in this document. In order to understand the intuitive meaning of Theorem 2.3, first of all, we have to remark that, given the functioning of the Simulated Annealing (i.e. given the fact that c is steadily decreased, in such a way that it tends towards zero), a limit for c tending towards zero is equivalent to a limit for time tending to infinity. So, Theorem 2.3 is giving us information about the properties of the Simulated Annealing, as the running time tends to infinite (asymptotic property).

Let us now try to answer the following question: what is the theorem telling us if the search space contains just one global optimum? The answer is straightforward: it is saying that, as time tends to infinity, the Simulated Annealing will tend to stabilize on that global optimum with a probability equal to 1, and on any other solution different from the global optimum with probability zero. Let us now try to understand what the theorem is telling us in case the search space contains two global optima. Also in this case, looking at the theorem's enunciate, it is not difficult to convince oneself that as time tends to infinity, the Simulated Annealing will tend to stabilize on one of those global optima with a probability equal to 0.5, on the other

global optimum with a probability equal to 0.5, and on any solution that is not a global optimum with probability zero.

Generalizing the previous reasoning, we can conclude that the theorem is telling us that, as time tends to infinity, the Simulated Annealing tends to stabilize on a global optimum, and the probability is uniformly distributed over all existing global optima. Interestingly, this property holds independently from the problem at hand, and, as such, from the shape of the fitness landscape.

Of course, this property is *not* telling us that the Simulated Annealing will find a global optimum in a humanly acceptable time. It is actually telling us that it will happen, but it is not saying anything about the convergence speed, and thus about the time in which it will happen. As already mentioned above, the optimization speed of the algorithm depends only on the parameter setting, which is a problem dependent task. In order to maximize our chances of finding a global optimum, all we can do is executing a large number of transitions for each value of the control parameter, and decreasing the control parameter slowly, in such a way that the total number of iterations performed is as large as possible. As it is intuitive, this also slows down the running time of the algorithm, and finding a good compromise between efficiency and effectiveness can be a hard task when we decide the values of the parameters.

Before concluding this section, it is worth discussing one point: using the *Law of Large Numbers* [Keane, 1995], one can easily infer that, given a potentially infinite amount of time, Random Search will find a global optimum for any problem. From this consideration, one may start wondering if there is really a difference in terms of effectiveness between Simulated Annealing[7] and Random Search, which may induce one to mistrust the real usefulness of the Simulated Annealing. Indeed, what Theorem 2.3 is telling us is much more than the simple application of the Law of Large Numbers for Random Search. Theorem 2.3 is telling us that the Simulated Annealing tends asymptotically towards a global optimum. From the intuitive meaning of limit [Rudin, 1986], and in informal terms, we could say that this entails that, after a certain amount of computation, the Simulated Annealing starts getting closer and closer to a global optimum. So, the Simulated Annealing is able to approximate a globally optimal solution, when it is not able to find it. In other words, we could say that an amount of time t spent executing the Simulated Annealing is "well spent", because, after this time, we have a significant probability of having found a solution that is better than the initial one, and this probability is generally higher as t gets bigger. On the olther hand, nothing like this can be said for Random Search, for which the probability of finding a good solution at any time t is identical to the one at time zero. These considerations allow us to conclude that asymptotic convergence towards a global optimum is a very important property, and for an algorithm to be considered "intelligent", such a property should hold.

[7] The reasoning proposed here holds also for any other algorithm for which it is possible to prove asymptotic convergence to a globally optimal solution.

Chapter 3
Genetic Algorithms

Genetic Algorithms (GAs) [Holland, 1975, Goldberg, 1989] are the oldest, and possibly the most known method belonging to the field of Evolutionary Computation (EC) [Eiben and Smith, 2015]. As the Simulated Annealing, studied previously:

- GAs are a technique for searching optimal solutions in a space of possible alternative solutions;
- GAs do not require an exhaustive analysis of the search space (which would be unpractical, given the vast size of the search space);
- GAs typically begin their search with randomly generated solutions and try to improve their quality in a stepwise refinement fashion, by means of an iterative algorithm.

However, unlike the Simulated Annealing, in GAs:

- The potential solutions (called *individuals* in the EC terminology) must necessarily be represented as stringhts of characters of a prefixed length, that does not change during the execution of the algorithm;
- The solutions that are considered at each iteration (called *generation* in the EC terminology) are more than one, in fact we talk of a *population* of individuals;
- The principle used as an inspiration is the theory of evolution of Darwin [Darwin, 1859], which is the element that characterizes all the methods beloging to the area of EC (Evolutionary Algorithms, EAs).

One of the fundamental elements of Darwin's theory of evolution is the idea of *natural selection*, thanks to which living beings would have evolved through the millennia from simple unicellular creatures to the vast variety of species that populate Earth today. Natural selection identifies five basic pillars, as the main responsibles of the evolution of the species:

- *Reproduction*, i.e. the fact that parents generate sons (or offspring), which allows species to survive.
- *Ability of Adaption to the Environment*, which characterizes, in different amounts, the individuals, and that has a direct implication on their probability of surviving and thus reproduce.

- *Hereditariness*, i.e. the fact that some characteristics of the parents are transmitted to their offspring.
- *Variation*, which is, in some sense, conflicting with hereditariness, and tells us that some characteristics of the offsprings are different from the ones of their parents, and this potentially allows the species to introduce elements of novelty along the generations.
- *Competition*, i.e. the idea that a limited amount of resources exists and, in order to survive, individuals have to compete to obtain them. Typically, the individuals that are more apt to the surrounding environment have more chances of obtaining resources and thus survive, while the others are likely to die and, in the long term, their species to estinguish.

The idea behind EC is to tranfer these concepts to computational optimization. GAs use a population of admissible solutions to the problem and function by means of an iterative process in which, at each generation, the solutions with the best fitness are probabilistically selected and genetic operators (typically crossover and mutation) are applied to those solutions, in order to create new and potentially better (in terms of fitness) solutions. The process is visualized in Figure 3.1. At a high level of

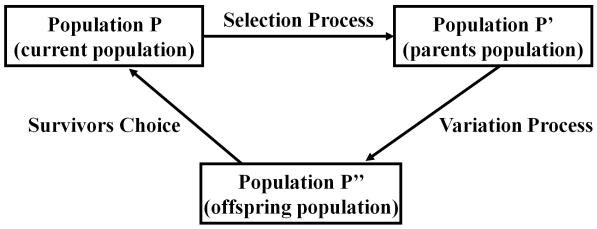


Fig. 3.1 Graphical representation of the general functioning of a GA.

abstraction, the GA process can be seen as the iteration of three phases of transformation of a population of solutions: selection phase, variation phase and survivors choice. The three transformed populations are usually called current population (P), which is initialized typically with random solutions in the beginning of the process, parents population (P') and offspring population (P''). Even though several variants exist, in the most standard version of GAs these three populations all have the same size N (i.e. they contain the same number N of individuals).

- The *selection process* mimics the principles of ability of adaption to the environment and competition of Darwin's theory of evolution. In simple words, solutions are chosen from P, based on their fitness, and copied in P'. No modification is applied to the solutions in this phase. The process is implemented as an iteration of N independent executions of a *selection algorithm*. The objective of a selection algorithm is to probabilistically chose one solution from the current population and insert it in the parents population. Some of the most well-known selection

algorithms are presented in Section 3.1. Iterating this process N times allows us to obtain a parents population of N individuals.

- The *variation process* mimics the principles of reproduction, hereditariness and variation of Darwin's evolution theory. It consists in the application of genetic operators to the individuals in the parents population, in order to modify them and generate new solutions. The individuals resulting from the application of the genetic operators are then inserted in the offspring population P''. Typical genetic operators are *crossover* and *mutation*, which mimic the homonymous biological processes, and are discussed in Section 3.2. The variation phase allows GAs to explore the search space.

- The *survivors choice* is the phase in which the new current population is created, in order to begin the next generation. The picking of the N individuals that will inhabit the current population in the next generation is usually made by choosing from the union of the $2N$ individuals that are in the offspring population and in the old current population. Various options are possible: from choosing the best N indviduals (in terms of fitness) from $P'' \cup P$ to applying the selection algorithms discussed in Section 3.1, or even to a simple random choice. One of the most used types of survivors choice, and the one used in this document from now on, is the so called *generational* choice of survivors, which consist simply in replacing the current population with the offspring population. In other words, with the generational choice of survivors the old current population is removed and the new current population becomes the offspring population ($P := P''$).

Before having a global vision of the functioning of GAs in Section 3.3, it is appropriate to understand the main bricks that compose the algorithm, i.e. the selection algorithms, presented in Section 3.1, and the genetic operators, presented in Section 3.2. The main difference between these two basic components of GAs is that, as in nature, selection acts on the *phenotype*, while genetic operators act on the *genotype* of individuals. In other terms, selection is based on fitness (or a function of it), and it is completely independent from the syntactic structure of the individuals. On the other hand, genetic operators act on the structure of the individuals and they are totally independent from fitness.

3.1 Selection Algorithms

Genetic operators are not applied to all the individuals in the population, but only to the ones that have been selected, according to the Darwinian principles of ability of adaption to the environment and competition. Selection algorithms enforce those principles, by choosing one individual from the current population. Several different selection algorithms exist, but all of them usually respect the following general properties:

1. Selection is probabilistic, in the sense that if the pick is repeated several times exactly in the same conditions, the outcome (i.e. the selected individual) in all these different experiments could change;

2. Selection has to be based on fitness, in the sense that given any two individuals i and j in the current population, if i has a better fitness than j, then i must have a higher probability than j of being selected;

3. All individuals in the population, even the worst ones, should have a probability different from zero of being selected. This allows the algorithm to maintain a certain degree of randomness, that is observable also in nature, where it is possible for an individual to survive not only because of its adaption to the environment, but also for purely fortuitous events;

4. Assuming that a selection algorithm is executed multiple times at different time intervals $1,2,...,N$, if an individual i has been selected at a given instant $t < N$, i remains in the current population also for the executions that will be performed at times $t+1,t+2,...,N$. In other words, all the executions are performed using the same current population P, and, after each execution, what is inserted in the parents population P' is a copy of the individual selected from P.

The previous principles are rather general, and one may imagine several different possible algorithms to implement them. The most used selection algorithms of GAs are:

- Fitness proportional selection (or "roulette wheel");
- Ranking selection;
- Tournament selection.

These three algorithms are presented in the continuation.

3.1.1 Fitness Proportional Selection (or "Roulette Wheel")

Let N be the number of individuals in population P (population size) and let $F = \{f_1, f_2, ..., f_N\}$ be the set of their fitness values. For any individual $i \in P$, the probability of selecting i, for maximization problems, is:

$$P(\text{sel } i) = p_i = \frac{f_i}{\sum_{j=0}^{N} f_j} \tag{3.1}$$

Given that the denominator is the same for all individuals (it is the sum of all the fitness values of the individuals in P), it is clear that the biggest f_i the higher the

probability of selecting i, so this selection algorithm respects Point 2 of the general selection algorithms principles discussed above. In some cases, in order to apply fitness proportionate selection, fitness values may need some rescaling. For instance, looking at Equation 3.1 it is straightforward to understand that if f_i is negative, then p_i is not a probability. If negative values exist in F, one may imagine, for instance, to rescale all fitness values by summing to each one a quantity larger or equal to $f_{min}+1$, where f_{min} is the minimum value in F, and use the rescaled values to calculate p_i in Equation (3.1). At the same time, it is easy to convince oneself that if for some i we have $f_i = 0$, then Point 3 of the selection algorithms principles is not respected. If one wants this principle to be respected, one may simply rescale all values in F by summing the same positive constant to each one of them. Last but not least, in order to extend this algorithm to minimization problems, one may simply consider $1/p_i$, instead of p_i as the probability of selecting and individual i.

Once we have established the probability of each solution P to be selected, implementing the selection is straightforward[1]. One simple way of simulating the algorithm is to consider a roulette wheel and partition in into N segments $\{s_1, s_2, ..., s_N\}$, one for each individual in the population, where the area of each s_i is directly proportional to f_i. When an individual needs to be selected, all we have to do is "play" a roulette game, letting a ball roll on the wheel. The selected individual will be the one corresponding to the segment in which the ball will terminate its trajectory. In other words, if the ball will stop in segment s_k, k will be the selected individual. In this way, it is clear that fitter individuals have a higher probability of being selected, because they are associated to a larger segment of the wheel (Point 2), but if all the individuals are associated to a segment, none of them has a probability equal to zero of being selected (Point 3). Executing the algorithm one more time is straightforward: it is enough to play another roulette game, letting the ball roll on the same wheel. In the new execution, the probabilities are the same (because we use the same wheel), but the outcome may be different (Point 4).

Example 3.1. Let us assume that we are trying to solve a maximization problem and we have a population $P = \{i_1, i_2, i_3\}$, where the fitness values are:

- $f_1 = 4$
- $f_2 = 12$
- $f_3 = 2$

The probabilities of selecting the various individuals are:

$$P(\text{sel } i_1) = 4/18 \approx 0.22$$
$$P(\text{sel } i_2) = 12/18 \approx 0.67$$
$$P(\text{sel } i_3) = 2/18 \approx 0.11$$

[1] The reader is referred to pages 34 and 49 for a discussion on how to implement a certain action with a given probability.

3.1.2 Ranking Selection

In this case, the selection algorithm begins by sorting the individuals in the population on the basis of their fitness, typically from the worst to the best. For each individual i, the selection probability is a function ϕ of the position occupied by i in the ranking. In principle, any monotonically growing function ϕ can be used. Typical cases are linear, logarithmic and exponential functions.

Example 3.2. Let us take back the case of Example 3.1. The ranking from the worst to the best is:

1. i_3
2. i_1
3. i_2

An example of a probability of selecting individual i_1, in case a simple linear function is used, is:

$$P(\text{sel } i_1) = 2/6 \approx 0.33$$

To calculate this probability, we have used the position occupied by i_1 in the ranking, divided by the sum of the ranking indexes of all the individuals.

The major difference between fitness proportionate selection and ranking selection should be clear to the reader: fitness proportionate selection is sensitive to the differences in fitness, while ranking selection is not: in ranking selection, only the position in the ranking matters. To convince oneself, one may consider the fitness values of the individuals in Example 3.1 and Example 3.2, and then change drastically the fitness values, without altering the ranking. The selection probabilities are likely to change for fitness proportionate selection, but remain the same for ranking selection. For instance, starting from the fitness values in Example 3.1, let us consider that the fitness value of i_2 is now equal to 1200, instead of 12. Let us now recalculate the probabilities of selecting the various individuals with fitness proportionate selection, and we will see that not only the probability of selecting i_2 becomes much larger compared to Example 3.1, but consequently also the probabilities of selecting the other two individuals become much smaller. At the same time, and with the same fitness modification, the probabilities of selecting the individuals with ranking selection remain as in Example 3.2.

3.1.3 Tournament Selection

With this technique, to select one individual, we do not have to evaluate the fitness of all the individuals in the population, but only a part of them. Every time that an individual has to be selected, a number k of individuals are chosen randomly, with uniform distribution, from the current population. The selected individual is

the best, in terms of fitness, among these k. The method is divided into two parts: the first part is completely independent from fitness and completely random (the choice of the k individuals taking part in the tournament), while the second part is completely deterministic and based on fitness (the selection of the best, among those k individuals).

Point 1 (at page 44) of the general selection algorithms principles is clearly respected, because the first part gives randomness to the technique. Point 2 is respected because of the second part, that uses fitness to ratify the winner of the tournament. Point 3 can be respected if we imagine that we execute the random choice of the k individuals with repetition, so that multiple copies of the same individual can participate in the tournament. In this case, all individuals, including the worst one in the population, have a positive probability of being selected. In fact, if the tournament is composed by k copies of the worst individual, it is the worst individual that will be selected. Finally, in order to iterate the method when the next individual needs to be selected, it is important to keep in mind that the *whole* algorithm needs to be repeated, including the generation of a new pool of k individuals, randomly chosen from the current population. This allows the different executions of the algorithm to be completely independent between each other.

Constant k is called *tournament size*, and it is an important parameter to establish the selection pressure, i.e. to establish how strong it is the probability of the best individuals to survive, at the expense of the others. A small value of k causes a low selection pressure; a large value of k causes a high selection pressure. All in all, tournament selection is an interesting algorithm for at least the following two reasons:

- In some conditions, it may be more efficient than the other studied selection algorithms, since it may relieve us from the burden of having to evaluate all the individuals in the population (but this is not the case if elitism is used, as explained in Section 3.3);
- It allows us to tune the selection pressure in a very simple way, i.e. by modifying a single number (the tournament size). It is interesting to point out that tuning the selection pressure is harder in ranking selection (where it can be changed by modifying a function, like ϕ, instead of a number), while it cannot be tuned at all in fitness proportionate selection.

3.2 Genetic Operators

When the selection phase is terminated, the parents population (P' in Figure 3.1) is inhabited by individuals that have been chosen as apt for surviving and mating. In the variation phase, they are modified, in order to generate new individuals. This modification is implemented by applying genetic operators. In the standard formulation of GAs [Holland, 1975, Goldberg, 1989] , two genetic operators exist: *crossover* and *mutation*. Crossover exchanges some syntactic characteristics of a set of (usually two) individuals, in order to generate offspring individuals, that are a

recombination of their parents. Mutation creates a new individual modifying (typically at random) a small portion of the syntactic structure of an existing individual. Mutation is usually identified as an *innovation* operator, since it introduces in the population new genetic material, while crossover is a *conservation* operator, since it uses already existing genetic material and works by recombining it in new ways.

As discussed previously, while selection is based on the phenotype of individuals, genetic operators are based on their genotype, i.e. on their syntactic structure. So, while it was possible to present the selection algorithms without even mentioning what representation the individuals have, and using only fitness, to discuss the genetic operators, we need to define the structure of the individuals. As mentioned at page 41, in GAs the individuals are represented as strings of characters of a prefixed length. Characters are normally chosen from a predefined *alphabet*, that can be, according to the different situations, finite, infinite, discrete, continuous, etc. In the next examples, we will limit ourselves to the simple, but very important[2], case of *binary* strings, i.e. strings whose characters belong to the set $\{0, 1\}$.

The first thing we have to learn about GAs genetic operators is that a vast amount of possible varieties of crossovers and mutations have been defined so far. At least in this very first approach to the algorithm, we will define and discuss only the standard versions of these operators, i.e. the ones originally proposed by J. Holland in his pioneering book [Holland, 1975] and later widely studied by D. Goldberg [Goldberg, 1989].

Standard crossover

Standard crossover of GAs works by taking in input two individuals (the *parents*), selecting at random a position (called *crossover point*) between two consecutive characters in the string (the point has to be the same for both individuals) and exchanging the substrings that are at the left and at the right of this position, thus generating two new individuals (the *offspring*). This process implements a very simple model of biological crossover, in which the offspring's DNA is formed by a part of the DNA of one parent, and the remaining part of the other parent.

Example 3.3. Let us consider the following two parents, where the different font in the two strings is used only to distinguish them more easily:

$$x = \mathbf{0\,1\,1\,0\,0\,0\,1\,1\,0}$$
$$y = 0\,0\,0\,1\,1\,1\,1\,0\,1$$

Let us assume that the randomly chosen crossover point is in position 4. In this case, the crossover between x and y generates the following offspring:

[2] A vast set of real world optimization problems can be solved using binary representation. Just as a matter of example, one could consider the representation used for the knapsack problem and its numerous practical applications, discussed in Example 2.1 at page 12.

$$z = \mathbf{0\,1\,1\,0}\,1\,1\,1\,0\,1$$
$$w = 0\,0\,0\,1\,\mathbf{0\,0\,1\,1\,0}$$

As we can observe, child z inherited the left substring from parent x and the right substring from parent y, while child w inherited the left substring from parent y and the right substring from parent x.

Standard mutation

Standard mutation of GAs works by considering one individual as input, and by iterating on all the positions in the string (also called *genes*, inheriting the terminology of biological evolution). For each gene, the current character is replaced with another randomly chosen possible character, with a given probability p_m, called *mutation rate*. Obviously, the higher p_m, the more desruptive the mutation, that can, as a limit case, even modify the entire string representing the individual. Given that, usually, we do not want mutation to be very disruptive, it is typical to use very small values of the mutation rate, as it happens also in nature, where mutation is generally a rather rare event.

Example 3.4. Let us assume that we want to mutate the following individual:

$$x = 1\,0\,0\,0\,1\,0$$

Potentially, any individual could be generated by the mutation of individual x. Let us assume, for instance, that we are using a mutation rate $p_m = 0.33$ (which, generally speaking, has to be considered a very high value). In this case, we may expect that, on average, two of the six characters that form x will be modified (this is, of course, a probabilistic event, and something totally different may happen). Let us assume, for instance, that the event of modifying the gene is verified only for the third and the fifth character. In this case, the mutation of x generates the individual:

$$x = 1\,0\,\mathbf{1}\,0\,\mathbf{0}\,0$$

where the modified characters have been written is bold for the sake of visibility.

Before moving on with a discussion of the general functioning of GAs, it is worth spending some more minutes reflecting on what a probabilistic event is, and how it can be implemented. As already discussed page 34, the following sentence:

$$\text{Perform an action } \mathscr{A} \text{ with probability } \mathscr{P}$$

Can be rewritten, in terms that are "closer" to an algorithmic implementation, as:

If a random number, extracted with uniform probability from $[0,1)$,
is smaller than $\mathscr{P}$, **then** perform action $\mathscr{A}$, **else** do not perform action $\mathscr{A}$

If we try to execute such an event several times, in some cases we might perform action $\mathscr{A}$, and in some cases we might not perform it. In fact, the event depends on the value of a random number that is, by definition, not predictable *a priori*[3].

All this considered, a realistic implementation of the standard mutation of an individual x, represented as a string of characters of length ℓ, and using a mutation rate equal to p_m, could be given by the pseudo-code of Algorithm 3, where Rand$[0,1)$ is a random number extracted with uniform probability from $[0,1)$.

Algorithm 3: Implementation of standard GA mutation of an individual x, represented as a string of characters of length ℓ, using a mutation rate equal to p_m.

```
for each i from 1 to ℓ do
    if Rand[0,1) < p_m then
        Replace the i^th character of individual x with a random admissible character;
    end
end
```

Probabilities of the Operators and Reproduction

As we have seen, mutation has a probability associated, p_m. It corresponds to the probability of modifying each gene in the string that represents an individual, and it bestows on the operator some stochasticity, in the sense that if we apply the operator several times in the same conditions, the outcome can be different. Actually, all the genetic operators of GAs are stochastic. This reinforces the analogy between the genetic operators of GAs and the corresponding biological operators and, generally speaking, improves the exploration ability of the algorithm.

Under this perspective, also a probability associated to crossover exists, the *crossover rate* p_c. The crossover rate represents the probability of applying crossover to two parents. In case this probabilistic event does not happen, another operator is applied to those two individuals, called *reproduction*. Reproduction is simply the copy unchanged of the parents in the offspring population. Thanks to reproduction, individuals of different "ages" can coexist in the same population, and possibly mate in the next generations.

Crossover rate p_c and mutation rate p_m are two of the numerous parameters of GAs, that will be discussed later in this chapter.

[3] Given that probabilistic events are discussed in many parts of this document (modifying one gene in GAs mutation is just an example of it), the reader is invited to spend some more time trying to convince herself about this concept before continuing with the reading.

3.3 General Functioning of Genetic Algorithms

The pseudo-code of the standard version of GAs [Holland, 1975, Goldberg, 1989] is shown in Algorithm 4. Before analysing this algorithm in detail, it is worth remind-

Algorithm 4: The pseudo-code of standard generational GAs.

1. Create an initial population P of N individuals (typically generated at random);
2. **repeat until** (termination condition)
   - 2.1. Calculate the fitness of each individual in P; // can be avoided in some cases
   - 2.2. Create an empty population P';
   - 2.3. **repeat until** (population P' contains N individuals)
     - 2.3.1. Choose a genetic operator between *crossover* (with probability p_c) and *reproduction* (with probability $1 - p_c$);
     - 2.3.2. Select two individuals from P, using a *selection* algorithm;
     - 2.3.3. Apply the genetic operators chosen at Point 2.3.1 to the individuals selected at Point 2.3.2 ;
     - 2.3.4. Apply *mutation* to the individuals obtained at Point 2.3.3;
     - 2.3.5. Insert the individuals obtained at Point 2.3.4 in population P';

     end
   - 2.4. $P := P'$; // Replace population P with population P' (generational choice of survivals)

   end
3. **return** the best individual in P;

ing that this is only one of the possible existing variants of GAs, that can be found in the numerous existing bibliographic references. For instance, as it is clear from point 2.4 of the pseudo-code, this algorithm uses generational choice of survivals (in other words, it describes generational GAs), which, as we have already discussed, is only one of the possible existing processes for creating a new population, joining the old current population and the offspring population.

Looking at Algorithm 4, and comparing it with the schema represented in Figure 3.1, one major difference can be seen: in Figure 3.1, three populations (P, P' and P'') appear, while only two populations (P and P') are used in Algorithm 4. The reader is invited to reflect on the fact that, in spite of this difference, Algorithm 4 and the schema of Figure 3.1 are functionally completely equivalent. In fact, the process represented in Figure 3.1 assumes that once the parents are selected, they are stored in a population P', and only successively they are used to apply the genetic operators and generate the offpring. The role of population P' in Figure 3.1 is "passive": P' is only a "container", where individuals that already existed are copied, waiting

to be used later. In Algorithm 4, instead, once parents are selected, genetic opertors are immetiately applied. There is no storage of the parents. So, Algorithm 4 allows us to save the memory space needed to store a population (and given that, in real cases, populations usually contain numerous individuals, this saving can be significant). The schema in Figure 3.1 was useful to conceptually separate the phases of selection and variation, but when GAs have to be implemented, it is generally more efficient to follow Algorithm 4.

The pseudo-code in Algorithm 4 works for an even population size (i.e. when N is an even number), since it inserts a pair of individuals in P' at each generation. When the population size is odd, some dodge is needed to allow the algorithm to fill P' with N individuals. Several variants exists, among which: selecting one individual, mutating it and inserting it into P' without applying crossover; in one application of crossover, just insert one child in the population instead of both (for instance, the best of the two in terms of fitness, or a child picked at random); inserting into P' one random individual, etc.

Algorithm 4 has a potential flaw, since it is possible to "loose" the best individual in the population. This problem can be avoided by copying, at each iteration, one or more individuals unchanged in the next population. For those individuals, that are usually the best ones in the current population, the process selection-crossover-mutation is *de facto* bypassed, and those individuals simply survive in the next generation. This process is called *elitism*, and it is a very common practice in GAs, in particular when the population size is reasonably large.

When we try to solve a problem using GAs, first of all, we have to find a representation for the solutions, using strings of characters of prefixed length. After that, the following *parameters* need to be set:

- Length of the strings representing the individuals;
- Population size;
- Selection algorithm;
- Crossover rate p_c;
- Mutation rate p_m;
- Maximum number of generations;
- Presence/Absence of elitism (and, in case elitism is used, size of the elite).

Even though there are no rules for an optimal setting of these parameters, and appropriate values are strongly problem dependent, the following informal guidelines can be identified:

- The length of the strings representing the individuals is usually not a choice that we have, but it is part of the definition of problem. For instance, the reader is invited to consider the knapsack problem, discussed in Example 2.1 at page 12, where the length of the string is equal to the total number of available objects, or the travelling salesperson problem, discussed in Example 2.2 at page 14, where the length of the string is equal to the number of cities to visit. In other words, when we have defined an appropriate representation for the individuals, this is usually a parameter of which we do not have to care.

- For the population size, it is clear that one basic rule of thumb exists: the larger the population, the better the exploration power of the algorithm. Nevertheless, experimental evidence also tells us that a threshold exists, beyond which keeping adding individuals to the population does not compensate for the increment in the computational effort. Finding that limit, by means of a set of preliminary experiments, can be a good practice for appropriately setting this parameter.

- No golden rule exists for choosing the most appropriate selection algorithm, and all the existing ones have their pros and cons. However, tournament selection presents a set of advantages that could make it our first bet: if no elitism is used, the choice of tournament selection may allow us to avoid evaluating the fitness of all the individuals in the population at each generation (in other words, if we use tournament selection and no elitism, then it is possible to avoid executing Point 2.1 in Algorithm 4). Furthermore, tournament allows us to tune the selection pressure by modifying one simple number, the tournament size (which becomes a further parameter to set, in case tournament is the chosen selection algorithm).

- Concerning the crossover and mutation rate, it is customary to work with high values of p_c (say from 0.8 to 1) and low values of p_m (say from 0 to 0.2). This is similar to what happens in nature, where crossover is an event with much higher probability than mutation. This is also the configuration that allows GAs to obtain the best results most of the times. However, it is not infrequent, as we will see in the continuation of this chapter, to have problems where crossover is much less effective than mutation. In such cases, the rates of these two operators may be inverted.

- Last but not least, for the maximum number of generations, similar considerations can be made to those that we made for the population size. Also in this case, in general, the more we let the algorithm evolve, the more likely it will be to find good quality solutions. However, also in this case, experimental evidence tells us that there is a point beyond which evolving for more generations does not compensate, in terms of the quality of the solutions, for the computational effort it requires. Once again, an experimental investigation for this threshold can be a good practice for approriately setting this parameter.

When setting the parameters of GAs, it can be appropriate to consider that the evolution of the algorithm can be often partitioned into two, ideally distinct, phases: *exploration* and *exploitation*. In exploration, the algorithm is searching for new solutions, exploring different regions of the search space, while exploitation means to focus on a particular region of the sarch space, in which particularly promising solutions have been found during exploration, using already existing solutions and make refinements to further improve their fitness. For this reason, having dynamic parameters, that change during the course of the evolution, can be an appropriate strategy.

Example 3.5. (Functioning of GAs). Let us consider the simple problem of maximizing the function $f(x) = x^2$, with $x \in \mathbb{N}$ & $0 \leq x \leq 31$. As usual in GAs, the first step has to be to decide a representation for all admissible solutions as strings of prefixed length. In this case, the admissible solutions are the natural numbers included between 0 and 31. In order to reresent these numbers in binary, we need up to a maximum of 5 bits. So, completing the numbers that need a smaller number of bits with zeros at the right, we can represent all solutions using strings of bits of size 5.

Let us assume, for simplicity, that we use a population of size equal to 4, fitness proportionate selection, p_c equal to 1 and p_m equal to 0, no elitism and generation choice of survivals. The first step is the random generation of an initial population. In case of binary representation, like in this example, generating an initial population of randomly generated individuals is straightforward: to create 4 strings of bits of length 5, we simply flip a coin 20 times, using bit 1 if the outcome is head and 0 if it is tail for each one of the bits that have to be initialized. Let us assume, for instance, that the randomly generated population is: $P = \{01101, 11000, 01000, 10011\}$. Table 3.1 reports the decimal value and the fitness for each individual in P, along with an approximation of the value of the probability of selection. For each solution, the

Table 3.1 Initial population for Example 3.5. The first column contains an identificator for each individual, the second one the binary representation, the third one the decimal representation, the fourth one the fitness and the fifth one the probability of selection.

Individual	Binary	Decimal	Fitness	Probability of selection
x_1	01101	13	169	0.14
x_2	11000	24	576	0.49
x_3	01000	8	64	0.06
x_4	10011	19	361	0.31

fitness is given by the square of the value it represents, while, given that we are using fitness proportionate selection, the probability of selection is given by the fitness divided by the sum of all the fitnesses of all the individuals in the population. For instance, for the case of x_1, we have:

$$P(\text{sel } x_1) = \frac{169}{169 + 576 + 64 + 361} \approx 0.14$$

Both from the fitness values and the selection probabilities, we can clearly see that x_2 is the best individual in P, while x_3 is the worse one. As expected, all individuals including x_3 have a probability different from zero of being selected. Figure 3.2 reports a representation of a roulette wheel that could be used for simulating the selection process in this case. The reader is invited to consider the areas of the different slices of the wheel proportional to the fitness of the individual they have been assigned to. Let us now assume that, at the first roulette game, the ball stops in the area assigned to individual x_1, and at the second one in the area of individual x_2. Then, these two

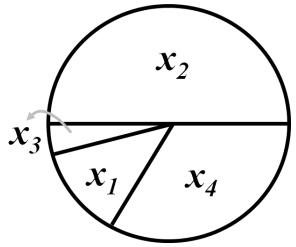


Fig. 3.2 Roulette wheel used for the initial population of Example 3.5.

individuals are the first pair that is selected as parents. Given that p_c is equal to 1, crossover between x_1 and x_2 has to be performed. This implies allignment of the two strings representing the individuals, random choice of a common crossover point, and substrings swap, in order to generate two new individuals. If we assume that the crossover point is point number 4, the application of the crossover is reported in Figure 3.3. As the figure shows, this crossover has generated individuals:

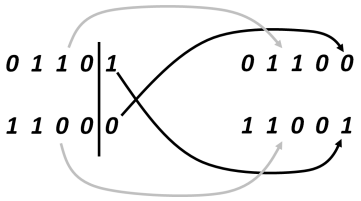


Fig. 3.3 A possible crossover (using the fourth crossover point) between individuals x_1 and x_2 of Example 3.5.

$$x_5 = 01100$$
$$x_6 = 11001$$

Given that p_m is equal to 0, we do not apply mutation, and so we insert x_5 and x_6 into the new population P'. So, P' has now two individuals. Two further individuals are needed to reach the target population size of 4. So, two further parents need to be selected. Using again the roulette wheel of Figure 3.2, and playing two further games, let us assume that the first time the ball stops in the area of x_2 and the second time in the area of x_4. So, x_2 and x_4 will be the parents of a new crossover. Assuming that the randomly chosen crossover point this time is point number two[4], the results of this crossover is shown in Figure 3.4. The individuals generated by this crossover are:

$$x_7 = 11011$$
$$x_8 = 10000$$

[4] Remark that the crossover point is randomly generated at each crossover event, so it is likely to be different from the one that was used previously.

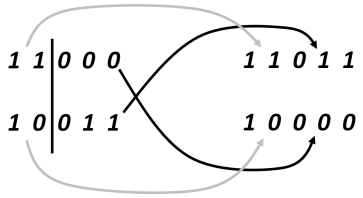


Fig. 3.4 A possible crossover (using the fourth crossover point) between individuals x_1 and x_2 of Example 3.5.

Given that p_m is equal to 0, once again no mutation is applied and these individuals are inserted in P', that now contains 4 individuals. This allow us to terminate the first generation of the algorithm, allowing P' to become the new current population P. Before analysing this new population, let us observe the behavior of selection in the first generation. We can see that:

- the individual with the best fitness (i.e. x_2) was selected twice for mating;
- the worst individual (i.e. x_3) was never selected, and so it got extinguished.

Even though these events are probabilistic, and so running the same experiment in the same conditions, different events may happen, this situation is rather typical: the fittest individuals in the population, being the ones with the highest selection probabilities, usually have several possibilities of engaging in a crossover, while the worst individuals in the population are likely to have very few, or even no chances of mating. Let us now visualize the new population, in Table 3.2. Comparing the

Table 3.2 The second population in Example 3.5. The first column contains an identificator for each individual, the second one the binary representation, the third one the decimal representation and the fourth one the fitness.

Individual	Binary	Decimal	Fitness
x_5	01100	12	144
x_6	11001	25	625
x_7	11011	27	729
x_8	10000	16	256

second population with the first one, reported in Table 3.1, we can notice that, in this case, both the best fitness and the average fitness in the population, after one generation, have improved.

3.4 Theory of Genetic Algorithms

Observing the evolutionary process of GAs, a main question arises natural: are GAs a good optimization algorithm? In other words: will this process allow us to find a globally optimal solution, or a reasonable approximation of it, for a given problem? In order to answer this question, other questions need to be answered first: what is the information that is processed by a GA? What does a GAs population contain generation after generation? In order to answer these questions, we need to deeply understand the way GAs work.

One of the basic hypothesis of GAs is a strong relationship between genotype and phenotype. Putting it in simple terms, we hypothesize that if an individual is good, i.e. it has a good fitness, then it has some structural/syntactic characteristics that make it good. We can also imagine that, potentially, several different structural characteristics that can have a beneficial effect on fitness. So, assume we select two individuals with good fitness: x_1 and x_2. If x_1 is good because it possesses a beneficial characteristic a and x_2 is good because it has another beneficial characteristic b, then we hope that if we remix the structures of x_1 and x_2 by means of crossover, the offspring will contain *both* characteristics a and b, and thus it will have a better fitness than both x_1 and x_2. At the same time, what we expect from mutation is that it acts on those parts of the chromosome of an individual that are not beneficial, in order to improve them. This leads to yet another question: how can we define a structural/syntactic characteristic of an individual? The first step to answer this question is to analyse data from some examples. This will allow us to formulate hypothesis, that later will be proven.

Let us take back the initial population of Example 3.5, represented in Table 3.1. Observing this population, for instance, one may hypothesize that having a bit equal to 1 in the leftmost position of the string can be a beneficial characteristic. Indeed, also considering larger samples, it is possible to observe that individuals with a 1 in the leftmost position always have a better fitness than individuals having a 0 in the leftmost position. So, now let us ask to ourselves: how can we represent this characteristic with a formal notation, that can be useful? One possible answer was given by Holland in his book [Holland, 1975], by means of the introduction of the concept of *schema*. For instance, the characteristic of *having a 1 in the leftmost position* can be represented as:

$$1 * * * *$$

In informal terms, the above notation can be interpreted as having a 1 in the leftmost position, independently from the information that appears in the other positions. The general definition of schema is:

Definition 3.1. *(Schema).* A schema is a set of individuals, that share the fact of having the same characters in some positions.

A rather simple way of representing a schema is by means of a string, where the characters come from the same alphabet used to build the individuals, plus one further "special" symbol, called the *don't care* symbol, and indicated by means of

a $*$. So, in the case of individuals that are represented as binary strings, schemata can be represented as strings wher each symbol comes from the alphabet $\{0, 1, *\}$. The meaning of the concept of schema can be better understood, if we think of it as a *pattern matching tool*: in the case of binary representation, an individual "matches" a schema (or, in other words, an individual belongs to a schema) if, in each corresponding position, a 1 matches a 1, a 0 matches a 0, and a $*$ matches either a 1 or a 0.

Example 3.6. Schema $*0000$ represents the set of individuals $\{00000, 10000\}$. In other terms, we can write:

$$*0000 = \{00000, 10000\}$$

Both individuals 00000 and 10000, in fact, match schema $*0000$. In fact, whenever in the schema there is a character different from $*$, the individuals have the same character in the same position. If the binary alphabet is used to represent the individuals, then the reader is invited to agree that no other individual different from 00000 and 10000 matches schema $*0000$. A schema represents the set containing all and only the individuals that match it. Analogously, we have, for instance:

$$*111* = \{01110, 01111, 11110, 11111\}$$

The larger the number of $*$ symbols in the schema, the larger the cardinality of the set of individuals it represents (i.e. the number of strings that match the schema).

Remark that $*$ is just a meta-symbol: it is not explicitly used by GAs. However, we hypothesize that, implicitly, GAs also work on schemata, and in the next pages, we are interested in trying to understand what is the effect of a GA on schemata. But before doing that, let us first perform some simple "exercises", that can help us acquire familiarity with the concept of schema.

- *Question*. If individuals can be represented as binary strings of length ℓ, how many possible schemata exist?
  Answer. The number of existing schemata is 3^ℓ, in fact the schemata are all the possible strings that can be build from the "extended" alphabet $\{0, 1, *\}$.

- *Question*. To how many different schemata does an individual belong to?
  Answer. Each binary string x is member of 2^ℓ different schemata. In fact, in each position of the string can have the value of the bit that actually appears in x in that position, or the $*$ symbol. For instance, string 111 belongs to the following schemata: $\{111, *11, 1*1, 11*, **1, 1**, *1*, ***\}$, corresponding to all the strings from the alphabet $\{0, 1, *\}$ that have, in each position, either 1, or $*$. Notice that a string without any *don't care* symbol can also be considered a schema, containing only one individual.

- *Question*. Let us consider a population of n individuals, each of which, as in the previous questions, represented as a string of bits of length ℓ. How many different schemata can the population (implicitly) contain? In order to answer

this question, one has to keep in mind that, by definition, we say that a population "contains" a schema if it contains at least one individual belonging to that schema.

Answer. The population can contain a number of schemata included between 2^ℓ and $(n \times 2^\ell)$, depending on the *diversity* of the population: if all the n individuals in the population are identical between each other, then the population contains only 2^ℓ schemata, that are the ones matching a single string, as we have seen in the previous question. On the other hand, if all the individuals in the population are different between each other, then the population can contain up to a maximum of $(n \times 2^\ell)$, i.e. 2^ℓ schemata for each one of the different strings in the population. Of course, this is just an upper bound to the possible number of schemata, since, given two individuals x and y in the population, generally the set of schemata matching x and the one matching y have a not empty intersection.

Before investigating how the concept of schema can be important to understand the dynamics of GAs, let us define two important concept related to schemata.

Definition 3.2. *(Order of a Schema)* The *order* of a schema H, denoted as $o(H)$, is the number of symbols different from $*$ that appear in H.

Example 3.7. For instance, we have:
$o(011*1**) = 4$
$o(**0**1*) = 2$

Definition 3.3. *(Length of a Schema)* The *length* of a schema H, denoted as $\delta(H)$, is the distance between the leftmost and the rightmost positions that contain a symbol different from $*$ in H.

Example 3.8. Considering the same schemata as in the previous example, we have:
$\delta(011*1**) = 5-1 = 4$
$\delta(**0**1*) = 6-3 = 3$

In order to answer the questions that we asked at the beginning of this section, we now ask other questions: of the schemata that are (implicitly) contained in a population, how many, and which onesm will be used by the GA, and how? How many of them will survive in the next generation? What is the modification in the number of individuals in the population matching a given schema, from a generation to the next? Answering these questions will be insightful for understanding the functioning of GAs.

In order to answer these questions, one may consider that the GA process is nothing but an iteration of a selection-crossover-mutation set of events. For this reason, it could be useful to study the effect of selection, crossover and mutation on the number and type of schemata that are contained in a population from one generation to the next. In particular, we are interested in understanding how many, and which ones, of the schemata will be represented by a steadily growing number of individuals, and which one will disappear from the population during the evolution. One interesting point is that, in GAs, selection, crossover and mutation are independent events. For this reason, the effect of these three events on the schemata can

be studied separately, and then it will be straightforward to obtain their combined effect.

Effect of Selection on the Schemata of a Population

In this section, we study the effect of *fitness proportionate selection* on the schemata of a population[5]. Focusing, without lost of generality, on maximization problems, we are interested in studying the *expected value* of the number of individuals belonging to a schema H in the population, at a given generation t. We indicate with $m(H,t)$ that expected value, and we want to study how this value changes with t. We have:

$$m(H,t+1) \;=\; m(H,t) \cdot n \cdot \frac{f(H)}{\sum_j f_j} \tag{3.2}$$

where $f(H)$ is the average fitness of the individuals matching H and belonging to the population; n is the population size; for each individual j, f_j is the fitness of j and the summation in the term $\sum_j f_j$ runs on all the individuals in the population.

In order to give an intuitive interpretation of Equation (3.2), let us begin with and attempt to understand the following term:

$$\frac{f(H)}{\sum_j f_j} \tag{3.3}$$

Knowing that, for each individual i in the population, the probability of selecting i with fitness proportionate selection is equal to $\frac{f_i}{\sum_j f_j}$, the term in Equation (3.3) is the average probability of selecting an individual that matches H, with *one step* of the selection algorithm. However, to generate the parents' population, the algorithm has to be repeated for n independent executions. This is why the term in Equation (3.3) is multiplied by n in Equation (3.2). So, the term:

$$n \cdot \frac{f(H)}{\sum_j f_j}$$

represents the average probability of selecting an individual that matches H in one generation. If we multiply this term by the expected value of the number of individuals matching H in the population at time t, what we obtain is the same expected value at time $t+1$, *assuming that only selection is applied* (remember that we are now considering only the effect of selection on the schemata of a population. The effect of the genetic operators will be studied later).

Equation (3.2) does not seem to give us much information. However, it is possible to make some developments of that equation, in order to turn it into a more

[5] Analogous results exist also for other types of selection algorithms, but they are out of the scope of this document.

insightful form. Let us begin by defining the average fitness of the individuals in the population $\tilde{f}$, that is, straightforwardly:

$$\tilde{f} = \frac{\sum_j f_j}{n}$$

From which, it immediately follows that:

$$\frac{n}{\sum_j f_j} = \frac{1}{\tilde{f}} \tag{3.4}$$

Now, replacing Equation (3.4) in Equation (3.2), we obtain a new form for Equation (3.2), given by Equation (3.5):

$$m(H,t+1) \;=\; m(H,t) \cdot \frac{f(H)}{\tilde{f}} \tag{3.5}$$

Equation (3.5) is more informative than Equation (3.2). In particular, from Equation (3.5), we can understand that the expected value of the number of individuals matching a schema H in the population *increases* if the average fitness of the individuals matching H is larger than the average fitness of all the individuals in the population, and *decreases* otherwise. In other words, schemata with an average fitness larger than average of the population "grow up" from one generation to the next (i.e. improve the number of representatives in the population), while the others "shrink", eventually tending towards extinction. If on the one hand, it could have been intuitive to imagine that schemata with a good average fitness would grow and the others would shrink, it was not so intuitive *a priory* to say that the threshold between the two types of the schemata is given by average fitness of the individuals in the population.

We are now interested in understanding *at what speed* the good schemata grow up, while the others shrink. To obtain this information, let us assume that a schema H has an average fitness larger than the average population fitness by a quantity equal to $c \cdot \tilde{f}$, where c is a constant. In other words, let us assume that:

$$f(H) = \tilde{f} \;+\; c \cdot \tilde{f} \tag{3.6}$$

Replacing Equation (3.6) in Equation (3.5), we obtain:

$$m(H,t+1) \;=\; m(H,t) \cdot \frac{\tilde{f} \;+\; c \cdot \tilde{f}}{\tilde{f}} \;=\; (1+c) \cdot m(H,t) \tag{3.7}$$

Starting from $t = 0$, and assuming a stationary value of c, we have:

$$
\begin{aligned}
&m(H,1) = m(H,0)\cdot(1+c)\\
&m(H,2) = m(H,1)\cdot(1+c) = m(H,0)\cdot(1+c)\cdot(1+c) = m(H,0)\cdot\ (1+c)^2\\
&m(H,3) = m(H,2)\cdot(1+c) = m(H,0)\cdot(1+c)^2\cdot(1+c) = m(H,0)\cdot\ (1+c)^3\\
&\qquad ...
\end{aligned}
$$

Extending the process to a generic time t, we have:

$$m(H,t) \ = \ m(H,0)\ \cdot\ (1+c)^t \tag{3.8}$$

Equation (3.8) clearly tells us that the growing speed of good schemata (as well as the shrinking speed of bad schemata) is exponential. In other words, selection allocates an exponentially increasing amount of individuals to schemata with larger average fitness than the average of the population, and an exponentially decreasing amount of individuals to schemata with smaller average fitness than the average of the population. It is clear that, in this way, bad schemata (i.e. those with smaller average fitness than the average of the population) will rapidly tend to extinguish.

Effect of Crossover on the Schemata of a Population

Selection alone does not do anything to promote exploration of new areas of the search space, since it does not produce any new individuals inside the population. On the other hand, crossover creates new individuals remixing structures that were already part of the population. To have a simple intuition of the effect of standard GA crossover on schemata, let us consider a simple binary string A of length ℓ equal to 7, and two schemata H_1 and H_2 that are matched by this string:

$$
\begin{aligned}
A &= 0111000\\
H_1 &= *1****0\\
H_2 &= ***10**
\end{aligned}
$$

Let us assume that the individual represented by string A has been selected for mating. Given that the legth of the string ℓ is equal to 7, there are 6 existing crossover points (i.e. points between two consecutive bits, that can be chosen to the the breaking point, to decompose the string into two substrings that will be exchanged). Let us suppose that we ussually a completely balanced dice to decide the crossover point. Let us assume, for instance, that the dice gives, as outcome, crossover point number 3. So, the strings will be partitioned into the substring until the third digit, and the rest of the string, as follows:

$$
\begin{aligned}
A &= 011|1000\\
H_1 &= *1*|***0\\
H_2 &= ***|10**
\end{aligned}
$$

As we can see, if we ignore the event of chosing a partner of A that matches one of the two substrings of H_1, H_1 will be "destroyed" by this crossover event. In fact, this crossover will create a child that has a bit equal to 1 in the second position, but does not have a bit equal to 0 in the seventh position, and a child that has a bit equal to 0 in the seventh position but does not have a bit equal to 1 in the second position. In other words, none of the offspring will match H_1. The situation is different for schema H_2: H_2 will survive this crossover event, because one of the two children will have both a bit equal to 1 in the fourth position and a bit equal to 0 in the fifth position.

Even though in the previous example we have used one particular crossover point, it is not hard to convince oneself that in general it is less probable that H_1 will survive crossover, compared to H_2. In order to quantify this intuition, let us count the number of crossover points that will destroy H_1. Unless the lucky event in which crossover point number 1 is chosen, in all other cases the two characters different from $*$ will be separated by crossover. So, the probability of destroying H_1 is:

$$P(\text{destr } H_1) = \frac{5}{6} \tag{3.9}$$

We now point out that 5 (i.e. the number of crossover points that, if chosen, cause the destruction of H_1) is equal to the length of H_1 (see definition 3.3), while 6 is the number of possible crossover points, which is equal to the number of characters minus one. So, we hypothesize that Equation (3.9) can be generalized by:

$$P(\text{destr } H_1) = \frac{\delta(H_1)}{\ell - 1}$$

Of course, from the probability of destrying H_1, it is immediate to obtain the probability of H_1 surviving crossover:

$$P(\text{surv } H_1) = 1 - \frac{\delta(H_1)}{\ell - 1} = \frac{1}{6}$$

Once again, it is worth mentioning that this probability has been calculated disregarding the event of a "lucky" crossover, in which H_1 survives thanks to the partner of A.

If we now consider schema H_2, we can see that the only way of destroying it is by choosing the fourth crossover point. In all other cases, H_2 will survive crossover. So:

$$P(\text{destr } H_2) = \frac{1}{6}$$

Once again, the length of H_2 is equal to 1 (see definition 3.3), so we can write:

$$P(\text{destr } H_2) = \frac{\delta(H_2)}{\ell - 1}$$

Of the probability of H_2 surviving crossover is:

$$P(\text{surv } H_2) = 1 - \frac{\delta(H_2)}{\ell - 1} = \frac{5}{6}$$

The observations made so far can be generalized: if crossover is performed with a probability equal to p_c, the probability of any schema H to survive to crossover can be expressed by the following lower limit:

$$P(\text{surv } H) \;\geq\; 1 \;-\; p_c \,\frac{\delta(H)}{\ell - 1} \tag{3.10}$$

Equation (3.10) is a generalization of the previous ones, and it is expressed as a lower limit because the right side of the equation takes into account only one crossover event (and schema H may survive also because of the other crossover events in the same generation) and because, as we already mentioned, it does not take into account "lucky" crossovers, where the schema survives thanks to the genetic material made available by the partner. We also point out that the right side of Equation (3.10) becomes like the previous equations if $p_c = 1$.

Given that we can assume that selection and crossover are two mutually independent events, we can obtain the combined effect of selection and crossover on the schemata of a population, by simply multiplying the two single effects, obtaining:

$$m(H,t+1) \;\geq\; m(H,t) \cdot \frac{f(H)}{\tilde{f}} \;\left[1 \;-\; p_c \,\frac{\delta(H)}{\ell - 1}\right] \tag{3.11}$$

With this new expression, we can notice that, if only selection and crossover are used, the factors that influence the growth (respectively, the shrinking) of a given schema in the population are if the average fitness is larger (respectively, smaller) than the population average and if the schema has a small (respectively, large) length. More specifically, schemata with:

- superior average fitness than the population average;
- small length

will steadily increase the number of representatives in the population from one generation to the next. In the multiplicative factor that estimates the probability of survival of a schema after crossover, there is no expression of time t. So, crossover does not change the fact that the increase in the number of representatives has an exponential speed.

Effect of Mutation on the Schemata of a Population

Standard GAs mutation consists in the random perturbation of each and every character in a string, with a given (and typically low) probability p_m. For a schema to

survive a mutation event, it is needed that all the positions with a character different from $*$ do *not* change. By definition (see Definition 3.2), the number of positions containing a character different from $*$ is the order of the schema. Given that each character in the string survives with a probability equal to $1-p_m$, and given that the event of mutating a character in, in general, independent from the mutation of the others in the string, the probability that a schema H will survive a mutation event is:

$$P(\text{surv } H) \;=\; \underbrace{(1-p_m)\,(1-p_m)\,...\,(1-p_m)}_{o(H)\text{ times}} \;=\; (1-p_m)^{o(H)}$$

If $p_m << 1$, which is typical in GAs, this probability can be reasonably approximated by:

$$P(\text{surv } H) \;=\; 1 \;-\; o(H)\;p_m$$

Intuitively, this equation is telling us that, as it may be intuitive, schemata with small order have a higher probability of surviving a mutation event than large order ones.

Schema Theorem

Joining together the effect on schemata of selection, crossover and mutation, which can be considered mutually independent events, we have that a generic schema H receives an expected number of representative individuals in the next generation of a GA given by:

$$m(H,t+1) \;\geq\; m(H,t)\;\cdot\;\frac{f(H)}{\tilde{f}}\;\left[1\;-\;p_c\;\frac{\delta(H)}{\ell-1}\right]\left[\,1\;-\;o(H)\;p_m\right] \tag{3.12}$$

For simplicity, Equation (3.12) can be rewritten as:

$$m(H,t+1) \;\geq\; m(H,t)\;\cdot\;\frac{f(H)}{\tilde{f}}\;\left[1\;-\;p_c\;\frac{\delta(H)}{\ell-1}-\;o(H)\;p_m\right] \tag{3.13}$$

where the term $(p_c \cdot p_m \cdot \frac{\delta(H)}{\ell-1} \cdot o(H))$ was ignored, because considered very small.

Intuitively, what Equation (3.13) is telling us is that, in GAs, schemata that are:

- short;
- of low order; and
- with an average fitness better than the average of the population

receive an exponentially growing number of representatives in the population, from one generation to the next. The information these schemata contain is commonly

called *building blocks*. Building blocks can be imagined as short and compact sequences of characters (different from $*$), such that the presence of those characters in those specific positions in an individual brings a benefit to the individual in terms of fitness. So, in a nutshell, the schema theorem is telling us that GAs work by maintaining and proliferating building blocks in the population.

At the same time, Equation (3.13) is also telling us is that schemata that are:

- long;
- of high order; and
- with an average fitness worst than the average of the population

receive an exponentially decreasing number of representatives in the population, from one generation to the next, until they eventually estinguish.

Example 3.9. (An empirical validation of the Schema Theorem).

Let us take back the case of Example 3.5. In other words, let us consider the problem of maximizing the function $f(x) = x^2$, with $x \in \mathbb{N}$ & $0 \leq x \leq 31$. As we might remember from Example 3.5, the initial (randomly generated) population was:

$$x_1 : 01101$$
$$x_2 : 11000$$
$$x_3 : 01000$$
$$x_4 : 10011$$

Let us now consider the following three schemata:

$$H_1 = 1****$$
$$H_2 = *10**$$
$$H_3 = 0***1$$

From Example 3.5, we should also remember that individual x_2 was selected twice, individuals x_1 and x_4 were selected once, while individual x_3 was never selected, and that the population at the second generation was:

$$x_5 : 01100$$
$$x_6 : 11001$$
$$x_7 : 11011$$
$$x_8 : 10000$$

Let us consider first schema H_1. Strings x_2 and x_4 in the initial population belong to this schema. In other words:

$$m(H_1, 0) = 2$$

After selection, the individuals belonging to H_1 are 3 (two copies of x_2, plus one copy of x_4). Let us check if this is confirmed by the Schema Theorem. If we take into account only selection, we have:

$$m(H_1,1) = m(H_1,0)\ \frac{f(H_1)}{\tilde{f}} = 3.20$$

This is a substantial confirmation of the fact that, with some approximation, the Schema Theorem is able to predict the number of representatives of H_1 after selection. Let us now consider crossover and mutation. Given that $\delta(H_1) = 0$, the probability that crossover destroys H_1 is equal to zero. In other terms, crossover does not affect H_1. Also, in the example we had considered a mutation rate $p_m = 0$, so also mutation has no effect on H_1. As a conclusion, the Schema Theorem predicts that, at generation 2, the number of representatives of schema H_1 in the population should be 3. Indeed, three individuals in the second population, namely x_6, x_7 and x_8, belong to schema H_1.

Let us now consider schema H_2. There are two individuals belonging to H_2 in the initial population and two individual belonging to H_2 in the second population. Applying the Schema Theorem, we have that, after selection:

$$m(H_2,1) \approx 2.18$$

The probability of survival of H_2 after crossover is:

$$P(\text{surv } H_2) = 1 - p_c\ \frac{\delta(H_2)}{\ell - 1} = 1 - 1\ \frac{1}{5-1} = 0.75$$

So, the expected number of individuals matching H_2 after selection and crossover is:

$$m(H_2,1) = 2.18 \times 0.75 = 1.64$$

which is still a good approximation of the observed result (two individuals belong to H_2 in the second population).

Finally, let us consider schema H_3. In the initial population, there is one individual belonging to H_3, while in the second population there is no individual belonging to H_3. This is confirmed by the Schema Theorem. In fact, the probability of schema H_3 of surviving after crossover is:

$$P(\text{surv } H_3) = 1 - p_c\ \frac{\delta(H_3)}{\ell - 1} = 1 - 1\ \frac{4}{5-1} = 0$$

In other words, crossover will surely destroy this schema. So, the number of expected representatives of H_3 in the second population is:

$$m(H_3,1)\ =\ m(H_3,0)\ \times\ \frac{f(H_3)}{\tilde{f}}\ \times\ 0\ =\ 0$$

Building Blocks Hypothesis

Schema Theorem tells us that GAs allocate an exponentially growing number of representatives to schemata that are short, of low order, and with average fitness bet-

ter than the population's average, in other words the building blocks. The Building Blocks Hypothesis says that allocating an exponentially growing number of representatives to the building blocks is *useful* to solve an optimization problem. In other words, it is beneficial in the process of searching for a globally optimal solution. There are foundamentally two ways of giving evidence of the truth of this hypothesis. The first one was shown by J. Holland [Holland, 1975]; it is rather complex and only a superficial intuition will be shown in this document. The second one is more intuitive, and will be presented later on with an example.

Let us now concentrate on Holland's empirical demonstration of the Building Blocks Hypothesis. This is based on a well-known game theory problem, called the *k-armed bandit problem*. In synthesis, the problem can be formulated as follows: n slot machines are given. For each $j = 1, 2, ..., n$, slot machine j has a known number of arms K_j. We assume that we have played t times and, for each one of these games, we assume that we know: the slot machine on which the game was played and the outcome of the game, that can only be one among win or defeat. The problem consist in answering the following question: at which slot machines should we allocate the next games/trials, in order to maximize our probability of winning?

Before further discussing this problem, and investigating his consequences on GAs, let us fix some terminology. Given a slot machine j, let S_j be the probability of obtaining a winning game on slot machine j, calculated on the basis of the t games that have been played until present. In other words:

$$S_j = \frac{\#\text{winning games using slot machine } j}{\#\text{games played using } j}$$

An important result concerning the k-armed bandit problem can be proven. In the continuation of this document, this result will be stated informally, and the proof will not be given. The interested reader is referred to [Holland, 1975].

Theorem 3.1. *(Informal Statement).*

If *in the next games we allocate an exponentially growing number of trials to slot machines j such that:*

- *S_j is larger than the average calculated over all the slot machines. In other words:*
$$S_j > \frac{\sum_{i=1}^{n} S_i}{n}$$
- *j has a "small" number of arms*

Then*:*

- *The probability of winning is maximal;*
- *The probability of winning tends to 1 as the number of future trials tends to infinity.*

Based on this result, Holland was able to demonstrate an important result on GAs. In simple terms, Holland's way of reasoning was to formalize the k-armed bandit problem and GAs, establishing the following correspondances:

k-armed bandit	GAs
slot machine	schema
number of arms of a slot machine	order and length of a schema
probability of winning of a slot machine (calculated on the previous t trials)	average fitness of the schema
winning in the future trials	finding a globally optimal solution
allocating a future trial to one arm of a slot machine	insert in the next population an individual belonging to a schema

From Theorem 3.1, and from this correspondance between the two systems, it is possible to deduce that maintaining in a GA's population an exponentially growing number of schemata with average fitness better than the population's average and small order and length maximizes the probability of finding a globally optimal solution and this probability tends to 1 as the number of generations of the GA tends to infinity. On the other hand, the Schema Theorem is telling us that maintaining in the population an exponentially growing number of schemata with average fitness better than the population's average and small order and length is exactly what GAs do. As a consequence, the following result can be asserted:

Theorem 3.2. *(Theorem of Asymptotic Convergence of Genetic Algorithms). Let $i(t)$ be the best solution in a GA's population at generation t, and let S_{opt} be the set of globally optimal solutions in the search space, then:*

$$\lim_{t\to\infty} P\big(i(t) \in S_{opt}\big) = 1$$

From Theorem 3.2, it immediately follows that the succession $[i(0), i(1), ..., i(t), ...]$ tends towards a solution $o \in S_{opt}$. Besides the Holland's heuristic reasoning, Theorem 3.2 can also be formally proven, in case of elitism, using the concept of Markov chains. The interested reader is referred to [Rudolph, 1997] for this proof.

As already mentioned, the Building Blocks Hypothesis can also have an intuitive justification. In order to discuss it, let us take back Example 3.5, consisting in the problem of maximizing the function $f(x) = x^2$, with $x \in \mathbb{N}$ & $0 \leq x \leq 31$. The graphical representation of this function is shown in Figure 3.5(a). The idea is that, since individuals can be seen as points on the abscissa of that plot, schemata can be seen as "areas". For instance, schema $H_1 = 1****$ contains all the individuals represented by a numeric value larger than 15. So, H_1 can be represented by the area highlighted in gray in Figure 3.5(b). Just as other examples, schema $H_2 = 10**$ can be represented by the the area in Figure 3.5(c), and schema $H_3 = **1*1$ can be represented by the the area in Figure 3.5(d). From these examples, we can notice that:

- The higher the average fitness of a schema, the "closer" the area that represents the schema is to the global optimum (that, in this example, corresponds to number 31).

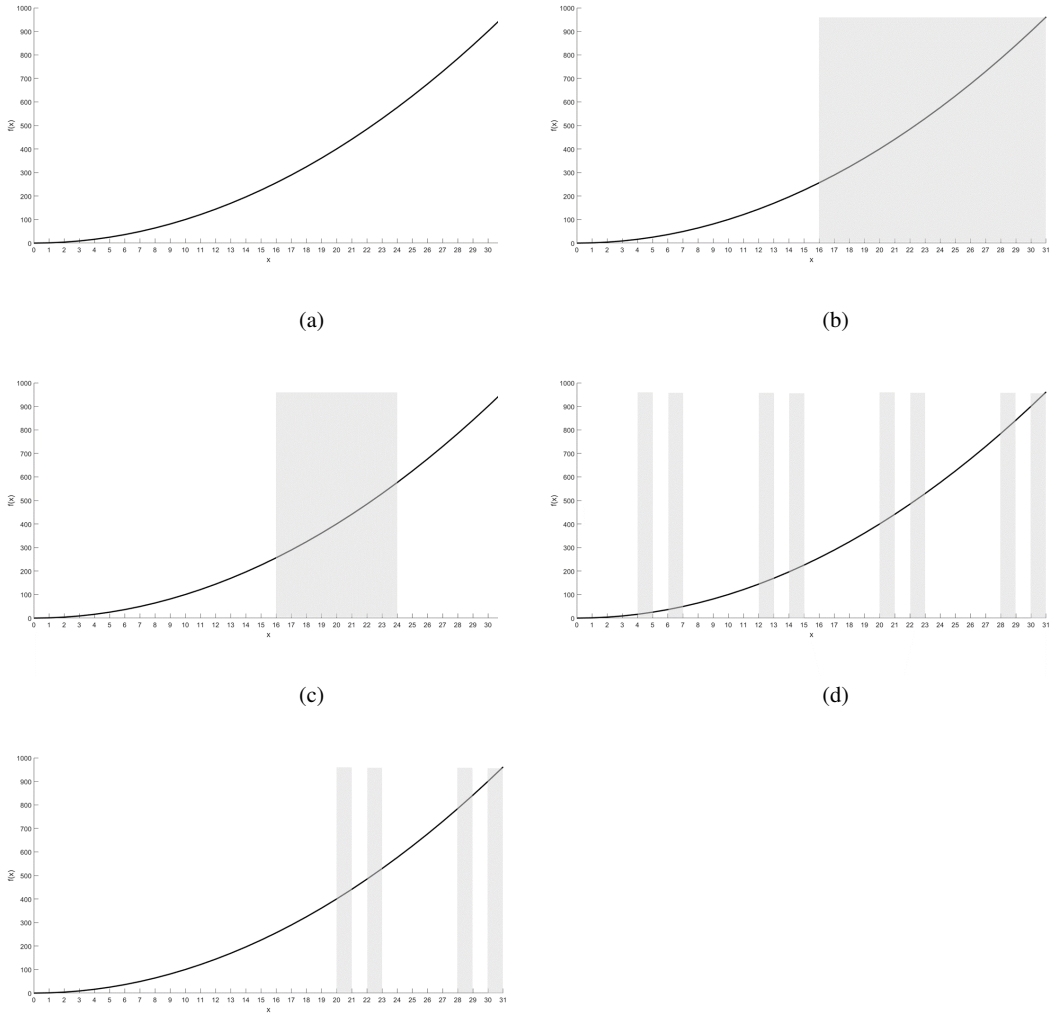


Fig. 3.5 Plot (a): graphical representation of the function $f(x) = x^2$, with $x \in \mathbb{N}$ & $0 \leq \mathrm{x} \leq 31$, used in Example 3.5. Plot (b): area represented by schema $1****$. Plot (c): area represented by schema $10**$. Plot (d): area represented by schema $**1*1$. Plot (e): area represented by schema $1*1*1$.

- The smaller the order of the schema, the smaller the area that represents it.

From a graphical viewpoint, crossover between two schemata H_i and H_j, that does not destroy neither H_i nor H_j, can be seen as the *intersection* of the areas represent-

ing H_i and H_j. For instance, given the schemata $H_1 = 1****$ and $H_3 = **1*1$, if position 1 is chosen as the crossover point, then crossover does not destroy neither H_1 nor H_3, and one of the resulting schemata is $H_4 = 1*1*1$, which is represented by the area highlighted in gray in Figure 3.5(e).

All this considered, it easy to understand that crossover between schemata of good fitness and small order gives origin to schemata represented by small areas, close to the global optimum. So, maintaining in the population these schemata (and the Schema Theorem tells us that this is what GAs do), and iterating the evolutionary process, it is possible to progressively get closer to the global optimum. The area representing the schemata in the population gets smaller and smaller and, for some of those schemata, closer and closer to the global optimum.

Another possible geometrical interpretation of GAs is by means of a hypercube. Let us consider, for simplicity, individuals represented by bit strings of length $\ell = 3$. In this case, the search space can be seen as a cube of side equal to 1 in a tridimensional space, like the one represented in Figure 3.6. In this cube:

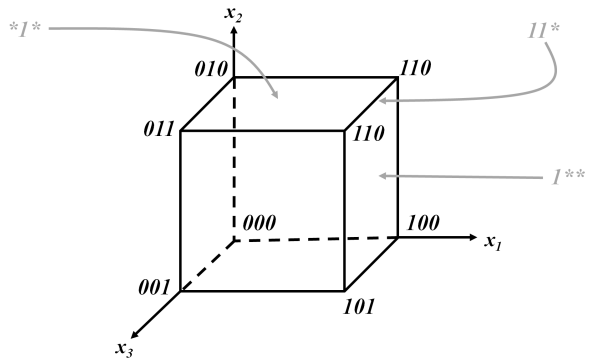


Fig. 3.6 Graphical representation of a GA in which individuals are represented as bit strings of length $\ell = 3$. The search space can be represented as a cube, where individuals are vertexes, schemata with one character equal to $*$ are sides and schemata with two characters equal to $*$ are faces.

- the vertexes represent the individuals;
- the sides represent schemata with one character equal to $*$.
- the vertexes of one side α represent the individuals matching the schema represented by α;
- the faces represent schemata with two characters equal to $*$.
- the vertexes of one face β represent the individuals matching the schema represented by β;
- the intersection side of two faces β_1 and β_2 can be interpreted as a crossover between the schemata represented by β_1 and β_2.

If the length of the strings representing the individuals is larger than 3, these ideas still hold, even though the search space (represented now by a hypercube) cannot be graphically represented.

3.5 Advanced Methods for Genetic Algorithms

As we have studied in the previous section, GAs converge asymptotically towards global optima. However, the Theorem of Asymptotic Convergence of GAs does not say anything about the speed with which a GA will approximate globally optimal solutions in real scenarios, and empirical evidence tells us that, in several cases, neither globally optimal solutions, nor reasonable approximations of those solutions, can be found in "reasonable" time by standard GAs[6]. For this reason, in the last three decades, effort was dedicated by researchers in the attempt of defining improvements for GAs, able to accelerate the process of convergence towards good quality solutions. The amount of produced work is so vast that even imagining of covering all of it in this document would be utopian. Nevertheless, it is worth discussing some of the most popular approaches. In particular, this section will be organized by identifying a number of possible drawbacks of the standard formulation of GAs, which may prevent them from finding good quality solutions in reasonable time. Then, possible workarounds for those problems will be discussed, thus defining a set of advanced GAs methods, that can potentially improve them. The problems of standard GAs that will be discussed in this section are:

- Premature convergence, or lack of diversity;
- Position problem of standard crossover;
- Unicity of the fitness function.

After defining each one of these potential issues, solutions that have been proposed in the literature will be discussed. In particuar, the premature convergence problem will be defined in Section 3.5.1, and workarounds presented for this problem will be fitness sharing, in Section 3.5.1.1, restricted mating, in Section 3.5.1.2, and diploid chromosome, in Section 3.5.1.3; the position problem of standard crossover will be presented in Section 3.5.2 and, in order to counteract this problem, we will discuss inversion and reordering operators in Section 3.5.2.1 and two new types of crossover, namely partially matched crossover and cycle crossover, in Section 3.5.2.2; finally, the unicity of the fitness function will be discussed in Section 3.5.3, and the solution to this issue that will be discussed is multi-objective optimization, in Section 3.5.3.1.

[6] From now on, we will use the term standard GAs to indicate the version of GAs introduced in [Holland, 1975, Goldberg, 1989], studied in Sections 3.1, 3.2, 3.3 and 3.4 and represented by the pseudo-code given in Algorithm 4.

3.5.1 Premature Convergence

The problem of premature convergence, or loss of diversity, is one of the most studied, and visible, problems of evolutionary algorithms. It can be explained in very simple terms: during the evolution, standard GAs tend to create populations, in which all the individuals are very similar between each other. In such a situation, typically the population contains few blocks of information, that are shared among all the individuals. It is intuitive that, if these blocks of information differ significantly from the characteristics of globally optimal solutions, the genetic operators will have difficulty in improving the quality of the individuals in the population.

To counteract this issue, several different strategies were developed by researchers along the years. Before studying some of them, however, it is important to define some measures that can allow us to quantify the diversity of a population, so that, using those measures, we are able to monitor and eventually prevent premature convergence. Even though several different measures of diversity exist, the two most employed measures are probably *entropy* and *variance*. Furthermore, both these measures can be used to quantify *phenotypic* diversity (i.e. the diversity in the fitness values, or functions of them) or *genotypic* diversity (i.e. diversity in the syntactic structures of the individuals). Let us begin with presenting entropy. The entropy of a population P is defined as:

$$H(P) = \sum_{j=1}^{N} F_j \log(F_j) \tag{3.14}$$

If phenotypic entropy is considered, F_j is te fraction n_j/N of individuals in P having a certain fitness value, where N is the total number of fitness values in P. To define genotypic entropy, two different techniques can be used: for the first one, F_j is defined as the fraction of individuals in population P having a certain genotype, and N is the total number of genotypes in P. The second technique consists in defining a distance measure between individual genotypes; in this case, F_j is the fraction of individuals having a certain distance value to a prefixed individual (often called "origin"), and N is the total number of distance values to the origin appearing in P. Experimental evidence indicates that this measure, at least qualitatively, is not strongly dependent on the choice of the particular individual used as origin. The logarithm in Equation (3.14) is normally used with basis equal to 2, even though the choice of the basis usually does not qualitatively affect the measurement of the diversity.

The variance of a population P is defined as:

$$V(P) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{3.15}$$

If phenotypic variance is considered, $\bar{x}$ is the average fitness of the individuals in P, x_i is the fitness of the i^{th} individual in P and n is the total number of individuals in P (i.e. the population size). To define genotypic variance, only the notion of distance is normally used. In this case, $\bar{x}$ is the average of all the individual distances to

the origin, x_i is the distance of the i^{th} individual in the population to the origin and, again, n is the total number of individuals in P.

Given that both genotypic entropy and phenotypic variance may need the notion of distance between individuals, it is a good idea to discuss a particular, and popular, measure commonly employed in GAs. Even though many other different measures can exist, and given that GA individuals are represented as strings of characters of a prefixed length, in GAs it is typical to use Hamming distance in case of discrete alphabet and Euclidean distance in case of continuous alphabet. Hamming distance between two strings x_1 and x_2, to be used if the characters in the strings are taken from a discrete alphabet, is defined as follows:

$$d_H(x_1,x_2) = \sum_{i=1}^{\ell} \Psi(x_1[i],x_2[i])$$

where, for each pair of characters a and b from the alphabet:

$$\Psi(a,b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

and where, for each string x, $x[i]$ represents the i^{th} character of x and ℓ is the string length.

Euclidean distance between two strings x_1 and x_2, to be used if the characters in the strings are taken from a continuous alphabet (typically, they are floating point numbers), is defined as follows:

$$d_E(x_1,x_2) = \sqrt{\sum_{i=1}^{n} (x_1[i] - x_2[i])^2}$$

3.5.1.1 Fitness Sharing

Fitness sharing is a method aimed at maintaining diversity in a population. The basic idea is to modify fitness, in such a way that individuals that are more diverse, compared to the rest of the population, receive a "reward", while individuals that are similar to the majority of the others in the population receive a "penalty". Given a measure of distance between individuals, and assuming that we are solving a maximization problem, fitness sharing can be applied to update the fitness of an individual x, by performing the following steps:

1. Calculate the distance of x to all the other individuals in the population;
2. Normalize all distances in the $[0,1]$ range (let $d_N(x,y)$ be the normalized distance of x to another individual y);
3. Apply a steadily decreasing function, called *sharing function* S, to the distances, so that sharing is large when the distance is small and viceversa.
4. Calculate the *sharing coefficient* of x like follows:

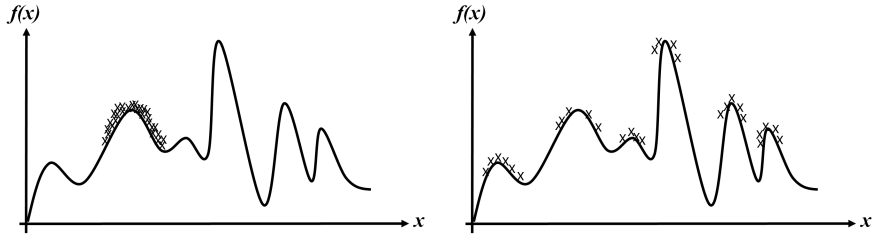
$$\mathscr{S}(x) = \sum_{y \in P, y \neq x} S(d_N(x,y))$$

5. Redefine the fitness of x as follows:

$$f_S(x) = \frac{f(x)}{\mathscr{S}(x)}$$

By applying this process, it is clear that when $\mathscr{S}(x)$ is large (i.e. when x is rather similar to the majority of the other individuals in the population) f_S receives a "penalty", while when $\mathscr{S}(x)$ is small, f_S receives a "reward". A simple way of implementing point 2 (normalization of the distances) in case Hamming distance is the chosen metric, is to define all distances by the string length. Analogously, a simple way of implementing point 3 (calculation of the sharing, by "inverting" the distance) is by using a simple linear sharing function like: $S(k) = 1 - k$.

It is clear that, if fitness sharing is applied, diverse individuals have a better fitness than what they would have if fitness sharing was not applied. This automatically gives those individuals a higher probability of being selected for mating, keeping a higher diversity in the population. The use of fitness sharing induces different population dynamics compared to standard GAs. This different behavior is idealized in Figure 3.7. In particular, Figure 3.7(a) shows the typical population dynamics, in



(a) (b)

Fig. 3.7 Plot (a): idealized search of a GAs population over a fitness landscape when fitness sharing is *not* used. Plot (b): typical population dynamics on the same fitness landscape when fitness sharing is used. The crosses indicate the positions in th fitness landscape assumed by the individuls in the population.

particular in an advanced stage of the search, when fitness sharing is not used. All individuals often tend to cluster around one single peak of the fitness landscape. On the other hand, Figure 3.7(b) shows the idealized dynamics induced by the use of fitness sharing: the individuals in the population are supposed to distribute more uniformly around the different peaks of the fitness landscape. The better exploration power bestowed by fitness sharing is payed by a larger computational cost of the algorithm. In fact, with fitness sharing, in order to evaluate the individuals, we have to calculate pairwise distances among all the individuals in the population.

3.5.1.2 Restricted Mating

As we can see from Figure 3.7, fitness sharing is implicitly partitioning the GAs population into *niches*. An intuition of the reason why, in some cases, organizing a population into niches can be beneficial can be given by observing the simple bidimensional fitness landscape represented in Figure 3.8. Assuming that the rep-

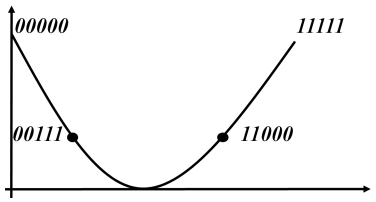


Fig. 3.8 Graphical representation of a simple fitness landscape for a maximization problem. Good quality individuals, in this case, have either a lot os 0s or a lot of 1s in their genotype.

resented problem is a maximization one, it is straightforward to remark that good quality individuals have either many bits equal to 0, or many bits equal to 1. If we allow a good individuals "with many 0s" to mate with a good individual "with many 1s", the offspring will have a hybrid chromosome, and will probably be less fit than their parents. For reasons like this, in many applications, it is appropriate to partition the population into niches, or species, promoting crossover between individuals in the same niche and discouraging crossover between individuals coming from different niches. Similar dynamics, obviously, exist also in biological populations and are believed to promote population diversity.

In order to foster the formation of niches in the population, methods imposing some mating restrictions have been proposed so far. One of the first contributions in this direction was proposed by Hollstien, who introduced a model in which crossover is allowed between two individuals i and j if and only if $d(i, j) < k$, where d is a prefixed distance metric and k a prefixed threshold value. In Hollstien's model, only in case the average population fitness does not improve for some generations, crossover between individuals that are more apart from each other (and so belonging to different niches) is allowed.

A different model was later introduced by Brooker. In Brooker's model each individual is represented using two substrings, that are usually called *template part* and *functional part*. The functional part contains the usual representation of the individual, as imposed by the definition of the problem. On the other hand, the template part is a schema (see Definition 3.1, at page 57). Intuitively, we could say that the template part of an individual x encodes in x's genome the "sexual tastes" of x, containing some characteristics of potential crossover partners of x. In order to know if two individuals can mate, we will compare the template part of one of them with the functional part of the other, and viceversa. Brooker defined the three following models:

- *Bidirectional match*. In this model, two individuals i and j are allowed to mate if the template part of i matches the functional part of j *and* the template part of j matches the functional part of i.
- *Unidirectional match*. In this model, two individuals i and j are allowed to mate if the template part of i matches the functional part of j *or* the template part of j matches the functional part of i.
- *Best partial match*. In this model, selection is modified in such a way that when an individual i is selected, the crossover partner of i is selected by choosing one of the individuals in the population whose functional part better matches the template part of i.

Example 3.10. Let us consider the three following individuals:

<Individual's ID>	<Template>	<Functional>
a	*10*	1010
b	*01*	1101
c	*00*	0000

In this case, we could informally say that "a is sexually attracted by b". In fact, the funcional part of b matches the template of a. But we can also say that the attraction of a is "reciprocated" by b, or that also "b is sexually attracted by a". In fact, also the funcional part of a matches the template of b. The conclusion is that individuals a and b are allowed to mate with any of the models presented above. On the other hand, none of the other individuals is attracted by c. However, it should be noted that the functional part of c matches the template part of c itself. So, if the implementation allows it, a crossover between two copies of c can take place.

In the Brooker's model, fitness is calculated using only the functional part (we could say that the functional part corresponds to the representation of an individual in standard GAs, and so it contains the information that is needed to calculate the fitness). On the other hand, before applying crossover and mutation, the template and functional part are concatenated, so to obtain a single string of double length. Then, crossover and mutation are applied to these longer strings. We could say that each individual is evolving its sexual preferences as a part of its genome. Under this viewpoint, as it is easy to hypothesize that it may happen also in natural populations, sexual preferences evolve (and thus change) with time.

3.5.1.3 Diploid Chromosome

Some organisms in nature are characterized by chromosomes that can be represented by means of a single string. This type of chromosome, called *aploid* chromosome, is typical of rather simple living beings. The chromosomes of more complex living beings, instead, can be represented by means of two strings of characters, and they are called *diploid* chromosomes. A diploid chromosome can be represented as in the following simple example:

A b C D e
a B C d e

As we can see, the two strings representing the chromosome have identical lengths and can be imagined, in some senses, as they were "aligned". The characters that appear in the same position in the two strings represent two alternative features for a determinate genetic trait. For instance, character B could represent the fact that the individual has brown eyes, while character b could represent the fact that the individual has blue eyes. Given that an individual cannot have these two traits at the same time, nature has developed a choice mechanism, called *dominance*. In Biology, we say that in a determinate *locus* (i.e. a position in a string), an *allele* (i.e. a character) dominates another allele. The former is called a *dominating* allele, while the latter is a *recessive* allele. Assuming by convention that the dominating characters are written with an upper case letter, the chromosome of the previous example could be transcribed as:

A B C D e

In other words, the dominating characters are always chosen, compared to the recessive ones, and only in case a recessive character does not have a dominating character in the corresponding position, it can be chosen.

At this point, a question comes natural: why do the most evolved individuals have diploid chromosomes? The answer to this question is, in large part, still object of scientific research, however the most accredited hypothesis is that a diploid chromosome allows the individuals a better adaption to environmental variations. For instance, in case of climate change, with significant lowering of the temperature, an animal possessing an allele that corresponds to long hair will probably have better ability of adaption to the new environment. Furthermore, it is an observed fact that diploid chromosome fosters a bigger population diversity, which allows for a better complexive adaptability.

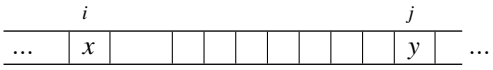
The idea of a dilpoid chromosome has been tranferred to GAs by several researchers. One of the first developed models was Bagley's model, in which each character of a string has an associated dominance value. Genetic operators are applied to both strings composing the chromosome, and then the character with higher dominance in each position is chosen, to generate a single string used to calculate fitness. The flaw of Bagley's model is that dominance values are prefixed, and genetic operators do not modify those values. In this way, after several generations, selection pressure causes few dominance values to survive in the population, thus vanishing the usefulness of having a diploid chromosome.

Another interesting model is the Hollstien-Holland model. In this model, each locus contains a pair of characters. In case of binary encoding, the first character comes from the alphabet $\{0,1\}$, while in general the second character comes from the alphabet $\{m,M\}$. The chromosome is diploid, in the sense that it is chacaterized by two strings, each of which contains this pair of characters in each locus. The string resulting from the dominance application, i.e. the string used to calculate

fitness, in case of binary encoding, is a binary string obtained applying to each locus a mapping, given by a prefixed dominance table. This model was later updated, by a diploid chromosome in which, in case of binary encoding, the two strings are formed by characters coming from the augmented alphabet $\{0,1,2\}$. In this second Hollstien-Holland model, genetic operators act on these two strings, and a dominance table is used to transform the two strings into a single binary string, used to calculate fitness. This table associates to each pair of characters in $\{0,1,2\}$ a character in $\{0,1\}$. Experimetal results confirm that this second model, also called *tri-allelic* model given that three characters are used in the chromosome, allows for a larger population diversity compared to the two previously discussed models. Furthermore, using schema theory, it is also possible to prove that, under some specific hypothesis, the tri-allelic model has better performance than the previously discussed models and also better than standard GAs [Goldberg, 1989].

3.5.2 Position Problem of Standard Crossover

The position problem of standard crossover can be explained in very simple terms: let us assume that, for some unknown reasons inherent to the faced problem, individuals with good fitness must have a certain allele x in a string locus i that is close to the left hand of the string, and a certain allele y in a locus j close to the right hand, like represented here:



It is clear that, in such a situation, standard crossover has a high probability of destroying good quality individuals. In fact, it is highly probable that the randomly chosen crossover point falls between alleles x and y. In this way, unless for a lucky contribution of the crossover partner, one offspring will contain allele x in locus i, but not allele y in locus j, while the other will contain allele y in locus j, but not allele x in locus i. If the feature that makes fitness good is to have *both* allele x in locus i *and* allele y in locus j, both offspring will have a worse fitness than the parent.

Using the schema theory terminology, we could say that, in GAs, building blocks do not have to necessarily be compact pieces of information, contained in contguous loci. In some situations, a useful building block could also be composed by pieces of information that are spread in the string, distributed in loci that are far apart from each other. Destroying those building blocks could be deleterious. Obviously, if we were able to:

- change the order of the characters in the string, so that positions i and j becolme close to each other;
- apply crossover;

- reorder the string;

the problem would be solved. But, unfortunately, we are in general not able to do this, because the informations on how to change the order of the characters in the string is not known. In the following sections, workarounds on how to limit this problem are proposed.

3.5.2.1 Inversion and Reordering Operators

A first attempt of couteracting the position problem of standard crossover was made by Bagley. In his model, at each locus in the string, a value and an index are associated. In other words, initially, individuals have the following aspect:

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \left[1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \right] \end{matrix}$$

In Bagley's model, a new operator, called inversion operator, was defined, allowing to swap two alleles chosen randomly. For instance, applying inversion to the previous individual, it would be possible to obtain:

$$\begin{matrix} 1 & 2 & 6 & 4 & 5 & 3 & 7 & 8 \\ \left[1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \right] \end{matrix}$$

where alleles 3 and 6 have been swapped. This operator can be applied once, or multiple times, according to the level or reshuffling we want to obtain. Crossover and mutation are applied to the string after the inversion, and before calculating fitness the string needs to be reordered. Of course, this method has the obvious problem of how to handle crossover between two strings in which indexes are not in the same positions. These strings are called *non-homologous*. For instance, consider a standard crossover between the following individuals:

$$\begin{matrix} 1 & 2 & 3 & 4 & | & 5 & 6 & 7 & 8 \\ \left[1 & 0 & 1 & 1 & | & 1 & 0 & 1 & 1 \right] \end{matrix}$$

$$\begin{matrix} 1 & 2 & 6 & 5 & | & 4 & 3 & 7 & 8 \\ \left[1 & 0 & 0 & 1 & | & 1 & 1 & 1 & 1 \right] \end{matrix}$$

Using the fourth crossover point, the offspring would be:

$$\begin{matrix} 1 & 2 & 3 & 4 & 4 & 3 & 7 & 8 \\ \left[1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \right] \end{matrix}$$

$$\begin{matrix} 1 & 2 & 6 & 5 & 5 & 6 & 7 & 8 \\ \left[1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \right] \end{matrix}$$

But clearly these strings cannot be reordered because some indexes are missing and some are appearing more than once. The method proposed in Bagley's model simply consists in prohibiting crossover between pairs of non-homologous strings. This solves the problem, but strongly limits the power of crossover. A possible alternative was suggested by Frantz, presenting several operators of inversion and mating. The inversion operators proposed by Frantz are:

- Linear Inversion. This is the inversion we have seen so far: two loci are chosen randomly and swapped.

- Linear+end inversion. In this type of inversion, linear inversion is applied with a given probability. If linear inversion has been applied, the algorithm terminates. Otherwise, with uniform probability, the left or the right extreme of the string are chosen, and then exchanged with a point chosen at random among all the points that have at least $\ell/2$ characters of distance from the chosen extreme, where ℓ is the string length.

With linear+end inversion, Frantz aimed at minimizing the tendency of linear inversion to swap alleles that are close to the central part of the string. Both these inversions can be applied with one of the following modalities:

- Modality of continuous inversion. Inversion is applied with a given probability p_i to each individual in the population.

- Modality of mass inversion. Once the new population is generated, and evaluated, half of the individuals are randmly chosen and the same inversion is applied to all of them.

The rationale beyond mass inversion is that it should tend to limit the cases of non-homologous strings. After applying inversion, Frantz proposed four types of mating:

- Mating between homologous. This is the same idea proposed by Bagley: crossover is allowed only among homologous strings.

- Viability mating. Crossover is allowed also if the parents strings are not homologous, but if the offspring do not contain all the indexes, they are disregarded and not inserted in the population.

- Any-pattern mating. One of the two parents is randomly chosen as the primary parent. The indexes of the other parent are changed, and they become identical to the ones of the primary parent. In this way, the two strings are forced to become homologous, and crossover can be applied.

- Best-pattern mating. Works like any pattern, but the primary individual is always the one with better fitness among the two parents.

Some studies have shown that Frantz's operators of inversion and mating can be useful for improving the performance of GAs in case of partcularly hard problems. However, empirical evidence also shows that inversion is more powerful if it is applied multiple times, and in different ways for the different individuals in the population, causing *de facto* a different random reshuffling of each string. A consequence of this strong inversion is that non-homologous strings become very frequent in the population, and Frantz's models become ineffective. Having a random reshuffling of individuals before applying crossover calls for the definition of recombination operators that are different from standard crossover. The next section presents two types of crossover that can be applied to non-homologous strings and, independently from the order of the indexes in the parent strings, by construction generate offspring containing all the indexes once and only once, in such a way that reordering is possible.

3.5.2.2 Partially Matched and Cycle Crossover

These two types of crossover, introduced in [Goldberg, 1989], can be applied to non-homologous individuals, independently from the type of inversion that has been applied to them, and both of them generate offspring that are a recombination of the chromosomes of the parents, and contain all the indexes. Let us begin by discussing *partially matched* crossover. The two parents that have been selected for mating (with any of the selection algorithms studied in Section 3.1) are aligned, and two crossover points are chosen randomly, like in the following example:

$$\begin{array}{c} 9\ 8\ 4\ |5\ 6\ 7\ |1\ 3\ 2\ 10 \\ [1\ 0\ 1\ |1\ 0\ 1\ |0\ 0\ 0\ 1\ ] \end{array}$$

$$\begin{array}{c} 8\ 7\ 1\ |2\ 3\ 10\ |9\ 5\ 4\ 6 \\ [0\ 0\ 1\ |0\ 1\ 1\ |1\ 1\ 0\ 0] \end{array}$$

The area of the string between the two crossover points in identified as a *window*. The algorithm starts by constructing the first offspring, in the following way: for each locus of the first parent (where loci are analyzed one by one, from left to right, in the order they appear in the string), if the index is not in the window than the locus is copied in the corresponding position of the offspring. Otherwise, if the index belongs to the window, in one of the two parents, the locus that is copied in the offspring is the corresponding one in the other parent. It is worth reinforcing that a locus is, by definition, a pair composed by an index and a character from the alphabet. So, when a locus is inherited by the offspring, both its index and its content are inherited. Following this process, the first offspring generated by the crossover between the two previous parents is:

$$\begin{array}{c} 9\ 8\ 4\ |2\ 3\ 10\ |1\ 6\ 5\ 7 \\ [1\ 0\ 1\ |0\ 1\ 1\ |0\ 0\ 1\ 1\ ] \end{array}$$

The reader is invited to notice that not only the loci inside the window have been exchanged, but also the three rightmost loci of the first parent. In fact, the indexes of those three loci also belong to the window, even though the window of the second parent, and so they have been exchanged with the loci in corresponding positions from the first parent. As we can see, the first offspring contains all the indexes, and each one appears once and only once. So, the string can be reordered and the individual inserted in the new population in a straightforward way. In order to construct the second offspring, the same method is applied, but this time the loci of the second parent are analyzed one by one. In the considered example, this allows us to create the following second offspring:

$$\begin{array}{c} \begin{array}{cccc|ccc|cccc} 8 & 10 & 1 & & 5 & 6 & 7 & 9 & 2 & 4 & 3 \end{array} \\ \left[\begin{array}{cccc|ccc|cccc} 0 & 1 & 1 & & 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \end{array}$$

The reader is invited to remark that, also in this case, not only the loci inside the window have been exchanged, but also the loci in the 2^{nd}, 8^{th} and 10^{th} positions, since their indexes belong to the window in the first parent.

Let us now present *cycle* crossover, and also in this case, let us discuss it with an example. For instance, let us consider the following parents, both of which have undergone inversion:

$$\begin{array}{c} \begin{array}{cccccccccc} 9 & 8 & 2 & 1 & 7 & 4 & 5 & 10 & 6 & 3 \end{array} \\ \left[\begin{array}{cccccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

$$\begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \left[\begin{array}{cccccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{array}\right] \end{array}$$

Let us start by building the first offspring. The algorithm starts by selecting a random locus from a random parent, and copy it in the corresponding position of the offspring. For simplicity, in this case, let us assume that the chosen locus is the first one of the first parent. So, the offspring that was built so far is:

$$\begin{array}{c} \begin{array}{cccccccccc} 9 & & & & & & & & & \end{array} \\ \left[\begin{array}{cccccccccc} 1 & - & - & - & - & - & - & - & - & - \end{array}\right] \end{array}$$

As we can see, only the first locus of the first parent was copied in the offspring, so far. All the rest of the offspring is, up to now, "empty", in the sense that all the other loci are still to be defined. Let us now take a further step. What makes us perform the next step is the following reasoning: given that locus with index equal to 9 has been inherited by the first parent, also locus with index equal to 1 has to be inherited by the first parent. In fact, let us assume that locus with index 1 was inherited by the second parent. If we accept that loci have to be copied in the offspring in the same positions in which they appear in the parents, we simply could not copy that locus in

the offspring. In fact, locus with index 1 appears in the first position of the string of the second parent. So, it should be copied in the first position of the offspring. But the first position of the offspring is already occupied. Obviously, the only remaining option is inheriting also the locus of index 1 from the first parent. So, now the part of offspring built so far is:

$$\begin{matrix} {\scriptstyle 9} & & & {\scriptstyle 1} & & & & & & \\ [1 & - & - & 1 & - & - & - & - & - & -] \end{matrix}$$

Now, if we repeat the same reasoning, given that locus with index 1 was inherited from the first parent, also locus with index 4 has to be inherited by the first parent. In fact, locus with index 4 appears in the second parent in the same position as locus with index 1 in the first parent. So, adding locus with index 4 from the first parent, the part of the offspring that was built so far is:

$$\begin{matrix} {\scriptstyle 9} & & & {\scriptstyle 1} & {\scriptstyle 4} & & & & & \\ [1 & - & - & 1 & - & 1 & - & - & - & -] \end{matrix}$$

Once again, the fact of having inherited locus with index 4 from the first parent "forces" to inherit also the locus with index 6 from the first parent, and so the construction of the offspring becomes:

$$\begin{matrix} {\scriptstyle 9} & & & {\scriptstyle 1} & {\scriptstyle 4} & & {\scriptstyle 6} & & \\ [1 & - & - & 1 & - & 1 & - & - & 0 & -] \end{matrix}$$

Iterating the reasoning, and observing that locus with index 9 appears in the second parent in the same position as locus with index 6 in the first parent, now we could say that the fact of having inherited locus with index 6 from the first parent induces us to also inherit locus with index 9 from the first parent. But, as we can observe, locus with index 9 was already inherited from the first parent and it already appears in the offspring. This event allows us to terminate a cycle (hence the name *cycle* crossover) that has allowed us to decide which loci the offspring should inherit from the first parent. The offspring is now completed by simply inheriting all the missing loci from the second parent. So, the final string representing the offspring is:

$$\begin{matrix} {\scriptstyle 9} & \textcircled{\scriptstyle 2} & \textcircled{\scriptstyle 3} & {\scriptstyle 1} & \textcircled{\scriptstyle 5} & {\scriptstyle 4} & \textcircled{\scriptstyle 7} & \textcircled{\scriptstyle 8} & {\scriptstyle 6} & \textcircled{\scriptstyle 10} \\ [1 & \textcircled{0} & \textcircled{1} & 1 & \textcircled{1} & 1 & \textcircled{1} & \textcircled{1} & 0 & \textcircled{0}] \end{matrix}$$

where, for the sake of readability, loci inherited from the second parent have been highlighed by a circle. As the reader can easily observe, the first offspring has all the indexes, each of which appearing once, and thus the string can be reordered straightforwardly. At this point, building the second offspring is simple: in each string position, the second offspring simply inherits the locus from the parent that

did not pass the locus to the first offspring. So, in this example, the second offspring is simply:

$$\begin{matrix} \textcircled{1} & 8 & 2 & \textcircled{4} & 7 & \textcircled{6} & 5 & 10 & \textcircled{9} & 3 \\ [\,\textcircled{0} & 0 & 1 & \textcircled{0} & 0 & \textcircled{1} & 0 & 0 & \textcircled{0} & 1\,] \end{matrix}$$

As we can see, also the second offspring has all the indexes, and each one of them appears once, and so its string can be reordered. This property is general, in the sense that, as for the partially matched crossover, also cycle crossover is able to build offspring recombining parents' loci, in such a way that the offspring always have all the indexes. One event that can happen is that, during the construction of the first offspring, the cycle that allows us to inherit the loci from the first parent only terminates when all the loci have been inherited. In that case, cycle crossover generates offspring that are identical to the parents. Given that this is rather rare, particularly for long strings, this event is normally accepted, with no further controls or restrictions. Partially matched crossover and cycle crossover have been successful in several applications, in particular when dealing with hard problems.

3.5.3 Unicity of the Fitness Function

This issue, shared also by all the other optimization algorithms studied so far, consists in the fact that, in standard GAs, the fitness function is unique. So, only one criterion at the time can be optimized. However, the real world is full of cases in which, at the same time, we have to optimize more than one, in some cases many, criteria at the time. One may think, for instance, of the simple problem of buying a new car: it is typical to look for a car that has some qualities, like power, speed, or comfort, but also that has a contained cost. In many cases, like this one, the criteria that have to be optimized are conflicting: cars with mant qualities are typically expensive, while cheap cars usually do not have many qualities.

Intuitively, one simple way to deal with this problem is to have several functions, able to quantify the different criteria to be optimized, and to define statistics over all these measurements, like for instance the average or the median, in order to unify all criteria in one, single, fitness function. If, on the one hand, this strategy has the advantage of being simple, and allow us to use GAs without any modification, on the other hand, it may create a number of new issue, that may be very hard, or even impossible, to solve in practical cases. First of all, in order for a statistic, like the average, to be a realistic expression of all the criteria, the different measurements should have the same scale. For instance, optimizing the average between a measurement α and a measurement β, in case α takes values in $[0,1]$ and β in $[0,1000]$, would clearly advantage the optimization of β, making any improvement in the optimization of α almost irrelevant. Even though normalization methods exist, in many cases normalizing measures of different nature can be tricky, and lead to misleading results. Also, calculating statistics over the different criteria may imply the necessity

of defining appropriate values for the weights of each one of the criteria, which may be an extremely hard task. For this reason, in the continuation of this section, we present methods that do not try, in any way, to join different criteria together in a unique fitness function. On the other hand, each criterion will define its own fitness function, and several different fitness functions will be optimized together.

A first attempt to integrate concepts of multi-objective optimization in GAs was proposed by Shaffer, who presented an environment called Vector Evaluated Genetic Algorithm (VEGA). In VEGA, given a population of n individuals and given k optimization criteria, selection allows us to choose n/k individuals using each one of the k criteria. Then, crossover and mutation are perfomed as usual on the whole population, with the rationale of integrating genetic material coming from the optimization of the different criteria. This method has the advantage of being simple, but the proposed selection mechanism tends to allow survival only to individuals that excel in one criterion, but inadequate in all the others. On the other hand, in multi-objective optimization, we are often interested in solutions that represent a good compromise between all the existing criteria. This drawback is overcome by Pareto multi-objective optimization, discussed in the continuation.

3.5.3.1 Pareto Multi-Objective Optimization in Genetic Algorithms

In multi-objective optimization, several fitness functions exist, that need to be optimized at the same time. In such a context, the concept of optimal solution is not obvious. One of the most used concepts is the one of *Pareto optimality*. Let us assume, for instance, that we want to build a product, so to optimize at the same time the average number of accidents provoked by that product and its prize. In this model, the fitness of a partcular product x can be seen as a pair:

$$f(x) = (\text{cost}, \text{average number of accidents})$$

For instance, let is assume that we have five products, A, B, C, D and E, with the following fitness values:

$$\begin{aligned} A &= (2,10) \\ B &= (4,6) \\ C &= (8,4) \\ D &= (9,5) \\ E &= (7,8) \end{aligned}$$

Which one of these products can be considered as "the best"? To help us answer this question, it can be useful to notice that the fitness values of the different products can be seen as points in a bidimensional space. So, it can be useful to plot those points in a Cartesian plane, like in Figure 3.9. At a first observation, we can immediately notice that there is an important difference between point A, B and C and the others: these three points have the property that no other solution is better on all optimization criteria. A, B and C are called *non-dominated* solutions. Concerning D

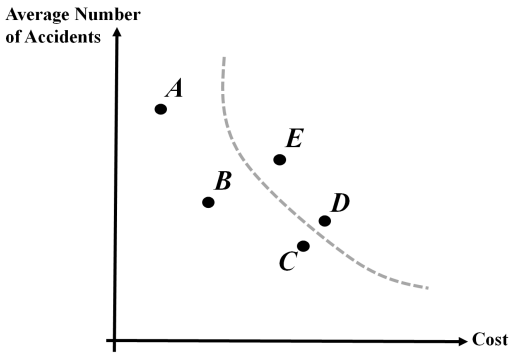


Fig. 3.9 Five solutions and the Pareto front, separating dominated from non-dominated ones (dashed gray line).

and E, we can, instead, observe that they are dominated. In fact, C is better than D on both criteria (we say that C dominates D) and B dominates E. So, instead of answering with a single solution, we can answer the previous question with a set of solutions: $\{A,B,C\}$. This is the set of non-dominated solutions, also called *Pareto optimal set*. The gray dashed line in Figure 3.9 is called *Pareto front*, and it is a line separating the non-dominated solutions from the dominated ones.

Actually, in multi-objective optimization, it is typical to have a set of optimal solutions, instead of a single one. In order to decide which one among A, B and C is better, we either need to give a weights to the criteria, or we can imagine to calculate a distance from an ideal "super-optimal" point. This point is a theoretical (and often inexistent) solution that joins the best known values on all criteria. For the previous example, it is represented in Figure 3.10, where it is indicated as α. If Euclidean

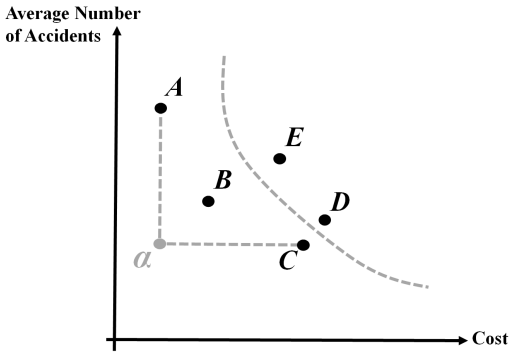


Fig. 3.10 Same solutions as in Figure 3.9, where also the ideal "super-solution" is shown. It consists in point α, that has the best known values for both criteria.

distance was used, in this case we can observe that B would be the closer solution to α among the ones belonging to the Pareto optimal set.

A method to integrate the concept of Pareto optimality inside the functioning of GAs was proposed by Baker. The method consists in sorting the individuals in a population on the basis of non-dominance and then apply selection, based on this ranking. The functioning of the method can be summarized by the pseudo-code in Algorithm 5. Baker applied this technique to multi-objective problems, in conjunction with methods to maintain diversity in the population, reporting very interesting experimental results.

Algorithm 5: The pseudo-code for applying selection in Pareto-based multi-objective GAs, according to Baker's method.

1. Insert all the individuals in the population in a set S;
2. *currentFlagValue* := 1;
3. **repeat until** (S is empty)
   - 3.1. Assign a flag equal to *currentFlagValue* to the non-dominated solutions in S;
   - 3.2. Remove the non-dominated solutions from S;
   - 3.3. *currentFlagValue* := *currentFlagValue* + 1;

   end
4. Assign to each individual in the population a selection probability that is *inversely* proportional to its flag value;

3.6 How to Organize an Experimental Comparison

One of the consequences of the No Free Lunch Theorem (Section 2.3) is that no formal method can exist to choose the best algorithm, or even the best algorithm configuration, to solve a particular problem. For this reason, in many different occasions, experimental comparisons are the only option. In this section, we discuss how two, or more, different algorithms, or configurations of algorithms, can be compared between each other.

Before entering in the details of the discussion, it is worth saying what, in general, an experimental comparison should *not* look like, thus revealing one of the most frequent and serious mistakes, when dealing with meta-heuristic optimization algorithms. If we want to compare the performance of two algorithm configurations on a given problem, we should *not* execute once both configurations and choose the one that returned the best result. In fact, all the algorithms that we have studied so

far, and also the majority of the algorithms we will study later in this document, are *non-deterministic*. In other words, they are based on random events and, in general, if we run twice the same configuration, we will obtain two different results. So, the outcome of a single run is not significant, since it may have been produced by fortuitous events. To perform a fair comparison, what is needed is to execute the algorithms for a number of runs much larger than one, and base the compariosn on statistics, calculated on the outcomes of those runs. The continuation of this section discusses how to organize such experiments, distinguishing between two cases:

- comparison of performance *against iterations*;
- comparison of performance *against computational effort* (or number of fitness evaluations).

Comparison against iterations is the simplest of the two, and can be made *only* when the configurations we are comparing are executed for the same number of iterations and, at each iteration, they perform a comparable amount of computational effort. This type of comparison is discussed in Section 3.6.1. When a comparison against iterations cannot be made, the alternative is a comparison against computational effort, that is discussed in Section 3.6.2.

3.6.1 Comparison Against Iterations

As previously mentioned, this type of comparison can only be done when the configurations we are comparing are executed for the same number of iterations and, at each iteration, they perform a comparable amount of computational effort. In metaheuristic optimization algorithms, like the ones studied so far, it is a typical and reliable approximation to estimate the amount of computational effort spent in one iteration using the number of fitness evaluations that have been performed in that iteration. In fact, fitness evaluation is usually the most computationally expensive activity of those algorithms, and the other actions performed by the algorithm usually have a negligible effect on computational effort, compared to it. For instance, Simulated Annealing usually performs one fitness evaluation per iteration (the solution that is evaluated is the chosen neighbor, that has to be compared with the current solution), while GAs perform n fitness evaluations per iteration, where n is the population size. So, for instance, two different configurations of Simulated Annealing, executed for the same number of iterations, can be compared between each other against iterations. Analogously, two different GAs configurations, both using the same population size and left to evolve for the same number of generations, can be compared against iterations. A configuration of the Particle Swarm Optimization (this algorithm will be studied in Chapter 4) and a GAs configuration, both using the same population size and left to evolve for the same number of generations, can also be compared against iterations. On the other hand, as we will clarify in the first part of Section 3.6.2, it is *incorrect* to compare against iterations, for instance, a configuration of Simulated Annealing executed for m iterations with a GAs configuration

with population size equal to $n > 1$, left to evolve for m generations. Analogously, it is incorrect to compare against iterations two GAs configurations left to evolve for the same number of generations, but using two different population sizes.

To fix the ideas and discuss how a comparison of performance against iterations can be organized, let us assume from now on, just as a matter of example, that we are interested in comparing two different GAs configurations $\mathscr{A}$ and $\mathscr{B}$, both of which use the same population size, left to evolve for the same number of generations. For instance, $\mathscr{A}$ and $\mathscr{B}$ could differ between each other for using two different types of crossover, or two different rates of the genetic operators.

Let us assume that both $\mathscr{A}$ and $\mathscr{B}$ are left to evolve for 4 generations, and, in both cases, the experiment is repeated for 3 independent runs[7]. Let us assume that we are facing a minimization problem, where the globally optimal fitness value is equal to 0. For each one of the 3 runs of configuration $\mathscr{A}$, we report, at each iteration, the value of the fitness of the best individual in the population. Let us assume that the obtained results are the ones reported in Figure 3.11, where each one of the tables reports the outcome of a different run.

Generations	Best Fitness
1	20
2	15
3	10
4	5

Generations	Best Fitness
1	18
2	13
3	12
4	10

Generations	Best Fitness
1	15
2	10
3	5
4	0

Fig. 3.11 Results obtained in 3 different independent runs by configuration $\mathscr{A}$ of the example used in Section 3.6.1. Each table reports the results obtained in a single run. For each generation, the best fitness value in the population is reported.

Let us now repeat the same set of experiments for configuration $\mathscr{B}$, and let us assume that the obtained results are the ones reported in Figure 3.12.

To compare the results of configuartions $\mathscr{A}$ and $\mathscr{B}$, we can now calculate, for each generation of each run, a measure called the Average Best Fitness (ABF). It simply consists in computing the average of the best fitness obtained at a given generation, over all performed runs. The ABF results for configuration $\mathscr{A}$ are reported in Table 3.3. What is reported in Table 3.3 can be interpreted, in some senses, as a sort of "average run". The table has, in fact, the same dimensions as each one of the tables

[7] It is crucial, for the understanding of this section, that the reader clearly understands the difference between the concept of run and the concept of generation: a generation is an iteration of the GA, aimed at building a new population. A run indicates a complete exection of a GA evolution, from the moment in which the population is initialized, to the moment in which the process terminates and a final solution is returned. Obviously, the numbers used in this example (4 generations and 3 runs) are toy quantities, which have been chosen only for the sake of simplicity; both need to be much larger in real experiments.

Generations	Best Fitness
1	18
2	17
3	16
4	12

Generations	Best Fitness
1	20
2	10
3	3
4	0

Generations	Best Fitness
1	10
2	5
3	2
4	0

Fig. 3.12 Results obtained in 3 different independent runs by configuration $\mathscr{B}$ of the example used in Section 3.6.1. Each table reports the results obtained in a single run. For each generation, the best fitness value in the population is reported.

Table 3.3 Using the data of Figure 3.11, we can obtain the ABF results for configuration $\mathscr{A}$ reported here.

Generations	ABF
1	$\frac{20+18+15}{3} \approx 17.667$
2	$\frac{15+13+10}{3} \approx 12.667$
3	$\frac{10+12+5}{3} = 9$
4	$\frac{5+10+0}{3} = 5$

reporting the outcome of a single run, shown in Figure 3.11. The reader is invited to remark that, in order to obtain the results of Table 3.3, no calculation was performed using data coming from the same run, but only using data coming from different runs. The analogous ABF results for configuration $\mathscr{B}$ are reported in Table 3.4. At

Table 3.4 Using the data of Figure 3.12, we can obtain the ABF results for configuration $\mathscr{B}$ reported here.

Generations	ABF
1	$\frac{18+20+10}{3} = 16$
2	$\frac{17+10+5}{3} \approx 7.33$
3	$\frac{16+3+2}{3} \approx 7$
4	$\frac{12+0+0}{3} = 4$

this point, the comparison between configurations $\mathscr{A}$ and $\mathscr{B}$ can be done in a visual way, by plotting in the same diagram the ABF of the two configurations for each generation, as in Figure 3.13. In some situations, it can be convenient to report the Median Best Fitness (MBF), instead of the ABF, because the median is more resitant to outliers than the average. Given that the plot of configuration $\mathscr{B}$ stands steadily below the plot of configuration $\mathscr{A}$, and given that we are facing a minimization problem, the indication we receive from Figure 3.13 is that configuration $\mathscr{B}$ seems to outperfom configuration $\mathscr{A}$.

However, in such a situation, it is *mandatory* to assess the statistical significance of the results. What we want is to understand if the differences between the results obtained by the two configurations are statistically significant or not. This can

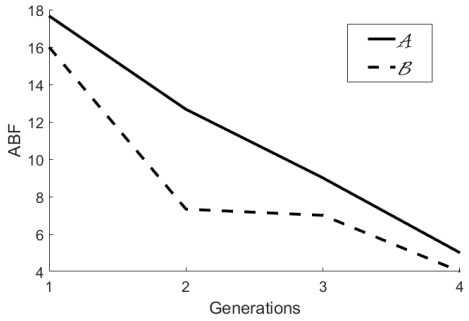


Fig. 3.13 A plot of the data of Table 3.3 and Table 3.4.

be obtained using several different statistical tests. Furthermore, in principle, single different tests can be perfomed at each generation, even though it is typical to run the tests only at termination (i.e. at the last performed generation). The first test that should be executed is aimed at understanding if the data are normally distributed or not. This can be obtained, for instance, using the Kolmogorov-Smirnov test [Massey, 1951, Dodge, 2008]. In case the data are normally distributed, a parametric test, like the Student's t-test can be used. In the opposite situation, i.e. when data are not normally distributed, which is also the most frequent sitatuation when analyzing experimental results coming from a set of independent runs, it is mandatory to use non-parametric tests, like for instance, the Mann-Whitley test, the Kruskal-Wallis test, the Friedman test or, most typically, the Wilcoxon rank-sum test for pairwise data comparison [Rey and Neuhäuser, 2011]. An explanation of those test is beyond the scope of this document, but the interested reader is referred, for instance, to [Lovric, 2011] for a deep introduction. In addition, a plot like the one in Figure 3.13 could be equipped with standard deviations, at each generation, in case the ABF is reported, or with boxplots [Benjamini, 1988] at termination, in case the MBF is reported.

The ABF and MBF give a visual understanding of the performance of an algorithm, or algorithm configuration, in a "typical" run, but do not say anything about the ability of finding globally optimal solutions, or reasonable approximations of them. On the other hand, in many situations, this is an interesting information. This information can be captured by another measure, called Success Rate (SR), which can be studied in conjunction with ABF and MBF, to give a more complete picture of the experimental results. SR is defined as follows:

$$\text{SR} = \frac{\#\text{successful runs}}{\#\text{performed runs}} \tag{3.16}$$

where, according to the situations, a run can be considered successful if the global optimum has been found, or if a solution reasonably approximates a global optimum has been found. In the latter case, the minimum difference between a fitness value and the optimal fitness needed to consider a solution a "reasonable" approximation

of a global optimum must be defined beforehand. In the previous example, assuming that 0 is the globally optimal fitness and that a run is successful only if a global optimum was found, if we apply Equation (3.16) to configurations $\mathscr{A}$ and $\mathscr{B}$, we obtain:

$$\mathrm{SR}(\mathscr{A}) = \frac{1}{3}$$

$$\mathrm{SR}(\mathscr{B}) = \frac{2}{3}$$

When, as in this example, a configuration, like $\mathscr{B}$, is better than another configuration, like $\mathscr{A}$, both using the ABF or the MBF and the SR, we can confidently conclude that $\mathscr{B}$ is preferable to $\mathscr{A}$. However, these measures do not necessarily return consistent results, and if it does not happen, the choice of the best configuration can be not trivial. Finally, it worth pointing out that it is not always possible to calculate the SR, because a globally optimal solution, or any reasonable approximation, could never be found in any of the performed runs.

3.6.2 Comparison Against Computational Effort

Let us consider, now, a comparison between two configurations of GAs $\mathscr{A}$ and $\mathscr{B}$, that use two different population sizes. This is just one of the numerous examples in which the comparison cannot be done against the number of generations. In fact, it is clear that the comptational cost of performing one generation is bigger for the model that uses the larger population. Just to fix the ideas, let us assume that model $\mathscr{A}$ uses a population size of 6 individuals, while model $\mathscr{B}$ uses a population size of 3 individuals. If we consider the number of fitness evaluations as an appropriate surrogate of the computational effort, which is customary for optimization meta-heuristics, it is clear that performing one generation of model $\mathscr{A}$ costs the double than performing one generation of model $\mathscr{B}$. In fact, the population of model $\mathscr{A}$ contains 2 times more individuals than the population of model $\mathscr{B}$.

To fairly compare the performance of model $\mathscr{A}$ and model $\mathscr{B}$, all we have to do is to store, at each run and for each generation, together with the generation number and the ABF, also the cumulative number of fitness evaluations that a model has executed until that generation. Considering that the number of fitness evaluations at each generation is, with few exceptions that can be ignored, equal to the number of individuals in the population, we simply have to add the population size to this cumulative counter, at each generation. Also, the two models have to be executed for a number of generations that allows us to obtain the same number of fitness evaluations at the end. So, in our case, model $\mathscr{B}$ has to be executed for a number of generations that is 2 times larger than model $\mathscr{A}$.

Let us make a numeric example, that should clarify the issue. Let us assume, again, that we are solving a minimization problem, where the globally optimal fit-

ness is equal to zero. Also, let us assume that we execute the two models for a number of runs equal to 3. Finally, let us assume that model $\mathscr{A}$ is evolved for 3 generations[8]. Let the results obtained by model $\mathscr{A}$ be the ones stored in the tables in Figure 3.14. Remark the third column of these tables: it contains the cumulative

Generations	Best Fitness	# Fit. Eval.
1	20	6
2	15	12
3	10	18

Generations	Best Fitness	# Fit. Eval.
1	18	6
2	13	12
3	12	18

Generations	Best Fitness	# Fit. Eval.
1	15	6
2	10	12
3	5	18

Fig. 3.14 Results obtained in 3 different independent runs by configuration $\mathscr{A}$ of the example used in this Section 3.6.2. Each table reports the results obtained in a single run.

number of fitness evaluations, and it is obtained simply by adding the population size at each generation.

Now, let the results obtained by model $\mathscr{B}$ be the ones stored in the tables of Figure 3.15. Remark, once again, that model $\mathscr{B}$ was executed for a number of gen-

Generations	Best Fitness	# Fit. Eval.
1	22	3
2	18	6
3	15	9
4	13	12
5	11	15
6	9	18

Generations	Best Fitness	# Fit. Eval.
1	19	3
2	16	6
3	14	9
4	13	12
5	12	15
6	11	18

Generations	Best Fitness	# Fit. Eval.
1	18	3
2	12	6
3	10	9
4	7	12
5	4	15
6	2	18

Fig. 3.15 Results obtained in 3 different independent runs by configuration $\mathscr{B}$ of the example used in this Section 3.6.2. Each table reports the results obtained in a single run.

[8] Once again, these are just toy values used for the sake of simplicity in this example. Much larger numbers have to be employed in real experiments.

erations that is double compared to model $\mathscr{A}$, which allows us to terminate with the same cumulative number of fitness evaluations.

Now, for both models, we calculate the averages of the second and third columns at each generation. For model $\mathscr{A}$ we have the results reported in Table 3.5, and for model $\mathscr{B}$ we have the results reported in Table 3.6.

Table 3.5 Using the data of Figure 3.14, we can obtain the average results for configuration $\mathscr{A}$ reported here.

Generations	ABF	# Fit. Eval.
1	$\frac{20+18+15}{3} \approx 17.667$	6
2	$\frac{15+13+10}{3} \approx 12.667$	12
3	$\frac{10+12+5}{3} = 9$	18

Table 3.6 Using the data of Figure 3.15, we can obtain the average results for configuration $\mathscr{B}$ reported here.

Generations	ABF	# Fit. Eval.
1	$\frac{22+19+18}{3} \approx 19.67$	3
2	$\frac{18+16+12}{3} \approx 15.33$	6
3	$\frac{15+14+10}{3} = 13$	9
4	$\frac{13+13+7}{3} = 11$	12
5	$\frac{11+12+4}{3} = 9$	15
6	$\frac{9+11+2}{3} \approx 7.33$	18

Finally, to draw the curves of the ABF, instead of plotting the first column against the second, we simply plot the third column against the second. In other words, we report the cumulative number of fitness evaluations on the horizontal axis, and the ABF on the vertical axis. The curves are reported in Figure 3.16. Notice that, for the sake of clarity, only the part in which both curves are present, i.e. for the number of evaluations between 6 and 18, has been reported in the figure. Also, it is important to remark that, if the same number of generations had been performed for both models, say 3 generations, and the results were reported against generations, the model with the best performance would have seemed $\mathscr{A}$. In fact, model $\mathscr{A}$ obtained a better (i.e. smaller) ABF in the first 3 generations. But this comparison would have been *wrong*. The correct comparison of ABF against computational effort tells us, instead, that model $\mathscr{B}$ outperforms model $\mathscr{A}$.

Before terminating this section, it is important to point out that also a comparison between a population-based approach like GAs and the Simulated Annealing[9]

[9] The same reasoning holds for all comparisons between any population-based meta-heuristic and any meta-heuristic that does not use a population, and processes one solution at each iteration.

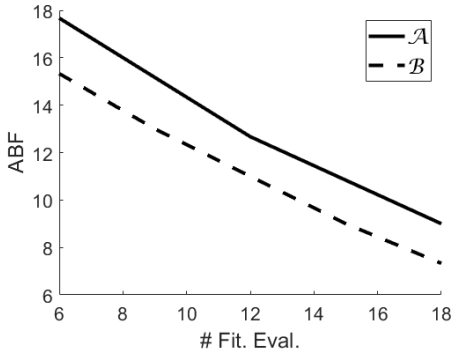


Fig. 3.16 A plot of the data of Table 3.5 and Table 3.6.

needs to be done against computational effort. In fact, the Simulated Annealing processes one solution at each iteration, and so performs one fitness evaluation at each iteration. So, if we want to produce tables like the ones in Figure 3.15 for the Simuated Annealing, the third column has to be increment by just one unit at each iteration. Also, if the Simulated Annealing is compared, say, to a GA that evolves a population of p individuals for g generations, the Simulated Annealing must be executed for $p \times g$ iterations, in order to have the same cumulative number of fitness evaluations at the end.

3.7 Genetic Algorithms for Continuous Optimization

Continuous optimization is a very frequent class of problems, with numerous real-life applications. Just as a matter of example, one may consider the optimization of the parameters of a device, or even the parameters of another algorithm. Even though several of the concepts studied so far in this chapter apply also to continuous optimization, some of the methods and examples were specific for discrete optimization, i.e. for the case in which the individuals are strings of characters extracted from a discrete alphabet. Some characteristics of GAs can to be changed/improved when the possible set of values of the alleles are extracted from a continuous set (like for instance the set of the real numbers $\mathbb{R}$). Specifically, more effective genetic operators can be defined. In fact, it is not hard to imagine that standard GAs crossover may be particularly weak when it comes to continuous optimization: by just swapping substrings, it is not able to generate new values for the alleles, and this largely limits the exploration ability of the algorithm. Before studying more effective operators, let us first define the continuous optimization problem.

In continuous optimization, individuals (i.e. feasible solutions) are vectors of the following shape:

$$\mathbf{x} = [x_1, x_2, ..., x_m]$$

where, for each $i = 1,2,...,m$, $x_i \in \mathbb{R}$ and $x_i \in [\alpha_i, \beta_i]$. In other words, the value of each allele has a well defined and known *a priory* range of variation. It is worth pointing out that, under these hypotheses, an individual can be interpreted as a point in a m-dimensional Cartesian space.

Several types of genetic operators can be defined for continuous optimization. Possibly, the most known are the so called geometric operators [Moraglio, 2008]. Geometric crossover works like follows: given two parents $[x_1, x_2, ..., x_m]$ and $[y_1, y_2, ..., y_m]$, is generates only one offspring, defined as:

$$[r_1 x_1 + (1 - r_1) y_1,\ r_2 x_2 + (1 - r_2) y_2,\ ...,\ r_m x_m + (1 - r_m) y_m]$$

where, for each $i = 1,2,...,m$, r_i is a random number extracted with uniform distribution from the interval $[0,1]$. The reason why this operator is called "geometric" is that, if individuals are interpreted as points in a Cartesian space, the offspring has precise geometric characteristics compared to the parents. More specifically, in the particular case in which all the r_is are indentical between each other, the offspring stands in the segment joining the parents. This geometric property has an important consequence: if fitness is directly proportional to a distance to a global optimum, then the offspring cannot be worse than the worst of the parents. This is shown by the example in Figure 3.17, where $m = 2$ for simplicity, and where we can clearly see that the offspring is closer to the reported global optimum than Parent 1.

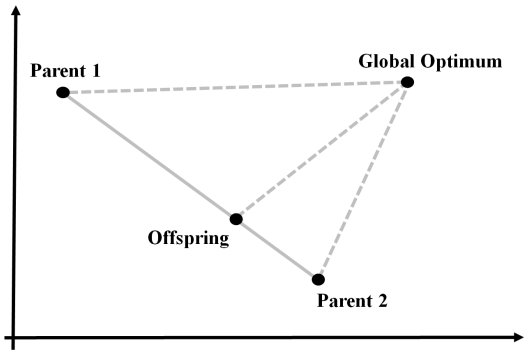


Fig. 3.17 A graphical representation of the effect of geometric crossover, in the simple bi-dimensional case.

Geometric semantic mutation (also called ball mutation or box mutation) works like follows: given an individual $[x_1, x_2, ..., x_m]$, it generates an individual defined as:

$$[x_1 + r_1,\ x_2 + r_2,\ ...,\ x_m + r_m]$$

where, for each $i = 1,2,...,m$, r_i is a random number drawn with uniform distribution from the interval $[-ms, ms]$, where ms is a parameter of the algorithm, called *mutation step*. In simple terms, geometric mutation consists in a (usually small)

random perturbation of the values of the alleles of an individual, where the importance of this perturbation can be tuned by means of a parameter. As Figure 3.18 shows for $m = 2$, geometric mutation can generate any point inside a "box" centered in $[x_1, x_2, ..., x_m]$. Since the offspring can appear in any position inside the box,

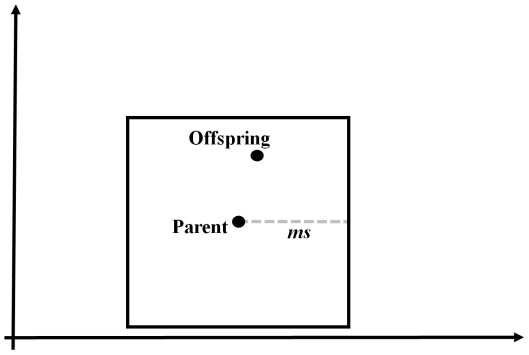


Fig. 3.18 A graphical representation of the effect of geometric mutation (box mutation), in the simple bi-dimensional case.

it is always possible that geometric mutation generates an offspring that is closer to a global optimum compared to the parent. This implies that, if fitness is directly proportional to a distance to a global optimum, geometric mutation induces a unimodal fitness landscape (i.e. a fitness landscape of no local optima, except for the global ones).

Chapter 4
Particle Swarm Optimization

Particle Swarm Optimization (PSO) [Kennedy, 2010, Kennedy and Eberhart, 1995] is an optimization algorithm, designed for continuous optimization. Like GAs, it is a population-based stochastic method, but unlike GAs it does not take its inspiration from the Theory of Evolution of Darwin, but from the social behavior of bird flocking or fish schooling. For instance, one may imagine a swarm of birds flying over a place, to find a point to land. In such a situation, the definition of which point the whole swarm should land is a complex problem, since it depends on many pieces of information, such as, for instance, maximizing the availability of food or minimizing the risk of existence of predators. In this context, one can interpret the movement of the birds as a sort of choreography; the birds synchronically move for a period until the best place to land is defined and all the flock lands at once. Given this natural ispiration, PSO is usually not categorized as belonging to the field of Evolutionary Computation, but to the field of Swarm Intelligence.

4.1 The Algorithm

PSO shares many similarities with Evolutionary Computation techniques such as GAs. For instance, the system is initialized with a population of random solutions and searches for optimal solutions, by updating generations. However, unlike GAs, PSO does not explore the search space using operators such as crossover and mutation. In PSO, we can imagine that the potential solutions, called particles, "move" through the problem space. Several studies indicated that all birds of a swarm searching for a good point to land are able to know the best point until it is found by one of the swarms members. By means of that, each member of the swarm balances its individual and its swarm knowledge experience, known as social knowledge. While in the natural scenario, the quality of a point for the birds to land is quantified by an estimation of the probability of survival, in PSO all solutions have fitness values which are given by the fitness function to be optimized, and have velocities which direct the movement of the particles. Each PSO solution

iteratively updates its position, influenced by its "best so far" achieved position and the swarm's current best position. In order to better understand the process, let us first fix some basic concepts and terminology.

- Following the definition of continuous optimization problem given at page 96, individuals of PSO (also called *particles*) are m-dimensional vectors of real numbers.
- A population (also called *swarm*) is an n-dimensional vector of particles.
- For each $i = 1,2,...,n$ we use the notation $\mathbf{x}_i$ to indicate the i^{th} particle of the swarm.
- For each $i = 1,2,...,n$ and for each $j = 1,2,...,m$, we use the notation x_{ij} to indicate the j^{th} coordinate of the i^{th} particle of the swarm.
- For each $i = 1,2,...,n$ we use the notation $\mathbf{v}_i$ to indicate the velocity of the i^{th} particle of the swarm.
- For each $i = 1,2,...,n$ and for each $j = 1,2,...,m$, we use the notation v_{ij} to indicate the j^{th} coordinate of the velocity of the i^{th} particle of the swarm.
- For each $i = 1,2,...,n$ we use the notation $\mathbf{b}_i$ to indicate the best position (i.e. the position with the best fitness) ever reached by the i^{th} particle of the swarm, since the beginning of the execution of the algorithm. This position is also called *local best* of the i^{th} particle of the swarm.
- For each $i = 1,2,...,n$ and for each $j = 1,2,...,m$, we use the notation b_{ij} to indicate the j^{th} coordinate of the local best of the i^{th} particle of the swarm.
- We use the notation $\mathbf{g}$ to indicate the best position ever reached by any particle of the swarm, since the beginning of the execution of the algorithm. This position is also called *global best*.
- For each $j = 1,2,...,m$, we use the notation g_j to indicate the j^{th} coordinate of the best position ever reached by any particle of the swarm.

We have that:

$$\forall i = 1,2,...,n : \mathbf{x}_i \in \mathbb{R}^{\mathrm{m}} \text{ (particle)}, \ \mathbf{v}_{\mathrm{i}} \in \mathbb{R}^{\mathrm{m}} \text{ (velocity)}, \ \mathbf{b}_{\mathrm{i}} \in \mathbb{R}^{\mathrm{m}} \text{ (local best)}$$

Furthermore:

$$\mathbf{g} \in \mathbb{R}^{\mathrm{m}} \text{ (global best)}$$

Finally, the fitness function f returns a real number for each particle, so:

$$f : \mathbb{R}^{\mathrm{m}} \rightarrow \mathbb{R}$$

With this in mind, we are now ready to study the general functioning of the PSO. Algorithm 6 presents the pseudo-code of PSO for the case of minimization problems. In that algorithm:

- w is a parameter, called *inertia constant*;
- c_1 and c_2 are two parameters, called *cognitive component* and *social component*, respectively;

Algorithm 6: The pseudo-code for Particle Swarm Optimization, in case of minimization problem.

1. $\forall i = 1,2,...,n:$ initialize $\mathbf{x}_i$ and $\mathbf{v}_i$;
2. $\forall i = 1,2,...,n:$ $\mathbf{b}_i := \mathbf{x}_i$;
3. $\mathbf{g} = \text{argmin}_{\mathbf{x}_i} f(\mathbf{x}_i)$, $\forall i = 1,2,...,n$;
4. **repeat until** (termination condition)
   - 4.1. **for** each $i = 1,2,...,n$ **do**
     - 4.1.1. $\mathbf{x}_i := \mathbf{x}_i + \mathbf{v}_i$;
     - 4.1.2. $\mathbf{v}_i := w\,\mathbf{v}_i + c_1\,\mathbf{r}_1 \circ (\mathbf{b}_i - \mathbf{x}_i) + c_2\,\mathbf{r}_2 \circ (\mathbf{g} - \mathbf{x}_i)$;
     - 4.1.3. **if** $(f(\mathbf{x}_i) < f(\mathbf{b}_i))$ **then** $\mathbf{b}_i := \mathbf{x}_i$;
     - 4.1.4. **if** $(f(\mathbf{x}_i) < f(\mathbf{g}))$ **then** $\mathbf{g} := \mathbf{x}_i$;

     end

   end
5. **return g**;

- $\mathbf{r}_1$ and $\mathbf{r}_2$ are two m-dimensional vectors of random numbers drawn at every different velocity update event, uniformly from the $[0,1]$ interval;
- the symbol $\circ$ represents the operator of element by element multiplication between two vectors.

Let us now comment the pseudo-code line by line, in order to understand every detail. Lines 1, 2 and 3 initialize the main employed structures. In particular, line 1 initializes the positions and the velocities of all the particles in the swarm. Generally, the particle positions $\mathbf{x}_i$ are initialized randomly. More precisely, for each $i = 1,2,...,n$ and for each $j = 1,2,...,m$, x_{ij} is initialized with a random number drawn with uniform probability from interval $[\alpha_j, \beta_j]$[1]. Also, it is customary to initialize all the velocities at zero ($\mathbf{v}_i := \mathbf{0}$). Line 2 initializes the local best of eac particle to the current position (that is also the only position that the particle has occupied so far), while line 3 initializes the global best to the particle with the best fitness in the initial swarm. Lines 4 and 4.1 represent the beginning of the main loops that characterize the algorithm. Line 4 represents the beginning of the external loop, that is repeated until a termination condition is achieved. As it is customary in optimization meta-heuristics, usually the termination condition is that a global optimum (or a reasonable approximation of it) has been found, or a prefixed maximum number of iterations has been executed. Line 4.1. represents the beginning of the internal loop, indicating a repetition of the same actions for each particle in the swarm. The actions that are executed are represented by lines 4.1.1 to 4.1.4. Line 4.1.1 modifies the position of each particle, by simply adding the current value of its velocity. This step is useless the first time, in case the velocity is initialized to zero. Line 4.1.2

[1] Following the definition of continuous optimization problem given at page 96, usually each coordinate j of the position vectors of the particles has a known *a priori* range of variation $[\alpha_j, \beta_j]$.

modifies the velocity, and it is probably the hardest point to understand in this algorithm. Basically, the new velocity is determined by the sum of three components:

- $w\ \mathbf{v}_i$: this term tends to maintain the movement of the particle in the same direction as it was in the previous iteration;
- $c_1\ \mathbf{r}_1 \circ (\mathbf{b}_i - \mathbf{x}_i)$: this term tends to let the particle move towards the position of its own local best (cognitive term);
- $c_2\ \mathbf{r}_2 \circ (\mathbf{g} - \mathbf{x}_i)$: this term tends to let the particle move towards the position of the global best of the swarm (social term).

The respective importance of these three terms can be tuned by setting the three parameters w, c_1 and c_2. However, it should be noticed that both the cognitive and the social term have a random component, that can potentially be different for each one of the coordinates, and that bestows stochasticity on the algorithm. Finally, points 4.1.3 and 4.1.4 update the local best and the global best, respectively, in case better solutions than the current ones were found. The final step of the algorithm, point 5, returns the current global best as the final result of the search.

The reader should notice that the algorithm is highly parallelizable. In particular, loop 4.1. executes the same action for all the particles independently, and with only one point of synchronization when it comes to updating the global best (point 4.1.4). All the other actions, including the update of the position, the update of the velocity and the calculation of the fitness of the new position (that is typically the most expansive operation, in terms of computational resources) can be run in parallel for each particle, since no synchronization is needed. Furthermore, the parallelism grain can even be finer, i.e. at the level of the single vector coordinates. In fact, the position and velocity update are vectorial operations, that can be executed coordinate by coordinate independently. In other words, loop 4.1 could be rewritten as:

for each $i = 1, 2, ..., n$ **do**
  for each $j = 1, 2, ..., m$ **do**
    $x_{ij} := x_{ij} + v_{ij}$;
    $v_{ij} := w\ v_{ij}\ +\ c_1\ r_1\ (b_{ij} - x_{ij})\ +\ c_2\ r_2\ (g_j - x_{ij})$;
  end
  if $(f(\mathbf{x}_i) < f(\mathbf{b}_i))$ **then**
    for each $j = 1, 2, ..., m$ **do** $b_{ij} := x_{ij}$;
  if $(f(\mathbf{x}_i) < f(\mathbf{g}))$ **then**
    for each $j = 1, 2, ..., m$ **do** $g_j := x_{ij}$;
end

where r_1 and r_2 are random numbers (scalars) extracted with uniform distribution from the $[0, 1]$ interval. The possibility of parallelizing PSO both at the particle level and at the coordinate level makes the algorithm potentially very efficient. Given that the calculations are mostly vectorial operations, PSO is particularly suitable to be implemented using Graphic Processing Unit (GPU), which typically makes the algorithm even faster.

As it is typical of all the techniques discussed in this document, the pseudo-code in Algorithm 6 has to be interpreted just as a general guideline, and several extensions and specifications need to be added when it comes to actually implement the method. For instance, one has to decide a strategy to "force" the algorithm to respect the limitation of the values of each coordinate position j into the predefined range $[\alpha_j, \beta_j]$. A possibility is, whenever Equation 4.1.1 of Algorithm 6 tends to exceed a margin, to set the value of the coordinate to the margin itself and to invert the sign of the velocity, or reinitialize it to zero, or to a random number. Reinitialization of the position coordinate with a random number in $[\alpha_j, \beta_j]$ is also an option. For instance, if the strategy consisting in setting the coordinate to the margin and inverting the sign of the velocity is adopted, the previous portion of pseudo-code could be extended to:

for each $i = 1, 2, ..., n$ **do**
  for each $j = 1, 2, ..., m$ **do**
    $x_{ij} := x_{ij} + v_{ij}$;
    if $x_{ij} < \alpha_j$ **then**
      $x_{ij} := \alpha_j$;
      $v_{ij} := -v_{ij}$;
    else if $x_{ij} > \beta_j$ **then**
      $x_{ij} := \beta_j$;
      $v_{ij} := -v_{ij}$;
    else
      $v_{ij} := w\ v_{ij}\ +\ c_1\ r_1\ (b_{ij} - x_{ij})\ +\ c_2\ r_2\ (g_j - x_{ij})$;
    end
  end
  if $(f(\mathbf{x}_i) < f(\mathbf{b}_i))$ **then**
    for each $j = 1, 2, ..., m$ **do** $b_{ij} := x_{ij}$;
  if $(f(\mathbf{x}_i) < f(\mathbf{g}))$ **then**
    for each $j = 1, 2, ..., m$ **do** $g_j := x_{ij}$;
end

4.2 Parameter Setting

The choice of PSO parameters can have a large impact on optimization effectiveness. Choosing appropriate parameter values is a complex task, that been the subject of much research. The PSO parameters can be tuned by using another optimization algorithm, or fine-tuned before or during the optimization. Like for any other optimization meta-heuristics, appropriate PSO parameter values are highly problem dependent. However, some simple rule of thumb can be discussed. These guidelines have to be interpreted only as suggestions to practitioners, and can by no means

replace parameter tuning. While the dimensions of the particles (m in the previous section) and the range of the different coordinates are usually determined by the problem, concerning the swarm size (i.e. the number of employed particles), it is typical to have significantly smaller values compared to GA populations. A typical range may be 20 to 40 particles, even though for simple problems even smaller values, like for instance 10 particles, can be large enough to get good results and for some difficult or special problems, one may need larger values, like 100 or 200 particles. The other side of the coin is that in PSO, it is typical to run the algorithm for a larger number of iterations, compared to GAs: from several thousands of iterations to even millions of iterations in case of parallel and GPU-based implementations.

Learning factors (c_1 and c_2) and inertia weight (w) are probably the hardest parameters to set. A common practice is to start the tuning by using values equal (or "close") to 2 for c_1 and c_2 and close to 1 for w, values that have revealed appropriate for several applications. However, other settings were also used in different studies. Even though exceptions can exist, appropriate values for c_1 and $c2$ typically range from 0 to 4. Furthermore, it is possible to also introduce another parameter, that is the maximum possible velocity. This can be an appropriate practice in some cases, to avoid too much large "jumps" of the positions of the particles in the search space. When it is used, a typical value for the maximum velocity for each coordinate j is the magnitude of the variation of the position in that dimension. In other words, if the particle position has a variation range in $[\alpha_j, \beta_j]$ for coordinate j, one may set the maximum velocity for that coordinate to $\beta_j - \alpha_j$. A method to force the algorithm to respect this limitation, for instance, can consist in resetting a coordinate of the velocity to zero whenever Equation 4.2.1 in Algorithm 6 tends to exceed the maximum velocity, or to choose a random value extracted from 0 to the maximum velocity.

4.3 Variants

Several variants of the basic PSO algorithm are possible [Cagnoni et al., 2008], and some of them have actually already been discussed in the previous section. For instance, one may imagine that each method employed to force the algorithm to respect the interval of variation of the position coordinates or to remain within the maximum possible velocity constitues, in itself, a variant. Besides this, given the clear analogies between PSO and GAs, a lot of effort was dedicated by researchers to the idea of generating hybrid versions, integrating PSO and GAs into a unique algorithm. If on one hand the update of the position of the particles can be seen as a particular type of mutation, on the other hand it is clear that selection and crossover are completely absent in PSO. The idea of enriching PSO with selection and crossover was explored, for instance, in [Miranda and Fonseca, 2002, Garg, 2016]. Furthermore, attemps of integrating PSO with other optimization algorithms were presented, for instance, in [Krink and Løvbjerg, 2002, Niknam and Amiri, 2010, Zhang and Xie, 2003].

Other research trends consist in using multiple swarms [Vanneschi et al., 2011, Cheung et al., 2014] and multi-objective optimization [Nobile et al., 2012].

Various simplified variants of PSO were presented in [Dan and Tim, 2008, Pedersen and Chipperfield, 2010]. Last but not least, PSO was extended to discrete optimization. A commonly used method is to map the discrete search space to a continuous domain, to apply a classical PSO, and then to demap the result. Such a mapping can be very simple (for example by just using rounded values) or more sophisticated [Roy et al., 2011]. Different approaches for using PSO for discrete optimization were presented in [Kennedy and Eberhart, 1997, Clerc, 2004, Jarboui et al., 2008, Chen et al., 2010].

Chapter 5
Genetic Programming

GAs, studied in Chapter 3, are capable of solving many problems and simple enough to allow solid theoretical studies. Nevertheless, the fixed-length string representation of individuals that characterizes GAs can be a limitation for a wide set of applications. In these cases, the most natural representation for a solution is a hierarchical computer program, rather than a fixed-length character string. For example, fixed-length strings do not readily support the hierarchical organization of tasks into subtasks typical of computer programs, they do not provide any convenient way of incorporating iteration and recursion and so on. But above all, GA representation schemes do not have any dynamic variability: the initial selection of string length limits in advance the number of internal states of the system and limits what the system can learn.

This lack of representation power (already recognized in [De Jong, 1988]) is overcome by Genetic Programming (GP) [Koza, 1992]. As GAs, GP belongs to the field of Evolutionary Computation, but contrarily to GAs, GP operates with general hierarchical computer programs. In other words, GP manages a population of computer programs. These programs are typically intialized randomly, and then evolved, using an algorithm that is inspired by the Theory of Evolution of Darwin, with the objective of automatically discovering programs that are appropriate for solving the problem at hand. So, GP is a method for automatically generating a program that solves a problem, starting from the high level specifications of the problem itself and, in general, without any intervention from a human programmer. Data models are particular computer programs, and, as such, GP can be seen as a Machine Learning method. Even though every programming language (e.g. Pascal, Fortran, C, etc.) is virtually capable of expressing and executing general computer programs, Koza chose the *LISP* (*LIS*t *P*rocessing) language to code GP individuals. The reasons of this choice can be summarized as follows:

- Both programs and data have the same form in LISP, so that it is possible and convenient to treat a computer program as data in the genetic population.

- This common form of programs and data is a parse tree and this allows in a simple way to decompose a structure in substructures (subtrees) to be manipulated by the genetic operators.
- LISP facilitates the programming of structures whose size and shape change dynamically and the handling of hierarchical structures.
- Many programming tools are commercially available for LISP.

In other words, GP, as originally defined by Koza, considers individuals as LISP-like tree structures. These structures are perfectly capable of capturing all the foundamental properties and features of modern programming languages. GP using this representation is called *tree-based GP*[1]. The tree-based representation of genomes, although the oldest and most commonly used, is not the only one that has been employed in GP. In particular, in the past decades, a growing attention has been dedicated by researchers to linear genomes (see for instance [Brameier and Banzhaf, 2001, McPhee et al., 2018]) , graph genomes (see for instance [Miller, 2001]). and grammars [O'Neill and Ryan, 2003].

5.1 The GP Process

In synthesis, the GP paradigm breeds computer programs to solve problems by executing the following steps:

1. Generate an initial population of computer programs (or individuals).

2. Iteratively perform the following steps until the termination criterion has been satisfied:

   2.1. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
   2.2. Create a new population by applying the following operations:
      2.2.1. Probabilistically select a set of computer programs to be reproduced, on the basis of their fitness (selection).
      2.2.2. Copy some of the selected individuals, without modifying them, into the new population (reproduction).
      2.2.3. Create new computer programs by genetically recombining randomly chosen parts of two selected individuals (crossover).
      2.2.4. Create new computer programs by substituting randomly chosen parts of some selected individuals with new randomly generated ones (mutation).

3. The best computer program appeared in any generation is designated as the result of the GP process at that generation. This result may be a solution (or an approximate solution) to the problem.

[1] Of course, modern tree-based GP is implemented in languages like C, C++, Java or Python, rather than LISP.

The attentive reader will not have failed to recognize clear similarities between the general functioning of GAs (presented in Chapter 3) and the one of GP. Actually, the high level algorithm is extremely similar, with typically the only major difference that, in GP, crossover and mutation are not applied sequantially. In other words, mutation is usually not applied to the offspring generated by crossover. Instead, some selected individuals undergo crossover, some of them mutation and some of them reproduction, according to prefixed probabilities, that are parameters of the algorithm. The individuals resulting from the application of these operators are usually directly inserted in the new population, and not used as inputs for other operators. This is justified by the fact that, as we will study later in this chapter, GP genetic operators are more "desruptive" than the ones of GAs, and thus applying more than one operator would risk to generate individuals that are too different from the parents. Besides this, the different representation of the individuals evolving in the population makes GP deeply different from GAs in its dynamics and interpretation. In the following sections, each feature of The GP process is analysed in details, focusing on those that distinguish GP from GAs.

5.1.1 Representation of GP Individuals

In tree-based GP, the set of all the possible structures that can be generated is the set of all the possible trees that can be built recursively from a set of function symbols $\mathscr{F} = \{f_1, f_2, \ldots, f_n\}$ (also called primitive functions, and used to label internal tree nodes) and a set of terminal symbols $\mathscr{T} = \{t_1, t_2, \ldots, t_m\}$ (used to label tree leaves). This potentially infinite search space is usually limited in size, by restricting it to only those trees whose depth is smaller or equal than a prefixed parameter d[2]. Each function in the fuction set $\mathscr{F}$ takes a fixed number of arguments, specifying its *arity*. Functions may include arithmetic operations ($+$, $-$, $*$, etc.), other mathematical functions (such as `sin`, `cos`, `log`, `exp`), boolean operations (such as `AND`, `OR`, `NOT`), conditional operations (such as `If-Then-Else`), iterative operations (such as `While-Do`) and/or any other domain-specific functions that may be defined. Each terminal is typically either a variable or a constant, defined on the problem domain.

Example 5.1. Given the following sets of functions and terminals:

$$\mathscr{F} = \{+, -\}, \quad \mathscr{T} = \{x, 1\}$$

a legal GP individual is represented in figure 5.1. This tree can also be represented by the following LISP-like S-expression (for a definition of LISP S-expressions see, for instance, [Koza, 1992]):

[2] We remind that the *depth* of a tree is defined as the longest possible path from the root of the tree to one of its leaves.

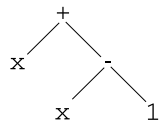


Fig. 5.1 A tree that can be built with the sets $\mathscr{F} = \{+, -\}$ and $\mathscr{T} = \{x, 1\}$.

$$(+\, x\, (-\, x\ \ 1))$$

which is the prefix notation representation for expression $x + (x - 1)$.

Closure and Sufficiency of Function Set and Terminal Set. The function and terminal sets should be chosen so as to verify the requirements of *closure* and *sufficiency*. The closure property requires that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. In other words, each function should be well defined for any combination of arguments that it may encounter. The reason why this property must be verified is clearly that programs must be executed in order to assign them a fitness, and a failure in one of the executions of one of the programs composing the population would lead either to a failure of the whole GP system or to the generation of unpredictable results. The function and terminal sets of the previous example clearly satisfy the closure property. The following ones, for instance, don't satisfy this property:

$$\mathscr{F} = \{*, /\},\ \ \mathscr{T} = \{x, 0\}$$

In fact, each evaluation of an expression containing an operation of division by zero would cause an error. In order to use division, but avoid this type of problem, this operation is usually modified in order to verify the closure property (see Section 5.3.1). Respecting the closure property in real-life applications is not always straightforward, since the use of many different data types could be necessary. A common example is a mix of boolean and numeric functions: function sets could exist composed by boolean functions (*AND*, *OR*, ...), arithmetic functions ($+$, $-$, $*$, $/$, ...), comparison functions ($>$, $<$, $=$, ...), conditionals (*IF THEN ELSE*), etc. and one might want to evolve expressions such as:

$$IF\ ((x > 10 * y)\ AND\ (y > 0))\ THEN\ z + y\ ELSE\ z * x$$

In such cases, introducing typed functions into the GP genome can help to force the closure property to be verified. GP in which each node carries its type as well as the types it can call, thus forcing functions calling it to cast the argument into the appropriate type, is called *strongly typed GP* [Banzhaf et al., 1998]. Using types might make even more sense in GP than with a human programmer, since a human programmer has a mental model of what she is doing, whereas the GP system

is completely random in its initialization and variation phases. Furthermore, type checking reduces the search space, which is likely to improve the search.

The sufficiency property requires that the set of terminals and the set of functions be capable of expressing a solution to the problem. For instance, the function and terminal sets of the previous example verify the sufficiency property if the problem at hand is an arithmetic one. It does not verify this property, for instance, if the problem faced is a logic one. For many domains, the requirements for sufficiency in the function and terminal sets are not clear and the definition of appropriate sets depends on the experience of the GP designer and his knowledge of the problem.

5.1.2 Initialization of a GP Population

Initialization of the population is the first step of the evolution process. It consists in the creation of the program structures that will later be evolved. The most common initialization methods in tree-based GP are the *grow* method, the *full* method and the *ramped half-and-half* method [Koza, 1992]. These methods will be explained in the following paragraphs, where the set of function symbols composing the trees will be indicated with $\mathscr{F}$, the set of terminal symbols with $\mathscr{T}$ and the maximum depth allowed for the trees with d.

Grow initialization. When the grow method is employed, each tree of the initial population is built using the following algorithm:

- a random symbol is selected with uniform probability from $\mathscr{F}$ to be the root of the tree;
- let n be the arity of the selected function symbol. Then n nodes are selected with uniform probability from the set $\mathscr{F} \cup \mathscr{T}$ to be its sons;
- for each function symbol between these n nodes, the method is recursively applied, i.e. its sons are selected from the set $\mathscr{F} \cup \mathscr{T}$, unless this symbol has a depth equal to $d-1$. In the latter case, its sons are selected from $\mathscr{T}$.

In other words, the root is selected with uniform probability from $\mathscr{F}$, so that no tree composed by a single node is created initially (even though, in some implementations, trees composed by one single node can be admitted in the initial popultaion). Nodes with depths between 1 and $d-1$ are selected with uniform probability from $\mathscr{F} \cup \mathscr{T}$, but once a branch contains a terminal node, that branch has ended, even if the maximum depth d has not been reached. Finally, nodes at depth d are chosen with uniform probability from $\mathscr{T}$. Since the incidence of choosing terminals from $\mathscr{F} \cup \mathscr{T}$ is random throughout initialization, trees initialized using the grow method are likely to have irregular shape, i.e. to contain branches of various different lengths.

Full initialization. The full initialization works like the grow initialization, with the difference that, instead of selecting nodes from $\mathscr{F} \cup \mathscr{T}$, the full method chooses

only function symbols (i.e. symbols in $\mathscr{F}$) until a node is at a depth equal to $d-1$. Then it chooses only terminals (i.e. symbols in $\mathscr{T}$). The result is that every branch of the tree goes to the full maximum depth.

Ramped half-and-half initialization. As first noted by Koza [Koza, 1992], population initialized with the above two methods may be composed by trees that are too similar between each other. In order to enhance population diversity, the ramped half-and-half technique has been developed. Let d be the maximum depth parameter. The population is divided equally among individuals to be initialized with trees having maximum depths equal to 1, 2, ..., $d-1$, d. For each depth group, half of the trees are initialized with the full technique and half with the grow technique.

5.1.3 Fitness Evaluation

Each program in the population is assigned a fitness value, representing its ability to solve the problem. This value is calculated by means of some well-defined explicit procedure. The two fitness measures most commonly used in GP are raw fitness and standardized fitness. They are described below.

Raw Fitness. Raw fitness, as defined by Koza, is "the measurement of fitness that is stated in the natural terminology of the problem itself". In other words, raw fitness is the most simple and natural way to calculate the ability of a program to solve a problem. For example, if the problem consists in driving a robot to make it pick up the maximum possible number of objects contained in a room, raw fitness of a program that drives the robot could be the number of objects effectively picked up after its execution.

Often, but not always, raw fitness is calculated over a set of *fitness cases*. A fitness case corresponds to a representative situation in which the ability of a program to solve a problem can be evaluated. For example, consider the problem of generating an arithmetic expression that appoximates a polynomial, like for instance $x^4+x^3+x^2+x$, over the set of natural numbers smaller than 10. Then, a fitness case is one of those natural numbers. Suppose x^2+1 to be one expression to evaluate, then we say that $2^2+1=5$ is the value assumed by this expression over the fitness case 2. Raw fitness is then defined as the sum of the distances over all fitness cases between values returned by perfect solutions and values returned by individuals to be evaluated. Formally, raw fitness f_R of an individual i, calculated over a set of N fitness cases, can be defined as:

$$f_R(i) = \sum_{j=1}^{N} |S(i,j) - C(j)|^k$$

where $S(i,j)$ is the value returned by the evaluation of individual i over fitness case j, $C(j)$ is the correct value for fitness case j and $k \in \mathbb{N}$.

Fitness cases are typically a small sample of the entire domain space and they form the basis for generalizing the results obtained to the entire domain space. The choice of how many fitness cases to use and which ones is often a crucial one and it may depend on the available data, on the experience of the GP designer and its knowledge of the problem.

Standardized Fitness. Standardized fitness restates the raw fitness so that a lower numerical value is always a better value. For cases in which lesser values of raw fitness are better, it can happen that standardized fitness equals raw fitness. In many cases, it is convenient and desirable to make the best value of standardized fitness equal zero. If not already the case, this can be obtained by subtracting or adding a constant. If, for a particular problem, a greater value of raw fitness is better, and the maximum possible value of raw fitness f_R^{max} is known, standardized fitness f_S of an individual i can be defined as:

$$f_S(i) = f_R^{max} - f_R(i)$$

where $f_R(i)$ is the raw fitness of i.

5.1.4 Selection

As pointed out in Chapter 3, selection depends on the phenotype of the individuals, while the genetic operators depend on their genotype. Given that what makes a difference between GP and GAs is the genotype, i.e. the structures representing the individuals evolving in the population, the selection operators of GP are identical to the ones of GAs. In particular, GP can implemented using fitness proportional selection (roulette wheel), ranking selection and tournament selection. The interested reader is referred to Section 3.1 for a presentation of these selection algorithms.

5.1.5 Genetic Operators

Given that GP differs from GAs in the genotype of the individuals evolving in the population, new genetic operators of crossover and mutation have to be defined for GP. The standard GP crossover and mutation are presented in the continuation.

Crossover. The crossover (sexual recombination) operator creates variation in the population by producing new offspring that consist of parts taken from each parent. The two parents, that will be called T_1 and T_2, are chosen by means of one of independent applications of a selection algorithm. Standard GP crossover [Koza, 1992] begins by independently selecting one random point in each parent (it will be called the crossover point for that parent). The crossover fragment for a particular par-

ent is the subtree rooted at the node lying underneath the crossover point. The first offspring is produced by deleting the crossover fragment of T_1 from T_1 and inserting the crossover fragment of T_2 at the crossover point of T_1. The second offspring is produced in a symmetric manner. Figure 5.2 shows an example of standard GP crossover.

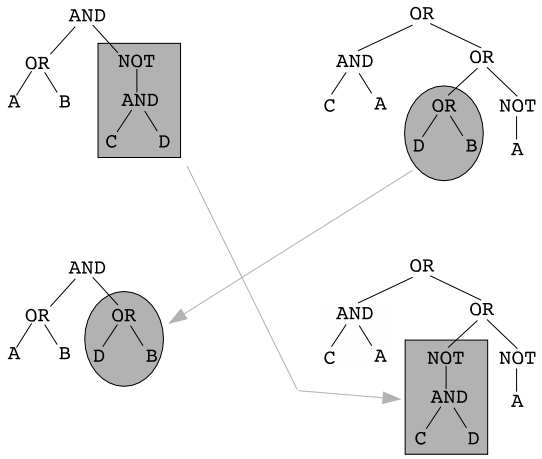


Fig. 5.2 An example of standard GP crossover. Crossover fragments are included into gray forms.

Because entire subtrees are swapped and because of the closure property of the functions, crossover always produces syntactically legal programs, regardless of the selection of parents or crossover points.

It is important to remark that in cases where a terminal and/or the root of one parent are located at the crossover point, generated offspring could have considerable depths. This may be one possible cause of the phenomenon of *bloat*, i.e. progressive growth of the code size of individuals in the population, that will be discussed later. For this reason, many variants of the standard GP crossover have been proposed in literature. The most common ones consist in assigning different probabilities of being chosen as crossover points to the various parents' nodes, depending on the depth level they are situated. In particular, it is very common to assign low probability of being selected as crossover points to the root and the leaves, so as to limit the influence of degenerative phenomena like the ones described above. Another different kind of GP crossover is *one-point crossover*, introduced in [Poli and Langdon, 1998a]. It deserves to be mentioned for the importance it has had in the development of a solid GP theory (see section 5.2).

Mutation. Mutation is asexual, i.e. it operates on only one parental program. Standard GP mutation, often called *subtree* mutation, begins by choosing a point at random, with uniform probability distribution, within the selected individual. This point is called mutation point. Then, the subtree laying below the mutation point is

removed and a new randomly generated subtree is inserted at that point. Figure 5.3 shows an example of standard GP mutation.

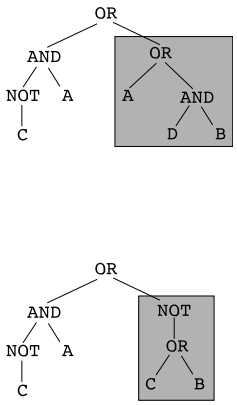


Fig. 5.3 An example of standard GP mutation.

This operation, as it is the case for standard crossover, is controlled by a parameter that specifies the maximum depth allowed and limits the size of the newly created subtree that is to be inserted. Nevertheless, the depth of the generated offspring can be considerably larger than the one of the parent.

As it happens for crossover, many variants of standard GP mutation have been developed too. The most commonly used are the ones aimed at limiting the probability of selecting the root and/or the leaves of the parent as mutation points. A special mention is deserved by the *point mutation* [Poli and Langdon, 1997] that exchanges a single node with a random node of the same arity, for the importance it has had in GP theory (see section 5.2). Other commonly used variants of GP mutation are: *permutation* (or *swap* mutation) that exchanges two arguments of a node and *shrink mutation* that generates a new individual from a parent's subtree.

5.1.6 Other GP Features

Steady State. In GP systems using steady state, one or more selected individuals, after undergoing variation, are immediately merged into the existing population. They take the place of one or two individuals that are thus removed from the population iiself. The removed individuals are often the ones with worst fitness values, but versions of GP systems also exist where the removed individuals are randomly selected or are the parents from which the new individuals have been generated. After the new individuals have been inserted into the population, selection is iterated and the new individuals are already taken into account for selection. In other words, steady state works as if only one single application of a genetic operator was performed

once at each generation. GP using steady state will be called *steady state GP* from now on and GP not using steady state will be called *generational GP*. Steady state exists also for GAs, and the analogous version are steady state GAs and generational GAs. In [Kinnear Jr., 1993], Kinnear shows a set of applications in which the use of steady state allows to obtain better performance than generational GP. However, generational GP continues to be the most popular and mostly used variant.

Constructs for Modularization. Automatically defined functions (ADFs) is one of the mechanisms by which GP implements the use of subroutines inside individuals. Simply speaking, ADFs are particular fractions of individuals (subtrees in tree-based GP). Each individual can be simply composed by nodes and/or by one or more ADFs. Each ADF may possess zero, one or more variables that play the role of formal parameters. ADFs allow parametrized reuse of code and hierarchical invocation of evolved code. The usefulness of ADFs has been shown by Koza in [Koza, 1994]. In the same vein, Koza defined automatically defined iterations (ADIs), automatically defined loops (ADLs), automatically defined recursions (ADRs) and automatically defined stores (ADSs) [Koza et al., 1999]. These mechanisms allow a high degree of reusability of code inside GP individuals.

5.1.7 GP Parameters

Once the GP user has decided the set of functions $\mathscr{F}$ and the set of terminals $\mathscr{T}$ used to represent potential solutions of a given problem and once the exact implementation of the genetic operators to employ have been chosen, still some parameters that characterize evolution need to be set. A list comprising some of these parameters is the following one:

- Population size.
- Stopping criterium.
- Algorithm used to initialize the population.
- Selection algorithm.
- Crossover type and rate.
- Mutation type and rate.
- Maximum tree depth.
- Presence or absence of steady state.
- Presence or absence of ADFs or other modular structures.
- Presence or absence of elitism (i.e. survival of the best individual(s) into the newly generated population).

The setting of each one of these parameters represents in general a crucial choice for the performance of the GP system. Much of what GP researchers know about these parameters is empirical and based on experience.

5.1.8 An Example Run

A very simple example of GP run is discussed here. The problem to be solved is the even parity 2 problem, and it consists in finding a Boolean function of 2 Boolean arguments, that returns *true* if an even number of its arguments evaluates to true and *false* otherwise (a generalization called the even parity k problem will be defined later in this chapter). Let A and B be the names of the two arguments, a Boolean function $f(A,B)$ perfectly solving this problem must respect the truth Table 5.1. Every line of this table represents a fitness case. For every boolean function of arguments A and B, its raw fitness is defined as the number of hits over all the fitness cases. Since the maximum raw fitness value, in this case, is equal to 4, we define the standardized fitness as equal to 4 minus the raw fitness. In this way, an optimal solution has a standardized fitness equal to 0 and the worst individuals have a standardized fitness equal to 4.

A	B	$f(A,B)$
false	false	true
false	true	false
true	false	false
true	true	true

Table 5.1 Truth table of the optimal individual for the the even parity 2 problem.

Let the following set of GP parameters be used: population size of 4 individuals, tournament selection of size 2, crossover rate equal to 50%, mutation rate equal to 25% and reproduction rate equal to 25%, maximum tree depth equal to 4 and ramped half-and-half initialization.[3] Let individuals be built with the set of functions $\mathscr{F} = \{and, or, not\}$ and the set of terminals $\mathscr{T} = \{A, B\}$.

The process begins with the generation of the initial population by the ramped half-and-half technique. Since the maximum tree depth is 4 and the population size is 4, the initial population will be probably composed of one individual of depth 1, one individual of depth 2, one individual of depth 3 and one individual of depth 4. Let the individuals composing the initial population be the following ones (prefix expressions instead of tree representations are given for the sake of simplicity):

1. $T_1 = not(A)$
2. $T_2 = or(and(A,B), and(A,A))$
3. $T_3 = and(and(A,B), or(A,A))$
4. $T_4 = not(or(B, not(or(A,B))))$

The following steps consist in evaluating the fitness of all the individuals composing the population. Tables of truth of individuals T_1, T_2, T_3, T_4 are shown in Figures 5.4(a), 5.4(b), 5.4(c) and 5.4(d), respectively. Let f_R be the raw fitness and f_S

[3] The reader is warned that these parameters are unrealistic and have been used for the sake of simplicity, just to show a first example of GP run.

A	B	T_1
false	false	true
false	true	true
true	false	false
true	true	false

(a)

A	B	T_2
false	false	false
false	true	false
true	false	true
true	true	true

(b)

A	B	T_3
false	false	false
false	true	false
true	false	false
true	true	true

(c)

A	B	T_4
false	false	false
false	true	false
true	false	true
true	true	false

(d)

Fig. 5.4 (a): Truth table of individual $not(A)$. (b): Truth table of individual $or(and(A,B),and(A,A))$. (c): Truth table of individual $and(and(A,B),or(A,A))$. (d): Truth table of individual $not(or(B,not(or(A,B))))$.

be the standardized fitness. From these tables, the following fitness values can be calculated:

1. $f_R(T_1) = 2 \rightarrow f_S(T_1) = 2$
2. $f_R(T_2) = 2 \rightarrow f_S(T_2) = 2$
3. $f_R(T_3) = 3 \rightarrow f_S(T_3) = 1$
4. $f_R(T_4) = 1 \rightarrow f_S(T_4) = 3$

Given the rates of reproduction, crossover and mutation, one individual will probably be selected for reproduction, two individuals for crossover and one for mutation. Let the tournament selection be first applied between T_1 and T_4. T_1 has a better fitness value and thus T_1 is reproduced, i.e. copied as it is in the new population. Analogously, let the two individuals chosen with the tournament technique for crossover be T_2 and T_3 and let

$$T_5 = and(and(A,B),and(A,A)), \quad T_6 = or(and(A,B),or(A,A))$$

be the two offspring to be inserted into the new population. Finally, let T_2 be the individual selected for mutation and let:

$$T_7 = or(and(A,B),and(A,not(B)))$$

be the generated offspring. Then, the new population is:

1. $T_1 = not(A)$
2. $T_5 = and(and(A,B),and(A,A))$
3. $T_6 = or(and(A,B),or(A,A))$
4. $T_7 = or(and(A,B),and(A,not(B)))$

At this point, the process is iterated on this new population. Truth tables analogous to the ones of Figure 5.4 could be written for T_1, T_5, T_6 and T_7. These tables would allow one to calculate the following values of standardized fitness:

1. $f_S(T_1) = 2$
2. $f_S(T_5) = 1$
3. $f_S(T_6) = 2$
4. $f_S(T_7) = 2$

Let T_6 be the individual selected for reproduction. Let T_1 and T_7 be the individuals selected for crossover and let

$$T_8 = A, \;\; T_9 = or(and(A,B), and(not(A), not(B)))$$

be the offspring to be inserted into the new population. Finally, let T_5 be the individual selected for mutation and let

$$T_{10} = and(and(A,B),B)$$

be the generated offspring. Then, the new population is:

1. $T_6 = or(and(A,B), or(A,A))$
2. $T_8 = A$
3. $T_9 = or(and(A,B), and(not(A), not(B)))$
4. $T_{10} = and(and(A,B),B)$

Iterating the process, and thus evaluating the fitness of all the individuals in the new population, T_9 is discovered to have a standardized fitness equal to 0, and thus it is an optimal solution. This leads the process to stop and to return T_9 as a solution to the given problem.

An important remark can be made on this example: consider, for instance, individual T_6: $or(and(A,B), or(A,A))$. This expression is clearly equivalent to $or(and(A,B),A)$ (in the sense that these two expressions have identical truth values). From this consideration, it is straightforward to deduce that GP individuals are not necessarily (and in general *are not*) in their most synthetic form. If the final user needs to have solutions in such a form, a *simplification* phase is necessary. The steps that enable to simplificate the structure of solutions are often called *rewrite rules*. This phase clearly depends on the type of language used to code GP individuals and it cannot be generalized.

5.2 GP Theory

After reading the introduction to GP given in Section 5.1, one question may come natural: why GP should work at all? This question can be made more precise by splitting it into the following ones: why the iterative GP process should allow to

build solutions of better and better fitness quality? And why should it allow to find a solution that is satisfactory for a given problem? Or even better: what is the probability of improving the fitness quality of solutions along with GP generations? What is the probability of finding a satisfactory solution to a given problem? The attempt to answer these questions has been an important research challenge in the GP field since its early years. The discussion contained in this section is inspired by the one in [Langdon and Poli, 2002], where Langdon and Poli offer a complete and detailed discussion of GP theory.

Being able to answer the above questions surely implies a deep understanding of what happens inside a GP population along with generations. One may think of somehow visualizing a population on a cartesian plane. In this way, one would often find that initially the population looks like a cloud of randomly scattered points but that, along with generations, this cloud "moves" in the seach space, following a well defined trajectory. This representation would probably provide interesting information about the dynamics regulating the GP process. But, since GP is a stochastic technique, in different runs different trajectories would be observed. Moreover, it is normally impossible to visualize a search space in all its features, given its high dimensionality and complexity. Thus, one may think of recording some numerical values concerning individuals of a population along with generations and of calculating statistics on them. These numerical values may be average individuals fitness, individuals length, differences between parent and offspring fitness and so on. Nevertheless, given the complexity of a GP system and its numerous degrees of freedom, any number of these statistical descriptors would be able to capture only a tiny fraction of the system's features.

For these reasons, the only way to understand the behavior of a GP system appears to be the definition of precise mathematical models. Theoretical studies on GAs have already been discussed in Section 3.4, where the GAs Schema Theorem was presented. Those studies that have inspired the formulation of a theory for GP. This theory, consisting of a rigorous probabilistic model for GP systems, can be considered as the milestone of GP theory. This section is not intended to explain this complex subject in details, but just to give a synthetic introduction.

Early GP Schema Theorems. Schema theory for GP had a slow start, one of the difficulties being that the variable size tree structure makes it harder to develop a definition of schema. Koza [Koza, 1992] was the first one to address schema theory in GP. However, his arguments are informal and only hint the existence of building blocks in GP. The first mathematical formulation of a schema theorem for GP is due to Altenberg [Altenberg, 1994]. He defined a schema as a subexpression and supplied equations for the proportion of a certain individual in the population at a certain generation and the probability that crossover picks up a certain expression from a randomly chosen program at a certain generation.

A formalization of a schema theorem for GP more similar to Holland's one for GAs is due to O'Reilly [O'Reilly and Oppacher, 1995]. Similarly to what happens for GAs, she defines a schema using a "don't care symbol" that allows to define an order and a length for schemata. She estimates the probability of disruption of

schemata by the maximum probability of disruption $P_d(H,t)$, producing the following schema theorem:

$$E[m(H,t+1)] \;\geq\; m(H,t)\;\frac{f(H,t)}{\overline{f}(t)}\;(1-p_c P_d(H,t))$$

where $f(H,t)$ is the mean fitness of all instances of schema H and $\overline{f}(t)$ is the average fitness in the population at generation t. The disadvantage of using the maximum probability is that it may produce a very conservative measure of the number of schemata at a given generation. Moreover, the maximum probability of disruption varies with the size of a given schema and makes it very difficult to predict which schemata will tend to multiply in the population and why.

Other interesting schema theorem formulations for GP are due to Whigham [Whigham, 1996], who produced a schema theorem for a GP system based on context free grammars, Rosca [Rosca, 1997] and Poli and Langdon [Poli and Langdon, 1998b], who gave a pessimistic lower bound (approximation) on the expected number of copies of a schema in the next generation. In the next section, schema theorems that give an exact expected number, rather than a bound will be considered.

Exact GP Schema Theorem. The development of an exact and general schema theory for GP is due to Poli and colleagues [Langdon and Poli, 2002]. Syntactically, a schema is a tree with some "don't care" nodes (labelled with the symbol "=") which represent exactly one node (primitive function or terminal). Semantically, a schema represents all programs that match its size, shape and defining (i.e. non-"don't care") nodes.

In addition to the defintion of schema, exact GP schema theory is based on the concept of *hyperschema* and *variable arity hyperschema* (VA hyperschema). An hyperschema is a tree composed of internal nodes from the set $\mathscr{F} \cup \{=\}$ and leaves from $\mathscr{T} \cup \{=,\#\}$, where the "#" symbol stands for any valid subtree. A VA hyperschema is a tree composed of internal nodes from the set $\mathscr{F} \cup \{=,\#\}$ and leaves from the set $\mathscr{T} \cup \{=,\#\}$, where the "=" "don't care" symbol stands for exactly one node, the terminal "#" stands for any valid subtree, while the function "#" stands for exactly one function of arity not smaller than the number of subtrees connected to it. If a function symbol "#' is matched by a function of greater arity than the number of subtrees connected to it, hen some arguments of that function are left unspecified.

Many formulations of exact schema theorems for GP have been developed, depending on the type of genetic operators used (one-point crossover, point mutation, standard crossover, etc.), and on weather each member of the population (*microscopic* schema theorems) or average population properties (*macroscopic* schema theorems) have to be considered. One of the most general forms of exact schema theorem for GP is probably:

Theorem 5.1 (Macroscopic Exact GP Schema Theorem for Standard Crossover).

Let selection and standard crossover be the genetic operators used by a GP system and let:

- $\alpha(H,t)$ *be the probability that the individuals produced by selection and crossover at generation* t *match schema* H *(also called total schema transmission probability for schema* H*).*
- p_{xo} *be the crossover rate.*
- $p(H,t)$ *the probability of selecting an individual matching schema* H *to be a parent at generation* t*.*
- $G_1, G_2, \ldots$ *be all the possible program shapes (i.e. all the fixed-size-and-shape schemata including only "=" symbols).*
- $N(K)$ *be the number of nodes in schema* K*.*
- $U(H,i)$ *be the hyperschema representing all the trees that match the portion of schema* H *above crossover point* i*; in other words* $U(H,i)$ *is the hyperschema obtained by replacing the subtree below crossover point* i *with a "#" node in* H*.*
- $L(H,i,j)$ *be the VA hyperschema representing all the trees that match the portion of schema* H *below crossover point* i*, but where the matching portion is rooted at some arbitrary node* j*.*

Then the following equality holds:

$$\alpha(H,t) = (1-p_{xo})\ p(H,t) + p_{xo} \sum_{k,l} \frac{1}{N(G_k)N(G_l)} \sum_{i \in H \cap G_k} \sum_{j \in G_l} p(U(H,i) \cap G_k, t)\ p(L(H,i,j) \cap G_l, t) \tag{5.1}$$

An even more general formulation of the exact schema theorem exists. It holds for *any* type of subtree swapping crossover and so it applies, for instance, to Koza's crossover, to one point crossover and many others. The interested reader can find this formulation in [Poli and McPhee, 2003a, Poli and McPhee, 2003b]. The contribution of GP exact schema theory to the comprehension of GP systems dynamics is undeniable. For instance, in [Langdon and Poli, 2002], Poli and Langdon have performed some experimental analysis based on GP exact schema theory, which has helped in deeply understanding how GP works on some concrete problems.

5.3 GP Benchmarks

GP has been applied to many fields in industry and science and has produced a large amount of results. The attempt to classify all the applications in which GP has been used since its early beginnings is probably hopeless, even though important efforts

can be found in [Langdon, 1996, Banzhaf et al., 1998]. In [Koza and Poli, 2003], Koza and Poli state that GP may be especially productive in areas having, among others, some or all of the following characteristics:

- where there is poor understanding of the problem at hand,
- where finding the size and shape of the ultimate solution is a major part of the problem,
- where there are good simulators to test performance of candidate solutions but poor methods to directly obtain good solutions,
- where conventional mathematical analysis does not, or cannot, provide analytic solutions,
- where an appoximate solution is acceptable.

One more characteristic can be added here:

- where it is impossible, or difficult, to write an algorithm to solve the problem.

Problems with one or more of these characteristics can, in some sense, be called "typical GP problems". In [Koza, 1992], Koza defines a set of problems that can be considered as belonging to this class and which have the relevant feature of being simple to define and to apply to GP. For this reason, they have been adopted by the GP research community as a, more or less, agreed upon set of benchmarks. Of course, the list presented here is not exhaustive, and many other benchmarks exist and are used in GP research, but it can be considered a good set of problems to be used by researchers to test their hypothesis, since it is composed by problems of different nature and often showing different behaviors. For a more complete list of GP benchmarks, that also contains pointers to freely available data, the reader is referred to [McDermott et al., 2012]. In particular, the first benchmark discussed here (even parity) is a boolean problem, the second one (symbolic regression) is a mathematical problem and the third one (the artificial ant) is a simple application of path planning in artificial intelligence.

5.3.1 Symbolic Regression

Given a set of vectors $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, where for all $i = 1, 2, ..., n, \mathbf{x}_i \in \mathbb{R}^m$, and a vector $\mathbf{t} = [t_1, t_2, ..., t_n]$, where for all $i = 1, 2, ..., n, t_i \in \mathbb{R}$, a symbolic regression problem can be generally defined as the problem of finding, or approximating, a function $\phi : \mathbb{R}^m \to \mathbb{R}$, also called target function, such that:

$$\forall i = 1, 2, ..., n : \ \phi(\mathbf{x}_i) = t_i$$

GP is typically used to solve symbolic regression problems using a set of primitive functions $\mathscr{F}$ that are mathematical functions, like for instance the arithmetic functions or others, and a set of terminal symbols $\mathscr{T}$ that contain at least m different real valued variables, and may also contain any set of numeric constants. In this way, a

GP individual (or program) P can be seen as a function that, for each input vector $\mathbf{x}_i$ returns the scalar value $P(\mathbf{x}_i)$. In symbolic regression, to measure the fitness of an individual P any distance metric (or error) between the vector $[P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)]$ and the vector $[t_1, t_2, ..., t_n]$ can be used. Just as a matter of example, one may use, for instance, the mean Euclidean distance, or root mean square error, and in this case the fitness $f(P)$ of a GP individual P is:

$$f(P) = \sqrt{\frac{\sum_{i=1}^{n} (P(\mathbf{x}_i) - t_i)^2}{n}} \quad , \tag{5.2}$$

or one may use the Manhattan distance, or absolute error, and in this case the fitness $f(P)$ of a GP individual P is:

$$f(P) = \sum_{i=1}^{n} |P(\mathbf{x}_i) - t_i|. \tag{5.3}$$

Using any error measure as fitness, a symbolic regression problem can be seen as a minimization problem, where the optimal fitness value is equal to zero (in fact, any individual with an error equal to zero behaves on the input data exactly like the target function ϕ). As it is customary in Machine Learning, vectors $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ are usually called input data, input vectors, training instances or *fitness cases*, while the values $t_1, t_2, ..., t_n$ are usually identified as the corresponding *target values*, or expected output values. $\mathbf{X}$ is usually called *dataset*. Finally, the values $P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)$ are usually called the *output values* of individual P on the input data.

Example 5.2. Let $\mathbf{X} = \{[3,12,1],[5,4,2]\}$ and let $\overrightarrow{t} = [27,13]$. For instance, GP individuals may be coded using as primitive functions the set of arithmetic operators $\mathscr{F} = \{+,-,*\}$ and as terminals a set of three real valued variables (since the input vectors have cardinality equal to 3) $\mathscr{T} = \{k_1, k_2, k_3\}$. In this way, the search space contains all the trees that can be built by composing the symbols in $\mathscr{F}$ and $\mathscr{T}$ (with the only exception that usually a maximum possible depth is imposed to the trees, as previously discussed). Using, for instance, the absolute error, one may calculate the fitness of an individual like, for instance:

$$P(k1, k2, k3) = k3 * (k1 - k2).$$

To do that, one has to first calculate the output values of P on the input data. In other words, one has to calculate $P(3,12,1)$ (obtained by replacing the values of the first input vector in the dataset $\mathbf{X}$ with k_1, k_2 and k_3 respectively in the expression of P) and $P(5,4,2)$ (obtained by replacing the values of the second input vector in the dataset $\mathbf{X}$ with k_1, k_2 and k_3 in P). So, the fitness of P is:

$$\begin{aligned} f(P) &= |P(3,12,1)-27| + |P(5,4,2)-13| = \\ &= |(1*(12-3))-27| + |(2*(5-4))-13| = \\ &= |9-27| + |2-13| = 18+11 = 29. \end{aligned}$$

It is not difficult to realize that, in this example, a global optimum, i.e. an individual that has a fitness equal to zero, is:

$$P_{opt}(k_1,k_2,k_3) = k_1 + k_2 + k_2.$$

From this, we can see that GP individuals do not have to necessarily use all the variables in $\mathscr{T}$ (for instance, P_{opt} is not using k_3).

In this simple example, only the binary operators of sum, subtraction and multiplication have been used. When division is also used, it is typical to "protect" it in some way from failure in case the denominator is equal to zero. The oldest and more popular method to protect division is to replace it with an operator that is equal to the division if the denominator is different from zero and that returns a constant value otherwise [Koza, 1992]. Nevertheless, several more sophisticated methods have been introduced [Keijzer, 2003].

Generalization. Symbolic regression can be seen as a supervised machine learning problem (supervised because the target values $t_1, t_2, ..., t_n$ are known for every vector of input data in the dataset). As such, the objective is usually finding a model of the data that may work not only for the information that is known (the data in the dataset), but that also works for other, unseen data. In this sense, we say that the model should be *general* and the computational method that generates the model (GP in our case) should have *generalization ability*.

Just as a matter of example, one may consider an application of drug discovery, where the various input vectors $\mathbf{x}_i$ are vectors of molecular descriptors, i.e. vectors of numbers that univocally represent a molecular compound that is a candidate new drug. For each one of these molecular compounds, the corresponding target value represents the toxicity of that compound. In this situation, the objective of the problem is finding a function (data model) that, when applied to the vector of molecular descriptors of a particular compound, returns as output an approximated value of the toxicity of that compound. But, what would be the point of having a function that works correctly in returning the toxicity *just* for the compounds that we have in the dataset? In the end, for *those* compounds, we *already* know the value of the toxicity: it is their target value, which is in the dataset. The objective is to discover the relationship between input data (molecular descriptors) and expected outputs (values of the toxicity), "coding" that relationship in the function (data model) that GP, or another computational method, is supposed to find. In this way, applying this function to *other* molecular compounds, that are not included in the dataset, and for which we *don't know* the value of the toxicity, we should be able to have a faithful estimate of it.

Several ways have been defined so far to test the generalization ability of GP. One of the most simple and intuitive ones consists in partitioning the dataset into two subsets, and using only one of them, called *training* set, to generate the data model. The remaining portion of the dataset, called *test* set, is then used only to verify that the model that has been found is general enough to faithfully approximate the target also on data that have not been used to generate it.

Symbolic Regression with Synthetic Functions. Symbolic regression has a huge number of real-life applications. Nevertheless, real world datasets are often very complex, and they may be noisy and contain errors. For this reason, when we want to study the properties of GP and other Machine Learning algorithms, it is often useful to create artificial, or synthetic, datasets with some known properties. A popular way of doing it is to consider a given known target function, and create a dataset that contains some inputs and some corresponding outputs, generated by that function. The objective is to test if GP, and/or other algorithms, are able to find that function, using only the considered points. For instance, one may consider a function (also called quartic polynomial) like: $f(x) = x^4 + x^3 + x^2 + x$. A dataset could be created by using as input data a set of values for x (say, for instance, the first 100 natural numbers), and as corresponding targets the corresponding values of $f(x)$.

The quartic polynomial is known to be a rather simple function for GP to find. It can be made harder to defining numerical coefficients for the different terms. A large set of other typical synthetic symbolic regression benchmarks can be found in [McDermott et al., 2012].

5.3.2 Boolean Problems

Boolean problems are similar to symbolic regression, with the only difference that both input data and the corresponding targets are Boolean values. Many real-life applications of Boolean problems exist, the most typical ones probably being the automatic synthesis of integrated electric circuits, on which GP reported several applicative successes so far [Koza, 1992]. As it happens for symbolic regression, also for Boolean problems it is possible that real-life datasets are too complex to be used when the properties of the algorithms need to be investigated. For this reason, it is typical to create artificial, or synthetic, Boolean problems, with known properties, and use them as benchmarks to test the dynamics of GP and other algorithms.

One of the most popular Boolean synthetic problems is the even-k parity problem [Koza, 1992]. The goal of this problem is to find a Boolean function of k Boolean arguments that returns *true* if an even number of its boolean arguments evaluates to true, and that returns *false* otherwise. A very simple case, the even parity 2, has been already introduced in Section 5.1.8, where a function perfectly solving this problem has been found. This function is:

$$f(A,B) = or(and(A,B), and(not(A), not(B)))$$

and its truth values for all possible values of its boolean arguments A and B are represented in Table 5.1. As this table shows, each time an even number (0 or 2, in this case) of arguments is true, f returns true and each time an odd number (1, in this case) of arguments is true, f returns false. In general, the number of fitness cases, i.e. the number of all possible permutations of the values of the k parameters is 2^k. Fitness is usually computed as 2^k minus the number of hits over the 2^k cases. Thus a perfect individual has fitness equal to 0, while the worst individuals have fitness equal to 2^k. Differently of what happens in Section 5.1.8, a typical set of functions employed for this problem is $\mathscr{F} = \{nand, nor\}$, while the terminal set is usually composed of k different boolean variables.

Besides the even-k parity problems, other known Boolean GP benchmarks are the h-multiplexer [Koza, 1992] and the FPGA [Vanneschi, 2004].

5.3.3 The Artificial Ant on the Santa Fe Trail

In this problem [Koza, 1992], an artificial ant is placed on a 32×32 toroidal grid. Some of the cells from the grid contain food pellets. The goal is to find a navigation strategy for the ant that maximizes its food intake. The ant starts in the upper left cell of the grid, identified by the coordinates (0, 0), facing east. It has a very limited view of its world. In particular, it has a sensor that can see only a single immediately adjacent cell in the direction the ant is currently facing. Food is placed on the grid according to the "Santa Fe trail", an irregular trail consisting of 89 food pellets. The trail is not straight and continuous, but instead has single gaps, double gaps and triple gaps at corners. Figure 5.5 shows the Santa Fe trail. Food is represented by solid black squares, while gaps are represented by gray squares. Numbers identify key features along the trail, for example the number 3 highlights the first corner, number 11 the first single gap, and so on.

The set of terminals used for this problem is usually $\mathscr{T} = \{Right, Left, Move\}$, and corresponds to the actions the ant can perform: turn right by 90°, turn left by 90° and move forward in the currently facing direction. When the ant moves into a square containing a food pellet, it eats the food (thereby eliminating the pellet from that square and erasing the trail). The set of primitive functions that are typically used for this problem is $\mathscr{F} = \{IfFoodAhead, Progn2, Progn3\}$. $IfFoodAhead$ is a conditional branching operator that takes two arguments and executes the first one if and only if a food pellet is present in the case that is adjacent to the ant in the direction the ant is facing, and the second one otherwise (it represents the information the ant can get by its sensor). $Progn2$ and $Progn3$ are common LISP operators; $Progn2$ takes two arguments and causes the ant to unconditionally execute the first argument followed by the second one. $Progn3$ is analogous, but it takes three arguments, that are executed in an ordered sequence. An individual built with these sets $\mathscr{F}$ and $\mathscr{T}$ can be considered as a "program", that allows the ant to navigate the grid. As fitness function, the total number of food pellets lying on the trail (89) minus the amount of food eaten by the ant in a fixed amount of time is considered. This turns the problem

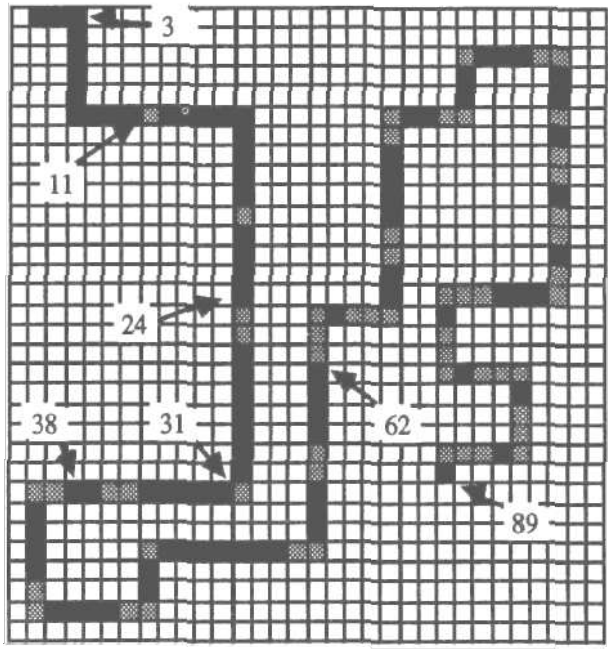


Fig. 5.5 The Santa Fe trail used for the artificial ant problem.

into a minimization one, like the previous ones. Each move operation and each turn operation is considered as taking one time step, and the ant is usually limited to 600 time steps. This time-out limit is sufficiently small to prevent a random walk of the ant to cover all the 1024 squares, and thus eating all the food, before timing out.

5.4 GP Open Issues

5.4.1 GP Problem Difficulty

The capability of GP to find good solutions varies from problem to problem. To convince oneself of this fact, one could perform the following experiment: execute the artificial ant on the Santa Fe trail problem (see Section 5.3.3) for, say, n generations; record the values $f_1, f_2, \ldots, f_n$, where, for each $g = 1, 2, ..., n$, f_g is the fitness of the best individual found in the population at generation g. Since GP is based on stochastic operators, these values may be fortuitous and surely lack statistical significance. For this reason, as we have studied for GAs in Section 3.6, it is suitable to repeat the execution a number of times, say 100, and consider average (or median) results. Thus, let $\overline{f}_g$ be the average of the best fitness values found at generation g over the 100 runs performed and plot on a cartesian bidimensional plane the val-

ues $\overline{f}_1, \overline{f}_2, \ldots, \overline{f}_n$ (on the ordinates) against generation numbers (on the abscissas). Now, repeat the same process for a symbolic regression problem (see Section 5.3.1), aimed at finding a simple equation, like for instance $x^4 + x^3 + x^2 + x$ (called quartic polynomial). Let the GP parameters used to solve these two different problems be identical (except, of course, for the sets of functions $\mathscr{F}$ and of terminal symbols $\mathscr{T}$). Figure 5.6 shows the results of this experiment, where the set of GP parameters for both problems was the following one: population size of 2500 individuals, ramped half-and-half initialization, tournament selection of size 10, crossover rate equal to 95% (standard GP crossover [Koza, 1992]), mutation rate equal to 0.1% (standard subtree mutation [Koza, 1992]), maximum tree depth for the initialization phase equal to 6, maximum depth for the crossover and mutation phases equal to 17, elitism (i.e. the best individual is always copied unmodified into the new population at each generation). In the case of the symbolic regression problem, 100 equidistant points included into the range $[0,5]$ have been used as fitness cases. The gray curve represents results of artificial ant and the black one results of symbolic regression. This figure clearly shows that, for the symbolic regression problem, the optimal so-

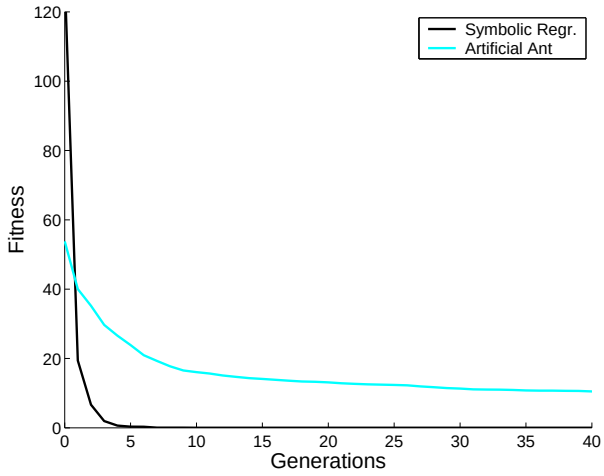


Fig. 5.6 Symbolic regression problem and artificial ant problem. Evolution of best fitness during the first 40 generations. The two curves have been obtained using the same GP parameters (see text) and are averages over 100 independent GP runs.

lution (fitness equal to zero) has been found (or at least appropriately approximated) in all the executed runs before generation 8. This is surely not the case for the artificial ant problem, since the gray curve does not touch the abscissas axis during the first 40 generations. Data not shown here confirm that, even if generations until 500 were considered, still the gray curve would not touch the abscissas axis. Moreover, less than 50 runs, over the total 100 runs executed, have lead to the discovery of one optimal solution before generation 500 for the artificial ant problem.

These different behaviors are usual in GP, when considering different problems. Thus, the following questions arise naturally: why does it happen? Why some problems are easily solved by GP, while others are so hard? The following observations can be made about the experiment shown in figure 5.6:

(i) The GP parameters used for the two considered problems are exactly the same.
(ii) The range of all possible fitness values that an individual of the symbolic regression problem can take is much larger than the ones that an individual of the artificial ant can take. In fact, fitness values for the ant problem range from 0 to 89 by definition (since 89 is the number of food pellets on the Santa Fe trail), while the best fitness value in the population at generation 0 for the symbolic regression problem was, on average, around 120 in the experiment shown (see figure 5.6). Furthermore, the "granularity" in the fitness function is very different: only 90 different fitness values are possible for the ant problem, while fitness is continuous for symbolic regression, and so an infinite number of values are possible.
(iii) Trees in the ant problem can be built using just three primitive functions (*IfFoodAhead*, *Progn2*, *Progn3*), while in the symbolic regression problem, they can be built using four primitive functions ($+$, $-$, $*$ and $//$). Furthermore, arities of the operators are also different: all the operators used for symbolic regression have an arity equal to 2, while two operators used for the ant have an arity equal to 3 and one of them has an arity equal to 2.

Observation (i) leads to the conclusion that GP parameters, even though they can have an influence on the performance of a GP system, do not bring a sufficient amount of information to fully understand its dynamics and justify its behavior. Observations (ii) and (iii) allow one to conclude that the size of the set of all possible fitness values and the size of the set of all possible trees, although probably not irrelevant, are not valid criteria to establish the difficulty of a problem for GP. In other words, the structure of the fitness lanscape, and not only its size, matters.

Thus, answering the question "what makes a problem easy or hard for GP?" does not seem to be an easy task. Several works where published in the last decade, trying to given an answer to this ambitious question, many of them collected in [Vanneschi, 2004]. The most relevant are arguably the ones aimed at defining mathematical measures, able to capture some characteristics of fitness landscapes that can have implications on the difficulty of the problem. The most successful of these measures, even though not without flaws, is the *fitness-distance correlation* [Tomassini et al., 2005], indicating that the relationship between fitness and distance to the goal can give reliable indication of the hardness of the problem, assuming that the distance is appropriately related to the genetic operators used to evolve [Gustafson and Vanneschi, 2005]. Another measure that can give some useful indications is the *negative slope coefficient* [Vanneschi et al., 2006], based on the concept of *fitness clouds* [Vanneschi et al., 2004].

5.4.2 Slowness and Computational Resources Consumption

In [Banzhaf et al., 1998], Banzhaf and colleagues analyse the reasons why GP is computationally intensive, identifying three main factors:

- GP generally needs large populations to produce acceptable results (according to Banzhaf et al., typical population sizes for real-life applications variates from 500 to 5000 individuals, but even larger populations are used in many recent applications),
- programs' size tends to grow along with generations (see Section 5.4.4),
- the number of fitness evaluations to execute is generally large, depending on the size of the fitness cases sets (according to Banzhaf et al., typical sizes of these sets for real life applications range from 50 to 5000 fitness cases, but modern datasets can contain much more in some applications).

Indeed, almost all of the time consumed by a GP run is spent performing fitness evaluations. These three factors have an impact both on CPU time and on memory occupation. Furthermore, for the majority of the applications, these activities have to be iterated for a large number of generations in order for the system to evolve towards good solutions.

To give an idea of the CPU time used by GP for an execution of a real-life problem, 20 independent runs of the FPGA problem have been performed (see [Fernández et al., 2001] for an introduction to this problem). The following GP parameters have been used in this experiment: population size of 500 individuals, ramped half-and-half initialization, tournament selection of size 10, crossover rate equal to 95%, mutation rate equal to 0.1%, maximum tree depth for the initialization phase equal to 6, maximum depth for the crossover and mutation phases equal to 50, elitism (i.e. copy of the best individual unchanged in the next population). Table 5.2 shows the times that have been spent to execute each one of these 20 runs until generation 500. The total CPU time needed to execute the totality of all these 20 runs as well as the average over the 20 runs are shown by the last two lines of this table. Note that the differences between the times used by each one of the 20 runs are due to the fact that GP operates on randomly generated populations and uses stochastic operators. All the runs have been executed on a single PC Intel Pentium III-500 with 128M RAM, so they may be smaller if more modern and powerful computers were used. The CPU time needed to execute all these 20 runs has been 1448' 31", i.e. approximately 24 hours. The obvious conclusion is that GP actually consumes large amounts of CPU time[4].

One significant way of reducing the running time of GP (as the one of any population-based algorithm) is the one of parallelizing it. Evolutionary algorithms can be parallelized very naturally, because they have an intrinsecally parallel nature. To convince oneself of this fact, one may for instance think that fitness evaluations

[4] The FPGA problem is known to be particularly costly, because good individuals have, in general, large size. Nevertheless, the problem of large computational resources needed by GP is common to many other problems.

	completion time
run 1	82’ 48”
run 2	47’ 06”
run 3	114’ 36”
run 4	41’ 06”
run 5	59’ 54”
run 6	88’ 30”
run 7	63’ 53”
run 8	59’ 32”
run 9	94’ 49”
run 10	78’ 57”
run 11	34’ 44”
run 12	48’ 31”
run 13	82’ 13”
run 14	54’ 41”
run 15	47’ 06”
run 16	87’ 46”
run 17	74’ 43”
run 18	92’ 12”
run 19	92’ 06”
run 20	103’ 18”
total	1448’ 31”
average	72’ 25”

Table 5.2 CPU times used by GP to execute 20 independent runs of the FPGA problem until generation 500. ’ means minutes and ” means seconds. The last two lines show the total amount of time and the average of these 20 results.

of te different individuals in the population are all independent between each other and so they can, in principle, be performed in parallel, each one on a different processor. Another popular way of parallelizing evolutionary algorithms, and GP in particular, is partitioning the population into smaller sub-populations, evolving independently, and communicating individuals from time to time. Besides reducing the global computational time of the process, multi-population GP was also demonstrated to be able to generate solutions of better quality, compared to traditional, single-population GP [Fernández et al., 2003].

5.4.3 Premature Convergence or Stagnation

After a certain number of generations, GP populations, as it is the case for GAs and any other evolutionary algorithm, tend to contain individuals that are similar, or even identical, between each other. To show this phenomenon, measures analogous to the ones introduced in Section 3.5.1 for GAs can be used. Many reasons can be given for premature convergence. First of all, the selection mechanism acts on phenotypes, thus assigning only to few good quality individuals high probabil-

ities of surviving. Secondly, according to schema theory, building blocks tend to multiply into the population, thus introducing similar substructures inside individuals [Langdon and Poli, 2002].

Many techniques have been developed by GP researchers to maintain diversity, both genotypic and phenotypic, inside GP populations (see, for instance, [Ekárt and Németh, 2002]). Although these techniques have often proved very useful to understand why premature convergence happens, their main drawback is generally that they are time consuming, since they demand the calculation of measures like structural distances, entropy or variance, at each generation. This generally causes an increment in the total completion time of GP systems, which may make them unusable in practice (even though efficient methods to approximate diversity measures have been defined [Burke et al., 2002]). Otherwise, new genetic operators, like different kinds of selections, crossovers or mutations, have been defined in order to produce offsprings that are as different as possible from the parents, or from the other individuals in the population, by construction. These techniques generally succeed in maintaing diversity inside populations, but substantially change the GP algorithm (in particular the genetic operators used) and thus the behavior of GP systems. [Fernández et al., 2003] demonstrated that parallelizing the GP process, and organizing it into separate and interacting sub-populations, helps in limiting premature convergence and its deleterious effects on GP. The migration of candidate solutions from one population to the other, in fact, allows individuals having different ancestors, and thus possibly different evolutionary histories and behaviors, to enter converging populations and to inject diversity inside them. Moreover, since (copies of) the fitter individuals of each subpopulation are sent to the other ones, good quality of solutions should be maintained together with diversity (one could informally say that not just "diversity", but "good diversity" is maintained).

5.4.4 The Problem of Bloat

The rapid growth of programs produced by GP has been widely studied (see for instance [McPhee and Poli, 2001, Poli, 2001, Silva and Costa, 2009]). Figure 5.7 shows the average number of nodes composing trees in the population (called average tree *length*) against generations for the artificial ant, the even parity 4, and the symbolic regression on the quartic polynomial with the same set of GP parameters as the ones used in Section 5.4.1. Results are averages over 100 independent runs. This growth, often called "code bloat", is not necessarily correlated to increases in fitness (in agreement with [Langdon and Poli, 2002]). Thus bloat consists primarly of code that does not change the semantics of the evolving programs. These "useless" pieces of code are often called *introns*. Some theories exist of why bloat happens. The oldest one is that introns are generated to "protect" useful code from the disruptive effects of crossover [Tackett, 1994]: fit individuals containing introns are less likely to be disrupted by crossover, so their children are more likely to be at least as fit as they are. This theory has been criticized in [Luke, 2000]. An-

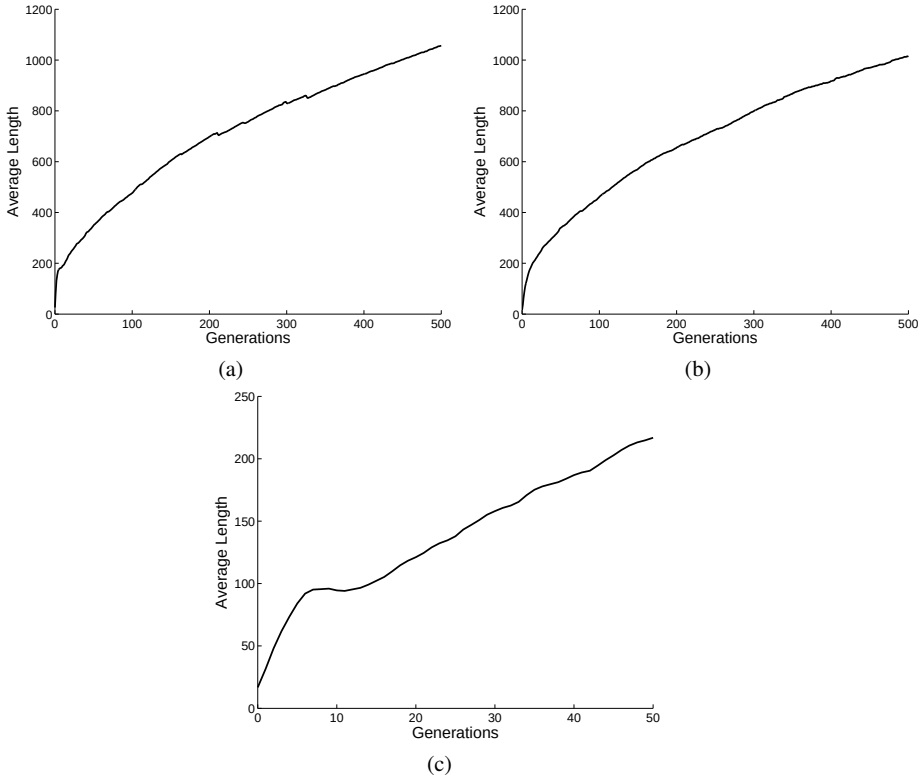


Fig. 5.7 Average tree length against generations for the following problems: (a): artificial ant on the Santa Fe trail, (b): even parity 4, and (c): symbolic regression on the quartic polynomial.

other theory is that children produced removing small subtrees are likely to have better fitness. Since the inserted subtree is randomly chosen, it will be on average bigger than that removed, and thus children will also be larger than their parents. This theory has been formulated in [Soule and Foster, 1998] and confirmed as a possible cause of bloat in [Langdon, 2000]. Finally, bloat has been described as a random spread along neutral networks, i.e. a network of programs of the same fitness connected by genetic operators [Langdon et al., 1998]. According to this theory, above a certain size threshold there are exponentially more larger programs of the same fitness as the current best individual than the ones of the same length or shorter [Langdon and Poli, 2002]. An important result has been offered by Poli, who produced a *size evaluation equation* for GP, that does not try to "explain" bloat, but that defines exactly what are the ingredients for bloat to occur. This equation has led to the definition of a method, called *Tarpeian* method, to control bloat [Poli, 2003].

As for the case of premature convergence, many techniques (other than Poli's Tarpeian method) have been proposed to counteract bloat. Almost all of them

operate directly on the GP algorithm, modifying it in different ways, with the goal of maintaining small sized and good fitting offprings and discarding the large ones (see for instance [Zhang, 2000, Silva, 2008, Silva and Costa, 2009, Silva and Vanneschi, 2012]). [Fernández et al., 2003] has also indicated that parallelizing GP systems can have positive effects on bloat. This seems to indicate a relationship between the size of the individuals in a population and the size of the population itself, as confirmed in [Poli et al., 2008].

5.5 Geometric Semantic Genetic Programming

5.5.1 Semantics in Genetic Programming

Geometric Semantic GP (GSGP) is the name we give to GP that uses two novel genetic operators, called geometric semantic operators, instead of the traditional crossover and mutation. These new operators are based on the concept of semantics, that is briefly introduced here. As discussed previously, let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ be the set of input data, or fitness cases, of a supervised learning problem, and $\mathbf{t} = [t_1, t_2, ..., t_n]$ the vector of the respective expected output or target values. Let P be a GP individual (or program). As discussed previously, P can be seen as a function that, for each input vector $\mathbf{x}_i$ returns a value $P(\mathbf{x}_i)$. Following [Moraglio et al., 2012], the *semantics* of P is given by the vector:

$$\mathbf{s_P} = [P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)]$$

In other words, from now on, we indicate with the term semantics the vector of the output values of a GP individual on the input data. Even though the concept of semantics is general, and applies to any type of supervised learning problem, it is particularly intuitive to discuss the case of symbolic regression. So, for simplicity, and without forgetting that it is just a particular case study while the concepts are general, let us assume that the problem at hand is a symbolic regression one. In this case, $\mathbf{s_P} \in \mathbb{R}^m$, and so $\mathbf{s_P}$ can be represented as a point in a n-dimensional Cartesian space, that we call *semantic space*. Remark that the target vector $\mathbf{t}$ itself is a point in the semantic space and, since we are dealing with supervised learning, it is known. What is not known is the tree structure of a GP individual that has $\mathbf{t}$ as its own semantics. Basically, when working with GP, one may imagine the existence of two different spaces: one, that we call genotypic or syntactic space, in which GP individuals are represented by tree structures (or, according to the different considered variant of GP, linear structures, graphs, grammars, etc.), and one, that we call semantic space, in which GP individuals are represented by their semantics, and thus by points. The situation is exemplified in Figure 5.8, where, for simplicity, the semantic space is represented in two dimensions, which corresponds to the toy case in which only two training instances exist. The figure also shows that to each tree in the genotypic space corresponds a point in the semantic space. This corre-

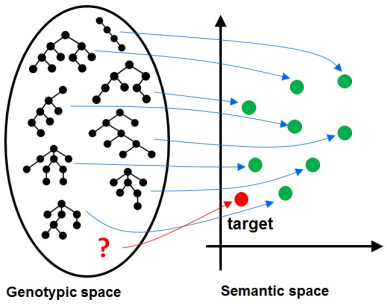


Fig. 5.8 When we work with GP, we can imagine the existence of two spaces: the genotypic space, in which individuals are represented by their structures, and the semantic space, in which individuals are represented by points, that are their semantics. In figure, the semantic space is represented in 2D, which corresponds to the unrealistic case in which only two training instances exist.

spondence is surjective and, in general, not bijective, since different trees can have the same semantics. Once we have the semantics of a GP individual, its fitness can be calculated as a distance between the semantics and the target, using any metric. For instance, Equation (5.2) at page 124 is the mean Euclidean distance between $\mathbf{s_P} = [P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)]$ and $\mathbf{t} = [t_1, t_2, ..., t_n]$, while Equation (5.3) at page 124 is the Manhattan distance between $\mathbf{s_P}$ and $\mathbf{t}$.

Example 5.3. Let us consider the same symbolic regression problem as in Example 5.2 (page 124). In that problem, the set of input data was $\mathbf{X} = \{[3,12,1],[5,4,2]\}$ and the vector of the corresponding target values was $\mathbf{t} = [27,13]$. In that example, we have considered the set of primitive functions $\mathscr{F} = \{+,-,*\}$ and a set of terminals composed by three real valued variables $\mathscr{T} = \{k_1, k_2, k_3\}$. Also, in that example, we have studied an individual, whose expression in infix notation is $P(k1,k2,k3) = k3 * (k1 - k2)$. What is the semantics of this individual?

The semantic of $P(k1,k2,k3) = k3 * (k1 - k2)$, in this case, is a vector of dimension equal to 2, because there are 2 training instances (the vectors contained in dataset $\mathbf{X}$), where the first component is equal to the evaluation of P on the first input vector and the second component is equal to the evaluation of P on the second input vector. In other words, the first component is obtained by replacing the values 3, 12 and 1 respectively to k_1, k_2 and k_3, and evaluating P. Analogously, the second component is obtained by replacing the values 5, 4 and 2 to k_1, k_2 and k_3. So:

$$\overrightarrow{s_P} = [P(3,12,1), P(5,4,2)] = [1 * (12-3), 2 * (5-4)] = [9,2]$$

Using, for instance, the absolute error between semantics and target, we have that the fitness of P is:

$$f(P) = d([9,2],[27,13]) = |9-27| + |2-13| = 18 + 11 = 29$$

where, in this case, d is the Manhattan distance. Comparing this calculation with the one of Example 5.2 (page 124) to calculate the fitness of P, we can clearly see that these two calculations are completely identical.

5.5.2 Similarity Between Semantic Space of Symbolic Regression and Continuous Optimization

The reader is now invited to have a look back at Section 3.7 at page 96, where continuous optimization problems were introduced, and geometric genetic operators of GAs were presented for those problems. Let us now consider a particular instance of a continuous optimization problem, with the following characteristics:

- $S = \{\mathbf{v} \in \mathbb{R}^n \mid \forall i = 1,2,...,n : v_i \in [\alpha_i, \beta_i]\}$
- $\forall \mathbf{v} \in S: \ f(\mathbf{v}) = d(\mathbf{v}, \mathbf{t})$
- Minimization problem.

where d is a distance measure (think, for simplicity, of the Euclidean distance, but the reasoning holds also for any other metric), and $\mathbf{t} \in \mathbb{R}^n$ is a predefined, and known, global optimum. In informal terms, solutions of this problem instance are vectors of n real numbers included in a predefined interval, and the fitness of an individual is equal to the distance of that individual to a unique, and known, globally optimal solution. Let us now assume that we are trying to solve this problem with GAs, using geometric mutation (also called box mutation, or ball mutation). As we have studied in Section 3.7, in this situation, the fitness landscape of this problem is unimodal. In other terms, there are no local optima, except for the global optimum. For this reason, the problem is characterized by a very good evolvability. In order to convince herself, the reader is advised to implement it. It will not be difficult to verify that, simply applying geometric mutation iteratively, it is possible to find solutions that are arbitrarily close to the global optimum. Just to fix a simple terminology, let a problem with these characteristics be called CONO, which stands for Continuous Optimization with Notorious Optimum. We find this acronym particularly appropriate, because the shape of the fitness landscape for this problem is actually a cone[5], where the vertex represents the global optimum. In synthesis:

IF box mutation is the operator used to explore the search space
THEN The CONO problem has a unimodal fitness landscape

Let us now observe the right part of Figure 5.8, showing the semantic space of a symbolic regression problem In this space, individuals are points and the target is known. It should not be hard to understand that this space is identical to the space of solutions of the CONO problem. This fact has a very important consequence on the solution of symbolic regression problems using GP. Similarly to the above property, we could now write:

[5] The word "cono" actually means cone in several languages of Latin origin, among which Italian and Spanish.

IF we define a GP operator that works like box mutation on the semantic space
THEN **any** *symbolic regression problem has a unimodal fitness landscape*

Let us repeat this property again, but with different words:

if we were able to define, on the syntax of the GP individuals (i.e. on their tree structure)[6] *an operator that has, on the semantic space, the same effect as box mutation, then the fitness landscape is unimodal for any symbolic regression problem.*

In other terms, if we were able to define such an operator, then we would be able to map any symbolic regression problem into an instance of the CONO problem that uses box mutation, in the sense that we would be able to perform exactly the same actions in a space (the semantic space of GP) that is identical. As such, we would inherit the same fitness landscape properties, i.e. a unimodal fitness landscape. It is worth stressing here one point:

This property would hold for any symbolic regression problem, independently on how large or complex the data of the problem are.

Since no locally optimal solution exists, actually *any* symbolic regression problem could be *easy* to optimize for GP (at least on the training set, were fitness is calculated), including problems characterized by huge amounts of data. This fact would clearly foster GP as a very promising method for facing the new challenges of *Big Data* [Fan and Bifet, 2013].

At this point, beginner readers, like for instance students, are invited to stop for a second and reflect on the importance and impact that this would have. For years, one of the main justifications that researchers have given to the limitations of GP was the fact that it was extremely difficult to study its fitness landscapes, because of the extreme complexity of the genotype/phenotype mapping, and because of the complexity of the neighborhoods induced by GP crossover and mutation. Introducing such an operator, we would not have to worry about this anymore! For any symbolic regression problem, we would have the certainty that the fitness landscape is unimodal, and thus easily evolvable. Furthermore: how many machine learning methods do you know in which the error surface is guaranteed to be unimodal for any possible application? Saying that this GP system would be the first machine learning system to induce unimodal error surfaces would probably be inappropriate; it is not impossible that other machine learning systems with this characteristic may have been defined so far. But still, it is absolutely legitimate to say that this characteristic is quite rare in machine learning, and should give a clear advantage to GP, compared to other well known systems, at least in terms of evolvability.

All we have to do for obtaining such an important result is to define an operator that, acting on trees, has an effect on the semantic space that is equivalent to box mutation. In other words, if we mutate a GP individual P_1, obtaining a new individual P_2, the semantics of P_2 should be like the semantics of P_1 except for a perturbation of its coordinates, whose magnitude is included in a predefined range. This is the objective of geometric semantic mutation, that is defined in the continuation.

[6] How could it be otherwise? GP is working with a population of trees, so the genetic operators can only act on them!

5.5.3 Geometric Semantic Mutation

The objective of Geometric Semantic Mutation (GSM) is the one of generating a transformation on the syntax of GP individuals that has the same effect on their semantics as box mutation. The situation is exemplified in Figure 5.9. More in par-

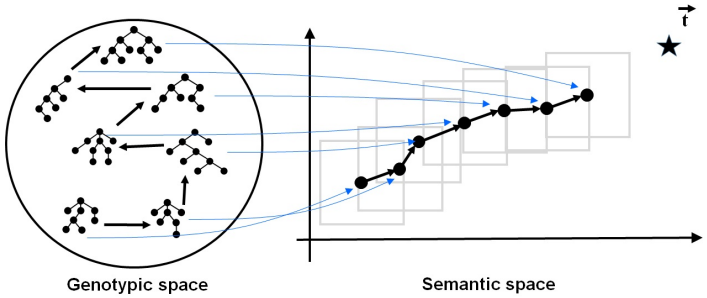


Fig. 5.9 A simple graphical representation, in the genotypic space, of a chain of solutions $\mathscr{C} = \{s_1, s_2, ..., s_h\}$ where, for each $i = 1, 2, ..., h-1$, s_{i+1} is a neighbor of s_i, and the corresponding points in the semantic space. The known vector of target values is represented in the semantic space by a star.

ticular, this figure 5.9 represents a chain of possible individuals that could be generated by applying GSM several times, with their corresponding semantics. Given that the semantics of the individuals generated by mutation can be any point inside a box of a given side centered in the semantics itself, GSM has always the possibility of creating an individual whose semantics is closer to the target (represented by a star symbol in Figure 5.9). As a direct consequence, the fitness landscape has no locally optimal solutions. Comparing the sematic space of Figure 5.9 with Figure 3.18 (page 98), it should not be difficult to see that if we use GSM, we *are* actually performing a CONO problem with box mutation on the semantic space. The definition of GSM, as given in [Moraglio et al., 2012], is:

Definition 5.1. Geometric Semantic Mutation (GSM). Given a parent function $P : \mathbb{R}^n \to \mathbb{R}$, the geometric semantic mutation with mutation step ms returns the function $P_M = P + ms \cdot (T_{R1} - T_{R2})$, where T_{R1} and T_{R2} are random functions.

It is not difficult to have an intuition of the fact that GSM has the same effect as box mutation on the semantic space. In fact, one should consider that each element of the semantic vector of P_M is a "weak" perturbation of the corresponding element in P's semantics. We informally define this perturbation as "weak" because it is given by a random expression centred in zero (the difference between two random trees). Nevertheless, by changing parameter ms, we are able to tune the "step" of the mutation, and thus the importance of this perturbation. Figure 5.10 gives a visual representation of the tree generated by GSM on a simple example. The tree P, that

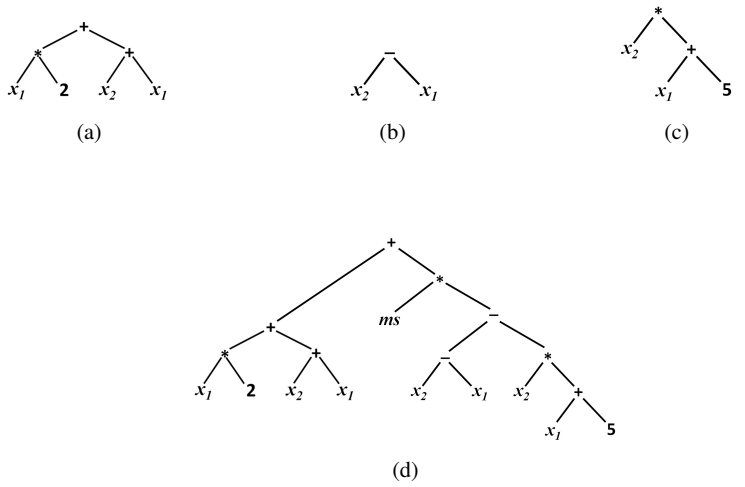


Fig. 5.10 A GP individual P that has to be mutated (plot (a)), two random trees T_{R1} and T_{R2} (plots (b) and (c) respectively) and the individual P_M generated by geometric semantic mutation (plot (d)).

is mutated, is represented in Figure 5.10(a). The two used random trees T_{R1} and T_{R2} are shown in Figure 5.10(b) and Figure 5.10(c) respectively. Finally, Figure 5.10(d) shows the resulting tree P_M generated by GSM.

In practice, and in order to make the perturbation even weaker, it is often useful to limit the codomain of the possible outputs of T_{R1} and T_{R2} into a given predefined range. This allows us to better "control" what mutation can do. The typical situation is that T_{R1} and T_{R2} are forced to assume values in $[0,1]$. This can be done easily, for instance, by "wrapping" T_{R1} and T_{R2} inside a logistic function. In other words, random trees are generated and the logistic is applied to their output before plugging them into P_M.

Before continuing, it is extremely important to stop for a while and convince oneself about the importance of having random *trees/expressions* (like T_{R1} and T_{R2}) in the definition of GSM, instead of just having random "numbers". The motivation is the following: when we mutate an individual, we want to perturb each one of the coordinates of its semantics by a *different* amount. It is easy to understand why this is important by considering a simple numeric example. Let the semantic of an individual P be, say, $\mathbf{s_P} = [5.6, 9.2]$, and let the target vector be $\mathbf{t} = [9, 10]$ (we are again considering the unrealistic bi-dimensional case for simplicity). If we used just random numbers for applying the perturbation, then we would be able to only mutate each one of the coordinates by the same amount. So, if we mutate by, say, 0.5, we obtain a new individual P_M whose semantics is $[6.1, 9.7]$. Even though P_M has a semantics that is closer to the target than P, it should not be difficult to convince oneself that if we iterate mutation, even in case we change the entity of the perturbation at each step, we will never have any chance of reaching the target. The only

possibility that we have to reach the target is to mutate the first coordinate *more* (i.e. of a larger amount) than the second one, simply because the first coordinate of $\mathbf{s_P}$ is further away from the corresponding coordinate of $\mathbf{t}$ than the second coordinate. So, we need a way of doing a perturbation that has to possess the following properties:

- it has to be random;
- it has to be likely to be different for each coordinate;
- it does not have to use any information from the dataset.

With the third point, we mean that the algorithm that makes the perturbation cannot have a form like:

"**if** ($\mathbf{x}_i$ = ...) **then** perturbation = ..."

because in this case the system would clearly overfit training data.

Under these hypotheses, the only way we could imagine of doing the perturbation was to sum to the value calculated by individual P the value calculated by a random expression. A random expression, in fact, is likely to have different output values for the different fitness cases. Last but not least, the fact that the difference between two random expressions is used ($T_{R1} - T_{R2}$) instead of just one random expression can be justified as follows. Especially in the final part of a GP run, it may happen that some of the coordinates have already been approximated in a satisfactory way, while it is not the case for others. In such a situation, it would be useful to have the possibility of modifying *some* coordinates and *not* modifying (or, which is the same, modifying by an amount equal to zero) others. The difference between two random expressions is a random expression *centered in zero*. This means that its output value is likely to be equal to zero, or close to zero, more than to be equal to any other value. In other words, by using the difference between two random expressions, we are imposing that some coordinates may have a perturbation that is likely to be equal, or at least as close as possible, to zero.

5.5.4 Geometric Semantic Crossover

GP practitioners may be surprised to notice that, so far, basically only mutation has been considered, practically ignoring crossover. Crossover, in fact, is known as the most powerful genetic operator, at least in standard GP. This section intends to fill this gap, by presenting a Geometric Semantic Crossover (GSC) that can behave, on the semantic space, as geometric crossover of GAs in continuous optimization, defined in Section 3.7. Geometric GAs crossover works by generating one offspring that has, for each coordinate, a linear combination of the corresponding coordinates of the parents with coefficients in $[0, 1]$, whose sum is equal to 1. Under these conditions, the offspring can geometrically be represented as a point that stands in the segment joining the parents. This is the behaviour that GSC must have on the semantic space, as exemplified in Figure 5.11. The objective of GSC is to generate the tree structure of an individual whose semantics stands in the segment joining

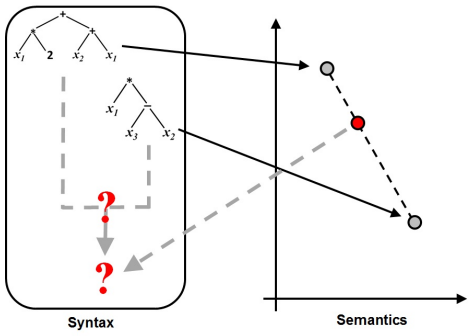


Fig. 5.11 Graphical representation of geometric semantic crossover, in the simple case of bi-dimensional semantic space. The offspring generated by this crossover has a semantics that stands in the segment joining the semantics of the parents.

the semantics of the parents. Following [Moraglio et al., 2012], GSC is defined as follows:

Definition 5.2. Geometric Semantic Crossover (GSC). Given two parent functions $T_1, T_2 : \mathbb{R}^n \to \mathbb{R}$, the geometric semantic crossover returns the function $T_{XO} = (T_1 \cdot T_R) + ((1 - T_R) \cdot T_2)$, where T_R is a random function whose output values range in the interval $[0, 1]$.

It is not difficult to see from this definition that T_{XO} has a semantics that is a linear combination of the semantics of T_1 and T_2, with random coefficients included in $[0, 1]$ and whose sum is 1. The fact that we are using a random expression T_R instead of a random number can be interpreted analogously to what we have done for explaining the use of random expressions in GSM. Furthermore, it is worth mentioning that in Definition 5.2 the fitness function is supposed to be the Manhattan distance; if Euclidean distance is used, then T_R should be a random constant instead. The interested user is referred to [Moraglio et al., 2012] for an explanation of this concept. Figure 5.12 gives a visual representation of the tree generated by GSC on a simple example.

The fact that the semantics of the offspring T_{XO} stands in the segment joining the semantics of the parents T_1 and T_2 has a very interesting consequence: the offspring generated by GSC cannot be worse than the worst of its parents, a propery that had already been studied in Section 3.7 for GAs geometric crossover. For a deeper discussion of this property, the reader is referred to [Moraglio, 2007, Moraglio et al., 2012].

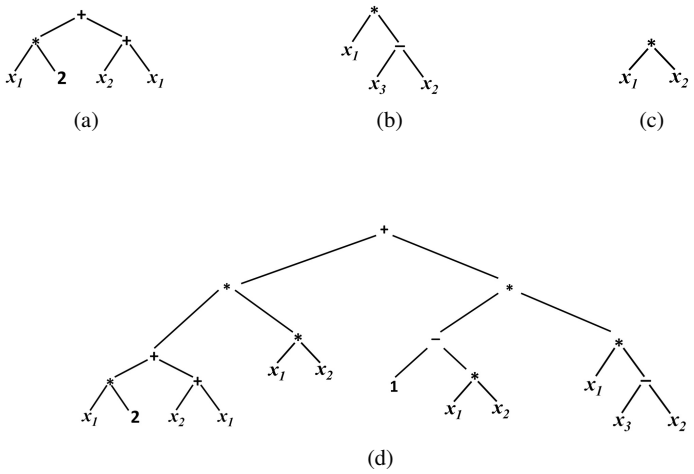


Fig. 5.12 Two parents T_1 and T_2 (plots (a) and (b) respectively), one random tree T_R (plot (c)) and the offspring T_{XO} of the crossover between T_1 and T_2 using T_R (plot (d)).

5.5.5 Code Growth and New Implementation

Looking at their definition (and at Figures 5.10 and 5.12), it is not hard to see that geometric semantic operators create offspring that contain the complete structure of the parents, plus one or more random trees and some additional arithmetic operators: the size of the offspring is thus clearly much larger than the size of their parents. The rapid growth of the individuals in the population, shown by Moraglio et al. [Moraglio et al., 2012], makes these operators unusable in practice: after a few generations the population becomes unmanageable because the fitness evaluation process becomes unbearably slow. The solution suggested in [Moraglio et al., 2012] consists in performing an automatic simplification step after each generation in which the individuals are replaced by (hopefully smaller) semantically equivalent ones. However, this additional step adds to the computational cost of GP and is only a partial solution to the progressive size growth. Last but not least, depending on the particular language used to code individuals and the used primitives, automatic simplification can be a very hard task.

In this section, we present a novel implementation of GP using these operators that overcomes this limitation, making them efficient without performing any simplification step. This implementation was first presented in [Vanneschi et al., 2013, Castelli et al., 2015]. Although the algorithm is described assuming the representation of the individuals is tree based, the implementation fits any other type of representation.

In a first step, we create an initial population of (typically random) individuals, exactly as in standard GP. We store these individuals in a table (that we call $\mathscr{P}$ from now on) as in the example reported in Figure 5.13(a), and we evaluate them. To

Id	Individual
T_1	$x_1 + x_2 x_3$
T_2	$x_3 - x_2 x_4$
T_3	$x_3 + x_4 - 2x_1$
T_4	$x_1 x_3$
T_5	$x_1 - x_3$

(a)

Id	Individual
R_1	$x_1 + x_2 - 2x_4$
R_2	$x_2 - x_1$
R_3	$x_1 + x_4 - 3x_3$
R_4	$x_2 - x_3 - x_4$
R_5	$2x_1$

(b)

Id	Operator	Entry
T_6	crossover	$\langle \mathrm{ID}(T_1), \mathrm{ID}(T_4), \mathrm{ID}(R_1) \rangle$
T_7	crossover	$\langle \mathrm{ID}(T_4), \mathrm{ID}(T_5), \mathrm{ID}(R_2) \rangle$
T_8	crossover	$\langle \mathrm{ID}(T_3), \mathrm{ID}(T_5), \mathrm{ID}(R_3) \rangle$
T_9	crossover	$\langle \mathrm{ID}(T_1), \mathrm{ID}(T_5), \mathrm{ID}(R_4) \rangle$
T_{10}	crossover	$\langle \mathrm{ID}(T_3), \mathrm{ID}(T_4), \mathrm{ID}(R_5) \rangle$

(c)

Fig. 5.13 Illustration of the example described in Section 5.5.5. (a) The initial population P; (b) The random trees used by crossover; (c) The representation in memory of the new population P'

store the evaluations we create a table (that we call $\mathscr{V}$ from now on) containing, for each individual in $\mathscr{P}$, the values resulting from its evaluation on each fitness case (in other words, it contains the semantics of that individual). Hence, with a population of n individuals and a training set of k fitness cases, table $\mathscr{V}$ will be made of n rows and k columns. Then, for every generation, a new empty table $\mathscr{V}'$ is created. Whenever a new individual T must be generated by crossover between selected parents T_1 and T_2, T is represented by a triplet $T = \langle \mathrm{ID}(T_1), \mathrm{ID}(T_2), \mathrm{ID}(R) \rangle$, where R is a random tree and, for any tree τ, $\mathrm{ID}(\tau)$ is a *reference* (or memory pointer)[7] to τ (using a C-like notation). This triplet is stored in an appropriate structure (that we call $\mathscr{M}$ from now on) that also contains the name of the operator used, as shown in Figure 5.13(c). The random tree R is created, stored in $\mathscr{P}$, and evaluated in each fitness case to reveal its semantics. The values of the semantics of T are also easily obtained, by calculating $(T_1 \cdot R) + ((1 - R) \cdot T_2)$ for each fitness case, according to the definition of geometric semantic crossover, and stored in $\mathscr{V}'$. Analogously, whenever a new individual T must be obtained by applying mutation to an individual T_1, T is represented by a triplet $T = \langle \mathrm{ID}(T_1), \mathrm{ID}(R_1), \mathrm{ID}(R_2) \rangle$ (stored in $\mathscr{M}$), where R_1 and R_2 are two random trees (newly created, stored in $\mathscr{P}$ and evaluated for their semantics). The semantics of T is calculated as $T_1 + ms \cdot (R_1 - R_2)$ for each fitness case, according to the definition of geometric semantic mutation, and stored in $\mathscr{V}'$. In the end of each generation, table $\mathscr{V}'$ is copied into $\mathscr{V}$ and erased. All the rows of $\mathscr{P}$ and $\mathscr{M}$ referring to individuals that are not ancestors[8] of the new population can also be erased. Note that, while $\mathscr{M}$ grows at every generation, by keeping the semantics of the individuals separated we are able to use a table $\mathscr{V}$ whose size is independent from the number of generations. Summarizing, this algorithm is based on the idea that, when semantic operators are used, an individual can be fully described by its semantics (which makes the syntactic component much less important than in standard GP), a concept discussed in depth in [Moraglio et al., 2012]. Therefore, at every generation we update table $\mathscr{V}$ with the semantics of the new individuals,

[7] Simple references to *lookup table* entries can be used in the implementation instead of real memory pointers (see [Vanneschi et al., 2013, Castelli et al., 2015]). This makes the implementation possible also in programming languages that do not allow direct manipulation of memory pointers, like for instance Java or MatLab.

[8] The term "ancestors" here is a bit abused to designate not only the parents but also the random trees used to build an individual by crossover or mutation.

and save the information needed to build their syntactic structures without explicitly building them.

In terms of computational time, it is worth emphasizing that the process of updating table $\mathscr{V}$ is very efficient as it does not require the evaluation of the entire trees. Indeed, evaluating each individual requires (except for the initial generation) a constant time, which is independent from the size of the individual itself. In terms of memory, tables $\mathscr{P}$ and $\mathscr{M}$ grow during the run. However, table $\mathscr{P}$ adds a maximum of $2 \times n$ rows per generation (if all new individuals are created by mutation) and table $\mathscr{M}$ (which contains only memory pointers) adds a maximum of n rows per generation. Even if we never erase the "ex-ancestors" from these tables (and never reuse random trees, which is also possible), we can manage them efficiently for several thousands of generations. Let us briefly consider the cost in terms of time and space of evolving a population of n individuals for g generations. At every generation, we need $O(n)$ space to store the new individuals. Thus, we need $O(ng)$ space in total. Since we need to do only $O(1)$ operations for any new individual (since the fitness can be computed using the fitness of the parents), the time complexity is also $O(ng)$. Thus, we have a linear space and time complexity with respect to population size and number of generations. This computational complaxity is very reasonable, and for sure competitive with several other machine learning systems.

The final step of the algorithm is performed after the end of the last generation. In order to reconstruct the individuals, we may need to "unwind" the compact representation and make the syntax of the individuals explicit. Therefore, despite performing the evolutionary search very efficiently, in the end we may not avoid dealing with the large trees that characterize the standard implementation of geometric semantic operators. However, most probably we will only be interested in the best individual found, so this unwinding (and recommended simplification) process may be required only once, and it is done offline after the run is finished. This greatly contrasts with the solution proposed by Moraglio et al. of building and simplifying *every* tree in the population at each generation online with the search process. If we are not interested in the form of the optimal solution, we can avoid the "unwinding phase" and we can evaluate an unseen input with a time complexity equal to $O(ng)$. In this case the individual is used as a "black-box" which, in some cases, may be sufficient. Excluding the time needed to build and simplify the best individual, the proposed implementation allowed us to evolve populations for thousands of generations with a considerable speed up with respect to standard GP.

Example 5.4. Let us consider the simple initial population P shown in Table (a) of Figure 5.13 and the simple pool of random trees that are added to P as needed, shown in Table (b). For simplicity, we will generate all the individuals in the new population (that we call P' from now on) using only crossover, which will require only this small amount of random trees. Besides the representation of the individuals in infix notation, these tables contain an identifier (Id) for each individual (T_1, ..., T_5 and R_1, ..., R_5). These identifiers will be used to represent the different individuals, and the individuals created for the new population will be represented by the identifiers T_6, ..., T_{10}. The individuals of the new population P' are simply represented by the set of entries exhibited in Table (c) of Figure 5.13. This table contains, for each

new individual, a *reference* to the ancestors that have been used to generate it and the name of the operator used to generate it (either "crossover" or "mutation"). For example, the individual T_6 is generated by the crossover of T_1 and T_4 and using the random tree R_1.

Let us assume that now we want to reconstruct the genotype of one of the individuals in P', for example T_{10}. The tables in Figure 5.13 contain all the information needed to do that. In particular, from Table (c) we learn that T_{10} is obtained by crossover between T_3 and T_4, using random tree R_5. Thus, from the definition of geometric semantic crossover, we know that it will have the following structure: $(T_3 \cdot R_5) + ((1 - R_5) \cdot T_4)$. The remaining Tables (a) and (b), that contain the syntactic structure of T_3, T_4, and R_5, provide us with the rest of the information we need to completely reconstruct the syntactic structure of T_{10}, which is:

$$((x_3 + x_4 - 2\,x_1) \cdot (2\,x_1)) + ((1 - (2\,x_1)) \cdot (x_1\,x_3))$$

and upon simplification becomes:

$$-x_1\,(4\,x_1 - 3\,x_3 - 2\,x_4 + 2\,x_1\,x_3).$$

5.5.6 Real-life Applications

The literature reports on various results obtained on several different applicative domains using the implementation of GSGP presented above. In this section, a subset of those results is briefly discussed. The objective of this section is only to give the reader an idea of the quality of the results that can be obtained using GSGP. We do not intend in any way to be exhaustive about the results that have been obtained. For a deeper review of the literature, the reader is referred to [Vanneschi et al., 2014].

All the applications presented in this section are real-life symbolic regression problems. Table 5.3 summarizes the main characteristics of each one of them. As the table shows, we are using 6 different problems. The first three of them (%F, %PPB and LD50) are problems in pharmacokinetics and they consist in the prediction of a pharmacokinetic parameter (bioavailability, protein binding level and toxicity respectively) as a function of some molecular descriptors of a potential new drug. The PEC dataset has the objective of predicting energy consumption in one day as a function of several different types of data relative to the previous days. The PARK dataset contains data about a set of patients of the Parkinson's disease and the target value is a measure of the severity of the disease, according to a standard measure. Finally, the CONC dataset has the objective of predicting concrete strength as a function of a set of parameters that characterize a concrete mixture. For different reasons, all these problems are considered important in their respective applicative domains, and they have been studied so far using several different computational intelligence methods. The results that have been obtained concerning a comparison between GSGP and standard GP (ST-GP) on these problems are presented in Fig-

Table 5.3 The main characteristics of the test problems used in the experiments presented in this section.

Dataset Name (ID)	# Features	# Instances	Objective
%F	241	359	Predicting the value of human oral bioavailability of a candidate new drug as a function of its molecular descriptors
%PPB	626	131	Predicting the value of the plasma protein binding level of a candidate new drug as a function of its molecular descriptors
LD50	626	234	Predicting the value of the toxicity of a candidate new drug as a function of its molecular descriptors
PEC	45	240	Predicting the value of energy consumption in one day as a function of a set of meteorologic data, and other kinds of data, concerning that day
PARK	18	42	Predicting the value of Parkinsons disease severity as a function of a set of patients data
CONC	8	1028	Predicting the value of concrete strength as a function of a set of feature of concrete mixtures

ure 5.14 (%F, %PPB and LD50 datasets) and Figure 5.15 (PEC, PARK and CONC datasets).

The plots in these figures report, for each problem, the results obtained on the training set (leftmost plot) and on the test set (rightmost plot). A detailed discussion of the parameters used in both GP systems in these experiments is beyond the objective of this chapter. What we can generally observe from Figure 5.14 and Figure 5.15 is that GSGP is able to consistently outperform ST-GP both on the training set and on the test set for all the considered problems. Statistical tests also indicated that all these results are statistically significant.

The good results that GSGP has obtained on training data were expected: the geometric semantic operators induce an unimodal fitness landscape, which facilitates evolvability. On the other hand, on a first analysis, it has been a surprise to observe the excellent results that have been obtained on test data. These results even appeared a bit counterintuitive at a first sight: we were expecting that the good evolvability on training data would entail an overfitting of those data. However, in order to give an interpretation to the generalization ability of GSGP, one feature of geometric semantic operators was realized that was not so obvious previously. Namely:

> *the geometric properties of geometric semantic operators hold independently of the data on which individuals are evaluated, and thus they hold also on test data!*

In other words, for instance geometric semantic crossover produces an offspring that stands between the parents also in the semantic space induced by test data. As a direct implication, following exactly the same argument as Moraglio et al. [Moraglio et al., 2012], each offspring is, in the worst case, not worse than the

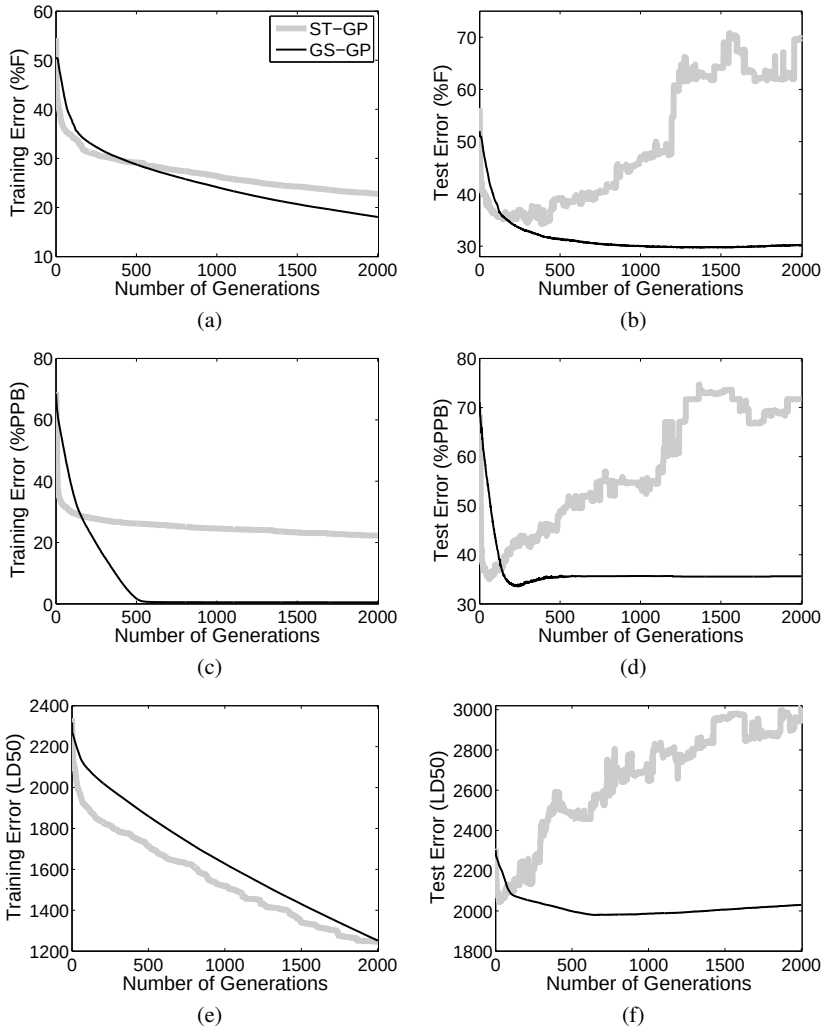


Fig. 5.14 Experimental comparison between standard GP (ST-GP) and Geometric Semantic GP (GS-GP). (a): %F problem, results on the training set; (b): %F problem, results on the test set; (c): %PPB problem, results on the training set; (d): %PPB problem, results on the test set; (e): LD50 problem, results on the training set; (f): LD50 problem, results on the test set.

worst of its parents on the test set. Analogously, as it happens for training data, geometric semantic mutation produces an offspring that is a "weak" perturbation of its parent also in the semantic space induced by test data (and the maximum possible perturbation is, again, expressed by the *ms* step). The immediate consequence for the behaviour of GSGP on test data is that, while geometric semantic operators do not guarantee an improvement in test fitness each time they are applied, they at

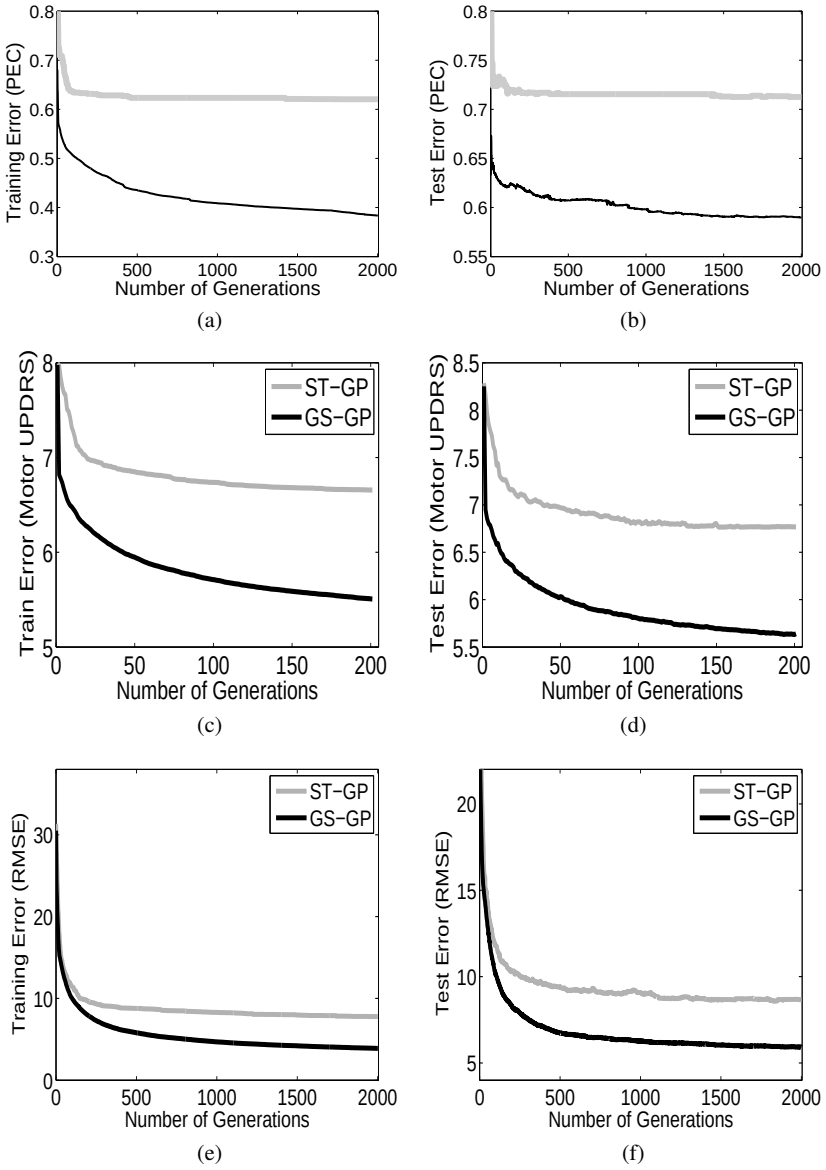


Fig. 5.15 Experimental comparison between standard GP (ST-GP) and Geometric Semantic GP (GS-GP). (a): PEC problem, results on the training set; (b): PEC problem, results on the test set. (c): PARK problem, results on the training set; (d): PARK problem, results on the test set. (e): CONC problem, results on the training set; (f): CONC problem, results on the test set.

least guarantee that the possible worsening of the test fitness is bounded (by the test

fitness of the worst parent for crossover, and by *ms* for mutation). In other words, *geometric semantic operators help control overfitting*. Of course, overfitting may still happen for GSGP (as it happens, slight but visible, for instance in plots (d) and (f) of Figure 5.14, reporting the results on %PPB and LD50 respectively), but there are no big "jumps" in test fitness like the ones observed for ST-GP. It is worth remarking that, without the novel implementation presented in Section 5.5.5, that allowed us to use GSGP on these complex real-life problems, this interesting property would probably have remained unnoticed.

Table 5.4 also reports an experimental comparison between GSGP and a wide set of non-evolutionary machine learning state-of-the-art methods on the CONC dataset. Analogous results can be found in literature also for all the other studied

Table 5.4 An experimental comparison between GSGP (last line), ST-GP (second to last line) and other machine learning strategies on the CONC dataset. The leftmost column contains the name of the method, the middle one the results obtained on the training set at termination, and the rightmost column contains the results obtained on the test set. Root Mean Square Error (RMSE) results are reported.

Method	Train	Test
Linear regression	10.567	10.007
Square Regression	17.245	15.913
Isotonic Regression	13.497	13.387
Radial Basis Function Network	16.778	16.094
SVM Polynomial Kernel (1 degree)	10.853	10.260
SVM Polynomial Kernel (2 degree)	7.830	7.614
SVM Polynomial Kernel (3 degree)	6.323	6.796
SVM Polynomial Kernel (4 degree)	5.567	6.664
SVM Polynomial Kernel (5 degree)	4.938	6.792
Artificial Neural Networks	7.396	7.512
Standard GP	7.792	8.67
Geometric Semantic GP	3.897	5.926

problems, and they show that GSGP is able to outperform all the other techniques both on training and test data.

References

[Aarts and Korst, 1989] Aarts, E. and Korst, J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, Inc., New York, NY, USA.

[Altenberg, 1994] Altenberg, L. (1994). Emergent phenomena in genetic programming. In Sebald, A. V. and Fogel, L., editors, *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 233–241. World Scientific.

[Antoniou and Lu, 2007] Antoniou, A. and Lu, W.-S. (2007). *Practical Optimization: Algorithms and Engineering Applications*. Springer Publishing Company, Incorporated, 1st edition.

[Applegate et al., 2007] Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. (2007). *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA.

[Auger and Teytaud, 2010] Auger, A. and Teytaud, O. (2010). Continuous lunches are free plus the design ofoptimal optimization algorithms. *Algorithmica*, 57(1):121–146.

[Banzhaf et al., 1998] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic programming, An Introduction*. Morgan Kaufmann, San Francisco CA.

[Benjamini, 1988] Benjamini, Y. (1988). Opening the box of a boxplot. *The American Statistician*, 42(4):257–262.

[Brameier and Banzhaf, 2001] Brameier, M. and Banzhaf, W. (2001). A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26.

[Burke et al., 2002] Burke, E., Gustafson, S., Kendall, G., and Krasnogor, N. (2002). Advanced population diversity measures in genetic programming. In Merelo, J. J., Adamidis, P., Beyer, H. G., Fernández-Villacanas, J.-L., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature - PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 341–350. Springer-Verlag, Heidelberg.

[Cagnoni et al., 2008] Cagnoni, S., Vanneschi, L., Azzini, A., and Tettamanzi, A. G. B. (2008). A critical assessment of some variants of particle swarm optimization. In Giacobini, M., Brabazon, A., Cagnoni, S., Di Caro, G. A., Drechsler, R., Ekárt, A., Esparcia-Alcázar, A. I., Farooq, M., Fink, A., McCormack, J., O'Neill, M., Romero, J., Rothlauf, F., Squillero, G., Uyar, A. Ş., and Yang, S., editors, *Applications of Evolutionary Computing*, pages 565–574, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Castelli et al., 2015] Castelli, M., Silva, S., and Vanneschi, L. (2015). A C++ framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 16(1):73–81.

[Černý, 1985] Černý, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51.

[Chen et al., 2010] Chen, W., Zhang, J., Chung, H. S. H., Zhong, W., Wu, W., and Shi, Y. (2010). A novel set-based particle swarm optimization method for discrete optimization problems. *IEEE Transactions on Evolutionary Computation*, 14(2):278–300.

[Cheung et al., 2014] Cheung, N. J., Ding, X., and Shen, H. (2014). Optifel: A convergent heterogeneous particle swarm optimization algorithm for takagisugeno fuzzy modeling. *IEEE Transactions on Fuzzy Systems*, 22(4):919–933.

[Chib and Greenberg, 1995] Chib, S. and Greenberg, E. (1995). Understanding the metropolis-hastings algorithm. *The American Statistician*, 49(4):327–335.

[Clerc, 2004] Clerc, M. (2004). *Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem*, pages 219–239. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Dan and Tim, 2008] Dan, B. and Tim, B. (2008). A simplified recombinant pso. *Journal of Artificial Evolution and Applications*, 2008.

[Darwin, 1859] Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection*. John Murray.

[De Jong, 1988] De Jong, K. A. (1988). Learning with genetic algorithms: an overview. *Machine Learning*, 3:121–138.

[Dodge, 2008] Dodge, Y. (2008). *Kolmogorov–Smirnov Test*, pages 283–287. Springer New York, New York, NY.

[Eiben and Smith, 2015] Eiben, A. E. and Smith, J. E. (2015). *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition.

[Ekárt and Németh, 2002] Ekárt, A. and Németh, S. Z. (2002). Maintaining the diversity of genetic programs. In Foster, J. A., Lutton, E., Miller, J., Ryan, C., and Tettamanzi, A. G. B., editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 162–171, Kinsale, Ireland. Springer-Verlag.

[Fan and Bifet, 2013] Fan, W. and Bifet, A. (2013). Mining big data: Current status, and forecast to the future. *SIGKDD Explor. Newsl.*, 14(2):1–5.

[Fernández et al., 2001] Fernández, F., Sánchez, J. M., and Tomassini, M. (2001). Placing and routing circuits on FPGAs by means of parallel and distributed genetic programming. In Liu, Y., Tanaka, K., Iwata, M., Higuchi, T., and Yasunaga, M., editors, *Evolvable Systems : From Biology to Hardware, 4th International Conference, Ices 2001*, volume 2210 of *Lecture Notes in Computer Science*, pages 204–215. Springer-Verlag, Heidelberg.

[Fernández et al., 2003] Fernández, F., Tomassini, M., and Vanneschi, L. (2003). An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–52.

[Fischer, 2019] Fischer, J. (2019). The boltzmann constant for the definition and realization of the kelvin. *Annalen der Physik*, 531(5):1800304.

[Garey and Johnson, 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[Garg, 2016] Garg, H. (2016). A hybrid pso-ga algorithm for constrained optimization problems. *Applied Mathematics and Computation*, 274:292 – 305.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

[Gustafson and Vanneschi, 2005] Gustafson, S. and Vanneschi, L. (2005). Operator-based distance for genetic programming: Subtree crossover distance. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., and Tomassini, M., editors, *Genetic Programming*, pages 178–189, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18.

[Holland, 1975] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan.

[Jarboui et al., 2008] Jarboui, B., Damak, N., Siarry, P., and Rebai, A. (2008). A combinatorial particle swarm optimization for solving multi-mode resource-constrained project scheduling problems. *Applied Mathematics and Computation*, 195(1):299 – 308.

[Keane, 1995] Keane, M. (1995). *The Essence of the Law of Large Numbers*, pages 125–129. Springer US, Boston, MA.

[Keijzer, 2003] Keijzer, M. (2003). Improving symbolic regression with interval arithmetic and linear scaling. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., and Costa, E., editors, *Genetic Programming*, pages 70–82, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kennedy, 2010] Kennedy, J. (2010). *Particle Swarm Optimization*, pages 760–766. Springer US, Boston, MA.

[Kennedy and Eberhart, 1995] Kennedy, J. and Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948.

[Kennedy and Eberhart, 1997] Kennedy, J. and Eberhart, R. C. (1997). A discrete binary version of the particle swarm algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 5, pages 4104–4108 vol.5.

[Kinnear Jr., 1993] Kinnear Jr., K. E. (1993). Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA. IEEE Press.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

[Kochenderfer and Wheeler, 2019] Kochenderfer, M. J. and Wheeler, T. A. (2019). *Algorithms for Optimization*. The MIT Press.

[Koza and Poli, 2003] Koza, J. and Poli, R. (2003). A genetic programming tutorial. In Burke, E., editor, *Introductory Tutorials in Optimization, Search and Decision Support*. Chapter 8. http://www.genetic-programming.com/jkpdf/burke2003tutorial.pdf.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[Koza, 1994] Koza, J. R. (1994). *Genetic Programming II*. The MIT Press, Cambridge, Massachusetts.

[Koza et al., 1999] Koza, J. R., Bennett III, F. H., Andre, D., and Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, San Francisco, CA.

[Krink and Løvbjerg, 2002] Krink, T. and Løvbjerg, M. (2002). The lifecycle model: Combining particle swarm optimisation, genetic algorithms and hillclimbers. In Guervós, J. J. M., Adamidis, P., Beyer, H.-G., Schwefel, H.-P., and Fernández-Villacañas, J.-L., editors, *Parallel Problem Solving from Nature — PPSN VII*, pages 621–630, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Langdon, 1996] Langdon, W. B. (1996). A bibliography for genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter B, pages 507–532. MIT Press, Cambridge, MA, USA.

[Langdon, 2000] Langdon, W. B. (2000). Size fair and homologous tree genetic programming crossover. *Genetic Programming and Evolvable Machines*, 1:95–119.

[Langdon and Poli, 2002] Langdon, W. B. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer, Berlin.

[Langdon et al., 1998] Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1998). The evolution of size and shape. In L. Spector, W. B. Langdon, U. M. O. and Angeline, P. J., editors, *Advances in Genetic Programming 3*, pages 163–190. The MIT Press.

[Lovric, 2011] Lovric, M., editor (2011). *International Encyclopedia of Statistical Science*. Springer.

[Luke, 2000] Luke, S. (2000). Code growth is not caused by introns. In Whitley, L. D., editor, *2000 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 228–235, Las Vegas, Nevada, USA.

[Martello and Toth, 1990] Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA.

[Massey, 1951] Massey, F. J. (1951). The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78.

[McCandlish, 2011] McCandlish, D. (2011). Visualizing fitness landscapes. *Evolution; international journal of organic evolution*, 65:1544–58.

[McDermott et al., 2012] McDermott, J., White, D. R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., and OReilly, U.-M. (2012). Genetic programming needs better benchmarks. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO 12, page 791798, New York, NY, USA. Association for Computing Machinery.

[McPhee et al., 2018] McPhee, N. F., Finzel, M. D., Casale, M. M., Helmuth, T., and Spector, L. (2018). *A Detailed Analysis of a PushGP Run*, pages 65–83. Springer International Publishing, Cham.

[McPhee and Poli, 2001] McPhee, N. F. and Poli, R. (2001). A schema theory analysis of the evolution of size in genetic programming with linear representations. In Miller, J., Tomassini, M., Lanzi, P. L., Ryan, C., Tetamanzi, A. G. B., and Langdon, W. B., editors, *Proceedings of the Fourth European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 108–125, Lake Como, Italy. Springer Verlag. Lecture notes in Computer Science vol. 2038.

[Miller, 2001] Miller, J. (2001). What bloat? Cartesian genetic programming on boolean problems. In Goodman, E. D., editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, San Francisco, California, USA.

[Miranda and Fonseca, 2002] Miranda, V. and Fonseca, N. (2002). Epso-evolutionary particle swarm optimization, a new algorithm with applications in power systems. In *IEEE/PES Transmission and Distribution Conference and Exhibition*, volume 2, pages 745–750 vol.2.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[Moraglio, 2007] Moraglio, A. (2007). *Towards a Geometric Unification of Evolutionary Algorithms*. PhD thesis, Department of Computer Science, University of Essex, UK.

[Moraglio, 2008] Moraglio, A. (2008). *Towards a geometric unification of evolutionary algorithms*. PhD thesis, University of Essex, Colchester, UK.

[Moraglio et al., 2012] Moraglio, A., Krawiec, K., and Johnson, C. (2012). Geometric semantic genetic programming. In Coello, C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., and Pavone, M., editors, *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31. Springer Berlin Heidelberg.

[Niknam and Amiri, 2010] Niknam, T. and Amiri, B. (2010). An efficient hybrid approach based on pso, aco and k-means for cluster analysis. *Applied Soft Computing*, 10(1):183 – 197.

[Nobile et al., 2012] Nobile, M. S., Besozzi, D., Cazzaniga, P., Mauri, G., and Pescini, D. (2012). A gpu-based multi-swarm pso method for parameter estimation in stochastic biological systems exploiting discrete-time target series. In Giacobini, M., Vanneschi, L., and Bush, W. S., editors, *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, pages 74–85, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Norouzi et al., 2012] Norouzi, M., Fleet, D. J., and Salakhutdinov, R. (2012). Hamming distance metric learning. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1061–1069, USA. Curran Associates Inc.

[O'Neill and Ryan, 2003] O'Neill, M. and Ryan, C. (2003). *Grammatical evolution - evolutionary automatic programming in an arbitrary language.*, volume 4 of *Genetic programming*. Kluwer.

[O'Reilly and Oppacher, 1995] O'Reilly, U.-M. and Oppacher, F. (1995). The troubling aspects of a building block hypothesis for genetic programming. pages 73–88. Morgan Kaufmann.

[Pedersen and Chipperfield, 2010] Pedersen, M. and Chipperfield, A. (2010). Simplifying particle swarm optimization. *Applied Soft Computing*, 10(2):618 – 628.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830.

[Pitzer and Affenzeller, 2012] Pitzer, E. and Affenzeller, M. (2012). *A Comprehensive Survey on Fitness Landscape Analysis*, pages 161–191. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Poli, 2001] Poli, R. (2001). General schema theory for genetic programming with subtree-swapping crossover. In Miller, J., Tomassini, M., Lanzi, P. L., Ryan, C., Tetamanzi, A. G. B., and Langdon, W. B., editors, *Proceedings of the Fourth European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 143–159, Lake Como, Italy. Springer Verlag. Lecture notes in Computer Science vol. 2038.

[Poli, 2003] Poli, R. (2003). A simple but theoretically-motivated method to control bloat in genetic programming. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., and Costa, E., editors, *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, volume 2610 of *LNCS*, pages 200–210, Essex. Springer-Verlag.

[Poli and Langdon, 1997] Poli, R. and Langdon, W. B. (1997). Genetic programming with one-point crossover and point mutation. Technical Report CSRP-97-13, University of Birmingham, B15 2TT, UK.

[Poli and Langdon, 1998a] Poli, R. and Langdon, W. B. (1998a). Genetic programming with one-point crossover. In Chawdhry, P. K., Roy, R., and Pant, R. K., editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, pages 23–27. Springer-Verlag, London.

[Poli and Langdon, 1998b] Poli, R. and Langdon, W. B. (1998b). Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252.

[Poli and McPhee, 2003a] Poli, R. and McPhee, N. F. (2003a). General schema theory for genetic programming with subtree swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66.

[Poli and McPhee, 2003b] Poli, R. and McPhee, N. F. (2003b). General schema theory for genetic programming with subtree swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206.

[Poli et al., 2008] Poli, R., McPhee, N. F., and Vanneschi, L. (2008). The impact of population size on code growth in gp: Analysis and empirical validation. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO 08, page 12751282, New York, NY, USA. Association for Computing Machinery.

[Rey and Neuhäuser, 2011] Rey, D. and Neuhäuser, M. (2011). *Wilcoxon-Signed-Rank Test*, pages 1658–1659. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Rosca, 1997] Rosca, J. P. (1997). Analysis of complexity drift in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA. Morgan Kaufmann.

[Roy et al., 2011] Roy, R., Dehuri, S., and Cho, S. B. (2011). A novel particle swarm optimization algorithm for multi-objective combinatorial optimization problem. *Int. J. Appl. Metaheuristic Comput.*, 2(4):4157.

[Rudin, 1986] Rudin, W. (1986). *Principles of Mathematical Analysis*. McGraw - Hill Book C.

[Rudolph, 1997] Rudolph, G. (1997). *Convergence properties of evolutionary algorithms*. Kovac.

[Russell and Norvig, 2009] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

[Samuel, 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM. Journal of Research and Development*, 3(3):210–229.

[Silva, 2008] Silva, S. (2008). *Controlling bloat : individual and population based approaches in genetic programming*. PhD thesis, University of Coimbra, Portugal.

[Silva and Costa, 2009] Silva, S. and Costa, E. (2009). Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179.

[Silva and Vanneschi, 2012] Silva, S. and Vanneschi, L. (2012). Bloat free genetic programming: Application to human oral bioavailability prediction. *Int. J. Data Min. Bioinformatics*, 6(6):585601.

[Soule and Foster, 1998] Soule, T. and Foster, J. A. (1998). Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309.

[Stadler, 2002] Stadler, P. F. (2002). *Fitness landscapes*, pages 183–204. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Tackett, 1994] Tackett, W. A. (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA.

[Tettamanzi and Tomassini, 2001] Tettamanzi, A. and Tomassini, M. (2001). *Soft Computing: Integrating Evolutionary, Neural and Fuzzy Systems*. Springer, New York.

[Tomassini et al., 2005] Tomassini, M., Vanneschi, L., Collard, P., and Clergue, M. (2005). A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 13(2):213–239.

[Vanneschi, 2004] Vanneschi, L. (2004). *Theory and Practice for Efficient Genetic Programming*. PhD thesis, Faculty of Sciences, University of Lausanne, Switzerland.

[Vanneschi et al., 2013] Vanneschi, L., Castelli, M., Manzoni, L., and Silva, S. (2013). A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In *Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013*, volume 7831 of *LNCS*, pages 205–216, Vienna, Austria. Springer Verlag.

[Vanneschi et al., 2014] Vanneschi, L., Castelli, M., and Silva, S. (2014). A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines*, 15(2):195–214.

[Vanneschi et al., 2004] Vanneschi, L., Clergue, M., Collard, P., Tomassini, M., and Vérel, S. (2004). Fitness clouds and problem hardness in genetic programming. In Deb, K., editor, *Genetic and Evolutionary Computation – GECCO 2004*, pages 690–701, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Vanneschi et al., 2011] Vanneschi, L., Dodecasa, D., and Mauri, G. (2011). A comparative study of four parallel and distributed pso methods. *New Generation Computing*, 29(1):129–161.

[Vanneschi et al., 2006] Vanneschi, L., Tomassini, M., Collard, P., and Vérel, S. (2006). Negative slope coefficient: A measure to characterize genetic programming fitness landscapes. In Collet, P., Tomassini, M., Ebner, M., Gustafson, S., and Ekárt, A., editors, *Genetic Programming*, pages 178–189, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Vassilev et al., 2003] Vassilev, V. K., Fogarty, T. C., and Miller, J. F. (2003). *Smoothness, Ruggedness and Neutrality of Fitness Landscapes: from Theory to Application*, pages 3–44. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Vlack, 2008] Vlack, L. H. V. (2008). *Elements of materials science and engineering*. Pearson Education, New Delhi, 6th. edition.

[Whigham, 1996] Whigham, P. A. (14 October 1996). *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *Trans. Evol. Comp*, 1(1):67–82.

[Zhang, 2000] Zhang, B.-T. (2000). Bayesian methods for efficient genetic programming. *Genetic Programming and Evolvable Machines*, 1(3):217–242.

[Zhang and Xie, 2003] Zhang, W.-J. and Xie, X.-F. (2003). In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483)*, volume 4, pages 3816–3821 vol.4.