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 problems: 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 elapses.

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 possession 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 settle on some ideas, let us consider the following examples of optimization problems:

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

L. Vanneschi and S. Silva, *Lectures on Intelligent Systems*, Natural Computing Series,
https://doi.org/10.1007/978-3-031-17922-8_2

- Example 2: given a set of photographs of human faces, find the one that most "resembles" 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 three-dimensional space at hand, and in order to assess 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 greater 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 in the size of the data). This is the main motivation for using Computational Intelligence methods to solve this kind of problem. Optimization problems can usually be specified by means of a set of *instances* of the problem.

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

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

  f quantifies the quality of the solutions in S and is called the 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 most "resembles" a given target face. Let us assume also that 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 existence 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 existence of an algorithm $\mathscr{A}$ 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; second, the possible instances can even be extremely different from 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 the book, 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 the distance between image features is more likely to correspond to our idea of "similarity" between the pictures than the 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 a *global optimum* or *globally optimal* solution, $f(o)$ or f_o is called the 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 many 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 with 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 deepening the study of optimization problems with an important theoretical result and several algorithms to solve them, 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:

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

  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 nonadmissible 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 nonadmissible 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 nonadmissible solution. The fact that all nonadmissible 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 identify a *gradient* also in the area of the nonadmissible solutions, which can be useful in some cases for the algorithms that will be studied in the continuation of this book. With the fitness defined in Equation (2.2), the fitness of nonadmissible 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 weight 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, with a total cost that is not larger than our predefined budget.

The Knapsack Problem is a well-known and widely studied optimization problem. The interested reader is referred, for instance, to [Martello and Toth, 1990].

Example 2.2. (*Traveling Salesperson Problem*). Given a set of cities and distances between each pair of cities, the objective of the Traveling 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 an $n \times n$ matrix D, where the element of indexes p and q ($D_{p,q}$) denotes the distance between the *pth* and the *qth* 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 a 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 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.

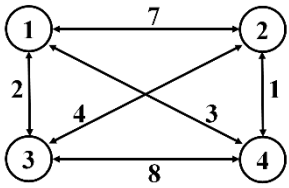

Fig. 2.1 Graph representation of the instance of the TSP discussed in Example 2.2

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, which 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 air traffic optimization.

The TSP is a well-known and widely studied optimization problem. The interested reader is referred, for instance, to [Applegate et al., 2007].

2.3 No Free Lunch Theorem

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

more particularly, *computational intelligence* optimization algorithms, with the objective of distinguishing them from classical, deterministic optimization algorithms. Although very different from 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 the 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 $\mathscr{A}$ finding a global optimum in its first m executions, 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 to find 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 book; 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 telling 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 the sum of the performance of an algorithm *over all existing optimization problems*. To convince one self about it, consider a countable[1] space of solutions S. 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 study some optimization algorithms, the fitness function is useful only to compare solutions with 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 is 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 identifies a possible sorting of fitness values, and all possible ways of sorting the fitness values 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, there cannot exist an algorithm (which 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 exists on which $\mathscr{A}_2$ outperforms $\mathscr{A}_1$.

This fact has an interesting consequence: every time that we are faced with a particular optimization 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 of 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 encourages the study of many different algorithms. As we will study in Chapter 5, the No Free Lunch Theorem can also be extended to Machine Learning [Wolpert, 1996]. So, all these considerations can be extended also to the field of Machine Learning. In conclusion, the No Free Lunch

[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.

Theorem motivates and paves the way for the existence of several software environments, in which several different Computational Intelligence and Machine Learning techniques are implemented, and a comparison between them is made particularly easy and automatic. 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 to a singular, and possibly counterintuitive, fact. At the beginning of this section, we defined the concept of heuristic 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 from 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 well as all the other heuristic optimization algorithms, including the ones that are considered more sophisticated or "intelligent", and, given any such algorithm, a set of problems exists on which random search outperforms it. 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 particular 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 natures. 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 described in a 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). Hill Climbing starts with an initial (typically random) solution i, which is made the current solution, and tries to improve it. 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, makes j 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 Hill Climbing is presented with more rigor 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, which we denote by $N(i)$, 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 existence of a precise algorithm $\mathscr{A}$ that allows us, 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 that transforms 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 than or equal to a given prefixed constant k. As we will see in the next examples, these two definitions often coincide, as it is 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 solution i often end up 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 $6th$ position of the string, which is different. On the other hand, the solution:

$$h = 1000101$$

is *not* a neighbor of i, in fact the bits in the $3rd$, $4th$, $5th$ and $6th$ 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 is the 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 feasible 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 fixed 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, which 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 contrary, a vast number of possible neighborhoods could be chosen. Just as an example, for the 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 $3rd$ to the $6th$ positions, which appear in the opposite order.

Given an instance of an optimization problem (S, f), and a neighborhood structure N, the pseudocode of Hill Climbing 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:

Algorithm 1: Pseudocode of 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 than or equal to $f(i)$) **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

- 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 good fitness should have certain characteristics, that information may 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 stepwise 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 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 than or equal to that of i, j becomes the new current solution. In minimization problems, $f(j)$ is better than or equal to $f(i)$ if $f(j) \leq f(i)$, while in maximization problems $f(j)$ is better than or equal to

[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 fixed 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 a tail and a 1 in case of a head, or vice versa.

$f(i)$ 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 if the generated neighbor has identical fitness to the current solution. In that case, which we call *strict Hill Climbing*, the term "better or equal" in the pseudocode 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 if the best neighbor of i has a fitness that is identical to that 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 we present an example that should clarify the functioning of 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 than or equal to all the other solutions in its neighborhood. In other words, in 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 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 Hill Climbing is always a local optimum. In fact, the termination condition of the algorithm exactly corresponds to the definition of local optimum. Furthermore, it should be noticed that *a global optimum is also a local optimum*. In fact, because a global optimum is the best solution among all the feasible ones, it is also the best of its neighborhood. We conclude that 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 Hill Climbing mainly depends on the initial solution i_{start}, which is typically generated at random, and on the particular neighborhood structure employed, which is a choice we must make when solving the problem.

Example 2.5. (*Execution of 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 study, the search space comprises only 16 feasible solutions: the natural numbers between 0 and 15 inclusive. As a fitness function, we use the number of 1s 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 between 0 and 15 inclusive that contains four 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 a neighbor of 8 and 10, and so on and so forth.

Let us, now, simulate the execution of 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, 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 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 pseudocode 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 analyzing 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, analyzing 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 Hill Climbing: advantages of 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 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 Hill Climbing returning solutions of better quality, one may imagine the following approaches:

- run Hill Climbing multiple times, each time using a different initial solution (possibly 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 strategies 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 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 as to become unmanageable. Indeed, one of the most widely accepted approaches for improving 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 Simulated Annealing, which will be studied in Section 2.6. But before we study the Simulated Annealing, the concept of Fitness Landscape is presented in Section 2.5.

2.5 Fitness Landscapes

Let us revisit 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 a *fitness landscape*, given that it visually resembles a landscape, with peaks, valleys, plateaux, etc. The behavior of 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, so "jumps" are allowed in the landscape. For instance, in the case of Example 2.5, Hill Climbing started its "motion" at abscissa 5 (the randomly generated initial solution

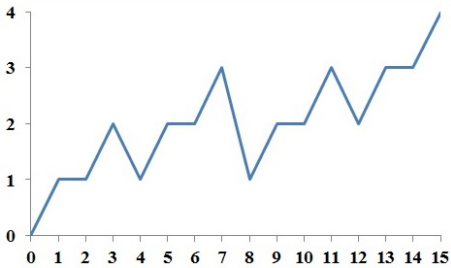

Fig. 2.2 Fitness landscape for the optimization problem instance and neighborhood studied in Example 2.5

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 behavior of Hill Climbing can be imagined as that of a "mountaineer", devoted to "climbing" the fitness landscape, and that stops every time it reaches a top (peak), whether or not it is 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. An FL is completely identified by the triple:

$$(S, f, N)$$

An FL gives visual intuition on the difficulty, or a 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 Hill Climbing, but also by any of the other existing algorithms, which 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 for the performance of optimization algorithms are discussed in [Vassilev et al., 2003].

Although the concept of FL is in general very useful for understanding the difficulty of a problem, it is generally impossible to draw an 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 multidimensional 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 on page 23); an 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 labeled with the value of the distance $d(i_1, i_2)$. An execution of Hill Climbing defines a walk on this graph. To make the representation more "visual", one may imagine "leaning" the graph on a horizontal plane, arranging the solutions in such a way that the distance between 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 three-dimensional plot can give a visual idea of the ruggedness of a landscape even in the 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 landscapes, drawing their shape whenever possible, and trying to imagine it otherwise.

Example 2.6. Let us recall Example 2.5, and let us extend the number of solutions in the search space, by increasing the upper bound of the natural numbers 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 equals 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 two-dimensional, we can still draw the FL using a two-dimensional plot as in Figure 2.3. As we can see, this landscape is very rugged. The reader is

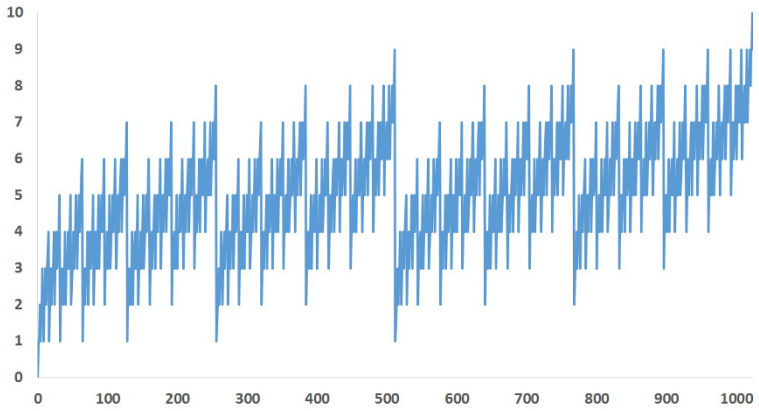

Fig. 2.3 Fitness landscape for the optimization problem instance and neighborhood studied in Example 2.6

invited to implement Hill Climbing and try to use it to solve this problem. It will quickly be observed that very often 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 Hill Climbing. It will immediately be observed that Hill Climbing will always be able to return the global optimum.

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

- $S = \{i \mid i \in \mathbb{N} \ \& \ 0 \leq \mathrm{i} \leq 1023\}$;

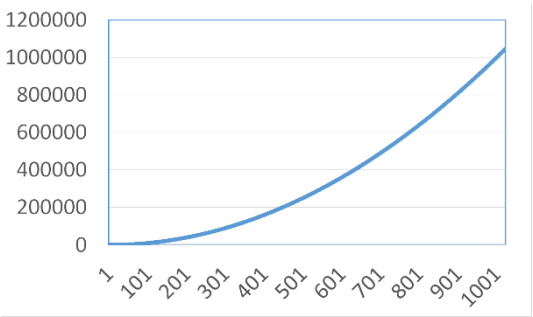

Fig. 2.4 Fitness landscape for the optimization problem instance and neighborhood studied in Example 2.7

- $\forall i \in S : f(i) =$ number of bits equal to 1 in the binary code of i (maximization).

And the following neighborhood structure:

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

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 neighbors 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 now has 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 be: if a solution is not the global optimum (which, 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 Hill Climbing to solve this problem. The reader will observe that, in a worst case of 10 steps, 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 an 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 two-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

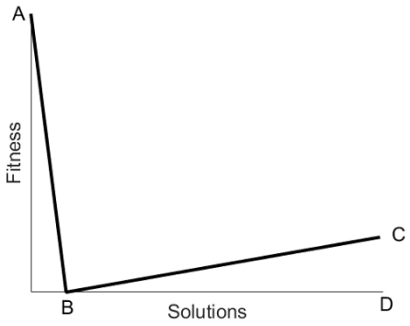

Fig. 2.5 Graphical representation of a deceptive fitness landscape

words, let us assume that the probability that a random number drawn with uniform distribution in $[0, D]$ is smaller than or equal to B is practically equal to zero. In such a situation, it should not be difficult to convince oneself that 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 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 that a solution of poor quality is returned is only one of the flaws that 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 on page 23), the returned solution is even the one that is furthest away from the global optimum according to d. In simple terms, 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 time, a steady attempt to improve 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 existence 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 Climbing. 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 the presence of deceptive or partially deceptive problems, Random Search can be a reasonable strategy.

- Always blindly following fitness, in the steady attempt to improve it, can be a 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 previously mentioned, one of the most useful strategies to improve Hill Climbing is to release the algorithm from the idea of always improving fitness. In Simulated Annealing, which 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 an 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 reaching the global optimum (abscissa 15) is accepting some downhill steps during the exploration. Thus, the idea behind Simulated Annealing seems reasonable and promising.

2.6 Simulated Annealing

Simulated Annealing [Kirkpatrick et al., 1983, Černý, 1985, Aarts and Korst, 1989] extends Hill Climbing, taking inspiration from a metallurgy and materials science heat treatment, called annealing [Vlack, 2008]. Annealing is a process that allows us to obtain materials in a 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 allowing 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 chosen, with some probability distribution, 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 Hill Climbing are visible, in the sense that we talk of a current state, similarly to how in Hill Climbing we talk of a current solution, and we try to update the current state by applying some transformations that may loosely remind us of the application of an operator to obtain a neighbor. Energy in annealing corresponds to fitness for optimization algorithms, and it is supposed to be minimized. The macroscopic difference from the Hill Climbing is clearly that the energy of the current state can increase with some probability. Simulated Annealing extends Hill Climbing to also envisage the case of a temporary 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:

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

where, as usual, in minimization problems $f(j)$ is better than or equal to $f(i)$ if $f(j) \leq f(i)$, while for maximization problems $f(j)$ is better than or equal to $f(i)$ if $f(j) \geq f(i)$. Compared to Equation (2.4), in Equation (2.5) the concept of the 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 the *control parameter* of Simulated Annealing and which, as we will see, plays an important role in the dynamics of the algorithm.

Once these similarities are established between Simulated Annealing and the Metropolis algorithm used for annealing, Simulated Annealing can be seen as an iteration of the Metropolis algorithm, using decreasing values of the control parameter c. The idea of beginning the execution with a high value 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 the function $\phi(c) = e^{-\frac{k}{c}}$, where k is a positive constant ($k = 2$ was used in the figure). This plot can be used to study the variation of the probability that Simulated Annealing accepts a worse solution than the current one, as c is modified. 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 deteriorations. 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 is easy to understand

[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 ever becoming equal to zero during the execution of Simulated Annealing.

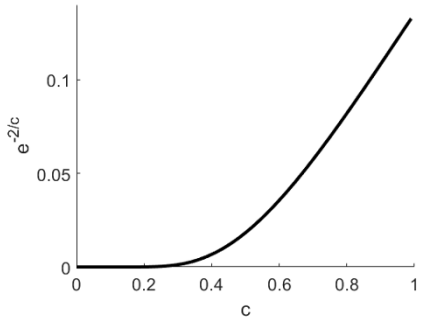

Fig. 2.6 Graphical representation of the function $\phi(c) = e^{-\frac{2}{c}}$, used to understand the contribution of the control parameter c to the probability of accepting a solution with worse fitness, compared to the current one

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

From all this, we infer that beginning with a high value 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 Hill Climbing, i.e., where the probability of accepting fitness deteriorations is very low. The rationale behind this idea is, in simple terms, that at 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 step over some hills. On the other hand, as long as the execution proceeds, we are possibly approaching one of the highest hills, and hopefully in 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 Hill Climbing would do.

The pseudocode of 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 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 or not to accept 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 discussed in the next paragraphs.

Algorithm 2: Pseudocode of 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 than or equal to $f(i)$) **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 encountered so far;

1. *External Loop and Termination Condition* (point 4). The external loop has a termination condition, which actually corresponds to the stopping criterion of the algorithm, that is usually satisfied when one of these two conditions is satisfied:

   - a "satisfactory" solution has been found, or
   - a previously fixed maximum number of iterations has been 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, we mean a solution x whose fitness value f_x is at a distance smaller than or equal to ε_f from f_o. Concerning the second point: the previously chosen maximum number of iterations can be con-

[4] The reader should observe that knowing the optimal fitness is rather usual in optimization problems, where the objective is finding a solution with such a fitness. To convince oneself about this, one may consider Example 1 on page 13. 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.

sidered 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.

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 is customary for 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 reach the global optimum (that is 15) is to accept 8 as the next solution. Such a situation can be generalized, that is, in some situations, always choosing the best neighbor of the current solution may jeopardize our chances of reaching a global optimum. Besides this, it is also straightforward to understand that, particularly in the 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 $\text{Rand}[0,1)$, which returns a random number between 0 and 1, drawn with uniform distribution[5]. The reader is invited to reflect on the portion of pseudocode:

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

This can be interpreted as the implementation of the sentence:

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

[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`.

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 even being equal to zero. In this way, the algorithm should be able to climb over several hills, basins of attraction of local optima, in the first phase of the execution, while it should "resemble" Hill Climbing in the 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 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 decrement 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 Simulated Annealing on a simple numeric example.

Example 2.10. Let us recall 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 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 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 Simulated Annealing encounters a local optimum, it typically remains 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 Hill Climbing, Simulated Annealing has the ability to escape from local optima, while still maintaining some positive characteristics of Hill Climbing, such as simplicity and generality. The convergence speed of the algorithm depends on several factors, including:

- 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 Simulated Annealing. The final result will be presented and commented on 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 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 pairwise neighbors. Use of 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 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, 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 the *stationary probability*, or *equilibrium distribution*, of Simulated Annealing, for control parameter c. Lemma 2.1 is not proven in this book. 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 book. With Theorem 2.3, we are now interested in understanding on what solution(s) Simulated Annealing will stabilize, after a large number of iterations, when c is modified.

Theorem 2.3. (Theorem of Asymptotic Convergence of Simulated Annealing). *Let (S, f) be an instance of a minimization problem, on which Simulated Annealing is executed using a completely interconnected neighborhood structure. Assuming that 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 search 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 recalling 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 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 the 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 side 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 than or equal to 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 side 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, substituting 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 book. In order to understand the intuitive meaning of Theorem 2.3, first of all, we have to remark that, given the functioning of 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 gives us information about the properties of Simulated Annealing as the running time tends to infinity (asymptotic properties).

Let us now try to answer the following question: what does the theorem tell us if the search space contains just one global optimum? The answer is straightforward: it says that, as time tends to infinity, 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 tells us if the search space contains two global optima. Also in this case, it is not difficult to convince oneself that as time tends to infinity, 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 tells us that, as time tends to infinity, 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 of the problem at hand, and, as such, from the shape of the fitness landscape.

Of course, this property does *not* tell us that Simulated Annealing will find a global optimum in a humanly acceptable time. It actually tells us that it will happen, but it does not say 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 execute a large number of transitions for each value of the control parameter, and decrease the control parameter slowly, in such a way that the total number of iterations performed is as large as possible. As is intuitively easy to see, 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 whether 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 Simulated Annealing. Indeed, what Theorem 2.3 tells us is much more than the simple application of the Law of Large Numbers for Random Search. Theorem 2.3 tells us that 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, Simulated Annealing starts getting closer and closer to a global optimum. So, 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 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 other 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 that 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.