1

Metaheuristics – Local Methods

COPYRIGHTED MATERIAL

1.1. Overview

The term *heuristic* in the expression heuristic optimization originates from ancient Greek, where *heuriskein* (*χευρισκειν*) means "to discover" or "to learn". There is an even more subtle meaning of this word, as revealed by the following example: assume that the readers of this book are challenged to find and mark in the text the position of the words *metaheuristic* or *metaheuristics*. Each of us has a specific strategy to meet the challenge. Nevertheless, in general, there are two categories of readers: readers who systematically analyze the text, trying not to miss occurrences, and readers who approach the text "diagonally", relying on their visual capacity of pattern recognition, without actually looking at every word. We say that the first category of readers performs an *exhaustive search* (like a computer), while the second category makes a *heuristic search* (like a living entity, not necessarily consciously). Obviously, the research duration of the first type of readers is usually longer than that of the second type of readers. However, it is very likely that the first type of readers will be able to find the most occurrences, while the second type of readers could miss enough occurrences. Finally, when comparing the performance of the two categories of readers, it can be seen that the number of occurrences found by the first category is surprisingly close to the number of occurrences marked by the second category, despite the big difference between the search durations.

Chapters 1 and 2 of this book are devoted to the description of a collection of optimization methods that simulate the second type of readers' attempt. Such methods actually are inspired either by the behavior of biological entities/systems or by the evolution of some natural phenomena. Next, the discussion focuses on a special class of optimization problems in engineering, more specifically on the class of *granular* optimization. This concept is explained in the following.

The optimization problem of concern is formulated in the context of a cost function (or criterion), as defined below:

$$f : \mathbf{S} \to \mathbf{R}, \qquad [1.1]$$

where *the search space* $\mathbf{S}$ is usually a bounded subset of $\mathbf{R}^{nx}$. (Sometimes, f is referred to as *fitness*.) What makes the criterion f so special? On the one hand, it has a fractal variation, with many ruptures (and thus with many local extremes). On the other hand, it is quite difficult (if not impossible) to evaluate its derivatives in complete form (provided that they exist). In terms of regularity, the f criterion is locally continuous or derivable, but this property does not extend to the global variation. (The criterion could be globally smooth, but this very seldom happens.) In turn, we can evaluate the f criterion for each point $\mathbf{x}$ of research space $\mathbf{S}$, according to some *a priori* known mathematical expression. Thus, in order to solve the optimization problem:

$$\underset{\mathbf{x}\in\mathbf{S}}{\text{opt}}\ f, \qquad [1.2]$$

which means to find the global optimum of f and the optimum point $\mathbf{x}^{\text{opt}} \in \mathbf{S}$, it is only possible to compare various criterion values in points of research space. As the search has to end after a reasonable duration, only a finite discrete subset of $\mathbf{S}$, say $\mathbf{D}$, can be used for this purpose. We aim not to miss the global optimum, and therefore the $\mathbf{D}$ subset has to include a very large number of points to test.

Problem [1.2] is then relaxed, being replaced by the following problem:

$$\underset{x \in \mathbf{D} \subset \mathbf{S}}{\text{opt}} \; f, \qquad [1.3]$$

where, as already mentioned, $\mathbf{D}$ is a finite discrete subset. Due to their large number, the test points of $\mathbf{D}$ are referred to as *granules* or *grains*. Consequently, [1.3] becomes a *granular optimization problem*. Solving problem [1.3] means in this context finding the grain of $\mathbf{D}$, which is located as close as possible to the global optimum point of f.

The following example can illustrate the principle of granulation. Assume that the following adage: "*Εν αρχεοσ ο αριτμοσ.*" (/*En arheos o aritmos.*/) has to be translated (it belongs to the famous mathematician and physicist Archimedes). We want the closest translation. Obviously, the criterion here is the map between the ancient Greek and a modern language, say English. The search space $\mathbf{S}$ is a dictionary with many words (a few tens of thousands), granular by nature (a grain being associated here with a word). In order to perform the translation, we may want first to insulate the "subdictionary" $\mathbf{D}$ including all words of $\mathbf{S}$ that begin with α (/a/), ε (/e/) and o (/o/). Next, an appropriate search strategy has to be set, as checking all words of $\mathbf{D}$ (still very large) is unthinkable. By adopting a heuristic-like strategy, the subdictionary size can be reduced, as soon as the next letters composing the words to translate are taken into account. Finally, $\mathbf{D}$ only includes the words to translate and we thus obtain a first but coarse translation: *in, ancient/antique, is/was, number*. However, a refinement is necessary so that the good meaning of the adage is found. A second optimization problem can thus be stated, but, this time, by accounting for all synonyms of already found words. We even can add all usual expressions containing those words and synonyms. Since the size of restricted subdictionary stays small, we can now adopt exhaustive search as suitable strategy. We then reach for the closest translation to the adage spirit: *the number was in the beginning*.

The methods for solving granular optimization problems should lead to numerical computer algorithms; otherwise, they are not really useful in engineering. The strategies adopted in the previous example can easily be followed by a human being, but the computer needs programming in order to perform similar reasoning. Thus, first, the words of the **S** dictionary are recorded to memory locations addressed by the American Standard Code for Information Interchange (ASCII) codes of their letters (**S** thus becomes an *electronic dictionary*). In this manner, the exhaustive search is avoided (no need to parse all dictionary addresses until the word is found). Second, an expert system of usual expression in the modern language has to be designed and implemented. Here, again, the memory addresses have to be generated in a way such that the exhaustive search can be avoided (if possible). Third, in order to increase the search speed, some statistic database pointing to the most employed words of dictionary can be built into the dictionary. The modern automatic translators are based on the principles of heuristic strategies, as stated above.

Concerning problem [1.3], the main goal is to design solving methods that can avoid the exhaustive search or, at least, that are only adopting this strategy as a final stage, on a restricted search subset. Moreover, such methods should lead to the numerical algorithms to implement on computer. Obviously, by avoiding the exhaustive search, the global optimum could be missed, but, in turn, there is a hope that the search speed increases sensibly, while the accuracy stays good enough. There is a preference for methods allowing the user to control the trade-off between the search speed and the optimal solution accuracy, in spite of their higher complexity (some of these methods are described in this chapter). For the other (usually less complex) methods, it is up to the user to select, or not, one of them in applications.

The heuristic methods that can be implemented on a computer are referred to as *metaheuristics*. They rely on the following basic principle: the search for optimum actually simulates either the behavior of a biologic system or the evolution of a natural phenomenon, including an intrinsic optimality mechanism. For this reason, a new optimizations branch has developed in the past 20 years,

namely *Evolutionary Programming*. All numerical algorithms designed from metaheuristics are included into this class of optimization techniques.

In general, all metaheuristics are using a pseudo-random engine to select some parameters or operations that yield estimation of optimal solution. Therefore, the procedures to generate *pseudo-random (numerical)* sequences (PRSs) are crucial in metaheuristics design. When speaking about pseudo-random numbers, their probability density should *a priori* be specified. In the case of metaheuristics, two types of probability densities are generally considered: uniform and normal (Gaussian). Thus, the following types of generators are useful:

1) *uniformly distributed pseudo-random sequences generator* (U-PRSG);

2) *prescribed probability distribution pseudo-random sequences generators* (P-PRSGs).

In Appendices 1 and 2 of this book, algorithms of both generator types are described. They are simple and effective. Sometimes (although rather seldom), more complex and sophisticated algorithms are preferred in applications.

Unfortunately, the use of pseudo-random numbers in conjunction with metaheuristics makes impossible the formulation of any general and sound result related to convergence. The only way to deal with convergence is to state some conjectures for those metaheuristics that seemingly are successful in the most applications. If the natural optimality mechanism is well simulated, there is a good chance that the corresponding metaheuristic converges toward optimum in most cases. This is a good foundation for a realistic convergence conjecture.

Two categories of metaheuristics are described in this book: local and global. In the case of local methods, we assume that either the criterion has a unique global optimum or the search is restricted to some narrow subsets including the global optimum. The aim of the global methods is to find the overall optimum (or, at least, a good approximation of it) by performing a search in several zones of the **S**

space. Usually, the search is following a scenario that allows a sudden change of focusing zone (with the help of PRS), in order to avoid attraction of local optima.

1.2. Monte Carlo principle

One of the first approaches related to granularity of numerical problems (not only of optimization kind) is known as *Monte Carlo method*. It was introduced by the physicist Stanislaw Ulam, in the end of the 1940s [MET 49], after trying without any success to build some realistic analytical models of "Solitaire" game (in correlation with an analytical model of matter atomic kernel). He noted that perhaps it is better to observe the game results over 1,000 plays and afterwards build an approximate associated statistic model, instead of writing the equations corresponding to this (extremely complicated) statistics. His idea was rapidly understood by John von Neumann, a researcher who had already performed similar research in the framework of some secret military projects. The two scientists have chosen a natural codename for this method, as inspired by gambling and casinos: *Monte Carlo* [ECK 87]. (Actually, the name was proposed by John von Neumann after learning about the passion for casinos of one of Stanislaw Ulam's uncles.)

The idea of this method is simple, but empirical. If a numerical description of a system or phenomenon with many entries is necessary, then it is worth stimulating this entity by a big number of random input values and to find a good result by following some simple computational rules. Very often, such a result is surprisingly close to the real numerical characteristic of the entity of interest. This is due to the Law of Large Numbers, in combination with ergodic hypotheses and the Central Limit Theorem from Statistics. The only requirement, for a correct efficiency of this approach, is to formulate the problem to solve in such a way that the method principle can be exploited.

The next example shows how the method works. Assume that a good approximation of Pythagoras' number π has to be found. Then, instead of using a Taylor's expansion of a trigonometric map (which

would constitute a sound approach), a simple geometrical property can be exploited. Since the unit square includes the unit circle and the ration between their areas is $4/\pi$, it suffices to approximate this number, without actually measuring the surfaces. According to the Monte Carlo method, to solve this problem, first we have to fill in the square with N randomly generated points, uniformly distributed. The number of points falling into the circle, say n, are then counted. Thus, an approximate value of π can be computed as follows:

$$\pi \cong 4\frac{n}{N}. \qquad [1.4]$$

The bigger the N, the more accurate the approximation. The result is illustrated by the succession of images in Figure 1.1. We start with 3,000 points. More points are subsequently added until the desired accuracy is obtained. For 30,000 points, $\pi \cong 3.1436$.

The success of the method tremendously depends on the probability distribution of generated points. If non-uniform, the error can rapidly grow. The entries here are the randomly generated points, which actually perform sampling of the two surfaces. It is easy to notice that the system is stimulated by an infinite number of inputs and, moreover, the inputs cannot be counted (as being uniformly spread over the square).

The computing rule is based on each point position (inside or outside the circle). To automatically decide the point position, it suffices to compute its distance from the circle center and compare it to the unit. In general, the problems to solve by Monte Carlo method should rely on simple enough computational rules. Otherwise, to reach for the prescribed accuracy, the procedure could be time-consuming (because the computational rules apply to each stochastic entry). Even in the case of the example above, the computational burden becomes important when the number of generated points increases beyond some limit.

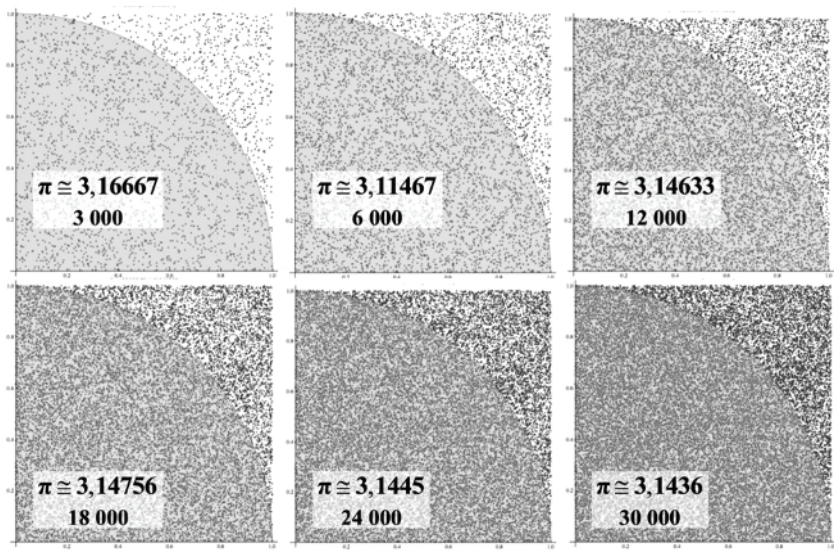


Figure 1.1. *Empirical estimation of Pythagoras' number by Monte Carlo method. For a color version of this figure, see www.iste.co.uk/stefanoiu/optimazation.zip*

As a general characteristic of Monte Carlo method, the procedure may be very greedy in terms of computational resources.

When looking back at the images of Figure 1.1, it is easy to see the granularity effect (the points are like grains pored on the square surface). It follows that, in the context of Monte Carlo method, finding a good approximation of π number is a granular computational problem (not necessarily of optimization type).

The method is frequently employed in surface or volume evaluation, in the case of complicated geometrical shapes. More specifically, by this method, multiple integrals such as the one given below can be evaluated:

$$\int_{\mathrm{D}} f(\mathbf{x})\mathrm{d}\mathbf{x}\,, \qquad [1.5]$$

where the map f usually has simple expression, whereas the border of integration domain $\mathbf{D} \subset \mathbf{R}^{nx}$ is described by complex equations, sometimes in implicit form. It is much easier to test whether or not some points of $\mathbf{R}^{nx}$ space belong to the integration domain, instead of computing the integral [1.5]. This is the heart of Monte Carlo principle.

The integral [1.5] can be approximated by means of the grains technique:

– the computational problem is formulated in the context of $\mathbb{R}^{nx+1}$ space, where f generates a hypersurface over the $\mathbf{D}$ domain. The integral is actually the volume of the solid body framed by $\mathbf{D}$ and the hypersurface, say V_f;

– the solid body can be included into a hypercylinder with a basis hyperdisk covering $\mathbf{D}$. For now, the hypercylinder height is unknown;

– many grains are uniformly pored on the hyperdisk;

– one checks each grain location (inside or outside the $\mathbf{D}$ domain);

– for each grain inside $\mathbf{D}$, the value of f is computed in order to determine the minimum and maximum heights of hypersurface;

– the hypercylinder height is set so that it completely includes the hypersurface;

– the hypercylinder volume is computed, say V_{hc};

– the available grains layer on hypercylinder basis is uniformly replicated at different heights, until the whole hypercylinder is filled in. All grains that can be projected onto the $\mathbf{D}$ domain are thus located inside the solid body, provided that their height is between 0 and the (already computed) values of f;

– all grains from the solid body are counted (by comparing their heights with f values). Denote this number by n;

– all grains of hypercylinder, say N, are counted as well;

– the volume of the solid body is approximated by:

$$V_f \cong \frac{n}{N} V_{hc} \,. \qquad [1.6]$$

In spite of its empirical spirit, the Monte Carlo method is employed in many modern applications such as in the fields of physics, engineering, biology, statistics, economy, telecommunications and computer vision.

Nowadays, a whole family of Monte Carlo methods exists, depending on various implementations of basic principle [FIS 95, KRO 11]. They can be used to perform not only approximations of expressions difficult to evaluate, but also some statistical information concerning the confidence degree in such results. Moreover, modern Monte Carlo methods often work with non-uniform probability distributions in order to alleviate the computational burden.

For the optimizations field, the use of Monte Carlo methods is considered a rudimentary approach. In the previous example, we have noticed that, during the integral evaluation, the extremes of f were somewhat empirically determined. Indeed, the more grains are pored on the $\mathbf{D}$ domain, the most accurate the approximations of f extremes. However, this attempt is not a real search for optimum, in the spirit of optimization procedures, where a direction toward optimum is usually estimated and upgraded, if necessary. The search here is "blind", without considering the properties of criterion to optimize. Moreover, in the case of a real optimization problem, the criterion can be expressed by very complex equations, which involves that the grains evaluation could be very slow.

The metaheuristics have adopted the stochastic granularity principle, though, as it could help to explore the whole search space for the global optimum.

To conclude this section, algorithm 1.1 introduces the basic Monte Carlo procedure for optimization purposes.

1) Input data:

– Search vicinity $\mathbf{V}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Optimization criterion, f .

– Minimum number of stochastic points to generate during the search, K .

– Accuracy threshold, $\varepsilon > 0$.

2) Initialization.

a) Select at random (but uniformly distributed) the starting point $\mathbf{x}_0 \in \mathbf{V}$. A U-PRSG of nx size has to be used in this aim (see section A2.4 of Appendix 2). If the starting point does not belong to $\mathbf{V}$, then it will be generated until this property is verified. (It is necessary to take into account the topology of $\mathbf{V}$.)

b) Evaluate the starting point performance: $f(\mathbf{x}_0)$.

c) Set the initial optimal point: $\mathbf{x}^{\text{opt}} = \mathbf{x}_0$.

d) Set the starting iterations number: $k = 0$.

3) For $k \in \overline{0, K-1}$:

3.1. Generate $\mathbf{x}_k^{k+1}$ inside the vicinity $\mathbf{V}$, by means of the U-PRSG, after being well calibrated in this aim.

3.2. If $\mathbf{x}_k^{k+1} \in \{\mathbf{x}_i\}_{i \in \overline{0,k}}$, the point already exists and it must be replaced by a different one. Repeat the previous step.

3.3. Otherwise, store $\mathbf{x}_{k+1} = \mathbf{x}_k^{k+1}$ in memory.

3.4. Evaluate the performance of newly generated point: $f(\mathbf{x}_{k+1})$.

3.5. If $f(\mathbf{x}_{k+1})$ is better than $f(\mathbf{x}^{\text{opt}})$, update the optimal point: $\mathbf{x}^{\text{opt}} = \mathbf{x}_{k+1}$.

3.6. Proceed with the next iteration: $k \leftarrow k+1$.

4) For $k \geq K$:

4.1. Generate a point $\mathbf{x}_k^{k+1}$ inside the vicinity $\mathbf{V}$, by using the U-PRSG.

4.2. If $\mathbf{x}_k^{k+1} \in \{\mathbf{x}_i\}_{i \in \overline{0,k}}$, the point already exists and it must be replaced by a different one. Repeat the previous step.

4.3. Otherwise, evaluate the performance of the new point: $f(\mathbf{x}_k^{k+1})$.

4.4. If $\left|f\left(\mathbf{x}_k^{k+1}\right)-f\left(\mathbf{x}^{\text{opt}}\right)\right|<\varepsilon$, stop the search, as no real improvement is obtained. If, moreover, $f\left(\mathbf{x}_k^{k+1}\right)$ is better than $f\left(\mathbf{x}^{\text{opt}}\right)$, then update the optimal point: $\mathbf{x}^{\text{opt}}=\mathbf{x}_k^{k+1}$. Go directly to the final stage.

4.5. Otherwise, store $\mathbf{x}_{k+1}=\mathbf{x}_k^{k+1}$ in memory.

4.6. If $f\left(\mathbf{x}_{k+1}\right)$ is better than $f\left(\mathbf{x}^{\text{opt}}\right)$, update the optimal point: $\mathbf{x}^{\text{opt}}=\mathbf{x}_{k+1}$.

4.7. Proceed with the next iteration: $k\leftarrow k+1$.

5) Return:

– The optimal point: $\mathbf{x}^{\text{opt}}$.

– The optimal performance: $f\left(\mathbf{x}^{\text{opt}}\right)$.

Algorithm 1.1. *Basic Monte Carlo procedure for local heuristic optimization*

Usually, the numerical procedure of algorithm 1.1 is time-consuming and greedy. Starting from a certain iteration, the search for duplicates among the already generated points can become much slower than the evaluation of performance for the newly generated point. However, this step is absolutely necessary in the third stage of the algorithm in order to ensure the minimum degree of granularity and to avoid inconsistent results. On the contrary, this step could be missed in the fourth stage, especially when the performance is evaluated faster than the search for duplicates (thus, step 4.2 can be removed from the procedure in order to speed up the search.)

This procedure is integrated into some of the metaheuristics from this chapter.

1.3. Hill climbing

We start with a metaheuristic of great simplicity, but that reveals, on the one hand, the basic principle of PRS generators use (that allow advancing toward the optimum) and, on the other hand, a link to the *artificial intelligence* (AI) [RUS 95]. Like this method, many other metaheuristics are (directly or indirectly) bound to AI techniques.

Moreover, this method is an example of how the Monte Carlo principle can be exploited in heuristic optimization.

From the beginning, we assume that the f criterion has to be maximized in a vicinity $\mathbf{V}$ inside $\mathbf{S}$. Since $\mathbf{S}$ is bounded, $\mathbf{V}$ inherits the same property. Thus, bounds are well known along each axis of the Cartesian space $\mathbb{R}^{nx}$. It is not mandatory, however, that the vicinity must have the shape of a hypercube or the criterion must have a smooth variation. Even though the criterion could exhibit more local extremes, the vicinity should be selected to embrace the global maximum.

Let $\mathbf{x}_0 \in \mathbf{V}$ be the initial point to start the search. This point could be seen as a place from which some tourist starts a journey to the top of a hill or mountain. Presumably, the tourist is not very well informed on the path to follow so that he/she has to advance quite blindly. His/her strategy is simple, though. Each time a higher position is conquered (comparing to the previous position), as decided by the "altitude" criterion f, the tourist tries to conserve it and avoids going beneath. Nevertheless, the tourist cannot leave the peak zone (i.e. the $\mathbf{V}$ vicinity). When located in the current position $\mathbf{x}_k \in \mathbf{V}$, the tourist seeks to find the path to the next position by selecting a direction at random. More specifically, the next position is evaluated as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_{k+1}, \quad \forall k \in \mathbf{N}, \tag{1.7}$$

where $\Delta\mathbf{x}_{k+1}$ is a randomly selected offset, in Monte Carlo spirit, such that:

$$\mathbf{x}_{k+1} \in \mathrm{V} \quad \text{and} \quad f(\mathbf{x}_{k+1}) > f(\mathbf{x}_k). \tag{1.8}$$

The tourist stops when one or several conditions given below are met:

– he/she cannot move, after a number of attempts, say $N \in \mathbf{N}^*$, to find the good path;

– the altitude difference between two successive positions remains too small, lower than some threshold, say $\delta > 0$, after a number of successive attempts, say $M \in \mathbf{N}^*$ (practically, the tourist cannot leave the same level contour line – the "isolevel" curve);

– the estimated gradient of his/her path, namely the vector:

$$\mathbf{f}_{k+1} = \left[\frac{f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k)}{x_{k+1,1} - x_{k,1}} \quad \frac{f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k)}{x_{k+1,2} - x_{k,2}} \quad \cdots \quad \frac{f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k)}{x_{k+1,nx} - x_{k,nx}} \right]^T \quad [1.9]$$

had too small a norm (below some threshold $\varepsilon > 0$), during the last $M \in \mathbf{N}^*$ attempts.

The last two conditions are practically equivalent from the tourist dynamics point of view, but the gradient could be almost null on the peak.

The basic procedure of hill (mountain) climbing is described in algorithm 1.2.

1) Input data:

– Search vicinity $\mathbf{V}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Altitude indicator, f (criterion to maximize).

– Maximum number of attempts to escape from current position, N.

– Maximum number of attempts to find a better position than the current one, M.

– Threshold to detect the isolevel curve, $\delta > 0$.

2) Initialization.

a) Select at random (but uniformly distributed) the tourist departure point $\mathbf{x}_0 \in \mathbf{V}$. A U-PRSG of nx size has to be used in this aim. If the departure point does not belong to $\mathbf{V}$, then it will be generated until this property is verified.

b) Evaluate the altitude of departure point: $f(\mathbf{x}_0)$.

c) Set a counter for the number of attempts to escape from the current position: $n = 0$.

d) Set a counter for remaining on the isolevel curve: $m = 0$.

e) Set the starting iterations number: $k = 0$.

3) Approaching the peak. For $k \geq 0$:

3.1. Perform attempts to escape from the current position:

3.1.1. Use the U-PRSG to generate an offset along some search direction: $\Delta\mathbf{x}_k^{k+1}$. The generator has to operate inside a hypercube that includes the vicinity, but stays as narrow as possible.

3.1.2. While $\mathbf{x}_k + \Delta\mathbf{x}_k^{k+1} \notin \mathbf{V}$, calibrate the U-PRSG to generate a new offset inside the hypercube $\left[\mathbf{0}, \Delta\mathbf{x}_k^{k+1}\right]$, where $\Delta\mathbf{x}_k^{k+1}$ is the most recently generated offset.

3.1.3. Set the offset: $\Delta\mathbf{x}_{k+1} = \Delta\mathbf{x}_k^{k+1}$. (Now, $\mathbf{x}_k + \Delta\mathbf{x}_{k+1} \in \mathbf{V}$ for sure.)

3.2. Test if the altitude increases if the tourist would move to the new position:

3.2.1. Evaluate the altitude of the proposed position: $f\left(\mathbf{x}_k + \Delta\mathbf{x}_{k+1}\right)$.

3.2.2. If $f\left(\mathbf{x}_k + \Delta\mathbf{x}_{k+1}\right) > f\left(\mathbf{x}_k\right)$, then:

3.2.2.1. Allow the tourist to move into the new position: $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_{n+1}$. (The altitude is already known: $f\left(\mathbf{x}_{k+1}\right) = f\left(\mathbf{x}_k + \Delta\mathbf{x}_{k+1}\right)$.)

3.2.2.2. Reset the counter related to blocking on the same position: $n \leftarrow 0$.

3.2.2.3. If $f\left(\mathbf{x}_{k+1}\right) - f\left(\mathbf{x}_k\right) < \delta$, then the tourist cannot leave the isolevel curve and, thus, the corresponding counter increases: $m \leftarrow m + 1$.

3.2.2.4. Otherwise, reset the corresponding counter: $m \leftarrow 0$.

3.2.3. Otherwise, the tourist has to conserve the current position ($\mathbf{x}_{k+1} = \mathbf{x}_k$) and the blocking counter increases: $n \leftarrow n + 1$.

3.3. If $n > N$ or $m > M$ stop the journey toward the peak and go to the final stage.

3.4. Otherwise, proceed with the next iteration: $k \leftarrow k + 1$.

4) Return:

– The tourist current position: $\mathbf{x}_{k+1}$.

– The tourist current altitude, $f\left(\mathbf{x}_{k+1}\right)$, assumed to approximate the hill peak height.

Algorithm 1.2. *Basic hill climbing procedure*

Algorithm 1.2 was presented from informatics perspective. Therefore, it could slightly be different from other hill climbing algorithms found in the AI literature. The configuring parameters (N, M, δ) normally have implicit values, in order to help the user making a choice. Usually, the user is more likely tempted to search his/her next good path starting from the current position than to advance toward the peak in small steps. Consequently, $N \in \overline{10,100}$, while $M \in \overline{5,10}$. Concerning the δ threshold, it actually is the least significant part of altitude numerical representation. Normally, this parameter should be set at the scale of altitude indicator f. Usually, if f does not change any more, up to the last 3–7 digits, then the search can be stopped. For example, if f varies in the range 0–1,000, then we can set $\delta = 10^{-3}$. But, if f is 1,000 times larger, maybe a better choice is $\delta = 1$.

In general, the performance of algorithm 1.2 is modest, both in terms of accuracy and convergence speed. One first critical step is 1.a (selection of departure point). If the $\mathbf{V}$ vicinity is defined by complex equations, then the algorithm can spend long time to solve this step. It is suitable to carefully analyze both the vicinity and the criterion before configuring the algorithm. Another critical step is 3.2. This time, there is the risk to enter a very slow loop before returning inside the $\mathbf{V}$ vicinity. Nevertheless, the loop is not infinite, as the U-PRSG is enforced to work with smaller and smaller hypercubes, which bring the newly generated point inside the vicinity, eventually. During the search, the algorithm can easily be captured by a local optimum, if isolated from the global one by large and deep valleys. Moreover, the search procedure could start to oscillate. Overall, the basic algorithm of hill climbing is easy to implement, but modest in performance.

An improved version of this procedure can be obtained if the tourist is supposed to have more traveling means, especially for guessing the next path to follow without testing too many possibilities. Of course, there is a characteristic to be accounted in this aim: the climbing directions to follow can be pointed by the gradient of altitude indicator, namely $\mathbf{f}_{\mathbf{x}}$. If the gradient could somehow be

approximated, then the tourist should use this new information like a compass in order to speed up the journey and, perhaps, to reach the peak with better accuracy.

A simple technique to estimate the gradient is to take into account two successive positions, such as $\mathbf{x}_k$ and $\mathbf{x}_{k+1}$, like in definition [1.9]. By convention, every time the denominator is null, the corresponding gradient component is null as well. The gradient estimation is assigned to tourist position $\mathbf{x}_{k+1}$. If a gradient estimator is available, then the variable step Cauchy algorithm [BOR 13] can be employed to speed up the search. The tourist has now a compass to approximately set the orientation of the next direction to follow.

The new hill climbing procedure is described in algorithm 1.3.

1) Input data:

– Search vicinity $\mathbf{V}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Altitude indicator, f (criterion to maximize).

– Maximum number of attempts to escape from current position, N.

– Maximum number of attempts to find a better position than the current one, M.

– Threshold to detect the isolevel curve, $\varepsilon > 0$.

2) Initialization.

a) Select at random (but uniformly distributed) the tourist departure point $\mathbf{x}_0 \in \mathbf{V}$. A U-PRSG of nx size has to be used in this aim. If the departure point does not belong to $\mathbf{V}$, then it will be generated until this property is verified.

b) Evaluate the altitude of departure point: $f(\mathbf{x}_0)$.

c) Set the initial gradient for the departure point, $\mathbf{f}_0 \in \mathbf{R}^{nx}$ (usually, $\mathbf{f}_0 = \mathbf{0}$).

d) Set the initial advancement step, $\alpha_0 \in \mathbf{R}$ (usually, $\alpha_0 = 1$).

e) Set a counter for the number of attempts to escape from the current position: $n = 0$.

f) Set a counter for remaining on the isolevel curve: $m = 0$.

g) Set the starting iterations number: $k = 0$.

3) Approaching the peak. For $k \geq 0$:

3.1. Perform attempts to escape from the current position:

3.1.1. Use the U-PRSG to generate an offset along some search direction: $\Delta \mathbf{x}_k^{k+1}$.

3.1.2. While $\mathbf{x}_k + \Delta \mathbf{x}_k^{k+1} \notin \mathbf{V}$, calibrate the U-PRSG to generate a new offset inside the hypercube $\left[\mathbf{0}, \Delta \mathbf{x}_k^{k+1}\right]$, where $\Delta \mathbf{x}_k^{k+1}$ is the most recently generated offset.

3.1.3. Evaluate the altitude of proposed position: $f\left(\mathbf{x}_k^{k+1}\right)$.

3.1.4. Estimate the gradient for the presumably next position, by using $\mathbf{x}_k$ and $\mathbf{x}_k^{k+1}$ in definition [1.9] (with $\mathbf{x}_k^{k+1}$ instead of $\mathbf{x}_{k+1}$). Thus, we obtain $\mathbf{f}_{k+1}$ (with null components corresponding to zero divide, if any), which is stored in memory.

3.1.5. Estimate the optimal advancement step: $\alpha_{k+1} = \alpha_k - \mathbf{f}_{k+1}^T \mathbf{f}_k$. (The previous gradient, $\mathbf{f}_k$, can be found in memory).

3.1.6. While $\mathbf{x}_k + \alpha_{k+1} \mathbf{f}_{k+1} \notin \mathbf{V}$, calibrate the U-PRSG to generate a new advancement step in $\left[0, \alpha_{k+1}\right]$, where α_{k+1} is the most recently generated step (starting from the optimal step of 3.1.5.).

3.1.7. Set the final offset: $\Delta \mathbf{x}_{k+1} = \alpha_{k+1} \mathbf{f}_{k+1}$.

3.2. Test if the altitude increases if the tourist moves to the new position:

3.2.1. Evaluate the altitude of the proposed position: $f\left(\mathbf{x}_k + \Delta \mathbf{x}_{k+1}\right)$.

3.2.2. If $f\left(\mathbf{x}_k + \Delta \mathbf{x}_{k+1}\right) > f\left(\mathbf{x}_k\right)$, then:

3.2.2.1. Allow the tourist to move into the new position: $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta \mathbf{x}_{n+1}$. (The altitude is already known: $f\left(\mathbf{x}_{k+1}\right) = f\left(\mathbf{x}_k + \Delta \mathbf{x}_{k+1}\right)$).

3.2.2.2. Reset the counter related to blocking on the same position: $n \leftarrow 0$.

3.2.2.3. If $\|\mathbf{f}_k\| < \varepsilon$, then the tourist practically keeps the same altitude and the corresponding counter increases: $m \leftarrow m + 1$.

3.2.2.4. Otherwise, reset the corresponding counter: $m \leftarrow 0$.

3.2.3. Otherwise, the tourist has to conserve the current

position ($\mathbf{x}_{k+1} = \mathbf{x}_k$) and the blocking counter increases: $n \leftarrow n+1$.

3.3. If $n > N$ or $m > M$ stop the search and go to the final stage.

3.4. Otherwise, proceed with the next iteration: $k \leftarrow k+1$.

4) Return:

– The tourist current position: $\mathbf{x}_{k+1}$.

– The tourist current altitude, $f(\mathbf{x}_{k+1})$, assumed to approximate the hill peak height.

Algorithm 1.3. *Improved hill climbing procedure, by using the Cauchy compass*

The complexity of algorithm 1.3 is slightly increased in comparison to algorithm 1.2. In order to better understand this procedure, the readers should analyze first algorithm 2.8 (the variable step Cauchy algorithm) from [BOR 13]. The same configuring rules apply here as in the case of the previous algorithm. Nevertheless, the M parameter can go down to null, since the estimated gradient is more sensitive to the directions leading to criterion extremes. In turn, the small gradient values are increasing the risk of oscillation. In this case, the search stops as soon as the norm of estimated gradient falls for the first time below the prescribed threshold ε. By difference from δ threshold in the previous algorithm, ε has to be calibrated depending on the altitude indicator variations and the scale of searching vicinity (see definition [1.9] again).

Algorithm 1.3 performs better than algorithm 1.2 in terms of accuracy and convergence speed. Due to the estimated gradient, the optimum can be found after a smaller number of iterations (even though this estimation is not that much accurate). The tourist is not now advancing as blindly as before, but with the help of a compass, which constitutes the main advantage.

This procedure can be adapted to find minima instead of maxima and it does not raise any implementation issues. However, the overall performance of improved hill climbing algorithm is inferior to other metaheuristics.

1.4. Taboo search

1.4.1. *Principle*

The metaheuristic described in this section belongs to greedy descent local methods class. For this type of method, the search starts from a possible solution $\mathbf{x}_i$ of $\mathbf{S}$. The strategy is then to focus on a local vicinity $\mathbf{V}(\mathbf{x}_i)$ in order to find another solution $\mathbf{x}_j$ that can improve the criterion's current performance. The vicinity $\mathbf{V}(\mathbf{x}_i)$ corresponds to the set of all accessible solutions, after applying a single admissible movement, displacement or transition from $\mathbf{x}_i$. Usually, this set is a hypercube or a hypersphere including the current solution $\mathbf{x}_i$.

1.4.2. *Greedy descent algorithm*

Assume that the goal is to minimize the f criterion. Then the search for a minimal point is performed over the vicinity $\mathbf{V}(\mathbf{x}_i)$ (corresponding to current solution $\mathbf{x}_i$), and a better solution $\mathbf{x}_j$ is recorded if $f(\mathbf{x}_j) < f(\mathbf{x}_i)$. In order to build the $\mathbf{V}(\mathbf{x}_i)$ vicinity, it suffices to generate some small offsets at random around the minimal point $\mathbf{x}_i$ such that the resulted subset is included in the search space $\mathbf{S}$. The variation range of uniformly distributed PRS (U-PRS) can be set according to the diameter of search space:

$$\mathbf{D}(\mathbf{S}) = \sup_{\mathbf{x},\mathbf{y}\in\mathbf{S}} \{\|\mathbf{x}-\mathbf{y}\|\}. \qquad [1.10]$$

For example, if $\mathbf{V}(\mathbf{x}_i)$ is set as a hypercube, then:

$$\begin{aligned}\mathrm{V}(\mathbf{x}_i) = &\left[x_{i,1} - \gamma_{i,1}, x_{i,1} + \delta_{i,1}\right] \times \left[x_{i,2} - \gamma_{i,2}, x_{i,2} + \delta_{i,2}\right] \times \cdots \\ &\times \left[x_{i,nx} - \gamma_{i,nx}, x_{i,nx} + \delta_{i,nx}\right]\end{aligned} \quad [1.11]$$

where the bounds $\left\{\gamma_{i,n}\right\}_{n \in \overline{1,nx}}$ and $\left\{\delta_{i,n}\right\}_{n \in \overline{1,nx}}$ are numbers of some U-PRS.

In the case of spherical vicinities, the below definition can be employed:

$$\mathrm{V}(\mathbf{x}_i) = \left\{\mathbf{x} \in \mathrm{S} \,\middle|\, \|\mathbf{x} - \mathbf{x}_i\| < \rho_i\right\}, \quad [1.12]$$

where the radius ρ_i is obtained with the help of a U-PRSG, after being calibrated by the diameter $\mathbf{D}(\mathbf{S})$.

In this framework, the greedy descent procedure is synthesized in algorithm 1.4.

1) Input data:

– Search space $\mathbf{S}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Depth indicator, f (criterion to minimize).

– The U-PRSG bounds to generate vicinities. (Usually, such bounds are expressed as fractions of search space diameter).

– Minimum number of grains for Monte Carlo method, N.

– Accuracy threshold for Monte Carlo method, $\varepsilon > 0$.

2) Initialization.

a) Select at random (but uniformly distributed) the starting point $\mathbf{x}_0 \in \mathbf{S}$. A U-PRSG of nx size has to be used in this aim. If the starting point does not belong to $\mathbf{S}$, then it will be generated until this property is verified. (It is necessary to take into account the topology of $\mathbf{S}$).

b) Evaluate the depth of starting point: $f(\mathbf{x}_0)$.

c) Set the starting iterations number: $k = 0$.

3) For $k \geq 0$:

3.1. Call the U-PRSG (after calibration according to prescriptions) to generate the vicinity $\mathbf{V}(\mathbf{x}_k)$.

3.2. While $\mathbf{V}(\mathbf{x}_k) \not\subset \mathbf{S}$, generate a new vicinity by using U-PRSG, but after calibration according to the last generated vicinity.

3.3. Now, $\mathbf{V}(\mathbf{x}_k) \subset \mathbf{S}$. This allows finding the best solution of $\mathbf{V}(\mathbf{x}_k)$ vicinity, say $\mathbf{x}_k^{\text{opt}}$. The local search inside the vicinity $\mathbf{V}(\mathbf{x}_k)$ can be performed by using the Monte Carlo method (algorithm 1.1), with granularity N and accuracy ε (already known).

3.4. Evaluate the depth of the best local solution: $f\left(\mathbf{x}_k^{\text{opt}}\right)$.

3.5. If $f\left(\mathbf{x}_k^{\text{opt}}\right) \geq f(\mathbf{x}_k)$, stop the search and go to the final stage.

3.6. Otherwise, update the minimal point: $\mathbf{x}_{k+1} = \mathbf{x}_k^{\text{opt}}$.

3.7. Proceed with the next iteration: $k \leftarrow k+1$.

4) Return:

– The current minimal point: $\mathbf{x}_k$.

– The current minimal depth: $f(\mathbf{x}_k)$.

Algorithm 1.4. *Greedy descent procedure*

The local search procedure of step 3.3 in algorithm 1.4 (Monte Carlo method) can be replaced by a different optimization technique, even withdrawn from the exact methods class [BOR 13], if allowed by the f criterion. In general, if possible, it is suitable to combine metaheuristics and exact methods in the same procedure in order to increase the accuracy of the final result. By means of metaheuristics, some vicinity of global optimum can be insulated, whereas with exact methods the vicinity can be exploited efficiently. Nevertheless, if the criterion is too fractal in nature, the use of exact methods should be avoided.

The approach given above is easy to implement, but there is the risk of rapidly stopping the search on a local minimum. A strategy to avoid this ending is to start the search from several initial points, uniformly spread on the search space. Many improvements of greedy descent procedure have been introduced in the literature so far. One of them, which is quite interesting, is based on taboo search, as described in what follows.

1.4.3. *Taboo search method*

While keeping the principle of local search for minimizing a criterion, by this method, there is the possibility of jumping out from the capturing vicinity and to explore a different zone. Hereafter, the term *movement* will stand for any modification allowing the search to focus on vicinity in the neighborhood of the current vicinity. As usual, the search starts from some initial point (solution) $\mathbf{x}_i$, in the vicinity $\mathbf{V}(\mathbf{x}_i)$, but it is permitted to relocate the exploitation around another point (solution) $\mathbf{x}_j \in \mathbf{V}(\mathbf{x}_i)$, even though the criterion degrades $f(\mathbf{x}_j) > f(\mathbf{x}_i)$. This is actually a movement toward another zone of interest. However, in order to avoid infinite search loops, once a solution is focused on, it will never be focused on again in the future. Thus, the focused solutions become untouchable, "taboo" (that gives the name of the method) [GLO 89, GLO 90, ENN 04, GHE 07].

Assume that the solution $\mathbf{x}_i$ has been visited at the *k*th iteration. Then the last $N_{k,i}$ visited solutions are taboo (due to the performed movements). Denote by $\mathbf{T}_{k,i}$ the list of the last taboo solutions (*the taboo list*). According to the principle of the method, it is forbidden to perform movements leading to solutions from the taboo list $\mathbf{T}_{k,i}$.

The currently exploited vicinity is then:

$$\mathbf{V}_k(\mathbf{x}_i) = \mathbf{V}(\mathbf{x}_i) \setminus \mathbf{T}_{k,i}. \qquad [1.13]$$

Starting from the solution $\mathbf{x}_i$, a set of possible movements, say $\mathbf{M}_{k,i}$, can be built, during the kth iteration. Let $m \in \mathbf{M}_{k,i}$ be such a movement. Conventionally, $\mathbf{x}_i \xrightarrow{m} \mathbf{x}_j$ stands for the transition from solution $\mathbf{x}_i$ to a new point $\mathbf{x}_j$, as a result of movement m. Then:

$$\mathbf{V}_k(\mathbf{x}_i) = \left\{ \mathbf{x}_j \in \mathbf{V}(\mathbf{x}_i) \;\middle|\; \exists m \in \mathbf{M}_{k,i},\ \mathbf{x}_i \xrightarrow{m} \mathbf{x}_j \ \&\ \mathbf{x}_j \notin \mathbf{T}_{k,i} \right\}. \quad [1.14]$$

Practically, definition [1.14] shows that it is only possible to generate new solutions in the vicinity of the current solution.

Although this approach is quite interesting, there is no guarantee that the loops will be completely avoided. Moreover, some possible solutions could become taboo even before being tested.

This fault can be removed by relaxing the definitions related to taboo labels. Thus, according to the aspiration principle, any movement leading to a better solution can be allowed in taboo zones, which paves the way toward possible solutions not yet taken into consideration.

We can notice that, often, it is easier to estimate the criterion variation $\Delta f = f(\mathbf{x}_j) - f(\mathbf{x}_i)$ than to accurately compute it for each iteration, which allows decreasing the computational burden.

The set of admissible movements can vary tremendously, depending on the application. For example, in the case of the traveling salesman problem, recall that any of the N cities has to be visited once and only once in a succession allowing minimizing the overall traveled distance. The following movement types can be considered here in the string of visited cities:

– displacement of a city: CDEGAF**H**JKI $\xrightarrow{dis}$ CDE**H**GAFJKI;

– permutation of two cities: CDEGAFHJKI $\xrightarrow{per}$ CD**J**HGAF**E**KI;

– inversion of visiting succession for a cities group:

- CDEGAFHJKI $\xrightarrow{inv}$ CDFAGEHJKI.

In this example, the solution $\mathbf{x}_i$ is CDEGAFHJKI, which means that the cities can be visited in a certain order. The movements are changing this order on purpose to find another possible solution, $\mathbf{x}_j$.

1.4.4. *Taboo list*

The definition and the size of taboo list are important parameters of search. If the list is too large, then the search is restricted to small areas, but the risk to miss the global optimum becomes important. On the contrary, if the list is too small, it is very likely that the search is slowed down by a loop.

The common solution is to work with constant length taboo list. In this case, the most recently visited solution (as a result of the last performed movement) enters the taboo list, while the oldest solution is removed from the list, as soon as the list has reached its maximum length. The taboo list thus acts like a last-in-first-out (LIFO) stack.

However, in many applications, it is more useful to work with adaptive taboo list length, say $N_{k,i}$, which can be modified at each iteration between some bounds:

$$N_{\min} \le N_{k,i} \le N_{\max}, \quad \forall k,i \in \mathbf{N}. \qquad [1.15]$$

Many adaptation rules were introduced so far in the literature. Two of them seem to be effective in real applications:

– if the currently generated solution improves the criterion, then the taboo list length is decreased by 1: $N_{k,i} \leftarrow \max\left\{N_{k,i}-1, N_{\min}\right\}$. Thus, the two oldest solutions are removed from the list, while the most recent solution is entered into the list. If the list length falls below the lower limit, then either the oldest solution is removed or the length decreasing is postponed;

– if the currently generated solution does not improve the criterion, then the length is increased by 1: $N_{k,i} \leftarrow \min\left\{N_{k,i}+1, N_{\max}\right\}$. Thus, the solution enters the list and no other solution is removed. If the list length becomes too big (larger than the upper limit), then the oldest solution is removed.

Notice that the taboo labeling applied to the latest movements can block the search (as will be revealed in a later example). In this case, it is necessary to allow violation of taboo principle in order to escape from the trap and jump to another possible solutions to test.

1.4.5. *Taboo search algorithm*

The taboo search procedure is described in algorithm 1.5.

1) Input data:

– Search space $\mathbf{S}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Optimization criterion, f.

– Types of possible movements starting from a solution (all restrictions accounted, if any).

– Minimum number of solutions for the current vicinity, $N_{\mathbf{v}}$.

– Bounds of taboo list length: $N_{\min}$ and $N_{\max}$, with $N_{\min} < N_{\max}$.

– Maximum number of iterations, K.

– Accuracy threshold, $\varepsilon > 0$.

– Maximum number of found solutions that comply with the accuracy threshold, M.

– Maximum number of iterations to stop the search since the optimal solution did not change, N.

2) Initialization.

a) Select at random (but uniformly distributed) the starting point $\mathbf{x}_{0,0} \in \mathbf{S}$. A U-PRSG of nx size has to be used in this aim. If the starting point does not belong to $\mathbf{S}$, then it will be generated until this property is verified. (It is necessary to take into account the topology of $\mathbf{S}$).

b) Evaluate the starting point performance: $f\left(\mathbf{x}_{0,0}\right)$.

c) Initialize the optimal solution: $\mathbf{x}^{\text{opt}} = \mathbf{x}_{0,0}$.

d) Initialize the taboo list: $\mathbf{T}_{0,0} = \{\mathbf{x}_{0,0}\}$.

e) Set the initial counter associated to the found solutions that comply with the accuracy threshold (the *threshold counter*): $m = 0$.

f) Set the initial counter associated to the number of iterations during which the optimal solution did not change (the *blocking counter*): $n = 0$.

g) Set the position of the initial optimal solution: $i = 0$.

h) Set the starting iterations number: $k = 0$.

3) For $k \in \overline{0, K-1}$:

3.1. Specify all possible movements (by accounting all restrictions, if any), starting from the current solution, $\mathbf{x}_{k,i}$. Although the set of such movements if finite, its size could be very large.

3.2. Initialize the vicinity of current solution: $\mathbf{V}_k(\mathbf{x}_{k,i}) = \varnothing$.

3.3. While $\mathbf{V}_k(\mathbf{x}_{k,i})$ includes less than $N_{\mathbf{V}}$ points, do:

3.3.1. Call the U-PRSG to select a possible movement, pm. (If there are no possible movements to select, break the loop and jump to step 3.4.)

3.3.2. Perform the pm movement to get to a new point:

$$\mathbf{x}_{k,i} \xrightarrow{pm} \mathbf{x}_{k,i}^{pm}.$$

3.3.3. If the new point belongs to the search space ($\mathbf{x}_{k,i}^{pm} \in \mathbf{S}$), but does not belong to the taboo list ($\mathbf{x}_{k,i}^{pm} \notin \mathbf{T}_{k,i}$), then add the point to the vicinity:

$$\mathbf{V}_k(\mathbf{x}_{k,i}) \leftarrow \mathbf{V}_k(\mathbf{x}_{k,i}) \cup \{\mathbf{x}_{k,i}^{pm}\}.$$

3.3.4. keep the vicinity unchanged.

3.4. If the vicinity includes at least one point ($\mathbf{V}_k(\mathbf{x}_{k,i}) \neq \varnothing$):

3.4.1. Solve the problem below by exhaustive search:

$$\mathbf{x}_{k,j} = \underset{\mathbf{x} \in \mathbf{V}_k(\mathbf{x}_{k,i})}{\text{argopt}} f(\mathbf{x}).$$

Here, $\mathbf{x}_{k,j}$ is the new solution (the firstly found better point) and j points to its position inside the vicinity.

3.4.2. Update the new solution: $\mathbf{x}_{k+1,j} = \mathbf{x}_{k,j}$.

3.4.3. Add the new solution to the taboo list:

$$\mathbf{T}_{k+1,j} = \mathbf{T}_{k,i} \cup \{\mathbf{x}_{k+1,j}\}$$

3.4.4. Update the solution position: $i = j$.

3.4.5. If the performance of new solution ($f\left(\mathbf{x}_{k+1,i}\right)$, already computed) is better than the current optimal performance ($f\left(\mathbf{x}^{\text{opt}}\right)$, available):

3.4.5.1. Update the optimal solution: $\mathbf{x}^{\text{opt}} = \mathbf{x}_{k+1,i}$.

3.4.5.2. Reset the blocking counter: $n \leftarrow 0$.

3.4.5.3. If $\left|f\left(\mathbf{x}_{k+1,i}\right) - f\left(\mathbf{x}^{\text{opt}}\right)\right| < \varepsilon$, increment the threshold counter: $m \leftarrow m + 1$.

3.4.5.4. If $m > M$, stop the search, as no real progress is made. Go directly to final stage (no. 4).

3.4.5.5. If possible, remove one or two old solutions from taboo list, such that the number of remaining taboo solutions is at least $N_{\min}$. (If this number already is smaller than $N_{\min}$, then keep unchanged the taboo list.)

3.4.6. Otherwise, the optimal solution does not change and then:

3.4.6.1. Increment the blocking counter: $n \leftarrow n + 1$.

3.4.6.2. If $n > N$, stop the search, as very likely, the algorithm has reached the best solution that can be found with this procedure. Go directly to final stage (no. 4).

3.4.6.3. Otherwise, check whether the updated taboo list ($\mathbf{T}_{k+1,i}$) includes or not too many solutions (more than $N_{\max}$). In case the number is too big, remove the oldest solutions in order to keep only the most recent $N_{\max}$ taboo solutions.

3.4.6.4. Reset the threshold counter: $m = 0$.

3.4.7. Proceed with the next iteration: $k \leftarrow k + 1$.

3.5. Otherwise, since the current vicinity is void, the procedure is blocked and the main loop has to be broken. Go directly to final stage (no. 4).

4) Return:

– The current optimal point: $\mathbf{x}_k$.

– The current optimal performance: $f\left(\mathbf{x}_k\right)$.

Algorithm 1.5. *Taboo search procedure*

Like in the case of previous procedures, several stop tests are integrated into algorithm 1.5. First, the search cannot overpass a

maximum number of iterations, K. If this condition enforces the procedure to stop prematurely, it is recommended to increase K and to restart the algorithm from the current taboo list and optimal solution of previous run. The search can stop when either the last M optimal solutions of taboo list do not improve the criterion performance or the optimal solutions are resisted for at least N successive iterations, without being overthrown. The last two stop tests actually are normal exits for a well-designed taboo procedure.

The taboo list length has been limited in order to speed up the search (if the taboo list is too large, looking for duplicate solutions could be lengthy). In turn, the risk of the algorithm to be captured into a loop or to oscillate has increased. Note, however, that the parameters M and N are preventing infinite loops. The procedure is exposed to the danger to be captured by a local optimum with big attraction force (i.e. quite well insulated from the global optimum). In this case, it is wise to restart the taboo procedure several times from different initializations.

The configuration parameters of algorithm 1.5 (especially K, M, N and ε) can be set according to similar recommendations like for algorithm 1.3. For example, $K \in \overline{100,200}$, $M \in \overline{5,10}$, $N \in \overline{10,30}$, while ε is determined by the variation range of f criterion. Of course, the taboo list length has to be adjusted according to M and especially N.

Step 3.1 of algorithm 1.5 should be managed carefully. Return to the traveling salesman example in order to better understand how the possible movements can be selected. If CDEGAFHJKI is an initial solution, then recall that only three types of movements can be employed to find a different solution. Each one of such movements is well defined by some parameters as follows:

– displacement can be performed if the position of the city to be displaced and the position where to displace it are known;

– permutation requires knowing the two positions of the cities to be exchanged;

– inversion needs the position of the cities group and the number of cities within the group.

When varying all those parameters, a large number of possible movements is obtained. Testing all possible movements actually means to perform an exhaustive search, which might not be a good strategy. Therefore, the Monte Carlo principle could be adopted (like in step 3.3 of the algorithm). The effective implementation of this principle depends on each application (as the possible movements have to be indexed accordingly, to facilitate the use of some U-PRSG). The exhaustive search is a strategy to consider only if the number of possible movements is small enough.

Concerning the constraints, in the case of traveling salesman, we can imagine, for example, that the journey from city A to city B must include a passage through city C. Consequently, it is impossible to exchange cities A and C or B and C. The constraints can change from iteration to iteration, depending on the current optimal solution. For example, different constraints apply to solutions CDEGAFHJKI and CDFAGEHJKI. If the path from A to H has to include F, then, in the first solution (CDEGAFHJKI), A cannot exchange with F. In the second solution (CDFAGEHJKI), exchanging A and F is possible, but, in this case, the cities group GE cannot lie between A, F and H. So, it is necessary to move GE to another zone.

1.4.6. *Intensification and diversification*

In order to have a better guidance for the search process and, at the same time, to reduce its duration, two actions are possible:

– intensification, which yields more intense exploitation of the current zone;

– diversification, which allows exploration of various regions.

In general, a trade-off between the two actions has to be found in order to propose solutions with good accuracy, after reasonable search durations.

The most commonly employed intensification approaches are the following:

– reducing the length of taboo list, as soon as the solution improves;

– selecting some preferred solutions more often, with performance closer to the best found solutions.

Regarding diversification, the usual approaches are listed below:

– sudden change of focus in the zones where there are solutions waiting to be visited;

– multiple reruns, starting from various initializations, selected at random;

– exclusion of the most visited solutions;

– penalties applied to the solutions nearby the current solution.

1.4.7. *Application examples*

1.4.7.1. *Searching the smallest value on a table*

Table 1.1 exhibits a matrix of expense rates made by a traveling professional in charge of the maintenance of a computer network. Each row of the table is associated with a working point, whereas each column shows the expense rates after a traveler's journey passing through the working points. Somewhere in the table there is a minimum rate of expenses. The problem is then to find the table cell hosting the minimum expenses rate by using taboo search metaheuristic.

7	9	8	11	10	7	9	7	8	5	9	6	7	10	3	4	3
8	6	10	9	8	6	14	17	16	18	16	14	13	8	5	2	7
6	4	5	4	7	8	10	13	10	12	14	16	13	6	3	6	8
7	3	7	3	5	4	7	11	10	19	13	16	15	13	8	9	7
9	6	6	4	8	11	13	14	6	18	17	15	13	11	7	10	9
7	7	8	7	9	7	15	13	16	12	14	16	14	15	14	17	10

Table 1.1. *Expense rates for computer network maintenance missions*

Two randomly chosen initializations are marked in the table. Before starting the taboo search, all possible movements as well as the maximum length of the taboo list have to be specified. For each cell in the table, the admissible movements toward the neighbor cells are illustrated in Figure 1.2.

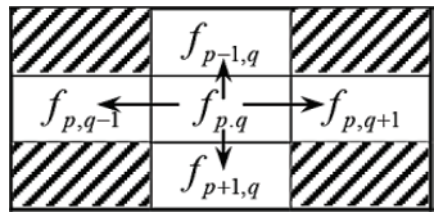


Figure 1.2. *Admissible movements in the problem of minimum expense rates for computer network maintenance missions*

It follows that the set of possible movements starting from the current solution $\mathbf{x}_i = (p,q)$ is the following, for each iteration $k \in \mathbf{N}$:

$$\begin{aligned} \mathrm{M}_{k,i} = \Big\{ &(p,q) \xrightarrow{\text{up}} (p-1,q),\ (p,q) \xrightarrow{\text{down}} (p+1,q),\ \ldots \\ &(p,q) \xrightarrow{\text{left}} (p,q-1),\ (p,q) \xrightarrow{\text{right}} (p,q+1) \Big\}. \end{aligned} \quad [1.16]$$

Since the number of movements is small, the exhaustive search is well suited (no need to implement the Monte Carlo method).

The taboo list can include up to 10 solutions and cannot be void (at least one solution has to receive the taboo label).

For the first initialization (on the left side in Table 1.1), the taboo search is blocked, as illustrated in Table 1.2. This is actually the effect of the test in step 3.5 of algorithm 1.5, which enforces the procedure to end prematurely. In this case, the optimal solution is:

$$\mathbf{x}^{\text{opt}} = (4,2), \quad f\left(\mathbf{x}^{\text{opt}}\right) = f_{4,2} = 3. \quad [1.17]$$

7	9	8	11	10	7	9	7	8	5	9	6	7	10	3	4	3
8	**6**	10	9	8	6	14	17	16	18	16	14	13	8	5	2	7
6	**4**	**5**	**4**	7	8	10	13	10	12	14	16	13	6	3	6	8
7	**3**	7	**3**	5	4	7	11	10	19	13	16	15	13	8	9	7
9	**6**	**6**	**4**	8	11	13	14	6	18	17	15	13	11	7	10	9
7	7	8	7	9	7	15	13	16	12	14	16	14	15	14	17	10

Table 1.2. *Taboo search leading to blocking*

The corresponding taboo list includes 10 solutions:

$$\mathbf{T}_{9,2} = \{(2,2),(3,2),(4,2),(5,2),(5,3),(5,4),(4,4),(3,4),(3,3),(4,3)\}\,, \quad [1.18]$$

while the number of iterations is 9 (count the number of arrows in Table 1.2).

The last found position in the current vicinity is $(4,3)$ (after a movement down). In fact, the last vicinity only includes two elements: $\{(2,3),(4,3)\}$ (according to the taboo method principle).

The second initialization leads to the iterations in Table 1.3.

7	9	8	11	10	7	9	7	8	5	9	6	7	10	3	4	3
8	6	10	9	8	6	14	17	16	18	16	14	13	8	**5**	**2**	7
6	4	5	4	7	8	10	13	**10**	**12**	**14**	16	13	6	**3**	6	8
7	3	7	3	5	4	7	11	**10**	19	**13**	**16**	15	13	**8**	9	7
9	6	6	4	8	11	13	14	**6**	18	17	**15**	**13**	**11**	**7**	10	9
7	7	8	7	9	7	15	13	16	12	14	16	14	15	14	17	10

Table 1.3. *Taboo search leading to optimum solution*

The optimum solution is then:

$$\mathbf{x}^{\text{opt}} = (2,16)\,, \quad f\left(\mathbf{x}^{\text{opt}}\right) = f_{2,16} = 2\,. \qquad [1.19]$$

(The solution [1.19] is even the global one – read Table 1.3 carefully).

Without a termination criterion the algorithm always seeks a better solution. We can easily see that, if the procedure is continued, the optimum solution [1.19] cannot change and the algorithm exits from step 3.4.6.2, where the invariance of the best solution for N iterations is detected.

If the taboo list is limited to 10 elements, then, starting from the movement:

$$\begin{array}{l}(5,14)\xrightarrow{\text{right}}(5,15),\\ f=11 \qquad\quad f=7\end{array} \qquad [1.20]$$

its length should be adjusted. Since the current optimal value of criterion is $f\left(\mathbf{x}^{\text{opt}}\right)=f_{5,9}=6$, while the new solution performance is $f_{5,15}=7$, the search procedure is at step 3.4.6.3 of algorithm 1.5. Consequently, the solution $(5,9)$ (corresponding to the starting point) has to be removed from the taboo list. To compensate the loss, the new solution $(5,15)$ is added to the list. This operation does not affect the optimal solution though. The blocking counter is thus enforced to increment (see step 3.4.6.1 of the algorithm).

The adjustment of taboo list continues in the same way until the movement below:

$$\begin{array}{l}(4,15)\xrightarrow{\text{up}}(3,15),\\ f=8 \qquad\quad f=3\end{array} \qquad [1.21]$$

when the optimal solution has to change from $f\left(\mathbf{x}^{\text{opt}}\right)=f_{5,9}=6$ to $f\left(\mathbf{x}^{\text{opt}}\right)=f_{3,15}=3$. Now, the procedure is at step 3.4.5.5 of the algorithm and the two oldest solutions of taboo list have to be removed. More specifically, the solutions $(3,9)$ (with $f_{3,9}=10$) and

(3,10) (with $f_{3,10} = 12$) are the removed candidates. In turn, the new solution, (3,15) (with $f_{3,15} = 3$), is added to the list.

1.4.7.2. *The problem of N queens*

The problem is placing N queens on a chessboard of size $N \times N$ such that no queen threatens the others (two queens cannot lie on the same row, column or diagonal unless they are in conflict with each other). The taboo search metaheuristic can be used to solve this problem.

In the following example, seven queens are considered. Moreover, the taboo list length is set to 3. The only accepted movements are row permutations. In order to better understand how the taboo procedure works, beside the chessboard in current configuration, to the right side, a table shows all permutations leading to the minimum costs (i.e. number of conflicts), stating from that configuration. The Monte Carlo procedure can be employed to generate possible movements, as their total number per iteration is:

$$1+2+3+4+5+6 = \frac{6 \cdot 7}{2} = 21. \qquad [1.22]$$

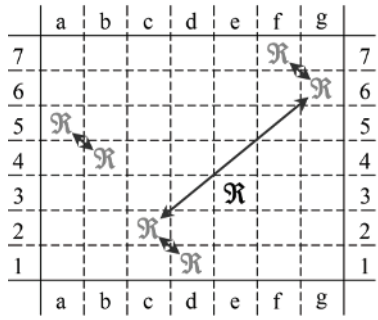
In this application, algorithm 1.5 has been slightly modified: the taboo list does not include solutions anymore (i.e. chessboard configurations showing the queens' placement), but possible movements. In theory, this list can include up to 21 elements (writing this version of taboo search algorithm can be a useful exercise for the readers). The movements leading to the taboo solutions are tagged by a star. Such movements actually become taboo as well.

In this example, the aspiration principle will be applied in the end. According to this principle, a taboo movement is allowed, provided that it leads to a better solution than the current one (and maybe to the optimum solution). Note, however, that Algorithm 1.5 did not integrate this principle. Thus, after analyzing this example, perhaps

the readers will be tempted to design another (more sophisticated) version of algorithm 1.5 so as to include the aspiration principle.

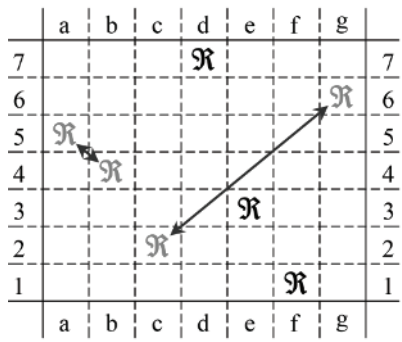
The taboo procedure iterations are described below:

– Initial configuration (four conflicts, as pointed to by the arrows)



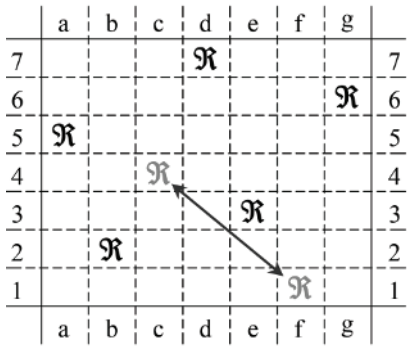
$\mathbf{M}_{0,0}$	Conflitcs	Cost
$1 \leftrightarrow 7$	a5 ⇌ b4 c2 ⇌ g6	2
$2 \leftrightarrow 4$	c4 ⇌ f7 f7 ⇌ g6	2
$2 \leftrightarrow 6$	a5 ⇌ b4 c6 ⇌ g2	2
$5 \leftrightarrow 6$	c2 ⇌ d1 e3 ⇌ g5	2
$1 \leftrightarrow 5$	c2 ⇌ g6 d5 ⇌ f7 f7 ⇌ g6	3

– Iteration no. 1 (after the movement $1 \leftrightarrow 7$ that becomes taboo; two conflicts)



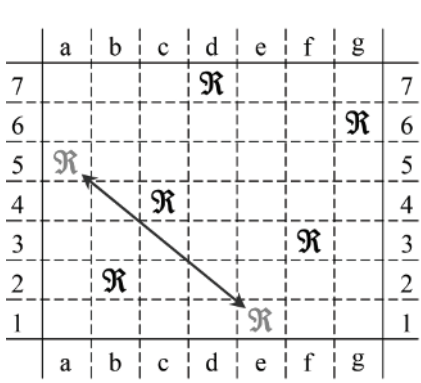
$\mathbf{M}_{1,1}$	Conflicts	Cost
$2 \leftrightarrow 4$	c4 ⇌ f1	1
$1 \leftrightarrow 6$	a5 ⇌ b4 e3 ⇌ g1	2
$2 \leftrightarrow 5$	b4 ⇌ c5 c5 ⇌ e3	2
$1 \leftrightarrow 2$	a5 ⇌ b4 c1 ⇌ e3 e3 ⇌ f2	3
$1 \leftrightarrow 3$	a5 ⇌ b4 b4 ⇌ e1 c2 ⇌ g6	3

– Iteration no. 2 (after the movement $2 \leftrightarrow 4$ that becomes taboo; one conflict)



$\mathbf{M}_{2,1}$	Conflicts	Cost
$1 \leftrightarrow 3$	a5 ⇌ e1	1
$6 \leftrightarrow 7$	b2 ⇌ g7 c4 ⇌ f1	2
$1 \leftrightarrow 7^*$	c4 ⇌ f7 f7 ⇌ g6	2
$2 \leftrightarrow 4^*$	a5 ⇌ b4 c2 ⇌ g6	2
$4 \leftrightarrow 5$	a4 ⇌ d7 c5 ⇌ e3	2

– Iteration no. 3 (after the movement $1 \leftrightarrow 3$ that becomes taboo; one conflict)

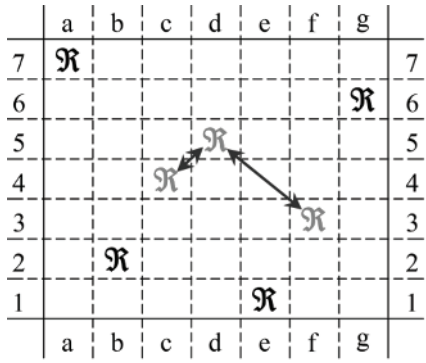


$\mathbf{M}_{3,1}$	Conflicts	Cost
$1 \leftrightarrow 3^*$	c4 ⇌ f1	1
$1 \leftrightarrow 7^*$	d1 ⇌ f3	1
$5 \leftrightarrow 7$	c4 ⇌ d5 d5 ⇌ f3	2
$6 \leftrightarrow 7$	a5 ⇌ e1 b2 ⇌ g7	2
$1 \leftrightarrow 2$	b1 ⇌ g6 c4 ⇌ e2 e2 ⇌ f3	3

– Iteration no. 4 (after the movement $5 \leftrightarrow 7$ that becomes taboo; two conflicts).

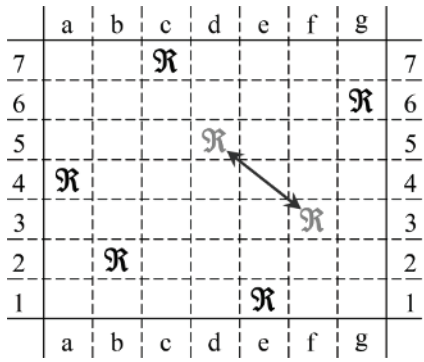
This iteration requires prior fulfillment of the taboo principle. Consequently, the number of conflicts increased with respect to the previous iteration. The aspiration principle cannot be applied here, as no forbidden movement (the first two of $\mathbf{M}_{3,1}$) is leading to the optimum solution (without conflicts). Since at the previous iteration a single conflict was obtained, the only acceptable better solution is the optimum one (with no conflicts).

The taboo list here surpasses the limit of four movements: $1 \leftrightarrow 7$, $2 \leftrightarrow 4$, $1 \leftrightarrow 3$ and $5 \leftrightarrow 7$. It follows that the oldest movement, namely $1 \leftrightarrow 7$, has to be removed from the list, while conserving the other three.



$\mathbf{M}_{4,3}$	Conflicts	Cost
$4 \leftrightarrow 7$	d5 ⇌ f3	1
$5 \leftrightarrow 7^*$	a5 ⇌ e1	1
$1 \leftrightarrow 5$	b2 ⇌ e5 d1 ⇌ f3	2
$2 \leftrightarrow 5$	b5 ⇌ c4 d2 ⇌ e1	2
$2 \leftrightarrow 4$	b4 ⇌ e1 c2 ⇌ g6 d5 ⇌ f3	3

– Iteration no. 5 (after the movement $4 \leftrightarrow 7$ that becomes taboo; one conflict)

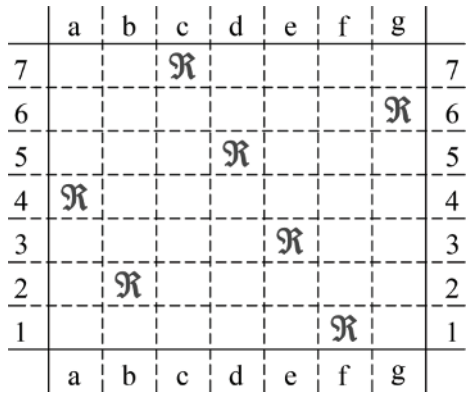


$\mathbf{M}_{5,1}$	Conflits	Coût
$1 \leftrightarrow 3^*$	–	0
$5 \leftrightarrow 6$	c7 ⇌ d6	1
$5 \leftrightarrow 7$	a4 ⇌ d7	1
$1 \leftrightarrow 6$	d5 ⇌ e6 d5 ⇌ f3	2
$1 \leftrightarrow 7$	b2 ⇌ c1 d5 ⇌ f3	2

Here, the movement $2 \leftrightarrow 4$ is removed from taboo list, being replaced by the movement $4 \leftrightarrow 7$.

By looking now at the table on the right-hand side, we can notice that the movement $1 \leftrightarrow 3$ leads to the optimum solution (no conflicts between queens), despite its taboo label. Thus, the aspiration principle can now be applied in order to complete the procedure.

– Iteration no. 6 (after the taboo movement $1 \leftrightarrow 3$; zero conflicts)



1.5. Simulated annealing

1.5.1. *Principle of thermal annealing*

The optimization technique based on simulated annealing allows escaping from local optimums. It was inspired by the technique of controlled annealing, as employed in metallurgy. When a melted metal is cooled down very fast, a metastable solid is obtained. Moreover, this solid state corresponds to a local minimum of the internal energy. On the contrary, if the metal is cooled down slowly, the resulted crystalline structure corresponds to an absolute minimum of internal energy. If the temperature decreases too fast, the crystallization is not perfect. In this case, the imperfections can be attenuated by first reheating the metal, in order to release a part of the energy, and then cooling it again slowly. The succession of heating–cooling operations establishes the annealing phenomenon.

The modern annealing technique takes into account some properties that the final metallic product should have. Very often, this product is made of steel. In this case, two properties are necessary: elasticity and stiffness. Unfortunately, the two properties are opposite to each other not only in terms of realization, but also as requirements

applied to internal chemical composition of steel. As already known, steel is obtained by mixing the iron with at least 2.11% carbon and other chemical substances from Mendeleev's Periodic table (such as Si, Mn, P, S, Cr, Ni, V, Ti and Mo). Depending on their proportions, the mixed substances can lead to a large panoply of steep properties, provided that the thermal annealing is carried out according to a carefully designed program. It is out of the scope of this chapter to elaborate the details of steel manufacturing. The only aim here is to outline a phenomenon related to annealing, which includes an intrinsic optimization mechanism, possibly to simulate on computer. If the liquid steel is cooled down too fast, then many chemical links between the molecules of its components cannot be established and internal tensions accumulate rapidly. As a result, the final product is very stiff, but breakable and inelastic. If, on the contrary, the liquid steel is cooled down too slowly, then some of the chemical components (especially the carbon) are burned, even though they already established molecular links with other substances. In this case, the final product may be elastic, but too soft, without enough stiffness. In order to ensure a good trade-off between the elasticity and stiffness of final steel-made product, it is thus necessary to design a "good" cooling down program, i.e. an optimal annealing procedure. This is an optimization problem based on two opposed criteria. Since in metallurgy such annealing programs have already been designed and implemented, there is a good chance to be able to simulate them in the framework of optimizations.

The simulated annealing algorithm is actually reproducing a simplified process. The internal energy is here the criterion to minimize, whereas the temperature is a parameter directly related to the convergence. The principle of the algorithm is as follows: as long as the temperature decreases, for each one of its values, the state of minimum internal energy is wanted. Such a state offers the minimum amount of internal tensions, which constitutes the premise for a good trade-off elasticity-stiffness in the end. Like for other metaheuristics, this is only an intuitive interpretation of a natural phenomenon. The numerical procedure of simulated annealing can be applied to various

optimization problems, regardless of their correlation with the thermal annealing from metallurgy.

1.5.2. *Kirkpatrick's model of thermal annealing*

A model of thermal annealing, largely accepted as being quite realistic, was introduced by Kirkpatrick [KIR 83].

The criterion to minimize, f, is referred to as *energy*. The energy is computed for states of a dynamical system. Starting from some optimal state $\mathbf{x}_i$ of energy $\mathbf{E}_i = f(\mathbf{x}_i)$, another neighbor state $\mathbf{x}_j$ can or cannot be selected as the next optimal state depending on its energy, $\mathbf{E}_j = f(\mathbf{x}_j)$. If the energy decreases ($\mathbf{E}_j < \mathbf{E}_i$), then the $\mathbf{x}_j$ state replaces the $\mathbf{x}_i$ state. Otherwise, the $\mathbf{x}_i$ state can be replaced by the $\mathbf{x}_j$ state, only if some requirements are met.

If the energy variation $\Delta\mathbf{E}_{j,i} = \mathbf{E}_j - \mathbf{E}_i$ is positive, denote by:

$$\mathbf{p}(\Delta\mathbf{E}_{j,i}, T) = \exp\left(-\frac{\Delta\mathbf{E}_{j,i}}{T}\right) \qquad [1.23]$$

the probability to obtain this variation at the temperature $T > 0$. When varying the normalized energy and the temperature, the probability surface of Figure 1.3 can be drawn. This image actually is the kernel of Kirkpatrick's model associated with annealing phenomenon.

The shape of this surface quite realistically describes the annealing phenomenon. Regardless of the temperature, it is very likely that small energy differences exist between neighbor states. The probability to have big energy variation between states decreases more or less rapidly (see the topology of cliff over the temperature axis). For small temperatures, the probability decreases rapidly. The bigger the temperature, the smaller the energy decreasing rate (see the top left corner of the surface). In other words, the big energy difference

between neighbor states seemingly appears at high temperature and is less probable at low temperature.

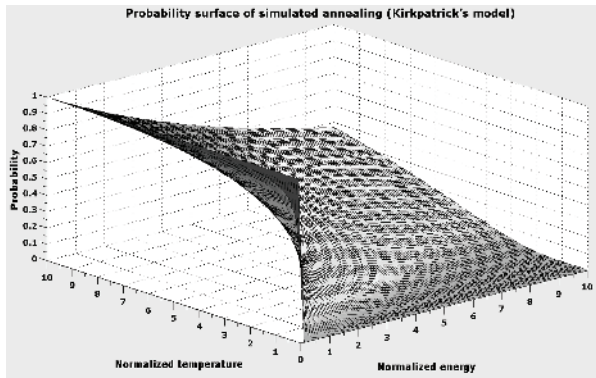


Figure 1.3. *Characteristic probability surface of annealing Kirkpatrick's model. For a color version of this figure, see www.iste.co.uk/stefanoiu/optimization.zip*

Changing the probability surface means practically changing the heart of the simulated annealing algorithm. This surface plays the same role for simulated annealing as does the cooling program for the natural annealing.

To decide whether a state $\mathbf{x}_j$ of higher energy will replace or not the current state $\mathbf{x}_i$ (i.e. in case $\Delta\mathbf{E}_{j,i} = \mathbf{E}_j - \mathbf{E}_i > 0$), the probability surface can be employed. Thus, some U-PRSG is calibrated to generate a probability threshold $p_i \in (0,1)$ in order to compare it to the computed probability of Kirkpatrick's model [1.23]. If $\mathbf{p}\left(\Delta\mathbf{E}_{j,i}, T\right) > p_i$, then $\mathbf{x}_j$ will replace $\mathbf{x}_i$. Otherwise, $\mathbf{x}_j$ cannot overthrow $\mathbf{x}_i$ and the search will focus on a new state $\mathbf{x}_j$ in the neighborhood of $\mathbf{x}_i$. From the probability theory point of view, the event "$\mathbf{x}_j$ replaces $\mathbf{x}_i$" occurs with the probability $\mathbf{p}\left(\Delta\mathbf{E}_{j,i}, T\right)$. (Since a U-PRSG was calibrated to generate numbers within the $(0,1)$ interval, the probability to obtain a number below $\mathbf{p}\left(\Delta\mathbf{E}_{j,i}, T\right)$ is just $\mathbf{p}\left(\Delta\mathbf{E}_{j,i}, T\right)$).

If no improvement is obtained after a certain number of iterations, the search is restarted in the same way, but after decreasing the temperature (T). To stop the search, we can count the number of successive temperature decreasing attempts while the optimization criterion still does not improve.

1.5.3. *Simulated annealing algorithm*

The main steps of simulated annealing procedure (based on Kirkpatrick's model) are listed in algorithm 1.6.

1) Input data:

– Search vicinity $\mathbf{V}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Energy f (criterion to minimize).

– Accuracy threshold, $\varepsilon > 0$.

– Maximum number of successive states for which the energy is not minimized anymore, M.

– Maximum number of successive temperature changing for which the current minimal state does not change, K.

– Maximum number of attempts to change the current minimal state, at constant temperature, N.

– Maximum temperature to test, T. (The successive temperatures are generated inside the interval $(\alpha T, T)$, where $\alpha \in (0,1)$ is *a priori* known).

2) Initialization.

a) Select at random (but with uniform distribution) the starting state $\mathbf{x} \in \mathbf{V}$. A U-PRSG of nx size has to be used in this aim. If the starting state does not belong to $\mathbf{V}$, then it will be generated until this property is verified.

b) Evaluate the initial state energy: $\mathrm{E} = f(\mathbf{x})$.

c) Set the first minimal solution: $\mathbf{x}^{\min} = \mathbf{x}$ and $\mathrm{E}^{\min} = \mathrm{E}$.

d) Set a counter related to the number of successive temperature changes before replacing the current state (the *temperature counter*): $k = 0$.

e) Set a counter related to the number of successive states that cannot really improve the energy beyond the accuracy threshold (the *energy counter*): $m = 0$.

f) Set a counter related to the number of attempts to change the minimal state at constant temperature (the *immobility counter*): $n = 0$.

3) Perform cooling down. For the current state $\mathbf{x}$:

3.1. Use a U-PRSG to generate an offset and a direction: $\Delta\mathbf{x}$. The generator has to operate in a hypercube including the vicinity, but as narrow as possible.

3.2. While $\mathbf{x} + \Delta\mathbf{x} \notin \mathbf{V}$, calibrate the U-PRSG to generate a new offset and direction, but inside the hypercube $[\mathbf{0}, \Delta\mathbf{x}]$, where $\Delta\mathbf{x}$ is the most recent vector offset.

3.3. Set the offset: the first generated $\Delta\mathbf{x}$, for which the vicinity limits are not violated. Thus, $\mathbf{x}^{\text{rep}} = \mathbf{x} + \Delta\mathbf{x} \in \mathbf{V}$ is the state that might replace the current state.

3.4. Evaluate the new state energy: $\mathrm{E}^{\text{rep}} = f\left(\mathbf{x}^{\text{rep}}\right)$.

3.5. Compute the energy difference with respect to the current minimal state: $\Delta\mathrm{E} = \mathrm{E}^{\text{rep}} - \mathrm{E}^{\min}$.

3.6. If $\Delta\mathrm{E} < 0$, then:

3.6.1. Replace the current minimal state and energy: $\mathbf{x}^{\min} \leftarrow \mathbf{x}^{\text{rep}}$ and $\mathrm{E}^{\min} \leftarrow \mathrm{E}^{\text{rep}}$.

3.6.2. Replace the current solution: $\mathbf{x} \leftarrow \mathbf{x}^{\text{rep}}$ and $\mathrm{E} \leftarrow \mathrm{E}^{\text{rep}}$.

3.6.3. Reset the temperature counter: $k \leftarrow 0$.

3.6.4. Reset the immobility counter: $n \leftarrow 0$.

3.6.5. If $\Delta\mathrm{E} > -\varepsilon$, then:

3.6.5.1. Increment the energy counter: $m \leftarrow m + 1$.

3.6.5.2. If $m > M$, stop the search, as no real improvement is made. Go directly to the final stage (no. 4).

3.6.6. Otherwise, reset the energy counter: $m \leftarrow 0$.

3.7. Otherwise, the energy of the possible new solution is at least equal to the energy of the current solution and then:

3.7.1. Increment the immobility counter: $n \leftarrow n + 1$.

3.7.2. If $n \le N$, then:

3.7.2.1. Compute the energy difference with respect to the current state: $\Delta \mathrm{E} = \mathrm{E}^{\mathrm{rep}} - \mathrm{E}$.

3.7.2.2. If $\Delta \mathrm{E} < 0$, replace the current solution: $\mathbf{x} \leftarrow \mathbf{x}^{\mathrm{rep}}$ and $\mathrm{E} \leftarrow \mathrm{E}^{\mathrm{rep}}$.

3.7.2.3. Otherwise:

a) Compute the Kirkpatrick probability: $\mathbf{p}(\Delta \mathbf{E}, T)$ (by using definition [1.23]).

b) Use a U-PRSG to generate a number $p \in (0,1)$.

c) If $\mathbf{p}(\Delta \mathbf{E}, T) > p$, then replace the current solution: $\mathbf{x} \leftarrow \mathbf{x}^{\mathrm{rep}}$ and $\mathrm{E} \leftarrow \mathrm{E}^{\mathrm{rep}}$. Otherwise, keep the current solution.

3.7.3. Otherwise, the temperature has to be decreased:

3.7.3.1. Use a U-PRSG to generate a new temperature, a smaller one, in the interval $(\alpha T, T)$. The new temperature replaces the current temperature, T.

3.7.3.2. Reset the immobility counter: $n \leftarrow 0$.

3.7.3.3. Increment the temperature counter: $k \leftarrow k + 1$.

3.7.3.4. If $k > K$, stop the search, as the minimal solution cannot change any more. Go directly to the final stage (no. 4).

3.8. Resume the search from step 3.1.

4) Return:

– The minimal current state: $\mathbf{x}^{\min}$.

– The current minimal energy: $\mathrm{E}^{\min} = f\left(\mathbf{x}^{\min}\right)$.

Algorithm 1.6. *Simulated annealing procedure*

This algorithm has been designed so as to avoid infinite loops. The two main counters (of temperature k and immobility n) are accountable for preventing the procedure to be captured in such a loop. Like in the case of previous algorithms (or the following ones), there are a number of configuring parameters that have to be specified before initiating the run. Usually, the user does not know how to set

such parameters. Therefore, some preliminary running tests are necessary in order to obtain a fine-tuning of the algorithm. Normally, the maximum number of states that are not changing the optimal solution at the same temperature, N, varies from 50 to 200. The maximum number of temperature change, K, generally is smaller, between 5 and 25. The maximum number of states for which the energy does not actually improve, M, is even smaller, between 1 and 10. The accuracy threshold ε depends on the criterion representation scale as usual.

The initial temperature, T, has to be sufficiently large in order to avoid capturing in a local minimum. The temperature decreasing strategy employed in algorithm 1.6 is not unique (the Monte Carlo principle actually was implemented here). Different strategies can be adopted depending on each application particularities. In any case, such a strategy should have a unique main goal: to increase the chance of global minimum to be found, in the given vicinity.

Finally, the decreasing parameter α is set between 0.8 and 0.9.

Another way to reduce the temperature is to split the $(0,T)$ interval into several smaller segments (uniform or not) and to operate with their bounds.

1.6. Tunneling

1.6.1. *Tunneling principle*

The main purpose of tunneling algorithms is to allow escaping from the local optima [LEV 85, BAR 91, HAM 05, GHE 07]. Their principle is quite simple: each time the search has reached a local optimum, a tunnel is carved, in order to find a valley in the criterion variation, which could lead to another optimum. The principle is illustrated in Figure 1.4.

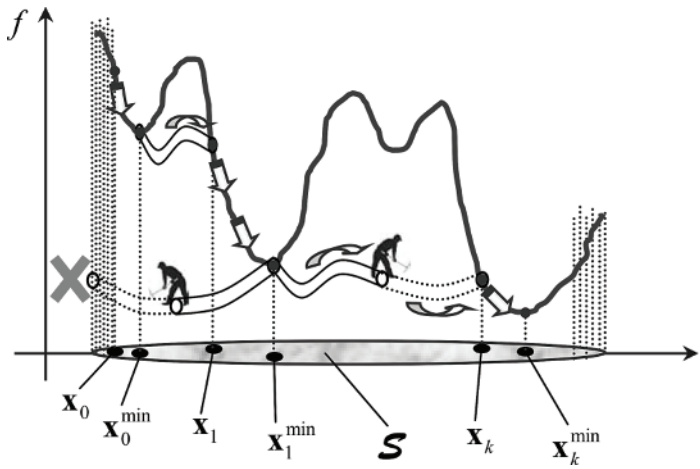


Figure 1.4. *Illustration of tunneling principle*

As usual, assume that the criterion has to be minimized. The procedure starts with a minimization phase initiated from a point $\mathbf{x}_0$. After reaching the local minimum $\mathbf{x}_0^{\min}$, a tunneling phase follows. The search space is thus pierced until another initialization is found, $\mathbf{x}_1$. The local search is thus relocated on the attraction zone of another minimum, $\mathbf{x}_1^{\min}$. This strategy is repeated until one of the stop tests is verified. If the carved tunnel touches or trespasses the search space (or some vicinity) before finding the next valley, then the search is aborted and a new tunnel is carved, following a different direction.

Thus, basically, a tunneling algorithm includes two phases:

1) local minimization on a valley (the *exploiting phase*);

2) migration toward another search zone by tunneling, in order to find another valley to exploit (the *tunneling phase*).

In the exploiting phase, a local optimization method can be invoked. Thus, starting from the initial solution $\mathbf{x}_k$ ($k \in \mathbf{N}$), a local minimum $\mathbf{x}_k^{\min}$ is found (as shown in Figure 1.4). In the tunneling

phase, actually, a new initial solution is wanted at the end of the carved tunnel.

1.6.2. *Types of tunneling*

If the tunneling directions are selected at random, with the help of some U-PRSG, the search can be time-consuming. During the search, the main issue is to avoid the already exploited zones. Therefore, two types of tunneling are considered for implementation: stochastic tunneling and tunneling with penalties.

1.6.2.1. *Stochastic tunneling*

This technique was introduced by Wenzel and Hamacher [WEN99]. The main idea is to apply a nonlinear transformation to the criterion in order to better isolate the visited valleys from the non-visited ones. For example, here is such a transformation:

$$f(\mathbf{x}) \leftarrow f_{\gamma,k}^{\text{sto}}(\mathbf{x}) = 1 - \exp\left[-\gamma\left(f(\mathbf{x}) - f\left(\mathbf{x}_k^{\min}\right)\right)\right], \quad \forall \mathbf{x} \in \mathbf{S}, \quad [1.24]$$

where:

– $\mathbf{x}_k^{\min}$ is the current minimal point, the best one found up to the kth iteration;

– $\gamma > 0$ is a parameter defining the deforming degree of search space (unit, by default).

This function conserves the original placement of the already discovered minimum points and changes each time a better solution is found. However, the criterion is scaled differently. Thus, all the points leading to values larger than the current performance $f\left(\mathbf{x}_k^{\min}\right)$ enforce the criterion to be projected into the $[0,1]$ interval. This allows the search procedure to easily detect the initializations that seemingly are not worth being considered. (Some of them may be employed though, as they could be located on deeper valleys than the

current one). On the contrary, the criterion becomes negative (varying between $(-\infty,0)$) for the points with better performance than $f\left(\mathbf{x}_k^{\min}\right)$. Thus, the zones that have not yet been explored can easily be detected. Now, the goal is to perform exploitation mainly around the points with negative values of criterion [1.24].

1.6.2.2. *Tunneling with penalties*

According to this approach, the tunneling phase benefits from some penalties applied on the performance of some points. For example, in the case of the traveling salesman problem, the distance between some cities can be artificially increased, which yields escaping from some local minimums and enlarging the search horizon. Also, penalties allow performing the search in virgin, unvisited zones.

For this type of tunneling, the penalties have to be defined according to each problem to solve. There is no general rule to set penalties and therefore only the principle can be stated in this context.

1.6.3. *Tunneling algorithm*

The procedure integrated into algorithm 1.7 is based on stochastic tunneling.

1. Input data:

– Search space $\mathbf{S}$ (equations allowing the user to decide whether a point belongs or not to this set).

– Criterion f to minimize.

– Parameter of search space deforming, $\gamma > 0$ (by default, $\gamma = 1$).

– Accuracy threshold $\varepsilon > 0$ (for local minimization).

– Maximum number of initializations to restart the local search, K.

– Maximum number of attempts to escape from the attraction of a minimal point, M.

2) Initialization.

a) Select at random (but with uniform distribution) the first departure point $\mathbf{x}_0 \in \mathbf{S}$. A U-PRSG of nx size has to be used in this aim. If the starting point does not belong to $\mathbf{S}$, then it will be generated until this property is verified.

b) Evaluate the performance of selected initial point: $f(\mathbf{x}_0)$.

c) Set a counter related to the number of tested initializations (the *initializations counter*): $k = 0$.

d) Set a counter related to the number of attempts to escape from the attraction of a minimal point (the *escape counter*): $m = 0$.

3) For each initial point, i.e. for $k = \overline{0, K-1}$:

3.1. Exploiting phase. Call a local optimization procedure (exact or metaheuristic), in order to find a local minimum $\left\{\mathbf{x}_k^{\min}, f\left(\mathbf{x}_k^{\min}\right)\right\}$, with ε accuracy, starting from the initial point $\mathbf{x}_k$.

3.2. Stochastic tunneling phase. For $m \geq 0$:

3.2.1. Select the tunnel departure: $\mathbf{x}_{k,0} = \mathbf{x}_k^{\min}$.

3.2.2. For $n \geq 0$:

3.2.2.1. Use a U=PRSG of nx size to generate a vector offset $\Delta\mathbf{x}_{k,n+1}$. The offset length has to be sufficiently small (with respect to the search space diameter).

3.2.2.2. Define the next point on the tunnel path:

$$\mathbf{x}_{k,n+1} = \mathbf{x}_{k,n} + \Delta\mathbf{x}_{k,n+1}.$$

3.2.2.3. If $\mathbf{x}_{k,n+1}$ falls outside the search space ($\mathbf{x}_{k,n+1} \notin \mathbf{S}$), then the tunnel is abandoned:

a) Increment the escape counter: $m \leftarrow m+1$.

b) If $m > M-1$, stop the search, as the search cannot escape from the attraction of the current minimal point. Go directly to the final stage (no. 4).

c) Otherwise, continue at step 3.2.1.

3.2.2.4. Otherwise, use the stochastic criterion [1.24] to test whether a new zone (not yet explored) has been discovered or not.

a) If $f_{\gamma,k}^{\text{sto}}\left(\mathbf{x}_{k,n+1}\right) < 0$, then a new local minimization has to be carried out. In this aim, set the initial point ($\mathbf{x}_{k+1} = \mathbf{x}_{k,n+1}$) for the local minimization procedure and jump to step 3.1. This is allowed only if, after incrementing the initializations counter k, the upper limit K is not violated. Otherwise, the search has to be stopped and the current optimal solution has to be returned – see the final stage, no. 4).

b) Otherwise, continue to carve the tunnel: $n \leftarrow n+1$.

4) Return:

– The current minimal point: $\mathbf{x}_k^{\min}$.

– The current performance: $f\left(\mathbf{x}_k^{\min}\right)$.

Algorithm 1.7. *Stochastic tunneling procedure*

1.7. GRASP methods

The abbreviation GRASP stands for: *greedy randomized adaptive search procedure(s)*. The concept was introduced by Feo and Resende in [FEO 95] and it actually reveals a principle that already has been suggested by the previous metaheuristic (the tunneling one). Thus, local optimization is performed, starting from a number of initial solutions, after being generated at random. The procedure is *adaptive* in the sense that the selection of new initial solutions takes into account the results of previous iterations. Although for the local optimization any algorithm can be employed, we aim to choose the most efficient procedure. Nevertheless, the overall procedure usually is greedy. In turn, there is a low risk to be captured by a local optimum, due to multiple reruns, from various initializations.

The readers may find it interesting to generalize the previous metaheuristics according to the GRASP principle. Nowadays, the modern optimization employs several techniques in an algorithm in order to solve complex problems. Exact methods [BOR 13] and metaheuristics can be combined together, if possible. Usually, in this

attempt, we start from low-complexity procedures to find the initial solutions. Then, more sophisticated procedures are employed to refine the search around each initialization until the optimal solution is found (with prescribed accuracy). During the first optimization phase, the convergence speed prevails, as the accuracy is not of real concern. We aim here to rapidly find some attraction centers around which local exploiting zones can be delimited. In the next phase, the accuracy becomes more important than the convergence speed.

Overall, applying the GRASP principle is seemingly the most successful way to deal with modern optimization problems.