International Series in
Operations Research & Management Science

Michel Gendreau · Jean-Yves Potvin
Editors

Handbook of Metaheuristics

Third Edition

Springer

Michel Gendreau • Jean-Yves Potvin
Editors

Handbook of Metaheuristics

Third Edition

Springer

Editors
Michel Gendreau
Department of Mathematics
and Industrial Engineering
Polytechnique Montréal
Montreal, QC, Canada

Jean-Yves Potvin
Département d'informatique et de
recherche opérationnelle
Université de Montréal
Montreal, QC, Canada

ISSN 0884-8289 ISSN 2214-7934 (electronic)
International Series in Operations Research & Management Science
ISBN 978-3-319-91085-7 ISBN 978-3-319-91086-4 (eBook)
https://doi.org/10.1007/978-3-319-91086-4

Library of Congress Control Number: 2018953159

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

Contributors

Ricardo Aceves
Universidad Nacional Autónoma de México, Mexico City, Mexico

Roberto Battiti
University of Trento, Trento, Italy

Christian Blum
Artificial Intelligence Research Institute, Bellaterra, Spain

Jack Brimberg
Royal Military College of Canada, Kingston, ON, Canada

Mauro Brunato
University of Trento, Trento, Italy

Edmund K. Burke
University of Leicester, Leicester, UK

Supatcha Chaimatanan
Geo-Informatics and Space Technology Development Agency, Siracha, Thailand

Maurice Clerc
Independent Consultant, Groisy, France

Carlos Cotta
Universidad de Málaga, Málaga, Spain

Teodor Gabriel Crainic
École des Sciences de la Gestion, Université du Québec à Montréal, Montréal, QC, Canada

CIRRELT, Montréal, QC, Canada

Daniel Delahaye
École Nationale de l'Aviation Civile, Toulouse, France

Marco Dorigo
Université Libre de Bruxelles, Brussels, Belgium

Abraham Duarte
Universidad Rey Juan Carlos, Madrid, Spain

Michel Gendreau
Polytechnique Montréal, Montréal, QC, Canada

CIRRELT, Montréal, QC, Canada

Bruce Golden
R.H. Smith School of Business, University of Maryland, College Park, MD, USA

Swati Gupta
Simons Institute for the Theory of Computing, University of California, Berkeley, CA, USA

Walter J. Gutjahr
University of Vienna, Vienna, Austria

Pierre Hansen
École des Hautes Études Commerciales, Montréal, QC, Canada

GERAD, Montréal, QC, Canada

Matthew R. Hyde
University of Nottingham, Nottingham, UK

Graham Kendall
University of Nottingham Malaysia Campus, Semenyih, Malaysia

Maria Teresa León
Universidad de Valencia, Valencia, Spain

Xiaodong Li
RMIT University, Melbourne, VIC, Australia

Manuel López-Ibáñez
Alliance Manchester Business School, University of Manchester, Manchester, UK

Helena Ramalhinho Lourenço
Universitat Pompeu Fabra, Barcelona, Spain

Andrea Mariello
University of Trento, Trento, Italy

Rafael Martí
Universidad de Valencia, Valencia, Spain

Olivier C. Martin
Université Paris-Sud, Orsay, France

Nenad Mladenović
Mathematical Institute, SANU, Belgrade, Serbia

Marcel Mongeau
École Nationale de l'Aviation Civile, Toulouse, France

Roberto Montemanni
Dalle Molle Institute for Artificial Intelligence, University of Applied Sciences of Southern Switzerland, Manno, Switzerland

Jose M. Moreno-Vega
Universidad de La Laguna, San Cristóbal de La Laguna, Spain

Pablo Moscato
The University of Newcastle, Newcastle, NSW, Australia

Gabriela Ochoa
University of Stirling, Stirling, UK

Ender Özcan
University of Nottingham, Nottingham, UK

José A. Moreno Pérez
Universidad de La Laguna, San Cristóbal de La Laguna, Spain

David Pisinger
Technical University of Denmark, Lyngby, Denmark

Jean-Yves Potvin
Université de Montréal, Montréal, QC, Canada

CIRRELT, Montréal, QC, Canada

Jakob Puchinger
CentraleSupélec, Gif-sur-Yvette, France

Günther R. Raidl
Institute of Logic and Computation, TU Wien, Vienna, Austria

Mauricio G. C. Resende
Amazon.com, Seattle, WA, USA

University of Washington, Seattle, WA, USA

Celso C. Ribeiro
Universidade Federal Fluminense, Niterói, Brazil

Stefan Ropke
Technical University of Denmark, Lyngby, Denmark

John Silberholz
Ross School of Business, University of Michigan, Ann Arbor, MI, USA

Thomas Stützle
Université Libre de Bruxelles, Brussels, Belgium

Xingyin Wang
Singapore University of Technology and Design, Singapore, Singapore

Darrell Whitley
Colorado State University, Fort Collins, CO, USA

John R. Woodward
Queen Mary University of London, London, UK

Chapter 1
Simulated Annealing: From Basics to Applications

Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau

Abstract Simulated Annealing (SA) is one of the simplest and best-known metaheuristic method for addressing difficult *black box* global optimization problems whose objective function is not explicitly given and can only be evaluated via some costly computer simulation. It is massively used in real-life applications. The main advantage of SA is its simplicity. SA is based on an analogy with the physical annealing of materials that avoids the drawback of the Monte-Carlo approach (which can be trapped in local minima), thanks to an efficient Metropolis acceptance criterion. When the evaluation of the objective-function results from complex simulation processes that manipulate a large-dimension state space involving much memory, population-based algorithms are not applicable and SA is the right answer to address such issues. This chapter is an introduction to the subject. It presents the principles of local search optimization algorithms, of which simulated annealing is an extension, and the Metropolis algorithm, a basic component of SA. The basic SA algorithm for optimization is described together with two theoretical properties that are fundamental to SA: statistical equilibrium (inspired from elementary statistical physics) and asymptotic convergence (based on Markov chain theory). The chapter surveys the following practical issues of interest to the user who wishes to implement the SA algorithm for its particular application: finite-time approximation of the theoretical SA, polynomial-time cooling, Markov-chain length, stopping criteria, and simulation-based evaluations. To illustrate these concepts, this chapter presents the straightforward application of SA to two classical and simple classical NP-hard combinatorial optimization problems: the knapsack problem and the

D. Delahaye (✉) · M. Mongeau
École Nationale de l'Aviation Civile, Toulouse, France
e-mail: daniel.delahaye@enac.fr; marcel.mongeau@enac.fr

S. Chaimatanan
Geo-Informatics and Space Technology Development Agency, Siracha, Thailand
e-mail: supatcha@gistda.or.th

M. Gendreau, J.-Y. Potvin (eds.), *Handbook of Metaheuristics*,
International Series in Operations Research & Management Science 272,
https://doi.org/10.1007/978-3-319-91086-4_1

traveling salesman problem. The overall SA methodology is then deployed in detail on a real-life application: a large-scale aircraft trajectory planning problem involving nearly 30,000 flights at the European continental scale. This exemplifies how to tackle nowadays complex problems using the simple scheme of SA by exploiting particular features of the problem, by integrating astute computer implementation within the algorithm, and by setting user-defined parameters empirically, inspired by the SA basic theory presented in this chapter.

1.1 Introduction

Simulated Annealing (SA) is one of the simplest and best-known metaheuristic methods for addressing difficult *black box* global optimization problems, whose objective function is not explicitly given and can only be evaluated via some costly computer simulation. It is massively used in real-life applications. The expression "simulated annealing" yields over one million hits when searching through the Google Scholar web search engine dedicated to the scholarly literature.

This chapter is an introduction to the subject. It is organized as follows. The first section introduces the reader to the basics of the simulated annealing algorithm. Section 1.2 deals with two theoretical properties of SA: statistical equilibrium and asymptotic convergence. Practical issues of interest when implementing SA are discussed in Sect. 1.3: finite-time approximation, polynomial-time cooling, Markov-chain length, stopping criteria and simulation-based evaluations. Section 1.4 illustrates the application of SA to two classical NP-hard combinatorial optimization problems: the knapsack problem and the traveling salesman problem. A real-life application, large-scale aircraft trajectory planning problem, is finally tackled in Sect. 1.5 in order to illustrate how the particular knowledge of an application and astute computer implementation must be integrated within SA in order to tackle nowadays complex problems using the simple scheme of SA.

1.2 Basics

In the early 1980s three IBM researchers, Kirkpatrick et al. [11], introduced the concepts of annealing in combinatorial optimization. These concepts are based on a strong analogy with the physical annealing of materials. This process involves bringing a solid to a low energy state after raising its temperature. It can be summarized by the following two steps (see Fig. 1.1):

- Bring the solid to a very high temperature until "melting" of the structure;
- Cool the solid according to a very particular temperature decreasing scheme in order to reach a solid state of minimum energy.

In the liquid phase, the particles are distributed randomly. It is shown that the minimum-energy state is reached provided that the initial temperature is sufficiently high and the cooling time is sufficiently long. If this is not the case, the solid will be found in a metastable state with non-minimal energy; this is referred to as *hardening*, which consists in the sudden cooling of a solid.

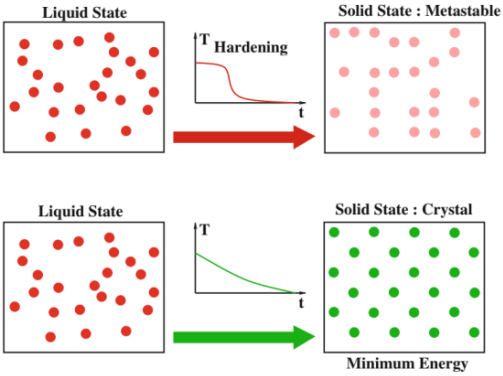

Fig. 1.1 When the temperature is high, the material is in a liquid state (left). For a hardening process, the material reaches a solid state with non-minimal energy (metastable state; top right). In this case, the structure of the atoms has no symmetry. During a slow annealing process, the material reaches also a solid state but for which atoms are organized with symmetry (crystal; bottom right)

Before describing the simulated annealing algorithm for optimization, we need to introduce the principles of local search optimization algorithms, of which simulated annealing is an extension.

1.2.1 Local Search (or Monte Carlo) Algorithms

These algorithms optimize the cost function by exploring the neighborhood of the current point in the solution space.

In the next definitions, we consider (S, f) an instantiation of a combinatorial optimization problem (S: set of feasible solutions, f: objective function to be minimized).

Definition 1 *Let $\mathcal{N}$ be an application that defines for each solution $i \in S$ a subset $S_i \subset S$ of solutions "close" (to be defined by the user according to the problem of interest) to the solution i. The subset S_i is called the neighborhood of solution i.*

In the next definitions, we consider that $\mathcal{N}$ is a neighborhood structure associated with (S, f).

Definition 2 *A generating mechanism is a mean for selecting a solution j in any neighborhood S_i of a given solution i.*

A local search algorithm is an iterative algorithm that begins its search from a feasible point, randomly drawn in the state space. A generation mechanism is then successively applied in order to find a better solution (in terms of the objective function value), by exploring the neighborhood of the current solution. If such a solution is found, it becomes the current solution. The algorithm ends when no improvement can be found, and the current solution is considered as the approximate solution of the optimization problem. One can summarize the algorithm by the following pseudo-code for a minimization problem:

Local Search

1. **Draw an initial solution** i**;**
2. **Generate a solution** j **from the neighborhood** S_i **of the current solution** i**;**
3. **If** $f(j) < f(i)$ **then** j **becomes the current solution;**
4. **If** $f(j) \geq f(i)$ **for all** $j \in S_i$ **then END;**
5. **Go to step 2;**

Definition 3 *A solution* $i^* \in S$ *is called a local optimum with respect to* $\mathcal{N}$ *for* (S, f) *if* $f(i^*) \leq f(j)$ *for all* $j \in S_{i^*}$.

Definition 4 *The neighborhood structure* $\mathcal{N}$ *is said to be exact if, for every local optimum with respect to* $\mathcal{N}$, $i^* \in S$, i^* *is also a global optimum of* (S, f).

Thus, by definition, local search algorithms converge to local optima unless one has an exact neighborhood structure. This notion of exact neighborhood is theoretical because it generally leads, in practice, to resort to a complete enumeration of the search space.

Intuitively, if the current solution "falls" in a subdomain over which the objective function is convex, the algorithm remains trapped in this subdomain, unless the neighborhood structure associated with the generation mechanism can reach points outside this subdomain.

In order to avoid being trapped in local minima, it is then necessary to define a process likely to accept current state transitions that momentarily reduce the performance (in terms of objective) of the current solution: this is the main principle of simulated annealing.

Before describing this algorithm, it is necessary to introduce the Metropolis algorithm [15] which is a basic component of SA.

1.2.2 Metropolis Algorithm

In 1953, three American researchers [15] developed an algorithm to simulate the physical annealing process, as described in Sect. 1.2. Their aim was to reproduce faithfully the evolution of the physical structure of a material undergoing annealing.

This algorithm is based on Monte Carlo techniques which consist in generating a sequence of states of the solid in the following way.

Starting from an initial state i of energy E_i, a new state j of energy E_j is generated by modifying the position of one particle.

If the energy difference, $E_i - E_j$, is positive (the new state features lower energy), the state j becomes the new current state. If the energy difference is less than or equal to zero, then the probability that the state j becomes the current state is given by:

$$Pr\{\text{Current state} = j\} = e^{\left(\frac{E_i - E_j}{k_b \cdot T}\right)},$$

where T represents the temperature of the solid and k_B is the Boltzmann constant ($k_B = 1.38 \times 10^{-23}$ J/K).

The acceptance criterion of the new state is called the *Metropolis criterion*. If the cooling is carried out sufficiently slowly, the solid reaches a state of equilibrium at each given temperature T. In the Metropolis algorithm, this equilibrium is achieved by generating a large number of transitions at each temperature. The thermal equilibrium is characterized by the *Boltzmann statistical distribution*. This distribution gives the probability that the solid is in the state i of energy E_i at the temperature T:

$$Pr\{X = i\} = \frac{1}{Z(T)} e^{-\left(\frac{E_i}{k_b T}\right)},$$

where X is a random variable associated with the current state of the solid and $Z(T)$ is a normalization coefficient, defined as:

$$Z(T) = \sum_{j \in S} e^{-\left(\frac{E_j}{k_b T}\right)}.$$

1.2.3 Simulated Annealing (SA) Algorithm

In the SA algorithm, the Metropolis algorithm is applied to generate a sequence of solutions in the state space S. To do this, an analogy is made between a multi-particle system and our optimization problem by using the following equivalences:

- The state-space points (solutions) represent the possible states of the solid;
- The function to be minimized represents the energy of the solid.

A control parameter c, acting as a temperature, is then introduced. This parameter is expressed with the same units as the objective that is optimized.

It is also assumed that the user provides for each point of the state space, a neighborhood and a mechanism for generating a solution in this neighborhood. We then define the acceptance principle:

Definition 5 *Let (S, f) be an instantiation of a combinatorial minimization problem, and i, j two points of the state space. The* acceptance criterion *for accepting solution j from the current solution i is given by the following probability*:

$$Pr\{\ accept\ j\} = \begin{cases} 1 & if\ f(j) < f(i) \\ e^{\left(\frac{f(i)-f(j)}{c}\right)} & else. \end{cases}$$

By analogy, the principle of generation of a neighbor corresponds to the perturbation mechanism of the Metropolis algorithm, and the principle of acceptance represents the Metropolis criterion.

Definition 6 *A* transition *represents the replacement of the current solution by a neighboring solution. This operation is carried out in two stages: generation and acceptance.*

In the sequel, let c_k be the value of the temperature parameter, and L_k be the number of transitions generated at some iteration k. The principle of SA can be summarized as follows:

Simulated Annealing

1. **Initialization** ($i := i_{start}$, $k := 0$, $c_k = c_0$, $L_k := L_0$);
2. **Repeat**
3. **For** $l = 0$ **to** L_k **do**
   - **Generate a solution** j **from the neighborhood** S_i **of the current solution** i;
   - **If** $f(j) < f(i)$ **then** $i := j$ (j **becomes the current solution**);
   - **Else,** j **becomes the current solution with probability** $e^{\left(\frac{f(i)-f(j)}{c_k}\right)}$;
4. $k := k + 1$;
5. **Compute**(L_k, c_k);
6. **Until** $c_k \simeq 0$.

One of the main features of simulated annealing is its ability to accept transitions that degrade the objective function.

At the beginning of the process, the value of the temperature c_k is high, which makes it possible to accept transitions with high objective degradation, and thereby to explore the state space thoroughly. As c_k decreases, only the transitions improving the objective, or with a low objective deterioration, are accepted. Finally, when c_k tends to zero, no deterioration of the objective is accepted, and the SA algorithm behaves like a Monte Carlo algorithm.

1.3 Theory

This section addresses two theoretical properties that are fundamental to SA: statistical equilibrium and asymptotic convergence. More details and proofs of the theorems cited in this section can be found in the books [1, 13].

1.3.1 Statistical Equilibrium

Based on the *ergodicity hypothesis* that a particle system can be considered as a set having observable statistical properties, a number of useful quantities can be deduced from the equilibrium statistical system: mean energy, energy distribution, entropy. Moreover, if this particle set is stationary, which is the case when the statistical equilibrium is reached, the probability density associated with the states in the equilibrium phase depends on the energy of the system. Indeed, in the equilibrium phase, the probability that the system is in a given state i with an energy E_i is given by the Boltzmann law:

Theorem 1 *After a sufficient number of transitions with a fixed control parameter c and using the following probability of acceptance:*

$$P_c\{accept\ j|S_i\} = \begin{cases} 1 & if\ f(j) < f(i) \\ e^{\left(\frac{f(i)-f(j)}{c}\right)} & else, \end{cases}$$

the simulated annealing algorithm will find a given solution $i \in S$ with the probability:

$$P_c\{X = i\} = q_i(c) = \frac{1}{N_0(c)} e^{\left(-\frac{f(i)}{c}\right)},$$

where X is the random variable representing the current state of the annealing algorithm, and $N_0(c)$ is the *normalization coefficient*:

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

Definition 7 *Let A and B be two sets such that $B \subset A$. We define the characteristic function of B, noted $\kappa_{(B)}$, to be the function such that:*

$$\kappa_{(B)}(a) = \begin{cases} 1 & if\ a \in B \\ 0 & else. \end{cases}$$

Corollary 1 *For any given solution i, we have:*

$$\lim_{c \to 0^+} P_c\{X = i\} = \lim_{c \to 0^+} q_i(c) = q_i^* = \frac{1}{|S_{opt}|} \kappa_{(S_{opt})}(i),$$

where S_{opt} represents the set of global optima.

This result guarantees the asymptotic convergence of the simulated annealing algorithm towards an element of the set of global optima, provided that the stationary distribution $q_i(c)$, $i \in S$, is reached at each value of c. For a discrete state space, such distributions are discrete and one can compute the probability to reach one particular point x_i in the state space with an objective value y_i:

$$q_i(c) = \frac{e^{\left(-\frac{y_i^c}{c}\right)}}{\sum_{j \in S} e^{\left(-\frac{y_j^c}{c}\right)}}.$$

The expected value of the function f to optimize at equilibrium for any positive value of c is denoted $\langle f \rangle_c$ and the variance is denoted $\langle f^2 \rangle_c$.

At a very high temperature c, the SA algorithm moves randomly in the state space. With each point x_i generated by this process, is associated an objective value y_i by the mapping $y_i^c = f(x_i)$. If we consider this process for a long period, it is possible to build the distribution of the objective function values $y_i^c, (i = 1, 2, \ldots, N)$ generated by the SA process. This distribution depends on the temperature c and will be denoted $q(c)$. For large values of c, this distribution is equal to the objective distribution. Figure 1.2 gives an example of such a distribution. The figure shows a one-dimensional objective function for which the circles represent the samples of the SA algorithm at some high temperature c_1. The dashed horizontal line shows the mean of this distribution ($< f(c_1) >$), and on the left-hand side the associated distribution is represented by the dashed graph ($q(c_1)$). For a lower temperature c_2, some transitions in the SA process are not accepted, meaning that the associated distribution $q(c_2)$ is shifted to the lower levels (squares in the objective function on the right and solid graph on the left) with a lower expected value.

Definition 8 *The entropy at equilibrium is*

$$H_c = \sum_{i \in S} q_i(c) ln(q_i(c)).$$

Corollary 2 *One has:*

$$\frac{\partial \langle f \rangle_c}{\partial c} = \frac{\sigma_c^2}{c^2}$$
$$\frac{\partial H_c}{\partial c} = \frac{\sigma_c^2}{c^3}.$$

These last two expressions play an important role in statistical mechanics. We also deduce the following expressions:

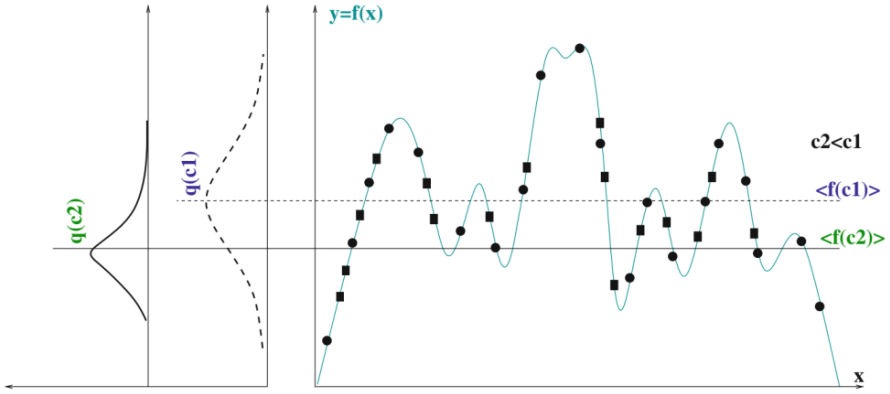

Fig. 1.2 Distribution of the objective function values at some high temperature c_1 and at a lower temperature c_2

Corollary 3

$$\lim_{c\to\infty}\langle f\rangle_c = \langle f\rangle_\infty = \frac{1}{|S|}\sum_{i\in S} f(i) \qquad \lim_{c\to 0}\langle f\rangle_c = \langle f\rangle_0 = f_{Opt},$$

$$\lim_{c\to\infty}\sigma_c^2 = \sigma_\infty^2 = \frac{1}{|S|}\sum_{i\in S}(f(i)-\langle f\rangle_\infty)^2 \qquad \lim_{c\to 0}\sigma_c^2 = \sigma_0^2 = 0,$$

$$\lim_{c\to\infty} H_c = H_\infty = ln(|S|) \qquad \lim_{c\to 0} H_c = H_0 = ln(|S_{Opt}|),$$

where f_{Opt} denotes the optimal value of f. This last formula represents the third law in thermodynamics (assuming that there is only one state of minimum energy, we then obtain: $S_0 = ln(1) = 0$).

In physics, the entropy measures the level of disorder associated with the system: a high entropy value indicates a chaotic structure, while a low value reflects organization.

In the context of optimization, the entropy is related to a measure of the degree of optimality achieved. During the successive SA iterations, the mathematical expectation of the objective function value and of the entropy only decrease and converge respectively towards f_{Opt} and $ln(|S_{Opt}|)$.

The derivative of the distribution $q_i(c)$ with the temperature c is given by the following expression:

$$\frac{\partial q_i(c)}{\partial c} = \frac{q_i(c)}{c^2}\left[\langle f\rangle_c - f(i)\right].$$

Since $\langle f\rangle_c \leq \langle f\rangle_\infty$, one can exhibit three regimes in the simulated annealing process. More precisely, one can show the following:

Corollary 4 *Let (S,f) be an instantiation of a combinatorial optimization problem with $S_{Opt} \neq S$, and let $q_i(c)$ be the stationary distribution associated with the annealing process. We then have:*

$$(i)\ \forall i \in S_{Opt}\ \frac{\partial q_i(c)}{\partial c} < 0;$$

$$(ii)\ \forall i \notin S_{Opt}\ \textit{such that}\ f(i) \geq \langle f \rangle_\infty : \frac{\partial q_i(c)}{\partial c} > 0;$$

$$(iii)\ \forall i \notin S_{Opt}\ \textit{such that}\ f(i) < \langle f \rangle_\infty, \exists \tilde{c}_i > 0\ \textit{satisfying:}$$

$$\begin{cases} \frac{\partial q_i(c)}{\partial c} > 0\ \textit{if}\ c < \tilde{c}_i \\ \frac{\partial q_i(c)}{\partial c} = 0\ \textit{if}\ c = \tilde{c}_i \\ \frac{\partial q_i(c)}{\partial c} < 0\ \textit{if}\ c > \tilde{c}_i. \end{cases}$$

This corollary indicates that the probability of finding an optimal solution increases monotonically when c decreases. Moreover, for any non-optimal solution, there exists a positive value $\tilde{c}_i$ such that for $c < \tilde{c}_i$, the probability of finding this solution decreases as c decreases.

Definition 9 *The* acceptance rate *associated with the simulated annealing algorithm is defined by:*

$$\chi(c) = \frac{\textit{Number of accepted transitions}}{\textit{Number of proposed transitions}}.$$

As a general rule, when c has a high value, all transitions are accepted and $\chi(c)$ is close to 1. Then, when c decreases, $\chi(c)$ decreases slowly until reaching 0, indicating that no transitions are accepted.

By observing the evolution of $\langle f \rangle_c$ and σ_c^2 as a function of c, we note that there exists a critical value called the transition threshold (denoted c_t), that delimits two distinct regions of the distribution at equilibrium. This threshold is the value c_t such that

$$\langle f \rangle_{c_t} \approx \frac{1}{2}\left(< f_\infty > + f_{Opt}\right),$$

and

$$\begin{aligned} \sigma_c^2 &\approx \sigma_\infty^2 \quad \text{if } c \geq c_t, \\ &< \sigma_\infty^2 \quad \text{if } c < c_t. \end{aligned}$$

For any given value of c, the search space S can therefore be partitioned into two regions:

1. Region R_1: where σ_c^2 remains roughly constant (close to σ_∞^2) when c decreases.
2. Region R_2: where σ_c^2 decreases when c decreases.

When c approaches the value of c_t, the acceptance rate is about 0.5 (i.e., $\chi(c_t) \approx 0.5$). Furthermore, one can show:

- In R_1, for large values of c, $\langle f \rangle_c$ is linear in c^{-1}, and σ_c^2 is roughly constant.
- In R_2, for small values of c, $\langle f \rangle_c$ is proportional to c, and σ_c^2 is proportional to c^2.

One can then propose the following approximation models for $\langle f\rangle_c$ and σ_c^2 :

$$\begin{cases} \langle f\rangle_c \cong f_< = f_{Opt} + N_t\left(\langle f\rangle_\infty - f_{Opt} - \frac{\sigma_\infty^2}{c}\right)\frac{c}{1-\gamma c} \text{ if } c \le c_t \\ \\ \langle f\rangle_c \cong f_> = \langle f\rangle_\infty - \frac{\sigma_\infty^2}{c} \text{ if } c > c_t \end{cases}$$

$$\begin{cases} \sigma_c^2 = \sigma_<^2 = N_t^2\sigma_\infty^2\left(\frac{c}{1-\gamma c}\right) \text{ if } c \le c_t \\ \\ \sigma_c^2 = \sigma_>^2 = \sigma_\infty^2 \text{ if } c > c_t \\ \\ \text{with} \\ \\ c_t = \frac{2\sigma_\infty^2}{\langle f\rangle_\infty - f_{Opt}} \text{ and } N_t = \frac{1-\gamma c_t}{c_t}, \end{cases}$$

where, roughly speaking, γ is the first-order approximation of $\langle f\rangle_c$. Finally, let us introduce the *specific heat*, noted $H(c)$ which is given by the following formula:

$$H(c) = \frac{d\langle f\rangle_c}{dc} = \frac{\langle f\rangle_c^2 - \langle f\rangle_c^2}{k_b c^2}$$

A large value of $H(c)$ indicates that the material starts to become solid: in this case, the decreasing rate of the temperature has to be reduced.

1.3.2 Asymptotic Convergence

The simulated annealing algorithm possesses the property of *stochastic convergence* towards a global optimum as long as it provides an infinitely-long temperature decay diagram with infinitely-small decay steps. This decay scheme is purely theoretical and one will try in practice to get closer to this ideal while remaining within reasonable times of execution.

Definition 10 *A* Markov chain *is a sequence of states, where the probability of reaching a given state depends only on the previous state. Let $X(k)$ be the state reached at the kth iteration. Then, the probability of transition at the kth iteration for each state pair i, j is given by $P_{ij}(k) = Pr\{X(k) = j | X(k-1) = i\}$. The associated matrix $[P_{ij}(k)]$ is called the transition matrix.*

In the simulated annealing context, a Markov-chain transition corresponds to a move in the state space (generation plus acceptance).

Definition 11 *The* transition probabilities *of the SA algorithm are given by:*

$$\forall i,j \in S \; P_{ij}(k) = P_{ij}(c_k) = \begin{cases} G_{ij}(c_k)A_{ij}(c_k) & \text{if } i \neq j \\ 1 - \sum_{l \neq i} P_{il}(c_k) & \text{if } i = j, \end{cases} \tag{1.1}$$

where $G_{ij}(c_k)$ denotes the probability of generating state j from state i; and $A_{ij}(c_k)$ is the probability of accepting the state j generated from the state i. For all $i,j \in S$, $A_{ij}(c_k)$ is given by:

$$A_{ij}(c_k) = e^{\left(-\frac{(f(j)-f(i))^+}{c_k}\right)}$$

$$\text{with } a^+ = \begin{cases} a \text{ if } a > 0 \\ 0 \text{ else .} \end{cases}$$

Theorem 2 *Let the transition probability associated with the SA algorithm be defined by (1). Suppose that the following condition is satisfied:*

$$\forall i,j \in S \; \exists p \geq 1, \exists l_0, l_1, \ldots, l_p \in S,$$

with $l_0 = i, l_p = j$, *and* $G_{l_k,l_{k+1}} > 0, k = 0,1,\ldots,p-1$.

Then, the Markov chain has a stationary distribution, denoted $q(c)$ *which is the distribution of the solutions visited by the SA algorithm at temperature c, whose components are given by:*

$$q_i(c) = \frac{1}{N_0(c)} e^{\left(-\frac{f(i)}{c}\right)}, \; \forall i \in S$$

where $N_0(c)$ *is the normalization coefficient.*

Furthermore,

$$\lim_{c \to 0} q(c) = q^*,$$

with $q^* = \frac{1}{|S_{opt}|} \kappa_{(S_{opt})}(i)$, $i \in S$.

Finally,

$$\lim_{c \to 0} \lim_{k \to \infty} Pr\{X_k^c = i\} q(c) = q^*,$$

and

$$\lim_{c \to 0} \lim_{k \to \infty} Pr\{X_k^c \in S_{opt}\} = 1,$$

where X_k^c denotes the kth iterate obtained at temperature c. This result indicates the convergence of the simulated annealing algorithm to one of the optimal solutions.

Generalization:

Theorem 3 *Assume that the probabilities of generation and acceptance satisfy the following assumptions:*

$$(G_1)\ \forall c_k > 0,\ \forall i,j \in S\ \exists p \geq 1,\ \exists\ l_0,l_1,\ldots,l_p \in S\ :\\ l_0 = i\ l_p = j \text{ and } G_{l_k l_{k+1}}(c_k) > 0\ k = 0,1,..,p-1;$$

$$(G_2)\ \forall c_k > 0,\ \forall i,j\ \in S\ :\ G_{ij}(c_k) = G_{ji}(c_k);$$

$$(A_1)\ \forall c_k > 0,\ \forall i,j,k\ \in S\ : \begin{cases} A_{ij}(c_k) = 1,\ \text{if } f(i) \geq f(j) \\ A_{ij}(c_k) \in]0,1[,\ \text{if } f(i) < f(j) \end{cases}$$

$$(A_2)\ \forall c_k > 0,\ \forall i,j,k\ \in S \text{ with } f(i) \leq f(j) \leq f(k),\ A_{ik}(c_k) = A_{ij}(c_k)A_{jk}(c_k)$$

$$(A3)\ \forall i,j\ \in S \text{ with } f(i) < f(j), \lim_{c_k \to 0^+} A_{ij}(c_k) = 0$$

Then, at any iteration k there exists a stationary distribution $q(c_k)$ whose components are given by:

$$q_i(c_k) = \frac{A_{i_{Opt}i}(c_k)}{\sum_{j \in S} A_{i_{Opt}j}(c_k)}\ \forall\ i \in S \text{ and } i_{Opt} \in S_{Opt}.$$

Moreover, for any $i_{Opt} \in S_{Opt}$, we have:

$$\lim_{c_k \to 0^+} q_i(c_k) = \frac{1}{|S_{Opt}|} \kappa_{(S_{Opt})}(i).$$

In practice, it is very hard to find acceptance distributions, other than exponential distributions, that satisfy A_1, A_2, A_3.

The theoretical results presented above are not directly applicable to a practical SA algorithm since they assume an infinite number of iterations for each value of c_k, which moreover decreases continuously towards zero.

In the case where the number of iterations at each temperature step is finite, the simulated annealing can be modeled using a Markovian inhomogeneous model for which similar results can be established.

The simulated annealing algorithm converges towards an optimal solution of the optimization problem but it reaches this optimum only for an infinite number of transitions. The approximation of the asymptotic behavior requires a number of iterations whose order of magnitude is equal to the cardinality of the state space, which is unrealistic in the context of NP-hard problems. It is therefore necessary to see the annealing as a mechanism for approaching the global solution of a combinatorial optimization problem, to which it will be necessary to add a local search method allowing an optimum to be reached exactly. In other words, the simulated annealing makes it possible to move in the right attraction basin, and a local method com-

pletes the optimization process by determining a local optimum within this basin of attraction corresponding to a global optimum of the problem.

1.4 Practical Issues

This section surveys the following practical issues of interest to the user who wishes to implement the SA algorithm for its particular application: finite-time approximation, polynomial-time cooling, Markov-chain length, stopping criteria, and simulatation-based evaluations.

1.4.1 Finite-Time Approximation

In practice, the convergence conditions will be approximated by choosing, at every iteration k, relatively small steps of decay of the parameter c_k and a sufficiently large number, L_k, of transitions at this temperature. Intuitively, the greater the decrement, the greater the length of the stabilization steps to achieve a *quasi-equilibrium* (defined below). There is therefore a trade-off to find between "large decrement" and "length" L_k.

A finite-time implementation of a simulated annealing algorithm can be achieved by generating homogeneous Markov chains of finite length for a finite decreasing sequence of values of the control parameter c.

Definition 12 *A cooling process is defined by:*

1. *A finite sequence of values of the control parameter c, that is to say:*
   - *An initial value c_0;*
   - *A decay function of parameter c;*
   - *A final value for c.*
2. *A finite number of transitions for each value of the control parameter, i.e. a finite length of the associated Markov chain.*

Definition 13 *Let ε be a sufficiently small positive value, k a given iteration number, L_k the length of the kth Markov chain and c_k the value of the control parameter. We say that we have a quasi-equilibrium if the probability distribution of the solutions after L_k iterations of the Markov chain (distribution denoted by $a(L_k, c_k)$) is sufficiently close to the stationary distribution $q(c_k)$:*

$$q_i(c_k) = \frac{1}{N_0(c_k)} e^{-\frac{f(i)}{c_k}} \ \forall\, i \in S,$$

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

That is:

$$||a(L_k, c_k) - q(c_k)|| < \varepsilon.$$

The cooling process using the quasi-equilibrium principle is based on the following observation. When the parameter c_k tends to ∞, the stationary distribution is given by a uniform law on the set of possible solutions S:

$$\lim_{c_k \to \infty} q(c_k) = \frac{1}{|S|}\mathbf{1},$$

where $\mathbf{1}$ is the vector of dimension $|S|$ whose components are all one.

Thus, for c_k sufficiently large, each point of the search space is visited with the same probability and a state of quasi-equilibrium is directly reached whatever the value of L_k. Then, the cooling process consists in determining the value (L_k, c_k) that will lead to a quasi-equilibrium at the end of each Markov chain.

There are many possible cooling processes but the two most common ones are the *geometric process* proposed by Kirkpatrick [11, 12] and the *polynomial-time cooling* proposed by Aarts and Van Laarhoven [2, 3].

1.4.2 Geometric Cooling

- **Initial temperature** c_0: A prior heating is performed so that we can find a value of c_0 large enough so that nearly all transitions are accepted at the first iterations. In order to find such a value, one starts with a small value c_0. Then, this value is progressively multiplied by a number greater than 1 until the acceptance rate $\chi(c_0)$ is close to 1.
- **Decay of the control parameter**: $c_{k+1} := \alpha c_k$ where typically $0.8 < \alpha < 0.99$.
- **Stopping criterion**: One decides that the algorithm is terminated when the current solution does not change any longer from one iteration to the next during a sufficiently large number of iterations.
- **Length of the chain**: In theory, it is necessary to allow each chain to reach a state of quasi-equilibrium. To this end, a sufficient number of acceptable transitions must be performed, which generally depends on the problem. Since the number of accepted transitions decrease over time with respect to the number of proposed transitions L_k, the latter must be lower bounded.

1.4.3 Cooling in Polynomial Time

Let us explain how the initial value of the temperature parameter can be set and how it should then be iteratively decreased.

1.4.3.1 Initial Temperature c_0

Let m_1 be the total number of transitions proposed that improves strictly the value of objective function, and let m_2 be the number of other (increasing) proposed transitions. Moreover, let $\bar{\Delta_f}^{(+)}$ be the average of the cost differences over all the increasing transitions. Then, the acceptance rate can be approximated by:

$$\chi(c) \simeq \frac{m_1 + m_2 e^{-\left(\frac{\bar{\Delta_f}^{(+)}}{c}\right)}}{m_1 + m_2},$$

which yields

$$c \simeq \frac{\bar{\Delta_f}^{(+)}}{ln\left(\frac{m_2}{m_2.\chi(c) - m_1.(1-\chi(c))}\right)}. \tag{1.2}$$

The proposed initial value of c_0 is then defined as follows:

Initially c_0 is set to zero. Thereafter, a sequence of m_0 transitions is generated for which the values of m_1 and m_2 are computed. The initial value of c_0 is then calculated from Eq. (1.2), where the value of the acceptance rate, $\chi(c)$, is defined by the user. The final value of c_0 is then taken as the initial value in the cooling process.

1.4.3.2 Decay of the Control Parameter

The quasi-equilibrium condition is replaced by:

$$\forall k \geq 0 \; ||q(k) - q(k+1)|| < \varepsilon,$$

Thus, for two successive values c_k and c_{k+1} of the control parameter, it is desired for the stationary distributions to be close. This can be quantified by the following formula:

$$\forall i \in S \; \frac{1}{1+\delta} < \frac{q_i(c_k)}{q_i(c_{k+1})} < 1 + \delta, \tag{1.3}$$

where δ is some small positive number *a priori* given. The following theorem provides a necessary condition for satisfying Eq. (1.3).

Theorem 4 *Let $q(c_k)$ be the stationary distribution of the Markov chain associated with the simulated annealing process at iteration k, and let c_k and c_{k+1} be two successive values of the control parameter with $c_{k+1} < c_k$, then (1.3) is satisfied if:*

$$\forall i \in S \quad e^{\Delta_i\left(\frac{1}{c_{k+1}} - \frac{1}{c_k}\right)} < 1 + \delta, \tag{1.4}$$

where $\Delta_i = f(i) - f_{Opt}$.

The necessary condition (1.4) can be rewritten as:

$$\forall i \in S \quad c_{k+1} > \frac{c_k}{1 + \frac{c_k . ln(1+\delta)}{f(i) - f_{Opt}}} . \tag{1.5}$$

One can show that the latter condition (1.5) can be approximated by:

$$\forall i \in S \quad c_{k+1} > \frac{c_k}{1 + \frac{c_k ln(1+\delta)}{3\sigma_{c_k}}} , \tag{1.6}$$

where σ_{c_k} is the standard deviation of $q(c_k)$ at temperature c_k.

The decrement of the temperature parameter c is then determined by the user-defined parameter δ. A large value of δ induces large decrements of c, and small value of δ produces small decrements.

1.4.3.3 Length of Markov Chains

In the SA cooling process, the length of the Markov chains must allow a significant percentage of the neighborhood S_i of a given solution $i \in S$ to be visited. The following theorem is used to quantify this percentage:

Theorem 5 *Let S be a set of cardinality $|S|$. Then, the average number of elements of S visited during a random walk with N iterations is given by:*

$$|S| . \left[1 - e^{-\frac{N}{|S|}} \right]$$

for large N and large $|S|$.

Thus, if no transition is accepted and if $N = |S_i|$, the percentage of solutions visited in the neighborhood S_i of a solution i is : $1 - e^{-1} \simeq 2/3$.

A good choice for the number of iterations of the inner loop (at temperature c_k) at iteration k is given by $L_k = |S_i|$ where, obviously, $|S_i|$ is problem dependent and has to be designed by the user.

1.4.3.4 Stopping Criterion

Let $\Delta \langle f \rangle_{c_k} = \langle f \rangle_{c_k} - f_{Opt}$. Then, the execution of the algorithm should terminate when $\Delta \langle f \rangle_{c_k}$ is "sufficiently" small with respect to $\langle f \rangle_{c_0}$. For sufficiently high values of c_0, we have $< f_{c_0} > \simeq \langle f \rangle_\infty$

Moreover, for $c_k << 1$:

$$\Delta \langle f \rangle_{c_k} \simeq c_k \frac{\partial \langle f \rangle_{c_k}}{\partial c_k} .$$

The end of the algorithm is then fixed by the following condition:

$$\frac{c_k}{\langle f \rangle_\infty} \frac{\partial \langle f \rangle_{c_k}}{\partial c_k} < \varepsilon_s \text{ for } c_k << 1$$

with some small tolerance ε_s to be set by the user.

1.4.3.5 Summary

The cooling process in polynomial time is thus parameterized by:

- The initial rate of acceptance: $\chi(c_0)$
- The distance between successive stationary distributions controlled by the parameter δ
- The stopping criterion, controlled by the parameter ε_s

The number of iterations of this cooling process is bounded and can be characterized by the following theorem:

Theorem 6 *Let the decrement function be given by:*

$$c_{k+1} = \frac{c_k}{1 + \alpha_k c_k},$$

where

$$\alpha_k = \frac{ln(1+\delta)}{3\sigma_{c_k}},$$

and let K be the first integer for which the stopping criterion is satisfied. Then, we have $K = O(ln(|S|))$.

Consequently, if $ln(|S|)$ is polynomial on the size of the problem (which is the case for many combinatorial optimization problems), then this type of cooling induces a polynomial execution of the algorithm.

There is an optimal annealing scheme for each problem and it is up to the user to define which one is the most suitable for his application. When one has no prior information about the optimal annealing scheme, which is generally the case, one should rely on a standard geometrical scheme for which the parameter c_k evolves as follows: $c_{k+1} := \alpha_k c_k$, and tune empirically the parameters α_k and L_k on some representative instances of the class of problem of interest.

This geometric approach is not optimal for all problems but has the advantage of being robust and ensures convergence towards an approximate solution, even though it requires more time to converge than it would do with an optimal annealing scheme.

1.4.4 Simulation-Based Evaluation

In many optimization applications, the objective function is evaluated thanks to a computer simulation process which requires a simulation environment. In such a case, the optimization algorithm controls the vector of decision variables, X, which are used by the simulation process in order to compute the performance (quality), y, of such decisions, as shown in Fig. 1.3.

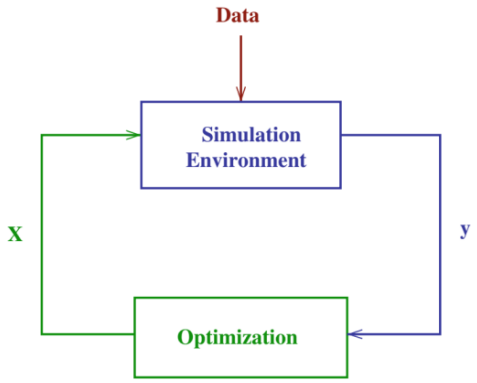

Fig. 1.3 Objective function evaluation based on a simulation process

In this situation, population-based algorithms may not be adapted to address such problems, mainly when the simulation environment requires huge amount of memory space as is often the case in nowadays real-life complex systems. As a matter of fact, in the case of a population-based approach, the simulation environment has to be duplicated for each individual of the population of solutions, which may require an excessive amount of memory. In order to avoid this drawback, one may think about having only one simulation environment which could be used each time a point in the population has to be evaluated. One first consider the first individual for which the simulation environment is initiated and the simulation associated with this first individual is run. The associated performance is then transferred to the optimization algorithm. After that, the second individual is evaluated, but the simulation environment must first be cleared from the events of the first simulation. The simulation is then run for the second individual, and so on until the last individual of the population is evaluated. In this case the memory space is not an issue anymore, but the evaluation time may be excessive and the overall process too slow, due to the fact that the simulation environment is reset at each evaluation.

In the standard simulated annealing algorithm, a copy of a state space point is requested for each proposed transition. In fact, a point $\mathbf{X}_j$ is generated from the current point $\mathbf{X}_i$ through a copy in the memory of the computer. In the case of state spaces of large dimension, the simple process of implementing such a copy may be inefficient and may reduce drastically the performance of simulated annealing.

In such a case, it is much more efficient to consider a *come back* operator, which cancels the effect of a generation. Let G be the generation operator which transforms a point from $\mathbf{X}_i$ to $\mathbf{X}_j$:

$$\mathbf{X}_i \xrightarrow{G} \mathbf{X}_j$$

The comeback operator is the inverse G^{-1} of the generation operator.

Usually, such a generation modifies only one component of the current solution. In this case, the vector $\mathbf{X}_i$ can be modified without being duplicated. Depending on the value obtained when evaluating this new point, two options may be considered:

1. the new solution is accepted and, in this case, only the current objective function value is updated.
2. else, the come back operator G^{-1} is applied to the new position in order to come back to the previous solution, again without any duplication in the memory.

This process is summarized in Fig. 1.4.

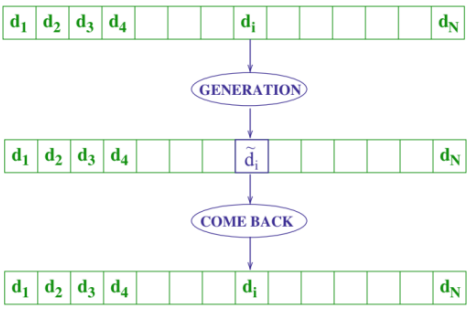

Fig. 1.4 Optimization of the generation process. In this figure, the state space is built with a decision vector for which the generation process consist of changing only one decision (d_i) in the current solution. If this modification is not accepted, this component of the solution recovers its former value. The only information to be stored is the integer i and the real number d_i.

The *come back* operator has to be used carefully because it can easily generate undesired distortions in the way the algorithm searches the state space. For example, if some secondary evaluation variables are used and modified for computing the overall evaluation, such variables must also recover their initial value, and the *come back* operator must therefore ensure the coherence of the state space.

1.5 Illustrative Applications

In this section, we will see how simulated annealing can be applied to two classical NP-hard combinatorial optimization problems: the knapsack problem and the traveling salesman problem.

1.5.1 Knapsack Problem

The knapsack problem can be defined as follows. Given a set of n item types, each with a weight and a value, and given a weight limit, determine the number of each item to include in a collection so that the associated total weight is less than or equal to the weight limit, and so that the total value is as large as possible. The knapsack problem derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

This problem often arises as a subproblem in resource allocation applications where there are financial constraints, such as:

- Cargo loading (truck, boat, cargo aircraft)
- Satellite channel assignment
- Portfolio optimization

In the following, we will consider the binary version of the problem, where there is only one item of each type. Thus, we have n items, each with value v_i and weight w_i , $i = 1,\ldots,n$. We must decide whether each item should be put (or not) in a knapsack of weight limit P, so as to maximize its total value. Before presenting the application of simulated annealing to such a problem, we first present a mathematical model for this optimization problem.

1.5.1.1 Mathematical Modeling

As for any real optimization problem to be solved, the modeling step is critical and has to be done carefully. It models the state space by defining the decision variables, and it expresses the objective function and the constraint functions in terms of the decision variables and the given data.

In the binary knapsack problem, we have a vector of binary decision variables $x = (x_1, x_2, \ldots, x_n)^T$, where $x_i = 0$ if item i is left out of the knapsack and $x_i = 1$ if item i is put in the knapsack. For a given vector x, the objective function value, which represents the total value of the items in the knapsack, is:

$$f(x) = \sum_{i=1}^{n} v_i x_i.$$

We want this value to be maximized. If there was no weight limit, there would be no optimization problem in the sense that all items would fit in the knapsack (i.e., the optimal decision vector would be $x = (1,1,\ldots 1)^T$). Thus, the weight limit makes the problem combinatorial. This weight limit is the main constraint of this problem and is modeled by the following inequality:

$$\sum_{i=1}^{n} w_i . x_i . \leq P.$$

Then, one must add the binary constraints:

$$x_i \in \{0,1\}, \text{ for } i = 1,2,\ldots,n.$$

The overall model is then

$$\begin{gathered} \max f(x) = \sum_{i=1}^{n} v_i x_i \\ s.t. \\ \sum_{i=1}^{n} p_i x_i \leq P \\ x_i \in \{0,1\}, i = 1,2,\ldots,n. \end{gathered}$$

This problem is easy to formulate but hard to solve due to the associated combinatorics. For n items, the number of potential solutions to consider is 2^n which grows very rapidly with n:

n	2^n
10	1.024×10^3
20	1.048×10^6
30	1.073×10^9
40	1.099×10^{12}
50	1.125×10^{15}
60	1.152×10^{18}
70	1.180×10^{21}
80	1.208×10^{24}
90	1.237×10^{27}
100	1.267×10^{30}

For large instances of the knapsack problem, one can consider applying metaheuristics like simulated annealing.

1.5.1.2 Simulated Annealing Implementation

For the knapsack problem, each solution is encoded as a binary vector X. From a point X_i, we generate a neighbor X_j by randomly flipping one component of X_i, as shown in Fig. 1.5 where the kth component is chosen.

In the unconstrained optimization context of SA, a classical relaxation can be considered to take into account the weight limit constraint. Basically, a term is added in the objective function to penalize the violation of this constraint. Here, we compute the weight excess Δ when the weight of the items in the knapsack exceeds its weight limit:

$$\Delta = \min(0, (\sum_{i=1}^{n} w_i x_i) - P).$$

and the objective function value is then penalized by subtracting from it $\mu \frac{\Delta}{P}$, where μ is a penalty parameter to be set by the user.

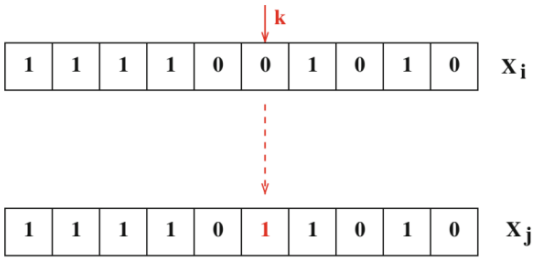

Fig. 1.5 In this example, with $n = 10$, the sixth position has been randomly selected in order to include the sixth object in the bag

In order to test the simulated annealing algorithm on this problem, we first build an instance of the problem by randomly generating 100 items for which the weights have also been selected randomly between 1 and 100 with a uniform probability density function. For this instance, the weight limit of the bag is set to $P = 2000$. We choose $\mu = 1$ for the penalty parameter and we apply the basic SA algorithm with the initial temperature set to a value of c_0 such that $\chi(c) = 0.8$, a geometric cooling schedule with $\alpha = 0.995$, and $L_k = 1000$ for every iteration k. The algorithm is stopped when the temperature reaches $\frac{c_o}{1000}$.

We propose as initial solution a uniformly-distributed random binary vector. The evolution of the penalized objective function with the number of iterations is shown in Fig. 1.6, and the associated evolution of the total weight and the value of the knapsack is shown in Fig. 1.7. At the beginning of the optimization process, the SA explores the solution space by accepting solutions that yield low value of the penalized objective function. This leads to high excess weight and high total value. The value of the penalized objective function increases as the algorithm converges to the optimal solution. Since the excess weight is high at the beginning, the solution is improved mainly by removing weight from the knapsack, therefore the total weight and total value decrease. As the excess weight reaches zero (feasible solution) the solution must be improved by increasing the value (while keeping the weight under the weight limit). Therefore, the total value increases until it reaches the maximum value.

1.5.2 Traveling Salesman Problem

The traveling salesman problem (TSP) asks the following question: "Given a list of n cities, among which an origin city, and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" This is again an important NP-hard combinatorial optimization

problem, particularly in the fields of operations research and theoretical computer science. The problem was first formulated in 1930 and is one of the most intensively-studied problems in discrete optimization.

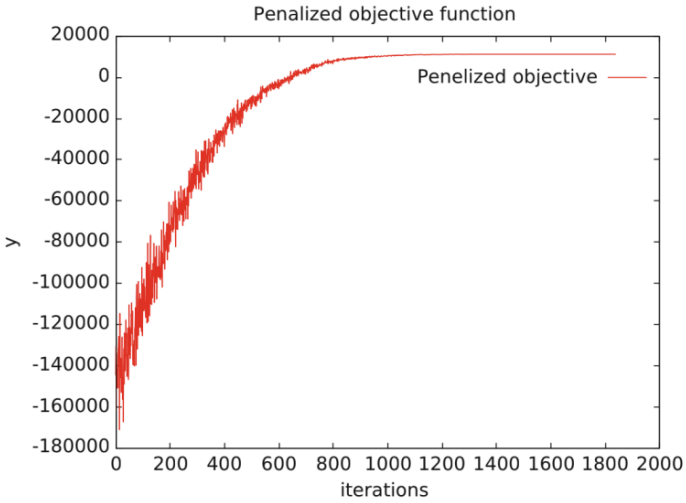

Fig. 1.6 Evolution of the penalized objective function with iterations

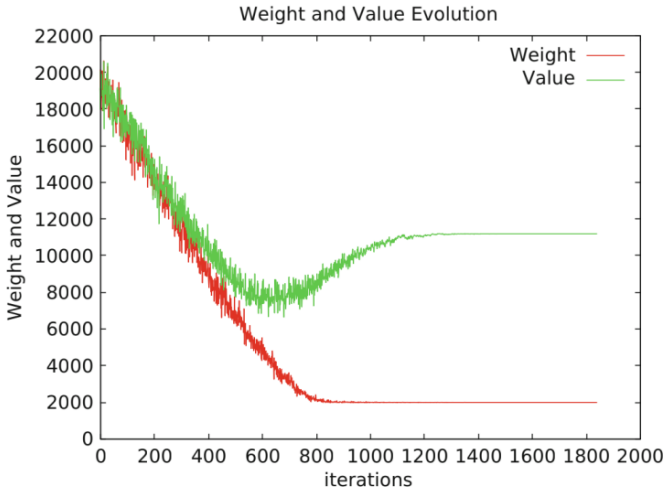

Fig. 1.7 Evolution of the total weight and value with iterations

As for the knapsack problem, we first present the mathematical modeling.

1.5.2.1 Mathematical Modeling

Let us consider a set of n cities where each city i has coordinates $(x_i, y_i), i = 1, 2, \ldots, n$. In this case, each point X of the state space, has to represent a potential permutation in the order we visit the n cities. For simplicity, we consider the following initial solution using the lexicographic order:

$$X_0 = \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{\ldots}\boxed{n}$$

The objective function evaluation consists in computing the length f of the tour corresponding to any vector X:

$$f(X) = \sum_{i=1}^{n-1} d(X_i, X_{i+1}) + d(X_N, X_1),$$

where, X_i is the ith element of X. If $X_i = k$ and $X_{i+1} = l$, the inter-city distance is:

$$d(X_i, X_{i+1}) = \sqrt{(x_l - x_k)^2 + (y_l - y_k)^2}.$$

Note that the last term, $d(X_N, X_1)$, in the above definition of f represents the last segment of the tour to come back to the origin city.

The complexity associated with the traveling salesman problem is known to be much higher than that of the knapsack problem. For a problem with n cities, the number of potential tours to be considered is $n!$, which grows with n much faster than 2^n:

n	2^n	n!
10	1.024×10^{3}	3.628×10^{6}
20	1.048×10^{6}	2.432×10^{18}
30	1.073×10^{9}	2.652×10^{32}
40	1.099×10^{12}	8.159×10^{47}
50	1.125×10^{15}	3.041×10^{64}
60	1.152×10^{18}	8.320×10^{81}
70	1.180×10^{21}	1.197×10^{100}
80	1.208×10^{24}	7.156×10^{118}
90	1.237×10^{27}	1.485×10^{138}
100	1.267×10^{30}	9.332×10^{157}

Just to give an idea of the complexity of the problem, if one evaluation of the objective function requests 10^{-9} s, then a naive enumeration algorithm evaluating every possible solution would require the following CPU time:

n	2^n	$n!$	ratio $\frac{n!}{2^n}$
10	1 µs	3.6 ms	3.6×10^{3}
20	1 ms	77 years	2.3×10^{12}
30	1 s	8.4×10^{15} years	2.47×10^{23}
40	18 min	2.5×10^{31} years	7.4×10^{35}
50	13 days	9.6×10^{47} years	2.7×10^{49}
60	36 years	2.6×10^{47} years	7.2×10^{63}
70	37×10^{3} years	3.8×10^{83} years	1×10^{79}
80	38×10^{6} years	2.2×10^{102} years	5.9×10^{94}
90	39×10^{9} years	4.7×10^{121} years	1.2×10^{111}
100	40×10^{12} years	2.9×10^{141} years	7.3×10^{127}

Even if the computer power is likely to double in the next 18 months, no need to say that it would not make such a naive algorithm practical.

1.5.2.2 Simulated Annealing Implementation

One of the simplest neighborhood operator for this problem consists of randomly exchanging two positions in the current solution vector X (see Fig. 1.8). This way of manipulating points of the state space ensures that the produced neighbor remains a permutation i.e. a tour of the n cities. Implementing such an operator within the SA algorithm yields acceptable results, but the performance of the SA can really be improved by using a neighborhood operator that exchanges all the positions between two randomly chosen indices (m,n), as shown in Fig. 1.9.

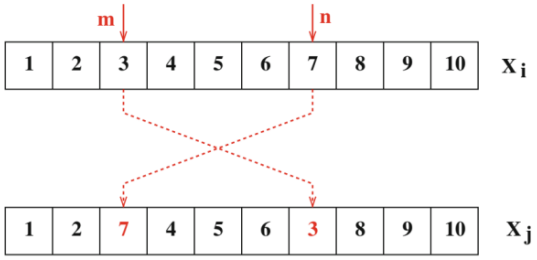

Fig. 1.8 A first neighborhood operator: randomly swapping two positions

Let us consider an instance with $n = 1000$ cities randomly generated in a square subset of the plane. The straightforward SA algorithm is implemented, again, with initial temperature c_0 such that $\chi(X) = 0.8$, a geometric cooling schedule with $\alpha = 0.995$, and $L_k = 1000$ for every iteration. The algorithm is stopped when the temperature reaches $\frac{c_0}{1000}$, and based on the second neighborhood operator (Fig. 1.9). The initial solution considered is the tour of total distance 1.16857164×10^{8} shown in Fig. 1.10.

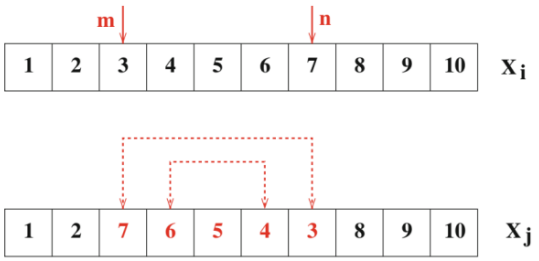

Fig. 1.9 A second neighborhood operator: swapping all positions between two randomly chosen positions (m,n)

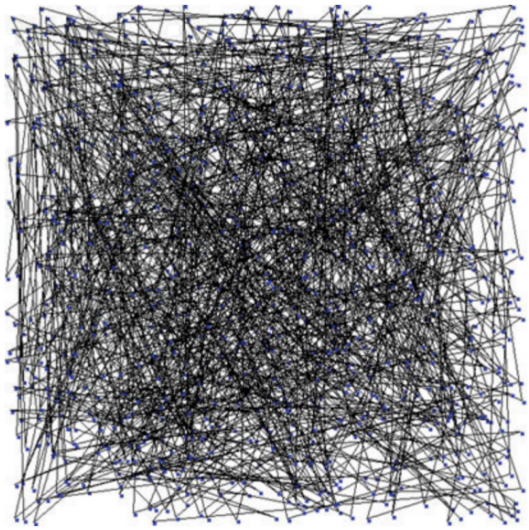

Fig. 1.10 Initial tour of the TSP with $n = 1000$ cities

After application of the simulated annealing algorithm on this problem, one obtains the tour displayed in Fig. 1.11. One minute of computation on a Unix platform with a 2.4 GHz processor and 8 GB of RAM was needed to get the final tour of total distance 360,482.

This is clearly not an optimal solution for this instance (there are some suboptimal crossings) but this solution is very easily obtained via a direct application of SA.

Simulated annealing has also been applied to many combinatorial problems coming from the industry and real-world operations. To mention just a few:

- Airline Crew Scheduling [8]
- Railway Crew Scheduling [9]
- Traveling Salesman Problem [4]
- Vehicle Routing Problem [14]
- Layout-Routing of Electronic Circuits [17]

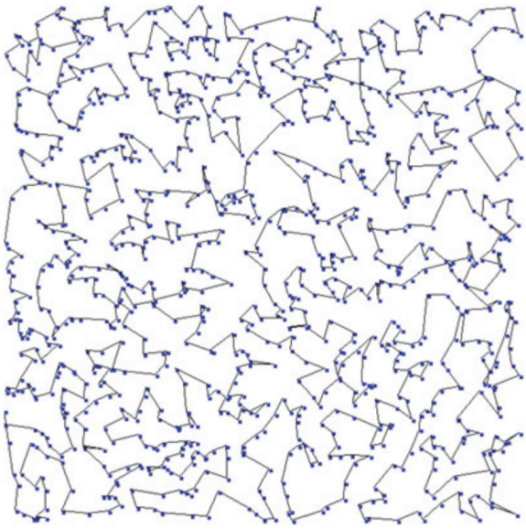

Fig. 1.11 Final tour of the TSP with $n = 1000$ cities

- Large Scale Aircraft Trajectory Planning [5, 10]
- Complex portfolio problem [7]
- Graph coloring problem [6]
- High-dimensionality minimization problems [16]

1.6 Large-Scale Aircraft Trajectory Planning

In this section, we present a methodology using SA to address a strategic planning of aircraft trajectories at the European continental scale, which involves nearly 30,000 flights per day. The goal is to separate the given set of 4D aircraft trajectories (three-dimension space plus time) by allocating an alternative route in the three-dimension space and an alternative departure time to each flight.

1.6.1 Mathematical Modeling

Our strategic trajectory planning problem considers a set of flight plans (origin, destination, departure time) for a given day. We rely on route or departure-time allocation to separate aircraft trajectories. In other words, for each flight, we can delay departure and/or impose an alternative route instead of the initially-planned direct route between the origin and the destination. This can be formulated as an optimization problem aimed at minimizing the number of *interactions* between trajectories,

where we count one interaction whenever two flights are *in conflict* i.e., separated at some point by less than 5 NM (nautical miles) horizontally or 1000 feet vertically.

Given Data. A problem instance is given by:

- A set of N initial (nominal) discretized 4D (direct-route) trajectories;
- For each flight i, for $i = 1,2,\ldots,N$:
  - The initial planned departure time: $t_{i,0}$;
  - The maximum allowed advance departure time shift: $\delta_a^i < 0$;
  - The maximum allowed delay departure time shift: $\delta_d^i > 0$;
  - The maximum allowed route length extension coefficient: $0 \leq d_i \leq 1$.
  - M: the number of allowed virtual waypoints to modify the route.

Decision Variables. In the time domain, one can use a departure-time shift, δ_i, associated with each flight i ($i = 1,2,\ldots,N$). Therefore, the resulting departure time of flight i is given by $t_i = t_{i,0} + \delta_i$, In the 3D space, one can rely on a vector, w_i, of virtual *waypoint locations* through which flight i must go (using straight-line segments), $w_i := (w_i^1, w_i^2, \ldots, w_i^M), i = 1,\ldots,N$. Let us set the compact vector notation: $\delta := (\delta_1, \delta_2, \ldots, \delta_N)$, and $\mathbf{w} := (w_1, w_2, \ldots, w_N)$. Therefore, the decision variables of our route / departure-time allocation problem can be represented by the vector: $u := (\delta, \mathbf{w})$.

Constraints. The above optimization variables must satisfy the following constraints:

- ***Allowed departure time shift***. Since it is not reasonable to delay or to advance departure times for too long, the departure time shift, δ_i, is limited to lie in the interval $[\delta_a^i, \delta_d^i]$. Common practice in airports led us to discretize this time interval. Given the (user-defined) time-shift step size δ_s, this yields $N_a^i := \frac{|\delta_a^i|}{\delta_s}$ possible advance slots, and $N_d^i := \frac{\delta_d^i}{\delta_s}$ possible delay slots for flight i. Therefore, we define the discrete set, Δ_i, of all possible departure time shifts for flight i by

$$\Delta_i := \{-N_a^i.\delta_s, -(N_a^i - 1).\delta_s, \ldots, -\delta_s, 0, \delta_s, \ldots, (N_d^i - 1).\delta_s, N_d^i.\delta_s\}. \quad (1.7)$$

- ***Maximal route length extension***. The alternative trajectory to be chosen increases the route length, which leads to an increase in fuel consumption and flight time. Therefore, the alternative choice should be limited for the new trajectory if it is to be accepted by the airline. Consequently, the alternative trajectory for flight i must satisfy:

$$L_i(w_i) \leq (1 + d_i), \quad (1.8)$$

  where $L_i(w_i)$ denotes the *normalized length* (i.e., assuming that the direct-flight path length is 1) of the alternative trajectory determined by the waypoint vector w_i.
- ***Allowed waypoint locations.*** To reduce the search space, prevent undesirable sharp turns, and restrain the route length extension, we bound the possible location of each virtual waypoint. Let W_{ix}^m and W_{iy}^m be the 2D sets of all possible normalized longitudinal and lateral locations, respectively, of the mth virtual

waypoint for trajectory i. The (normalized) longitudinal component, w_{ix}^m, must lie in the interval:

$$W_{ix}^m := \left[\left(\frac{m}{1+M} - b_i\right), \left(\frac{m}{1+M} + b_i\right)\right], m = 1, 2, \ldots, M, \tag{1.9}$$

where $0 \leq b_i \leq 1$ is a (user-defined) model parameter. The normalized lateral component, w_{iy}^m, is restricted to lie in the interval:

$$W_{iy}^m := [-a_i, a_i], \tag{1.10}$$

where $0 \leq a_i \leq 1$ is a (user-defined) model parameter chosen *a priori* so as to satisfy (1.8). This yields a rectangular shape for the possible locations of the virtual waypoint w_i^m (see Figure 1.12).

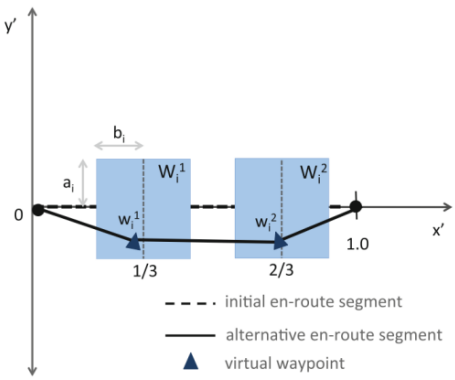

Fig. 1.12 Rectangular-shape sets of the possible locations of $M = 2$ virtual waypoints, for trajectory i

Objective Function

The objective is to minimize the number of *interactions between trajectories*, which correspond, roughly speaking, to situations that occur in the flight planning phase, when more than one trajectory compete for the same space at the same period of time. Consider for example the trajectories A, B and C in Fig. 1.13.

We define an *interaction at a trajectory point* $P_{i,k}(u_i)$ to be the sum of all the conflicts associated with point $P_{i,k}(u_i)$, where u_i the ith component of u. We further define the *interaction,* Φ_i*, associated with trajectory* i, as: $\Phi_i(u) := \sum_{k=1}^{K_i} \Phi_{i,k}(u)$ where K_i is the number of trajectory points obtained through some discretization of the trajectory of the ith flight. Figure 1.13 illustrates the case of trajectory $i = B$ at the trajectory point $P_{B,4}$. Finally, *interaction between trajectories*, Φ_{tot}, for a whole traffic situation is simply defined as:

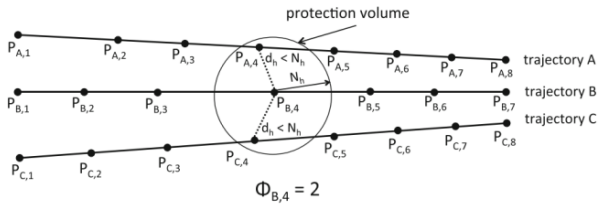

Fig. 1.13 Interactions, $\Phi_{B,4}$, at sampling point $P_{B,4}$ of trajectory B

$$\Phi_{tot}(u) := \sum_{i=1}^{N} \Phi_i(u) = \sum_{i=1}^{N} \sum_{k=1}^{K_i} \Phi_{i,k}(u). \tag{1.11}$$

The interaction minimization problem can be formulated as a mixed-integer optimization problem, as follows:

$$\min_{u=(\delta,\mathbf{w})} \Phi_{tot}(u)$$

subject to

$$\begin{aligned} &\delta_i \in \Delta_i, \quad \text{for all } i = 1,2,\ldots,N \\ &w_{ix}^m \in W_{ix}^m, \quad \text{for all } i = 1,2,\ldots,N, m = 1,2,\ldots,M \\ &w_{iy}^m \in W_{iy}^m, \quad \text{for all } i = 1,2,\ldots,N, m = 1,2,\ldots,M, \end{aligned} \tag{P1}$$

where the set Δ_i is defined in (1.7), and W_{ix}^m and W_{iy}^m are defined in (1.9) and (1.10), respectively.

In order to evaluate the objective function of a candidate solution, $(\mathbf{w}, \delta)$, one needs to compute the interaction, Φ_{tot}, between the N aircraft trajectories. To avoid the $\frac{N(N-1)}{2}$ time-consuming pair-wise comparisons, which is prohibitive in our large-scale application context, we propose a 4D grid-based conflict detection scheme as illustrated in Fig. 1.14 (see [5, 10] for further details). First, we define a four-dimensional (3D space + time) grid (see Fig. 1.14). The size of each cell in the x, y, and z directions is defined by the minimum separation requirements, $N_h = 5$ NM and $N_v = 1000$ ft. The size of the cell in the time domain is set according to some given discretization step size, t_s. To detect conflicts, the idea is to successively put each trajectory in this grid, and then check for conflicts only in the cells surrounding the current trajectory.

In the SA optimization process, the computation of the objective function, $\Phi_{tot}(u)$, is repeated many times. Therefore it must be computed as efficiently as possible. To avoid checking interactions over all the N trajectories even when only a subset of trajectories are modified in a new proposed solution, the interaction count is updated in a *differential* manner. More precisely, we proceed as follows. First, the 4D grid is initialized with every cell empty. Then, the initial N trajectories, corresponding to the initial value of the decision vector, u (with all its components at zero, i.e., direct flight), are placed in the 4D grid and the *current* interaction, Φ_{i_C}, associated with each trajectory, i, and the current total interaction between trajectories, Φ_{tot_C}, are computed.

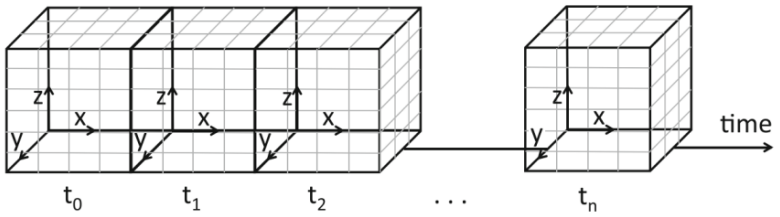

Fig. 1.14 Four dimension (space-time) grid

We assume now that during the optimization process, the decision variables of l flights are to be modified. Let I_{modif} be a list of length l containing the flight indices of the l flights. To update the value of total interaction, we first remove all the l corresponding trajectories from the 4D grid. Therefore, the interaction associated with each trajectory in I_{modif} is set to an intermediate value $\Phi_{i,inter}(u) = 0, \quad \forall i \in I_{modif}$. It should be noted that the interaction measurement is symmetrical: if $\Phi^{ij}(u)$ denotes the *contribution of trajectory i to the interaction associated with trajectory j*, then $\Phi^{ij}(u) = \Phi^{ji}(u)$. Let $\mathcal{N}_i$ be a set of trajectories currently interacting with trajectory i. The interaction associated with trajectory $j \in \mathcal{N}_i$ over all trajectories $i \in I_{modif}$, is set to an intermediate value $\Phi_{j,inter}(u) = \Phi_j(u) - \sum_{i \in I_{modif}} \Phi^{ij}(u)$. Thereafter, the *modified* trajectories corresponding to the new decision variable values, u_i, $i \in I_{modif}$, are placed in the 4D grid and the interaction detection procedure is performed over all trajectories $i \in I_{modif}$. Then, the interaction, Φ_i, associated with each trajectory $i \in I_{modif}$, is computed. Again, the interaction associated with each trajectory, j, interacting with the set of modified trajectories is updated as follows: $\Phi_j(u) = \Phi_{j,inter}(u) + \sum_{i \in I_{modif}} \Phi^{ij}(u)$. Finally, the total interaction between trajectories is simply computed as $\Phi_{tot}(u) = \sum_{i=1}^{N} \Phi_i(u)$. This interaction computation method allows us to update the value of the objective function when some trajectories are modified within a very short computation time, since we do not need to compute the change of interaction for decisions that are not modified at the current optimization iteration.

1.6.2 Computational Experiments with SA

The proposed methodology is tested with a continent-size air traffic instance for a full day of air-traffic over the European airspace, consisting of $N = 29{,}852$ en-route trajectories. The trajectories are sampled with a discretization step of t_s = 20 s. The initial trajectory set involves $\Phi_{tot} = 142{,}144$ total interactions between trajectories. Figure 1.15 illustrates the initial trajectory points (blue dots), and the locations where the initial interactions occur (red dots).

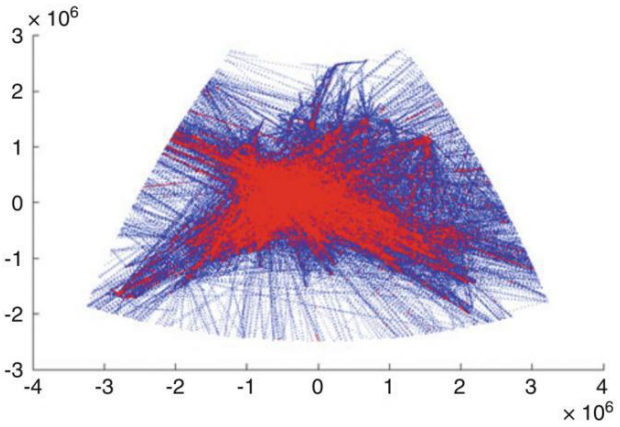

Fig. 1.15 Initial (direct-route) trajectory set involving 1-day en-route air traffic over the European airspace (29,852 flights) sampled with $t_s = 20$ s with initial location of interactions displayed as red color dots

The initial temperature is computed by first generating 100 deteriorating transformations at random and then by evaluating the average variations, $\Delta\Phi_{avg}$, of the objective function values. The initial temperature, c_0, is then deduced from the relation: $c_0 = e^{\frac{\Delta\Phi_{avg}}{\tau_0}}$, where τ_0 is the initial acceptance rate of degrading solutions (which will be empirically set). In order to reach an equilibrium, a sufficient number of iterations, denoted L_k, have to be performed at each temperature step k. In our case, we assume for simplicity purposes that the number of iterations, L_k, is constant and empirically set. The temperature is decreased following the geometrical law, $c_{k+1} = \alpha c_k$, where $0 \leq \alpha \leq 1$ is a pre-defined constant value.

To generate a solution in the neighborhood, we set a user-defined threshold value of interaction, denoted Φ_τ, such that the trajectory of a randomly chosen flight i will be modified only if $\Phi_i(u) \geq \Phi_\tau$, where u is the current solution. Then, for a chosen flight, i, we introduce another user-defined parameter, $P_w \leq 1$, to control the probability of modifying the value of the ith trajectory waypoint location decision vector, w_i. The probability to modify instead the departure time is thus $1 - P_w$. The algorithm terminates when the final temperature, c_f, is reached, or when an interaction-free solution is found. The parameter values chosen to specify the instance considered, and the empirically set parameters defining the overall SA problem-solving methodology are given in Table 1.1.

The SA adapted to solve the strategic trajectory planning problem is implemented in Java. We address this problem instance with an AMD Opteron 2 GHz processor with 128 Gb RAM. Numerical results obtained from the simulation are reported in Table 1.2. This SA implementation yields an interaction-free solution for this continent-scale problem instance after around 76 min of computation time. This is compatible with strategic (several days in advance) planning application requirements in the setting of regular airline schedules.

Table 1.1 Chosen (user-defined) parameter values defining the problem and the empirically-set (user-defined) parameter values of the resolution methodology

Parameters defining the problem		Parameters defining the SA	
Parameter	Value	Parameter	Value
$-\delta_a^i = \delta_d^i$	60 min	L_k	3500
δ_s	20 s	τ_0	0.3
d_i	0.12 (12%)	β	0.99
M	2	T_f	$(1/500).T_0$
a_i	0.126	P_w	0.5
b_i	0.067	Φ_τ	0.5 Φ_{avg}

Table 1.2 Numerical results for continent-size problem instance solved by SA (averages are computed over 10 runs)

Numerical results	Value
Number of iterations	497,000
Avg. computation time (minutes)	76.19
Avg. proportion of delayed/advanced flights	71.29%
Avg. proportion of extended flights	46.23%
Avg. departure time shifts (minutes)	30.14
Avg. route length extensions	1.95%

1.7 Conclusion

This chapter introduced the reader to simulated annealing (SA), a global optimization metaheuristic. The main advantage of SA is its simplicity. SA is based on an analogy with the physical annealing of materials that avoids the drawback of the Monte-Carlo approach (which can be trapped in local minima), thanks to an efficient Metropolis acceptance criterion. When the objective function evaluations require a lot of memory space, for example when it results from complex simulation processes that manipulate large-dimension state space involving much memory, population-based algorithms are not applicable and simulated annealing is the right answer to address such issues. An illustration was provided in section 1.6 where a large-scale complex aircraft trajectory planning problem involving nearly 30,000 flights over Europe was addressed by exploiting particular features of the problem and, in particular, by integrating clever implementation techniques within the algorithm, and by setting user-defined parameters empirically, along the lines of the basic SA theory.

References

1. E. Aarts, J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing* (Wiley, New York, 1989)
2. E. Aarts, P. Van Laarhoven, A new polynomial time cooling schedule, in *Proceedings of the IEEE International Conference on Computer-Aided Design, Santa Clara* (1985), pp. 206–208
3. E. Aarts, P. Van Laarhoven, Statistical cooling: a general approach to combinatorial problems. Philips J. Res. **40**, 193–226 (1985)
4. H. Bayram, R. Sahin, A new simulated annealing approach for travelling salesman problem. Math. Comput. Appl. **18**(3), 313–322 (2013)
5. S. Chaimatanan, D. Delahaye, M. Mongeau, A hybrid metaheuristic optimization algorithm for strategic planning of 4D aircraft trajectories at the continental scale. IEEE Comput. Intell. Mag. **9**(4), 46–61 (2014)
6. M. Chams, A. Hertz, D. de Werra, Some experiments with simulated annealing for coloring graphs. Eur. J. Oper. Res. **32**(2), 260–266 (1987)
7. Y. Crama, M. Schyns, Simulated annealing for complex portfolio selection problems. Eur. J. Oper. Res. **150**(3), 546–571 (2003)
8. T. Emden-Weiner, M. Proksch, Best practice simulated annealing for the airline crew scheduling problem. J. Heuristics **5**(4), 419–436 (1999)
9. R. Hanafi, E. Kozan, A hybrid constructive heuristic and simulated annealing for railway crew scheduling. Comput. Ind. Eng. **70**, 11–19 (2014)
10. A. Islami, S. Chaimatanan, D. Delahaye, Large-scale 4D trajectory planning, in *Air Traffic Management and Systems II*, ed. by Electronic Navigation Research Institute. Lecture Notes in Electrical Engineering, vol. 420 (Springer, Tokyo, 2017), pp. 27–47
11. S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing. IBM Research Report RC 9355, Acts of PTRC Summer Annual Meeting (1982)
12. S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing. Science **220**(4598), 671 (1983)
13. P. Laarhoven, E. Aarts (eds.), *Simulated Annealing: Theory and Applications* (Kluwer, Norwell, 1987)
14. W.F. Mahmudy, Improved simulated annealing for optimization of vehicle routing problem with time windows (VRPTW). Kursor J. **7**(3), 109–116 (2014)
15. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equation of state calculation by fast computing machines. J. Chem. Phys. **21**(6), 1087–1092 (1953)
16. P. Siarry, G. Berthiau, F. Durdin, J. Haussy, Enhanced simulated annealing for globally minimizing functions of many continuous variables. ACM Trans. Math. Softw. **23**(2), 209–228 (1997)
17. D.F. Wong, H.W. Leong, C.L. Liu, *Simulated Annealing for VLSI Design* (Kluwer Academic, Boston, 1988)

Chapter 2
Tabu Search

Michel Gendreau and Jean-Yves Potvin

Abstract This chapter presents the fundamental concepts of Tabu Search (TS) in a tutorial fashion. Special emphasis is put on showing the relationships with classical local search methods and on the basic elements of any TS heuristic, namely, the definition of the search space, the neighborhood structure, and the search memory. Other sections cover other important concepts such as search intensification and diversification and provide references to significant work on TS. Recent advances in TS are also briefly discussed.

2.1 Introduction

Over the last 30 years, hundreds of papers presenting applications of Tabu Search (TS), a heuristic method originally proposed by Glover in 1986 [30], to various combinatorial problems have appeared in the operations research literature. In several cases, the methods described provide solutions very close to optimality and are

M. Gendreau
Département de mathématiques et de génie industriel, Polytechnique Montréal, Montreal, QC, Canada

Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, Montreal, QC, Canada
e-mail: michel.gendreau@cirrelt.net

J.-Y. Potvin (✉)
Département d'informatique et de recherche opérationnelle, Université de Montréal, Montreal, QC, Canada

Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, Montreal, QC, Canada
e-mail: potvin@iro.umontreal.ca

M. Gendreau, J.-Y. Potvin (eds.), *Handbook of Metaheuristics*,
International Series in Operations Research & Management Science 272,
https://doi.org/10.1007/978-3-319-91086-4_2

can be interpreted as a form of controlled random walk in the space of feasible solutions. The emergence of SA indicated that one could look for other ways to tackle combinatorial optimization problems and spurred the interest of the research community. In the following years, many other new approaches were proposed, mostly based on analogies with natural phenomena (like TS, Ant Colony Optimization, Particle Swarm Optimization, Artificial Immune Systems) which, together with some older ones, such as Genetic Algorithms [38], gained an increasing popularity. Now collectively known under the name of metaheuristics (a term originally coined by Glover in [30]), these methods have become over the last 20 years the leading edge of heuristic approaches for solving combinatorial optimization problems.

2.3.2 Tabu Search

Building upon some of his previous work, Fred Glover proposed a new approach, which he called Tabu Search, to allow local search methods to overcome local optima [30]. In fact, many elements of this first TS proposal, and some elements of later TS elaborations, were introduced in [29], including short term memory to prevent the reversal of recent moves, and longer term frequency memory to reinforce attractive components. The basic principle of TS is to pursue LS whenever it encounters a local optimum by allowing non-improving moves; cycling back to previously visited solutions is prevented by the use of memories, called tabu lists, that record the recent history of the search, a key idea that can be linked to artificial intelligence concepts. It is also important to remark that Glover did not see TS as a proper heuristic, but rather as a metaheuristic, i.e., a general strategy for guiding and controlling inner heuristics specifically tailored to the problems at hand.

2.3.3 Search Space and Neighborhood Structure

As we just mentioned, TS is an extension of classical LS methods. In fact, a basic TS can be seen as simply the combination of LS with short-term memories. It follows that the two first basic elements of any TS heuristic are the definition of its search space and its neighborhood structure.

The search space of an LS or TS heuristic is simply the space of all possible solutions that can be considered (visited) during the search. For instance, in the CVRP example described in Sect. 2.2, the search space could simply be the set of feasible solutions to the problem, where each point in the search space corresponds to a set of vehicles routes satisfying all the specified constraints. While in that case the definition of the search space seems quite natural, it is not always so. In the Capacitated Plant Location Problem (CPLP), for instance, customers must be served from plants located in a subset of potential sites. In this context, one could use the full feasible search space made of binary location variables (a site is open or closed) and

continuous flow variables. A more attractive search space, though, is obtained by restricting the search space to the binary location variables, from which the complete solution can be obtained by solving the associated transportation problem to get the optimal flow variables. One could also decide to search for the extreme points of the set of feasible flow variable vectors, retrieving the associated location variables by noting that a plant must be open whenever some flow is allocated to it [17]. It is also important to note that it is not always a good idea to restrict the search space to feasible solutions; in many cases, allowing the search to move to infeasible solutions is desirable, and sometimes necessary (see Sect. 2.4.3 for further details).

Closely linked to the definition of the search space is that of the neighborhood structure. At each iteration of LS or TS, the local transformations that can be applied to the current solution, denoted S, define a set of neighboring solutions in the search space, denoted $N(S)$ (the neighborhood of S). Formally, $N(S)$ is a subset of the search space made of all solutions obtained by applying a single local transformation to S. In general, for any specific problem at hand, there are many more possible (and even, attractive) neighborhood structures than search space definitions. This follows from the fact that there may be several plausible neighborhood structures for a given definition of the search space. This is easily illustrated on our CVRP example that has been the object of several TS implementations. To simplify the discussion, we suppose in the following that the search space is the feasible space. Simple neighborhood structures for the CVRP involve moving at each iteration a single customer from its current route; the selected customer is inserted in the same route or in another route with sufficient residual capacity. An important feature of these neighborhood structures is the way in which insertions are performed: one could use random insertion or insertion at the best position in the target route; alternately, one could use more complex insertion schemes that involve a partial reoptimization of the target route, such as GENI insertions [25]. Before proceeding any further it is important to stress that while we say that these neighborhood structures involve moving a single customer, the neighborhoods they define contain all the feasible route configurations that can be obtained from the current solution by moving any customer and inserting it in the stated fashion. Examining the neighborhood can thus be fairly demanding.

More complex neighborhood structures for the CVRP, such as the λ-interchange [50], are obtained by allowing simultaneously the movement of customers to different routes and the swapping of customers between routes. In [54], moves are defined by ejection chains that are sequences of coordinated movements of customers from one route to another; for instance, an ejection chain of length 3 would involve moving a customer v_1 from route R_1 to route R_2, a customer v_2 from R_2 to route R_3 and a customer v_3 from R_3 to route R_4. Other neighborhood structures involve the swapping of sequences of several customers between routes, as in the Cross-exchange [63]. These types of neighborhoods have seldom been used for the CVRP, but are common in TS heuristics for its time-windows extension, where customers must be visited within a pre-specified time interval. We refer the interested reader to [9, 27] for a more detailed discussion of TS implementations for the CVRP and the Vehicle Routing Problem with Time Windows.

When different definitions of the search space are considered for a given problem, neighborhood structures will inevitably differ to a considerable degree. In the case of the CPLP, alluded to above, if the search space corresponds to the location variables only, one could use operators to change the status of these variables (from open to closed and conversely). If, however, the search space is made of the extreme points of the set of feasible flow variable vectors, one could instead consider moves defined by the application of pivots to the linear programming formulation of the transportation problem to move the current solution to an adjacent extreme point. Thus, choosing a search space and a neighborhood structure is by far the most critical step in the design of any TS heuristic. It is at this step that one must make the best use of the understanding and knowledge he/she has of the problem at hand.

2.3.4 Tabus

Tabus are one of the distinctive elements of TS when compared to LS. As we already mentioned, tabus are used to prevent cycling when moving away from local optima through non-improving moves. The key realization here is that when this situation occurs, something needs to be done to prevent the search from tracing back its steps to where it came from. This is achieved by declaring tabu (disallowing) moves that reverse the effect of recent moves. For instance, in the CVRP example, if customer v_1 has just been moved from route R_1 to route R_2, one could declare tabu moving back v_1 from R_2 to R_1 for some number of iterations (this number is called the tabu tenure of the move). Tabus are also useful to help the search move away from previously visited portions of the search space and thus perform more extensive exploration.

Tabus are stored in a short-term memory of the search (the tabu list) and usually only a fixed and fairly limited quantity of information is recorded. In any given context, there are several possibilities regarding the specific information that is recorded. One could record complete solutions, but this requires a lot of storage and makes it expensive to check whether a potential move is tabu or not; it is therefore seldom used. The most commonly used tabus involve recording the last few transformations performed on the current solution and prohibiting reverse transformations (as in the example above); others are based on key characteristics of the solutions themselves or of the moves.

To better understand how tabus work, let us go back to our reference problem. In the CVRP, one could define tabus in several ways. To continue our example where customer v_1 has just been moved from route R_1 to route R_2, one could declare tabu specifically moving back v_1 from R_2 to R_1 and record this in the short-term memory as the triplet (v_1, R_2, R_1). Note that this type of tabu will not constrain the search much and that cycling may occur if v_1 is then moved to another route R_3 and then from R_3 to R_1. A stronger tabu would involve prohibiting moving back v_1 to R_1, without consideration for its current route, and be recorded as (v_1, R_1). An even

stronger tabu would be to disallow moving v_1 to any other route and would simply be noted as (v_1).

Multiple tabu lists can be used simultaneously and are sometimes advisable. For example, when different types of moves are used to generate the neighborhood, it might be a good idea to keep a separate tabu list for each type. Standard tabu lists are usually implemented as circular lists of fixed length. It has been shown, however, that fixed-length tabus cannot always prevent cycling, and some authors have proposed varying the tabu list length during the search [31, 32, 58, 60, 61]. Another solution is to randomly generate the tabu tenure of each move within some specified interval; using this approach requires a somewhat different scheme for recording tabus that are then usually stored as tags in an array (the entries in this array will usually record the iteration number until which a move is tabu; see [25], for more details).

2.3.5 Aspiration Criteria

While central to TS, tabus are sometimes too powerful: they may prohibit attractive moves, even when there is no danger of cycling, or they may lead to an overall stagnation of the searching process. It is thus necessary to use algorithmic devices that will allow one to revoke (cancel) tabus. These are called aspiration criteria. The simplest and most commonly used aspiration criterion, which is found in almost all TS implementations, consists in allowing a move, even if it is tabu, if it results in a solution with an objective value better than that of the current best-known solution (since the new solution has obviously not been previously visited). Much more complicated aspiration criteria have been proposed and successfully implemented (see, for instance [19, 37]), but they are rarely used. The key rule in this respect is that if cycling cannot occur, tabus can be disregarded.

2.3.6 A Template for Simple Tabu Search

We are now in the position to give a general template for TS, integrating the elements we have seen so far. We suppose that we are trying to minimize a function $f(S)$ over some domain and we apply the so-called best improvement version of TS, i.e., the version in which one chooses at each iteration the best available move (this is the most commonly used version of TS).

Notation

- S, the current solution,
- S^*, the best-known solution,

- f^*, the value of S^*,
- $N(S)$, the neighborhood of S,
- $\tilde{N}(S)$, the admissible subset of $N(S)$ (i.e., non-tabu or allowed by aspiration),
- T, the tabu list.

Initialization

Choose (construct) an initial solution S_0.
Set $S \leftarrow S_0$, $f^* \leftarrow f(S_0)$, $S^* \leftarrow S_0$, $T \leftarrow \emptyset$.

Search

While termination criterion not satisfied do:

select S in $argmin_{S' \in \tilde{N}(S)}[f(S')]$;
if $f(S) < f^*$, then set $f^* \leftarrow f(S)$, $S^* \leftarrow S$;
record tabu for the current move in T (delete oldest entry if necessary).

2.3.7 Termination Criteria

One may have noticed that we have not specified in our template above a termination criterion. In theory, the search could go on forever, unless the optimal value of the problem at hand is known beforehand. In practice, obviously, the search has to be stopped at some point. The most commonly used stopping criteria in TS are:

- after a fixed number of iterations (or a fixed amount of CPU time);
- after some number of iterations without an improvement in the objective function value (the criterion used in most implementations);
- when the objective reaches a pre-specified threshold value.

In complex tabu schemes, the search is usually stopped after completing a sequence of phases, the duration of each phase being determined by one of the above criteria.

2.3.8 Probabilistic TS and Candidate Lists

In regular TS, one must evaluate the objective for every element of the neighborhood $N(S)$ of the current solution. This can prove extremely expensive from the computational standpoint. An alternative is to instead consider only a random sample $N'(S)$ of $N(S)$, thus reducing significantly the computational burden. Another attractive

2.4.3 Allowing Infeasible Solutions

Accounting for all problem constraints in the definition of the search space often restricts the searching process too much and can lead to mediocre solutions. This occurs, for example, in CVRP instances where the route capacity or duration constraints are too tight to allow moving customers effectively between routes. In such cases, constraint relaxation is an attractive strategy, since it creates a larger search space that can be explored with simpler neighborhood structures. Constraint relaxation is easily implemented by dropping selected constraints from the search space definition and adding to the objective weighted penalties for constraint violations. This, however, raises the issue of finding correct weights for constraint violations. An interesting way of circumventing this problem is to use self-adjusting penalties, i.e., weights are adjusted dynamically on the basis of the recent history of the search: weights are increased if only infeasible solutions were encountered in the last few iterations, and decreased if all recent solutions were feasible (see, for instance, [25] for further details). Penalty weights can also be modified systematically to drive the search to cross the feasibility boundary of the search space and thus induce diversification. This technique, known as strategic oscillation, was introduced as early as 1977 in [29] and used since in several successful TS procedures (an important early variant oscillates among different types of moves, hence neighborhood structures, while another oscillates around a selected value for a critical function).

2.4.4 Surrogate and Auxiliary Objectives

There are many problems for which the true objective function is quite costly to evaluate. When this occurs, the evaluation of moves may become prohibitive, even if sampling is used. An effective approach to handle this issue is to evaluate neighbors using a surrogate objective, i.e., a function that is correlated to the true objective, but is less computationally demanding, in order to identify a (small) set of promising candidates (potential solutions achieving the best values for the surrogate). The true objective is then computed for this small set of candidate moves and the best one selected to become the new current solution; an example of this approach is found in [16].

Another frequently encountered difficulty is that the objective function may not provide enough information to effectively drive the search to more interesting areas of the search space. A typical illustration of this situation is the variant of the CVRP in which the fleet size is not fixed, but is rather the primary objective (i.e., one is looking for the minimal fleet size allowing a feasible solution). In this problem, except for solutions where a route has only one or a few customers assigned to it, most neighborhood structures will lead to the situation where all elements in the neighborhood score equally with respect to the primary objective (i.e., all allowable moves produce solutions with the same number of vehicles). In such a case, it is absolutely necessary to define an auxiliary objective function to orient the search.

Such a function must measure in some way the desirable attributes of solutions. In our example, one could, for instance, use a function that would favor solutions with routes having just a few customers, thus increasing the likelihood that a route can be totally emptied in a subsequent iteration. It should be noted that coming up with an effective auxiliary objective is not always easy and may require a lengthy trial and error process. In some other cases, fortunately, the auxiliary objective is obvious for anyone familiar with the problem at hand (see [24], for an illustration).

2.5 Advanced Concepts

The concepts and techniques described in the previous sections are sufficient to design effective TS heuristics for many combinatorial problems. Early TS implementations, several of which were extremely successful, relied indeed almost exclusively on these algorithmic components. Modern TS implementations, however, exploit more advanced concepts and techniques. While it is clearly beyond the scope of an introductory tutorial, such as this one, to review this type of advanced material, we would like to give readers some insight into it (readers who wish to learn more about this topic should consider the key references provided in the next section).

Various techniques have been devised for making the search more effective. These include methods for exploiting better the information that becomes available during search and creating better starting points, as well as more powerful neighborhood operators and parallel search strategies (on this last topic, see the advances reported in [3] and the chapter on parallel metaheuristics in this Handbook; for specific implementation examples of TS on CPU-based parallel platforms, see [13, 42], and for GPU-based platforms, see [46, 67]). The numerous techniques for making better use of the information are of particular significance since they can lead to dramatic performance improvements. Many of these rely on elite solutions (the best solutions previously encountered) or on parts of these to create new solutions, the rationale being that fragments or elements of excellent solutions are often identified quite early in the searching process, but that the challenge is to complete these fragments or to recombine them [33, 35, 39, 53, 55, 64]. Other methods, such as the Reactive TS [6, 48], attempt to find ways of making the search move away from local optima that have already been visited. An important issue is the general approach for exploiting the search framework provided by TS. Some favor simplicity, that is, a search strategy with only a few parameters and based on simple neighborhood operators, as illustrated by the Unified TS [14, 15, 22]. Others propose complex neighborhood operators, thus leading to large or very large neighborhood searches [1, 2].

Another important research area in TS (this is, in fact, pervasive in the whole metaheuristics field) is hybridization, i.e., using TS in conjunction with other solution approaches such as adaptive large neighborhood search [69], genetic algorithms [41, 45, 47, 49], constraint programming [8, 10, 18, 52] or integer programming techniques (there is a whole chapter on this topic in [35]).

TS has also been successful in domains outside its traditional ones (graph theory problems, scheduling, vehicle routing), for example: continuous optimization [7, 11, 12, 21, 40, 68], multi-criteria optimization [36, 40], stochastic programming [5], mixed integer programming [51, 57], dynamic decision problems [26, 28, 56], etc. These domains confront researchers with challenges that ask for innovative extensions of the method.

2.6 Key References

Readers who wish to read other introductory papers on TS can choose among several ones [23, 31, 34, 37, 62]. The book by Glover and Laguna [35] is the ultimate reference on TS: apart from the fundamental concepts of the method, it presents a considerable amount of advanced material, as well as a variety of applications. It is interesting to note that this book contains several ideas applicable to TS that yet remain to be fully exploited. Also valuable are the books and special issues made up from selected papers presented at the recent Metaheuristics International Conferences (MIC) in 2011 [20], 2013 [44] and 2015 [4]. The last MIC conference was held in Barcelona in 2017 and the conference web site can be accessed at mic2017.upf.edu.

2.7 Tricks of the Trade

Newcomers to TS trying to apply the method to a problem that they wish to solve are often confused about what they need to do to come up with a successful implementation. This section is aimed at providing some help in this regard.

2.7.1 Getting Started

The following step-by-step procedure should provide a useful framework for getting started.
A step-by-step procedure

1. Read one or two good introductory papers to gain some knowledge of the concepts and of the vocabulary.
2. Read several papers describing in detail applications in various areas to see how the concepts have been actually implemented by other researchers.
3. Think a lot about the problem at hand, focusing on the definition of the search space and the neighborhood structure.
4. Implement a simple version based on this search space definition and this neighborhood structure.

5. Collect statistics on the performance of this simple heuristic. It is usually useful at this point to introduce a variety of memories, such as frequency and recency memories, to really track down what the heuristic does.
6. Analyze results and adjust the procedure accordingly. It is at this point that one should eventually introduce mechanisms for search intensification and diversification or other intermediate features. Special attention should be paid to diversification, since this is often where simple TS procedures fail.

2.7.2 More Tips

It is not unusual that, in spite of following carefully the preceding procedure, one ends up with a heuristic that nonetheless produces mediocre results. If this occurs, the following tips may prove useful:

1. If there are constraints, consider penalizing them. Letting the search move to infeasible solutions is often necessary in highly constrained problems to allow for a meaningful exploration of the search space (see Sect. 2.4).
2. Reconsider the neighborhood structure and change it if necessary. Many TS implementations fail because the neighborhood structure is too simple. In particular, one should make sure that the chosen neighborhood structure allows for a purposeful evaluation of possible moves (i.e., the moves that seem intuitively to move the search in the right direction should be the ones that are likely to be selected); it might also be a good idea to introduce a surrogate objective to achieve this (see Sect. 2.4).
3. Collect more statistics.
4. Follow the execution of the algorithm step-by-step on some reasonably sized instances.
5. Reconsider diversification. As mentioned earlier, this is a critical feature in most TS implementations.
6. Experiment with parameter settings. Many TS procedures are extremely sensitive to parameter settings; it is not unusual to see the performance of a procedure dramatically improve after changing the value of one or two key parameters (unfortunately, it is not always obvious to determine which parameters are the key ones in a given procedure).

2.7.3 Additional Tips for Probabilistic TS

While it is an effective way of tackling many problems, probabilistic TS creates problems of its own that need to be carefully addressed. The most important of these is the fact that, more often than not, the best solutions returned by probabilistic TS will not be local optima with respect to the neighborhood structure being used. This

is particularly annoying since, in that case, better solutions can be easily obtained, sometimes even manually. An easy way to come around this is to simply perform a local improvement phase (using the same neighborhood operator) from the best found solution at the end of the TS itself. One could alternately switch to TS without sampling (again from the best found solution) for a short duration before completing the algorithm. A possibly more effective technique is to add throughout the search an intensification step without sampling; in this fashion, the best solutions available in the various regions of the search space explored by the method will be found and recorded (similar special aspiration criteria for allowing the search to reach local optima at useful junctures are proposed in [34]).

2.7.4 Parameter Calibration and Computational Testing

Parameter calibration and computational experiments are key steps in the development of any algorithm. This is particularly true in the case of TS, since the number of parameters required by most implementations is fairly large and since the performance of a given procedure can vary quite significantly when parameter values are modified. The first step in any serious computational experimentation is to select a good set of benchmark instances (either by obtaining them from other researchers or by constructing them), preferably with some reasonable measure of their difficulty and with a wide range of size and difficulty. This set should be split into two subsets, the first one being used at the algorithmic design and parameter calibration steps, and the second reserved for performing the final computational tests that will be reported in the paper(s) describing the heuristic under development. The reason for doing so is quite simple: when calibrating parameters, one always run the risk of overfitting, i.e., finding parameter values that are excellent for the instances at hand, but poor in general, because these values provide too good a fit (from the algorithmic standpoint) to these instances. Methods with several parameters should thus be calibrated on much larger sets of instances than ones with few parameters to ensure a reasonable degree of robustness. The calibration process itself should proceed in several stages:

1. Perform exploratory testing to find good ranges of parameters. This can be done by running the heuristic with a variety of parameter settings.
2. Fix the value of parameters that appear to be robust, i.e., which do not seem to have a significant impact on the performance of the procedure.
3. Perform systematic testing for the other parameters. It is usually more efficient to test values for only a single parameter at a time, the others being fixed at what appear to be reasonable values. One must be careful, however, for cross effects between parameters. Where such effects exist, it can be important to jointly test pairs or triplets of parameters, which can be an extremely time-consuming task.

The work in [16] provides a detailed description of the calibration process for a fairly complex TS procedure and can be used as a guideline for this purpose.

38. J.H. Holland, *Adaptation in Natural and Artificial Systems* (The University of Michigan Press, Ann Arbor, 1975)
39. L.M. Hvattum, A. Lokketangen, F. Glover, Comparisons of commercial MIP solvers and an adaptive memory (tabu search) procedure for a class of 0-1 integer programming problems. Algorithm. Oper. Res. **7**, 13–20 (2012)
40. D.M. Jaeggi, G.T. Parks, T. Kipouros, P.J. Clarkson, The development of a multi-objective tabu search algorithm for continuous optimisation problems. Eur. J. Oper. Res. **185**, 1192–1212 (2008)
41. S.N. Jat, S. Yang, A hybrid genetic algorithm and tabu search approach for post enrolment course timetabling. J. Sched. **14**, 617–637 (2011)
42. J. Jin, T.G. Crainic, A. Lokketangen, A parallel multi-neighborhood cooperative tabu search for capacitated vehicle routing problems. Eur. J. Oper. Res. **222**, 441–451 (2012)
43. S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, Optimization by simulated annealing. Science **220**, 671–680 (1983)
44. H.C. Lau, G.R. Raidl, P. Van Hentenryck (eds.), New developments in metaheuristics and their applications. Special issue. J. Heuristics **22**(4), 359–664 (2016)
45. X. Li, L. Gao, An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem. Int. J. Prod. Econ. **174**, 93–110 (2016)
46. T.V. Luong, L. Loukil, N. Melab, E.-G. Talbi, A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem, in *ACS/IEEE International Conference on Computer Systems and Applications*, Hammamet (2010). https://doi.org/10.1109/AICCSA.2010.5587019
47. T. Lust, J. Teghem, MEMOTS: a memetic algorithm integrating tabu search for combinatorial multiobjective optimization. RAIRO—Oper. Res. **42**, 3–33 (2008)
48. F. Mascia, P. Pellegrini, M. Birattari, T. Stützle, An analysis of parameter adaptation in reactive tabu search. Int. Trans. Oper. Res. **21**, 127–152 (2014)
49. S. Meeran, M.S. Morshed, A hybrid genetic tabu search algorithm for solving job shop scheduling problems: a case study. J. Intell. Manuf. **23**, 1063–1078 (2012)
50. I.H. Osman, Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. Ann. Oper. Res. **41**, 421–451 (1993)
51. J.P. Pedroso, Tabu search for mixed integer programming, in *Metaheuristic Optimization via Memory and Evolution*, ed. by C. Rego, B. Alidaee (Kluwer Academic, Boston, 2005), pp. 247–261
52. G. Pesant, M. Gendreau, A constraint programming framework for local search methods. J. Heuristics **5**, 255–280 (1999)
53. C. Rego, B. Alidaee (eds.), *Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search* (Kluwer Academic, Boston, 2005)
54. C. Rego, C. Roucairol, A parallel tabu search algorithm using ejection chains for the vehicle routing problem, in *Meta-Heuristics: Theory and Applications*, ed. by I.H. Osman, J.P. Kelly (Kluwer Academic, Boston, 1996), pp. 661–675
55. Y. Rochat, É.D. Taillard, Probabilistic diversification and intensification in local search for vehicle routing. J. Heuristics **1**, 147–167 (1995)
56. A.G. Roesener, J.W. Barnes, An advanced tabu search approach to the dynamic airlift loading problem. Log. Res. **9**, 12:1–12:18 (2016)
57. L.H. Sacchi, V.A. Armentano, A computational study of parametric tabu search for 0-1 mixed integer program. Comput. Oper. Res. **38**, 464–473 (2011)
58. J. Skorin-Kapov, Tabu search applied to the quadratic assignment problem. ORSA J. Comput. **2**, 33–45 (1990)
59. P. Soriano, M. Gendreau, Diversification strategies in tabu search algorithms for the maximum clique problem. Ann. Oper. Res. **63**, 189–207 (1996)
60. É.D. Taillard, Some efficient heuristic methods for the flow shop sequencing problem. Eur. J. Oper. Res. **47**, 65–74 (1990)
61. É.D. Taillard, Robust taboo search for the quadratic assignment problem. Parallel Comput. **17**, 443–455 (1991)

62. E. Taillard, Tabu search, in *Metaheuristics*, ed. by P. Siarry (Springer International Publishing, Cham, 2016), pp. 51–76
63. É.D. Taillard, P. Badeau, M. Gendreau, F. Guertin, J.-Y. Potvin, A tabu search heuristic for the vehicle routing problem with soft time windows. Transp. Sci. **31**, 170–186 (1997)
64. C.D. Tarantilis, C.T. Kiranoudis, BoneRoute - an adaptive memory-based method for effective fleet management. Ann. Oper. Res. **115**, 227–241 (2002)
65. P. Toth, D. Vigo (eds.), *The Vehicle Routing Problem*. SIAM Monographs on Discrete Mathematics and Applications (SIAM, Philadelphia, 2002)
66. P. Toth, D. Vigo, The granular tabu search and its application to the vehicle routing problem. INFORMS J. Comput. **15**, 333–346 (2003)
67. C. Tsotskas, T. Kipouros, A.M. Savill, The design and implementation of a GPU-enabled multi-objective tabu-search intended for real world and high-dimensional applications. Procedia Comput. Sci. **29**, 2152–2161 (2014)
68. G. Waligóra, Simulated annealing and tabu search for discrete-continuous project scheduling with discounted cash flows. RAIRO—Oper. Res. **48**, 1–24 (2014)
69. I. Žulj, S. Kramer, M. Schneider, A hybrid of adaptive large neighborhood search and tabu search for the order-batching problem. Eur. J. Oper. Res. **264**, 653–664 (2018)

Chapter 3
Variable Neighborhood Search

Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez

Abstract Variable neighborhood search (VNS) is a metaheuristic for solving combinatorial and global optimization problems whose basic idea is a systematic change of neighborhood both within a descent phase to find a local optimum and in a perturbation phase to get out of the corresponding valley. In this chapter we present the basic schemes of VNS and some of its extensions. We then describe recent developments, i.e., formulation space search and variable formulation search. We then present some families of applications in which VNS has proven to be very successful: (1) exact solution of large scale location problems by primal-dual VNS; (2) generation of solutions to large mixed integer linear programs, by hybridization of VNS and local branching; (3) generation of solutions to very large mixed integer programs using VNS decomposition and exact solvers (4) generation of good

P. Hansen
École des Hautes Études Commerciales, Montréal, QC, Canada

GERAD, Montréal, QC, Canada
e-mail: pierre.hansen@gerad.ca

N. Mladenović (✉)
Mathematical Institute, SANU, Belgrade, Serbia
e-mail: nenad@mi.sanu.ac.rs

J. Brimberg
Department of Mathematics and Computer Science, Royal Military College of Canada, Kingston, ON, Canada
e-mail: jack.brimberg@rmc.ca

J. A. M. Pérez
IUDR and Department of Informatics and Systems Engineering, Universidad de La Laguna, Tenerife, Spain
e-mail: jamoreno@ull.es

M. Gendreau, J.-Y. Potvin (eds.), *Handbook of Metaheuristics*,
International Series in Operations Research & Management Science 272,
https://doi.org/10.1007/978-3-319-91086-4_3

feasible solutions to continuous nonlinear programs; (5) adaptation of VNS for solving automatic programming problems from the Artificial Intelligence field and (6) exploration of graph theory to find conjectures, refutations and proofs or ideas of proofs.

3.1 Introduction

Optimization tools have greatly improved during the last two decades. This is due to several factors: (1) progress in mathematical programming theory and algorithmic design; (2) rapid improvement in computer performances; (3) better communication of new ideas and integration in widely used complex softwares. Consequently, many problems long viewed as out of reach are currently solved, sometimes in very moderate computing times. This success, however, has led researchers and practitioners to address much larger instances and more difficult classes of problems. Many of these may again only be solved heuristically. Therefore thousands of papers describing, evaluating and comparing new heuristics appear each year. Keeping abreast of such a large literature is a challenge. Metaheuristics, or general frameworks for building heuristics, are therefore needed in order to organize the study of heuristics. As evidenced by the Handbook, there are many of them. Some desirable properties of metaheuristics [58, 59, 68] are listed in the concluding section of this chapter.

Variable neighborhood search (VNS) is a metaheuristic proposed by some of the present authors some 20 years ago [80]. Earlier work that motivated this approach can be found in [25, 36, 44, 78]. It is based upon the idea of a systematic change of neighborhood both in a descent phase to find a local optimum and in a perturbation phase to get out of the corresponding valley. Originally designed for approximate solution of combinatorial optimization problems, it was extended to address mixed integer programs, nonlinear programs, and recently mixed integer nonlinear programs. In addition VNS has been used as a tool for automated or computer assisted graph theory. This led to the discovery of over 1500 conjectures in that field and the automated proof of more than half of them. This is to be compared with the unassisted proof of about 400 of these conjectures by many different mathematicians.

Applications are rapidly increasing in number and pertain to many fields: location theory, cluster analysis, scheduling, vehicle routing, network design, lot-sizing, artificial intelligence, engineering, pooling problems, biology, phylogeny, reliability, geometry, telecommunication design, etc. References are too numerous to be listed here, but many of them can be found in [69] and special issues of *IMA Journal of Management Mathematics* [76], *European Journal of Operational Research* [68] and *Journal of Heuristics* [87] that are devoted to VNS.

This chapter is organized as follows. In the next section we present the basic schemes of VNS, i.e., variable neighborhood descent (VND), reduced VNS (RVNS), basic VNS (BVNS) and general VNS (GVNS). Two important extensions are presented in Sect. 3.3: Skewed VNS and Variable neighborhood decomposition

search (VNDS). A further recent development called Formulation Space Search (FSS) is discussed in Sect. 3.4. The remainder of the paper describes applications of VNS to several classes of large scale and complex optimization problems for which it has proven to be particularly successful. Section 3.5 is devoted to primal dual VNS (PD-VNS) and its application to location and clustering problems. Finding feasible solutions to large mixed integer linear programs with VNS is discussed in Sect. 3.6. Section 3.7 addresses ways to apply VNS in continuous global optimization. The more difficult case of solving mixed integer nonlinear programming by VNS is considered in Sect. 3.8. Applying VNS to graph theory *per se* (and not just to particular optimization problems defined on graphs) is discussed in Sect. 3.9. Brief conclusions are drawn in Sect. 3.10.

3.2 Basic Schemes

A deterministic optimization problem may be formulated as

$$\min\{f(x)|x \in X, X \subseteq \mathscr{S}\}, \tag{3.1}$$

where $\mathscr{S}, X, x$ and f denote the *solution space*, the *feasible set*, a *feasible solution* and a real-valued *objective function*, respectively. If $\mathscr{S}$ is a finite but large set, a *combinatorial optimization* problem is defined. If $\mathscr{S} = \mathbb{R}^n$, we refer to *continuous optimization*. A solution $x^* \in X$ is *optimal* if

$$f(x^*) \leq f(x), \ \forall x \in X.$$

An *exact algorithm* for problem (3.1), if one exists, finds an optimal solution x^*, together with the proof of its optimality, or shows that there is no feasible solution, i.e., $X = \emptyset$, or the solution is unbounded. Moreover, in practice, the time needed to do so should be finite (and not too long). For continuous optimization, it is reasonable to allow for some degree of tolerance, i.e., to stop when sufficient convergence is detected.

Let us denote $\mathscr{N}_k$, $(k = 1, \ldots, k_{max})$, a finite set of pre-selected neighborhood structures, and $\mathscr{N}_k(x)$ the set of solutions in the kth neighborhood of x. Most local search heuristics use only one neighborhood structure, i.e., $k_{max} = 1$. Often successive neighborhoods $\mathscr{N}_k$ are nested and may be induced from one or more metric (or quasi-metric) functions introduced into a solution space $\mathscr{S}$. An *optimal solution* x_{opt} (or global minimum) is a feasible solution where a minimum is reached. We call $x' \in X$ a *local minimum* of (3.1) with respect to $\mathscr{N}_k$ (w.r.t. $\mathscr{N}_k$ for short), if there is no solution $x \in \mathscr{N}_k(x') \subseteq X$ such that $f(x) < f(x')$. Metaheuristics (based on local search procedures) try to continue the search by other means after finding the first local minimum. VNS is based on three simple facts:

Fact 1 *A local minimum w.r.t. one neighborhood structure is not necessarily so for another;*

It has been observed that the best value for the parameter k_{max} is often 2 or 3. In addition, a maximum number of iterations between two improvements is typically used as the stopping condition. RVNS is akin to a Monte-Carlo method, but is more systematic (see, e.g., [81] where results obtained by RVNS were 30% better than those of the Monte-Carlo method in solving a continuous min-max problem). When applied to the p-Median problem, RVNS gave equally good solutions as the *Fast Interchange* heuristic of [102] while being 20 to 40 times faster [63].
(iii) The **Basic VNS** (BVNS) method [80] combines deterministic and stochastic changes of neighborhood. The deterministic part is represented by a local search heuristic. It consists in (1) choosing an initial solution x, (2) finding a direction of descent from x (within a neighborhood $N(x)$) and (3) moving to the minimum of $f(x)$ within $N(x)$ along that direction. If there is no direction of descent, the heuristic stops; otherwise it is iterated. Usually the steepest descent direction, also referred to as *best improvement*, is used. Also see Algorithm 2, where the best improvement is used in each neighborhood of the VND. This is summarized in Algorithm 5, where we assume that an initial solution x is given. The output consists of a local minimum, also denoted by x, and its value.

Function `BestImprovement`(x)
1 **repeat**
2 $\quad x' \leftarrow x$
3 $\quad x \leftarrow arg\min_{y \in N(x')} f(y)$
until $(f(x) \geq f(x'))$
return x

Algorithm 5: Best improvement (steepest descent) heuristic

As *Steepest descent* may be time-consuming, an alternative is to use a *first descent* (or *first improvement*) heuristic. Points $x^i \in N(x)$ are then enumerated systematically and a move is made as soon as a direction for descent is found. This is summarized in Algorithm 6.

Function `FirstImprovement`(x)
1 **repeat**
2 $\quad x' \leftarrow x; \quad i \leftarrow 0$
3 $\quad$ **repeat**
4 $\quad\quad i \leftarrow i+1$
5 $\quad\quad x \leftarrow arg\min\{f(x), f(x^i)\}, \quad x^i \in N(x)$
$\quad$ **until** $(f(x) < f(x')$ `or` $i = |N(x)|)$
until $(f(x) \geq f(x'))$
return x

Algorithm 6: First improvement (first descent) heuristic

The stochastic phase of BVNS (see Algorithm 7) is represented by the random selection of a point x' from the kth neighborhood of the shake operation. Note that

point x' is generated at random in Step 5 in order to avoid cycling, which might occur with a deterministic rule.

Function BVNS(x, k_{max}, t_{max})
1 $t \leftarrow 0$
2 **while** $t < t_{max}$ **do**
3 $k \leftarrow 1$
4 **repeat**
5 $x' \leftarrow$ Shake(x,k) // Shaking
6 $x'' \leftarrow$ BestImprovement(x') // Local search
7 $x,k \leftarrow$ NeighborhoodChange(x,x'',k) // Change neighborhood
until $k = k_{max}$
8 $t \leftarrow$ CpuTime()
return x

Algorithm 7: Basic VNS

Example. We illustrate the basic steps on a minimum k-cardinality tree instance taken from [72], see Fig. 3.1. The minimum k-cardinality tree problem on graph G (k-card for short) consists of finding a subtree of G with exactly k edges whose sum of weights is minimum.

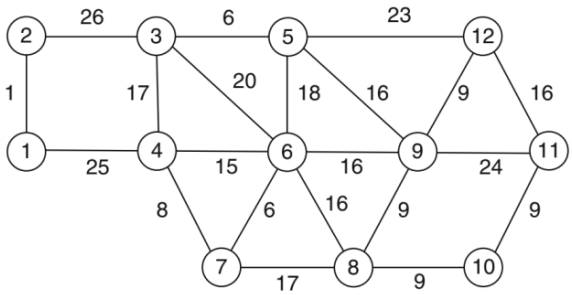

Fig. 3.1 4-Cardinality tree problem

The steps of BVNS for solving the 4-card problem are illustrated in Fig. 3.2. In Step 0 the objective function value, i.e., the sum of edge weights, is equal to 40; it is indicated in the right bottom corner of the figure. That first solution is a local minimum with respect to the edge-exchange neighborhood structure (one edge in, one out). After shaking, the objective function is 60, and after another local search, we are back to the same solution. Then, in Step 3, we take out 2 edges and add another 2 at random, and after a local search, an improved solution is obtained with a value of 39. Continuing in that way, the optimal solution with an objective function value equal to 36 is obtained in Step 8.

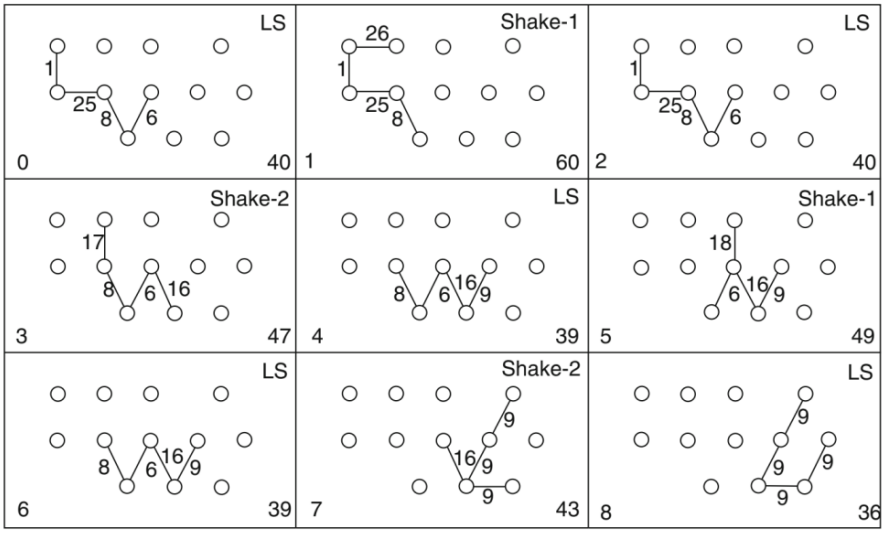

Fig. 3.2 Steps of the Basic VNS for solving 4-card tree problem

(iv) **General VNS.** Note that the local search step (line 6 in BVNS, Algorithm 7) may also be replaced by VND (Algorithm 2). This General VNS (VNS/VND) approach has led to some of the most successful applications reported in the literature (see, e.g., [1, 26–29, 31, 32, 39, 57, 66, 92, 93]). General VNS (GVNS) is outlined in Algorithm 8 below. Note that neighborhoods $N_1, \ldots, N_{l_{max}}$ are used in the VND step, while a different series of neighborhoods $N_1, \ldots, N_{k_{max}}$ apply to the Shake step.

Function GVNS $(x, \ell_{max}, k_{max}, t_{max})$
1 **repeat**
2 $k \leftarrow 1$
3 **repeat**
4 $x' \leftarrow$ Shake(x,k)
5 $x'' \leftarrow$ VND(x', ℓ_{max})
6 $x,k \leftarrow$ NeighborhoodChange(x,x'',k)
until $k = k_{max}$
7 $t \leftarrow$ CpuTime()
until $t > t_{max}$
return x

Algorithm 8: General VNS

3.3 Some Extensions

(i) The **Skewed VNS** (SVNS) method [62] addresses the problem of exploring valleys far from the incumbent solution. Indeed, once the best solution in a large region has been found it is necessary to go quite far to obtain an improved one. Solutions drawn at random in far-away neighborhoods may differ substantially from the incumbent, and VNS may then degenerate, to some extent, into a Multistart heuristic (where descents are made iteratively from solutions generated at random, and which is known to be inefficient). So some compensation for distance from the incumbent must be made, and a scheme called Skewed VNS (SVNS) is proposed for that purpose. Its steps are presented in Algorithms 9, 10 and 11. The `KeepBest`(x,x') function (Algorithm 9) in SVNS simply keeps the best of solutions x and x' The `NeighborhoodChangeS` function (Algorithm 10) performs the move and neighborhood change for the SVNS.

Function `KeepBest`(x,x')
1 **if** $f(x') < f(x)$ **then**
2 $\quad x \leftarrow x'$
return x

Algorithm 9: Keep best solution

Function `NeighborhoodChangeS`(x,x',k,α)
1 **if** $f(x') - \alpha\rho(x,x') < f(x)$ **then**
2 $\quad x \leftarrow x'; k \leftarrow 1$
else
3 $\quad k \leftarrow k+1$
return x,k

Algorithm 10: Neighborhood change for Skewed VNS

SVNS makes use of a function $\rho(x,x'')$ to measure the distance between the current solution x and the local optimum x''. The distance function used to define $\mathcal{N}_k$ could also be used for this purpose. The parameter α must be chosen to allow movement to valleys far away from x when $f(x'')$ is larger than $f(x)$ but not too much larger (otherwise one will always leave x). A good value for α is found experimentally in each case. Moreover, in order to avoid frequent moves from x to a close solution, one may take a smaller value for α when $\rho(x,x'')$ is small. More sophisticated choices for selecting a function of $\alpha\rho(x,x'')$ could be made through some learning process.

```
Function SVNS (x, k_max, t_max, α)
1  x_best ← x
2  repeat
3  |  k ← 1
4  |  repeat
5  |  |  x' ← Shake(x, k)
6  |  |  x'' ← FirstImprovement(x')
7  |  |  x, k ← NeighborhoodChangeS(x, x'', k, α)
8  |  |  x_best ← KeepBest (x_best, x)
   |  until k = k_max
9  |  x ← x_best
10 |  t ← CpuTime()
   until t > t_max
return x
```

Algorithm 11: Skewed VNS

(ii) The **Variable neighborhood decomposition search** (VNDS) method [63] extends the basic VNS into a two-level VNS scheme based upon decomposition of the problem. It is presented in Algorithm 12, where t_d is an additional parameter that represents the running time allowed for solving decomposed (smaller-sized) problems by Basic VNS (line 5).

```
Function VNDS (x, k_max1, t_max, t_d)
1  repeat
2  |  k ← 1
3  |  repeat
4  |  |  x' ← Shake (x, k); y ← x' \ x
5  |  |  y' ← BVNS(y, k_max2, t_d); x'' = (x' \ y) ∪ y'
6  |  |  x''' ← FirstImprovement(x'')
7  |  |  x, k ← NeighborhoodChange(x, x''', k)
   |  until k = k_max1
   until t > t_max
return x
```

Algorithm 12: Variable neighborhood decomposition search

For ease of presentation, but without loss of generality, we assume that the solution x represents a set of attributes. In Step 4 we denote by y a set of k solution attributes present in x' but not in x ($y = x' \setminus x$). In Step 5 we find the local optimum y' in the space of y; then we denote with x'' the corresponding solution in the whole space X ($x'' = (x' \setminus y) \cup y'$). We notice that exploiting some *boundary effects* in a new solution can significantly improve solution quality. That is why, in Step 6, the local optimum x''' is found in the whole space X using x'' as an initial solution. If this is time consuming, then at least a few local search iterations should be performed.

VNDS can be viewed as embedding the classical successive approximation scheme (which has been used in combinatorial optimization at least since the sixties, see, e.g., [48]) in the VNS framework. Let us mention here a few applications

```
Logical Function Accept (x, x', p)
1 for i = 0 to p do
2     if (f_i(x') < f_i(x)) then return TRUE
3     if (f_i(x') > f_i(x)) then return FALSE
4 return FALSE
```

Algorithm 15: Accept procedure with p secondary formulations

If Accept (x,x',p) is included in the LocalSearch subroutine of BVNS, then it will not stop the first time a non improved solution is found. In order to stop LocalSearch and thus claim that x' is a local minimum, x' should not be improved by any among the p different formulations. Thus, for any particular problem, one needs to design different formulations of the problem considered and decide the order in which they will be used in the Accept subroutine. Answers to those two questions are problem specific and sometimes not easy. The Accept (x,x',p) subroutine can obviously be added to the NeighborhoodChange and Shaking steps of BVNS from Algorithm 7 as well.

In [85], three evaluation functions, or acceptance criteria, within the Neighborhood Change step are used in solving the *Bandwidth Minimization Problem*. This min-max problem consists of finding permutations of rows and columns of a given square matrix to minimize the maximal distance of the nonzero elements from the main diagonal in the corresponding rows. Solution x may be represented as a labeling of a graph and the move from x to x' as $x \rightarrow x'$. Three criteria are used:

1. the bandwidth length $f_0(x)$ $(f_0(x') < f_0(x))$;
2. the total number of critical vertices $f_1(x)$ $(f_1(x') < f_1(x))$, if $f_0(x') = f_0(x)$;
3. $f_3(x,x') = \rho(x,x') - \alpha$, if $f_0(x') = f_0(x)$ and $f_1(x') = f_1(x)$. Here, we want $f_3(x,x') > 0$, because we assume that x and x' are sufficiently far from one another when $\rho(x,x') > \alpha$, where α is an additional parameter. The idea for a move to an even worse solution, if it is very far, is used within Skewed VNS. However, a move to a solution with the same value is only performed in [85] if its Hamming distance from the incumbent is greater than α.

In [86] a different mathematical programming formulation of the original problem is used as a secondary objective within the Neighborhood Change function of VNS. There, two combinatorial optimization problems on a graph are considered: the *Metric Dimension Problem* and *Minimal Doubly Resolving Set Problem*.

A more general VFS approach is given in [89], where the *Cutwidth Graph Minimization Problem* (CWP) is considered. CWP also belongs to the min-max problem family. For a given graph, one needs to find a sequence of nodes such that the maximum cutwidth is minimum. The cutwidth of a graph should be clear from the example provided in Fig. 3.3 for the graph with six vertices and nine edges shown in (a).

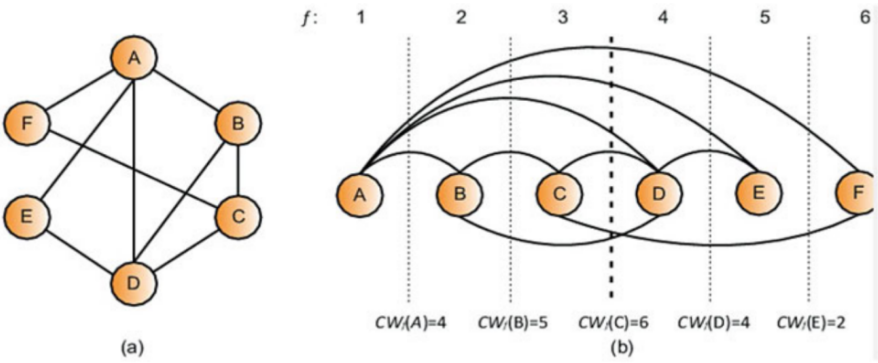

Fig. 3.3 Cutwidth minimization example as in [89]

Figure 3.3b shows an ordering x of the vertices of the graph in (a) with the corresponding cutwidth CW values of each vertex. It is clear that the CW represents the number of cut edges between two consecutive nodes in the solution x. The cutwidth value $f_0(x) = CW(x)$ of the ordering $x = (A,B,C,D,E,F)$ is equal to $f_0(x) = \max\{4,5,6,4,2\} = 6$. Thus, one needs to find an order x that minimizes the maximum cut-width value over all vertices.

Beside minimizing the bandwidth f_0, two additional formulations, denoted f_1 and f_2 , are used in [89], and implemented within a VND local search. Results are compared among themselves (Table 3.1) and with a few heuristics from the literature (Table 3.1), using the following usual data set:

- "*Grid*": This data set consists of 81 matrices constructed as the Cartesian product of two paths. They were originally introduced by Rolim et al. [94]. For this set of instances, the vertices are arranged on a grid of dimension width $\times$ height where width and height are selected from the set {3, 6, 9, 12, 15, 18, 21, 24, 27}.
- "*Harwell-Boeing*" (HB): This data set is a subset of the public-domain Matrix Market library.[1] This collection consists of a set of standard test matrices $M = (M_{ij})$ arising from problems in linear systems, least squares, and eigenvalue calculations from a wide variety of scientific and engineering disciplines. Graphs were derived from these matrices by considering an edge (i, j) for every element $M_{ij} \neq 0$. The data set is formed by the selection of the 87 instances were $n \leq 700$. Their number of vertices ranges from 30 to 700 and the number of edges from 46 to 41,686.

[1] Available at http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/.

Table 3.1 presents the results obtained with four different VFS variants, after executing them for 30 s over each instance. The column 'BVNS' of Table 3.1 represents a heuristic based on BVNS which makes use only of the original formulation f_0 of the CWP. VFS_1 denotes a BVNS heuristic that uses only one secondary criterion, i.e., f_0 and f_1. VFS_2 is equivalent to the previous one with the difference that now f_2 is considered (instead of f_1). Finally, the fourth column of the table, denoted as VFS_3, combines the original formulation of the CWP with the two alternative ones, in the way presented in Algorithm 15. All algorithms were configured with $k_{max} = 0.1n$ and start from the same random solution.

Table 3.1 Comparison of alternative formulations within 30 s for each test, by average objective values and % deviation from the best known solution

	BVNS	VFS_1	VFS_2	VFS_3
Avg.	137.31	93.56	91.56	90.75
Dev. (%)	192.44	60.40	49.23	48.22

Test are performed on "Grid" and "HB" data sets that contain 81 and 86 instances, respectively

It appears that significant improvements in solution quality are obtained when at least one secondary formulation is used in case of ties (compare e.g., 192.44% and 60.40% deviations from the best known solutions obtained by BVNS and VFS_1, respectively). An additional improvement is obtained when all three formulations are used in VFS_3.

Comparison of VFS_3 and state-of-the-art heuristics are given in Table 3.2. There, the stopping condition is increased from 30 s to 300 and 600 s for the first and the second set of instances, respectively. Besides average values and % deviation, the methods are compared based on the number of wins (the third row) and the total cpu time in seconds. Overall, the best quality results are obtained by VFS in less computing time.

Table 3.2 Comparison of VFS with the state-of-the-art heuristics over the "Grid" and "HB" data sets, within 300 and 600 s respectively

	81 'grid' test instances				86 HB instances			
	GPR [2]	SA [34]	SS [88]	VFS [89]	GPR [2]	SA [34]	SS [88]	VFS [89]
Avg.	38.44	16.14	13.00	12.23	364.83	346.21	315.22	314.39
Dev. (%)	201.81	25.42	7.76	3.25	95.13	53.30	3.40	1.77
#Opt.	2	37	44	59	2	8	47	61
CPU t (s)	235.16	216.14	210.07	90.34	557.49	435.40	430.57	128.12

3.5 Primal-Dual VNS

For most modern heuristics, the difference in value between the optimal solution and the obtained approximate solution is not precisely known. Guaranteed performance of the primal heuristic may be determined if a lower bound on the objective

function value can be found. To this end, the standard approach is to relax the integrality condition on the primal variables, based on a mathematical programming formulation of the problem. However, when the dimension of the problem is large, even the relaxed problem may be impossible to solve exactly by standard commercial solvers. Therefore, it seems to be a good idea to solve dual relaxed problems heuristically as well. In this way we get guaranteed bounds on the primal heuristic performance. The next difficulty arises if we want to get an exact solution within a branch-and-bound framework since having the approximate value of the relaxed dual does not allow us to branch in an easy way, for example by exploiting complementary slackness conditions. Thus, the exact value of the dual is necessary. A general approach to get both guaranteed bounds and an exact solution is proposed in [67], and referred as Primal-Dual VNS (PD-VNS). It is given in Algorithm 16.

Function `PD-VNS` (x, k_{max}, t_{max})
1 `BVNS` (x, k_{max}, t_{max}) // Solve primal by VNS
2 `DualFeasible`(x, y) // Find (infeasible) dual such that $f_P = f_D$
3 `DualVNS`(y) // Use VNS do decrease infeasibility
4 `DualExact`(y) // Find exact (relaxed) dual
5 `BandB`(x, y) // Apply branch-and-bound method

Algorithm 16: Basic PD-VNS

In the first stage, a heuristic procedure based on VNS is used to obtain a near optimal solution. In [67] it is shown that VNS with decomposition is a very powerful technique for large-scale simple plant location problems (SPLP) with up to 15,000 facilities and 15,000 users. In the second phase, the objective is to find an exact solution of the relaxed dual problem. Solving the relaxed dual is accomplished in three stages: (1) find an initial dual solution (generally infeasible) using the primal heuristic solution and complementary slackness conditions; (2) find a feasible solution by applying VNS to the unconstrained nonlinear form of the dual; (3) solve the dual exactly starting with the found initial feasible solution using a customized "sliding simplex" algorithm that applies "windows" on the dual variables, thus substantially reducing the problem size. On all problems tested, including instances much larger than those previously reported in the literature, the procedure was able to find the exact dual solution in reasonable computing time. In the third and final phase, armed with tight upper and lower bounds obtained from the heuristic primal solution in phase one and the exact dual solution in phase two, respectively, a standard branch-and-bound algorithm is applied to find an optimal solution of the original problem. The lower bounds are updated with the dual sliding simplex method and the upper bounds whenever new integer solutions are obtained at the nodes of the branching tree. In this way it was possible to solve exactly problem instances of sizes up to 7000 facilities $\times$7000 users, for uniform fixed costs, and 15,000 facilities $\times$15,000 users, otherwise.

3.6 VNS for Mixed Integer Linear Programming

The Mixed Integer Linear Programming (MILP) problem consists of maximizing or minimizing a linear function, subject to equality or inequality constraints and integrality restrictions on some of the variables. The mixed integer programming problem $(MILP)$ can be expressed as:

$$(MILP) \quad \begin{bmatrix} \min & \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \forall i \in M = \{1,2,\ldots,m\} \\ & x_j \in \{0,1\} & \forall j \in \mathscr{B} \\ & x_j \geq 0, \texttt{integer} & \forall j \in \mathscr{G} \\ & x_j \geq 0 & \forall j \in \mathscr{C} \end{bmatrix}$$

where the set of indices $N = \{1,2,\ldots,n\}$ is partitioned into three subsets $\mathscr{B}, \mathscr{G}$ and $\mathscr{C}$, corresponding to binary, general integer and continuous variables, respectively.

Numerous combinatorial optimization problems, including a wide range of practical problems in business, engineering and science, can be modeled as MILPs. Several special cases, such as knapsack, set packing, cutting and packing, network design, protein alignment, traveling salesman and other routing problems, are known to be NP-hard [46].

Many commercial solvers such as CPLEX [71] are available for solving MILPs. Methods included in such software packages are usually of the branch-and-bound (B&B) or of branch-and-cut (B&C) types. Basically, those methods enumerate all possible integer values in some order, and prune the search space for the cases where such enumeration cannot improve the current best solution.

3.6.1 Variable Neighborhood Branching

The connection between local search based heuristics and exact solvers may be established by introducing the so called *local branching constraints* [43]. By adding just one constraint into (MILP), as explained below, the kth neighborhood of (MILP) is defined. This allows the use of all local search based metaheuristics, such as Tabu search, Simulating annealing, VNS etc. More precisely, given two solutions x and y of (MILP), the distance between x and y is defined as:

$$\delta(x,y) = \sum_{j \in \mathscr{B}} | x_j - y_j |.$$

at line 24. There are four different outputs from subroutine `MIPSOLVE` provided by variable *stat*. They are coded in lines 11–20. The shaking step also uses the MIP solver. It is presented in the loop that starts at line 25.

3.6.2 VNDS Based Heuristics for MILP

It is well known that heuristics and relaxations are useful for providing upper and lower bounds on the optimal value of large and difficult optimization problems. A hybrid approach for solving 0-1 MILPs is presented in this section. A more detailed description may be found in [51]. It combines variable neighborhood decomposition search (VNDS) [63] and a generic MILP solver for upper bounding purposes, and a generic linear programming solver for lower bounding. VNDS is used to define a variable fixing scheme for generating a sequence of smaller subproblems, which are normally easier to solve than the original problem. Different heuristics are derived by choosing different strategies for updating lower and upper bounds, and thus defining different schemes for generating a series of subproblems. We also present in this section a two-level decomposition scheme, in which subproblems created according to the VNDS rules are further divided into smaller subproblems using another criterion, derived from the mathematical formulation of the problem.

3.6.2.1 VNDS for 0-1 MILPs with Pseudo-Cuts

Variable neighborhood decomposition search is a two-level variable neighborhood search scheme for solving optimization problems, based upon the decomposition of the problem (see Algorithm 12). We discuss here an algorithm which solves exactly a sequence of reduced problems obtained from a sequence of linear programming relaxations. The set of reduced problems for each LP relaxation is generated by fixing a certain number of variables according to VNDS rules. That way, two sequences of upper and lower bounds are generated, until an optimal solution of the problem is obtained. Also, after each reduced problem is solved, a pseudo-cut is added to guarantee that this subproblem is not revisited. Furthermore, whenever an improvement in the objective function value occurs, a local search procedure is applied in the whole solution space to attempt a further improvement (the so-called *boundary effect* within VNDS). This procedure is referred to as VNDS-PC, since it employs VNDS to solve 0-1 MILPs, while incorporating pseudo-cuts to reduce the search space [51].

If $J \subseteq \mathscr{B}$, we define the partial distance between x and y, relative to J, as $\delta(J,x,y) = \sum_{j\in J} | x_j - y_j |$. Obviously we have $\delta(\mathscr{B},x,y) = \delta(x,y)$). More generally, let $\bar{x}$ be an optimal solution of LP(P), the LP relaxation of the problem P considered (not necessarily MIP feasible), and $J \subseteq B(\bar{x}) = \{j \in N \mid \bar{x}_j \in \{0,1\}\}$ an arbitrary subset of indices. The partial distance $\delta(J,x,\bar{x})$ can be linearized as follows:

$$\delta(J,x,\bar{x}) = \sum_{j\in J} [x_j(1-\bar{x}_j) + \bar{x}_j(1-x_j)].$$

Let X be the solution space of problem P. The neighborhood structures $\{\mathscr{N}_k \mid k = k_{min},\ldots,k_{max}\}$, $1 \le k_{min} \le k_{max} \le p$, can be defined knowing the distance $\delta(\mathscr{B},x,y)$