THE DEVELOPMENT AND APPLICATION OF METAHEURISTICS FOR PROBLEMS IN GRAPH THEORY: A COMPUTATIONAL STUDY

A thesis submitted for the degree of

Doctor of Philosophy

by

Sergio Consoli

School of Information Systems, Computing and Mathematics

Brunel University

20^{th} November 2008

This thesis has been supervised by:

Prof. Kenneth Darby-Dowman

Examiner committee:

Dr. Eleni Hadjiconstantinou
Dr. Steven Noble
Prof. Said Salhi

Brunel
UNIVERSITY
WEST LONDON

The work described in this thesis has been carried out at the School of Information Systems, Computing and Mathematics at Brunel University, West London, as part of the E.U. Marie Curie EST-FP6 project NET-ACE (number MEST-CT-2004-006724).

The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvellous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity.

Albert Einstein

This thesis is dedicated, with deepest love and everlasting respect, to my parents Carmelo and Concetta, my aunt Mariagrazia, my brother Fabrizio, and my beloved half Lucia. Without their love, encouragement, and constant support I could not have reached this stage.

Acknowledgements

First and foremost I would like to express my most sincere gratitude to Prof. Kenneth Darby-Dowman for introducing me to the fields of operations research, graph theory, and combinatorial optimization, for his elaborate feedback and tremendous technical contributions, for his inspiration and competent support that led me to the realization of this work. There are simply no words to explain the encouragement and the affection I have received from him.

I would like to thank the School of Information Systems, Computing and Mathematics at Brunel University, particularly all the colleagues of the NET-ACE and CARISMA research groups, for the extensive feedback and for being always nice and friendly during these years. I acknowledge the director of NET-ACE, Prof. Geoff Rodgers, and the director of CARISMA, Prof. Gautam Mitra, for bearing with me with a support and understanding during my research years. This work was supported by an E.U. Marie Curie Fellowship for Early Stage Researcher Training (EST-FP6) under grant number MEST-CT-2004-006724 at Brunel University.

Many thanks go also to the DEIOC department of the University of La Laguna, Tenerife, and the User Experiences Group at Philips Research Eindhoven, where I had the opportunity to undertake research placements during my Ph.D. course. For helping me in the resolution of the problems that arise from addressing the combinatorial optimization field, and for being always kind in giving an answer to my questions and doubts, special thanks go to Prof. José Andrés Moreno-Pérez from the University of La Laguna, and Dr. Steffen Pauws, Dr. Jan Korst, and Dr. Gijs Geleijnse from Philips Research Eindhoven.

Many thanks also to Dr. Nenad Mladenović from Brunel University both for his initial encouragement of the research and for his continued wise counsel, involvement, and enthusiasm throughout my work. These researchers are also particularly thanked because they helped me in writing the manuscripts that form the greatest part of this thesis and that led me to submit papers for publication in international journals of the field.

There are a few special thanks that I would like to give. Firstly to my father, my mother, and my aunt Mariagrazia for the love, help and precious advice they constantly gave to me. I wouldn't be here without their support and affection. Particular thanks are for my brother Fabrizio who encouraged, with enthusiasm and affection, my journey towards the field of science and research, and, making use of his talent, gave me always many competent ideas and ample feedback, and challenged me with incredible insight and logic. Next I want to thank my friends Pierpaolo Vivo and Elisa Garimberti, for the ample feedback, reading preliminary versions, commenting, and discovering mistakes. My biggest thank go to my beloved half Lucia for her invaluable encouragement, psychological support, and patient understanding during the years of my research. The end of my Ph.D. course coincides with the beginning of a new stage of our lives together. I deeply love her.

For all the others who have been close to me during these years I have again the same but sincere words: thanks to everyone.

Abstract

It is known that graph theoretic models have extensive application to real-life discrete optimization problems. Many of these models are NP-hard and, as a result, exact methods may be impractical for large scale problem instances. Consequently, there is a great interest in developing efficient approximate methods that yield near-optimal solutions in acceptable computational times. A class of such methods, known as metaheuristics, have been proposed with success.

This thesis considers some recently proposed NP-hard combinatorial optimization problems formulated on graphs. In particular, the *minimum labelling spanning tree problem*, the *minimum labelling Steiner tree problem*, and the *minimum quartet tree cost problem*, are investigated. Several metaheuristics are proposed for each problem, from classical approximation algorithms to novel approaches. A comprehensive computational investigation in which the proposed methods are compared with other algorithms recommended in the literature is reported. The results show that the proposed metaheuristics outperform the algorithms recommended in the literature, obtaining optimal or near-optimal solutions in short computational running times. In addition, a thorough analysis of the implementation of these methods provide insights for the implementation of metaheuristic strategies for other graph theoretic problems.

Related publications

• S. Consoli, K. Darby-Dowman, N. Mladenović, and J. A. Moreno-Pérez (2008). Greedy randomized adaptive search and variable neighbourhood search for the minimum labelling spanning tree problem. *European Journal of Operational Research*, accepted for publication. doi:10.1016/j.ejor.2008.03.014.

• S. Consoli, K. Darby-Dowman, N. Mladenović, and J. A. Moreno-Pérez (2008). Solving the minimum labelling spanning tree problem using hybrid local search. Submitted to *Optimization Methods and Software*, Special Issue EURO XXII conference.

• S. Consoli, J. A. Moreno-Pérez, K. Darby-Dowman, and N. Mladenović (2008). Discrete particle swarm optimization for the minimum labelling Steiner tree problem. In N. Krasnogor, G. Nicosia, M. Pavone, and D. Pelta, editors, *Nature Inspired Cooperative Strategies for Optimization*, volume 129 of *Studies in Computational Intelligence*, pages 313-322. Springer-Verlag, New York.

• S. Consoli, J. A. Moreno-Pérez, K. Darby-Dowman, and N. Mladenović (2008). Discrete particle swarm optimization for the minimum labelling Steiner tree problem. Submitted to *Natural Computing*, Special Issue NICSO conference.

• S. Consoli, K. Darby-Dowman, N. Mladenović, and J. A. Moreno-Pérez (2008). Variable neighbourhood search for the minimum labelling Steiner tree problem. *Annals of Operations Research*, accepted for publication.

• S. Consoli, K. Darby-Dowman, G. Geleijnse, J. Korst, and S. Pauws (2008). Heuristic approaches for the quartet method of hierarchical clustering. Submitted to *IEEE Transactions on Knowledge and Data Engineering*.

Contents

Summary of Abbreviations

ACO	:	Ant Colony Optimization
AS	:	Ant System
DPSO	:	Discrete Particle Swarm Optimization
EDA	:	Estimation of Distribution Algorithm
EXACT	:	Exact Method
GA	:	Genetic Algorithm
GRASP	:	Greedy Randomized Adaptive Search Procedure
GLS	:	Guided Local Search
HYBRID	:	hybrid local search method
MA	:	Memetic Algorithm
max-CPU-time	:	maximum allowed CPU time
MGA	:	Modified Genetic Algorithm
MLST	:	minimum labelling spanning tree
MLSteiner	:	minimum labelling Steiner tree
MQC	:	maximum quartet consistency
MQTC	:	minimum quartet tree cost
MVCA	:	Maximum Vertex Covering Algorithm
NCD	:	Normalized Compression Distances
PILOT	:	Pilot Method
PSO	:	Particle Swarm Optimization
QGA	:	Quantum-inspired Genetic Algorithm
RHC	:	Randomized Hill Climbing
RVNS	:	Reduced Variable Neighbourhood Search
SA	:	Simulated Annealing
SS	:	Scatter Search
SVNS	:	Skewed Variable Neighbourhood Search
TS	:	Tabu Search
VNDS	:	Variable Neighbourhood Decomposition Search
VNS	:	Variable Neighbourhood Search
WWW	:	World Wide Web

List of Figures

List of Tables

My arguments will be open to all,
and may be judged of by all.

PUBLIUS

Chapter 1

Introduction

Combinatorial optimization (CO) is the general name given to the problem of finding the best solution out of a very large, but finite, number of possible solutions. It is one of the youngest and most active areas of discrete mathematics and operational research, related to computer science, algorithm theory, and computational complexity theory, and sitting at the intersection of several fields, including artificial intelligence, computer science, and software engineering. Its increasing interest arises from the fact that a large number of scientific and industrial problems can be formulated as abstract combinatorial optimization problems, through graphs and/or (integer) linear programs. To solve problems arising in the fields of transportation and telecommunications, the operational research analyst often has to use techniques that were first designed to solve classical combinatorial problems related to graph theory (Avis et al., 2005). For example, many combinatorial optimisation problems have been formulated on graphs, where the possible solutions are "optimal" spanning trees with respect to some measure. Typical measures include the total length or the diameter of the tree. Many real-life combinatorial optimisation problems belong to this class of problems and consequently there is a large and growing interest in both theoretical and practical aspects of the subject. Examples of such problems are network flow problems (e.g. shortest path problem, minimum spanning tree problem, maximum/minimum cost flow problem), matching problems (e.g. maximum cardinality matching problem, job assignment problem, maximum/minimum weight

matching problem), matroids (e.g. maximization/minimization problem for independent systems, matroid intersection problem, matroid partitioning problem), set covering problem, colouring problems (e.g. vertex-colouring problem, edge-colouring problem), max 3-sat problem, knapsack problem, bin-packing problem, network design problems (e.g. survivable network design problem, Steiner tree problem), and travelling salesman problem.

A combinatorial optimization problem $P = (S, f)$ may be specified as follows:
- A set of variables $X = \{x_1, x_2, \ldots, x_n\}$;
- Variable domains $D_1, \ldots, D_n$;
- Constraints among variables;
- An objective function f to be minimized (or maximized), where
$f : D_1 \times \ldots \times D_n \rightarrow \Re^+$.

The set of all possible feasible solutions is

$$S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} | v_i \in D_i \text{ and } s \text{ satisfies the constraints}\}, \quad (1.1)$$

and is usually called the *search (or solution) space*, as each element of the set can be seen as a candidate solution. To solve a combinatorial optimization problem means to find a solution $s^* \in S$ with minimum (or maximum) objective function value; that is, $f(s^*) \leq f(s)$, $\forall s \in S$. s^* is called a "globally optimal solution" of (S, f). Let the set $S^* \subseteq S$ be the "set of globally optimal solutions".

Some of these problems have polynomial-time ("efficient") algorithms, and they are said to be in the complexity class **P**. However, most of them are **NP-hard**, meaning that no algorithm with a number of steps polynomial in the size of the problem instances is known to exist, and it is not possible to guarantee, in general, that an exact solution to the problem can be found within an acceptable timeframe. For more details on the concepts of P and NP complexity, see Appendix A.

In practice, combinatorial optimization problems are often large-scale and difficult to solve. Thus, much attention has been given to studying computational complexity and algorithm design with a view to developing efficient solution procedures. The *No-Free-Lunch-Theorem* (Wolpert and Macready, 1997) states that, if an optimization algorithm performs well on a particular sub-class of combinatorial problems, having been designed to exploit the specific characteristics of that

sub-class, then it may have degraded performance on other combinatorial problems not belonging to that sub-class. The theorem is used as an argument against using generic searching algorithms (e.g. Genetic Algorithms and Simulated Annealing) without exploiting as much domain knowledge as possible. Alternatively, the theorem establishes that a general-purpose universal optimization strategy is not possible, and the only way one strategy can outperform another is when it is specially adapted to the problem under consideration (Ho and Pepyne, 2002).

Combinatorial optimization algorithms are classified as *complete (or exact) algorithms* and *approximate algorithms*. Complete strategies are guaranteed to find, for every instance of a specified combinatorial problem of finite size, an optimal solution in bounded time (with proof of its optimality), while in approximate methods, the guarantee of finding an optimal solution is sacrificed for the sake of getting good solutions in a significantly reduced amount of time. By considering the knapsack problem, for example, it is easy to understand the difficulty of finding an optimal solution. Suppose a hitchhiker has to fill up his knapsack by selecting, from among various possible objects, those that will give him maximum comfort: these and many other examples of knapsack problems can be mathematically formulated by numbering the objects from 1 to n, and introducing a vector of binary variables $x_j = (j = 1, \ldots, n)$ having the following meaning:

$$x_j = \begin{cases} 1 & \text{if object } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

Then, if p_j is a measure of the comfort given by object j, w_j its size, and c the size of the knapsack, the problem will be to select, from among all binary vectors x satisfying the constraint

$$H_b = \sum_{j=1}^{n} w_j x_j \leq c, \tag{1.3}$$

the one which maximizes the objective function f:

$$\max f = \max \sum_{j=1}^{n} p_j x_j. \tag{1.4}$$

There are many applications of the knapsack model. For example, suppose an investment of up to c dollars is to be made in one or more of n possible

investments. Let p_j be the profit expected from investment j, and w_j the required investment. It is self-evident that the optimal solution of the knapsack problem above will indicate the best possible choice of investments. A naive approach to solve the knapsack problem would be to examine all possible binary vectors x, selecting the best of those that satisfy the constraint. A full enumeration consists of 2^n vectors and thus, for a computer with a clock frequency of 3.6 GHz ($3.6 \cdot 10^9$ instructions per second, i.e. 1 instruction in $0.28 \cdot 10^{-9}$ sec), a lower bound (LB) of the time to compute the 2^n vectors is given by:

$$LB = 2^n \cdot (0.28 \cdot 10^{-9}) \text{ sec} = \frac{1}{365 \cdot 24 \cdot 3600} \cdot 0.28 \cdot 2^n \cdot 10^{-9} \text{ years.} \qquad (1.5)$$

For example, with $n = 60$, $LB \approx 10$ years, with $n = 61$, $LB \approx 20$ years, and with $n = 65$, LB almost 4 centuries!

The complexity of many combinatorial problems arising in operational research and other fields is such that an exact solution may not be found within a reasonable time. Moreover, in some cases where a problem admits a polynomial algorithm, the power of this polynomial may be so large that many realistic instances cannot be solved in reasonable time (these are also called long-term problems in the complexity class P). There are also problems that are known to have a polynomial time algorithm but no-one knows what the algorithm actually is. In all these contexts, making use of exact algorithms to reach optimality may be impractical, and approximate techniques need to be used in order to provide good feasible solutions. In the last 20 years, a new kind of approximate algorithms, commonly called *metaheuristics*, have emerged in this class, which basically try to combine heuristics in high level frameworks aimed at efficiently and effectively exploring the search space. Metaheuristics perform intelligent searches in the search space, starting with one or more candidate solutions, and improving them by means of intense searches in promising areas and by using diversification mechanisms for moving towards attractive areas.

The research reported in this thesis focusses on metaheuristic techniques applied to problems in graph theory. The aim of the thesis is twofold. On the one hand, it seeks to bring together, in a systematic and consistent way, several features of different metaheuristic techniques. Classical and novel metaheuristics are presented in Chapter 2. This chapter covers many theoretical and practical

aspects of metaheuristics, outlining their main concepts and components, similarities and differences, advantages and disadvantages.

Subsequent chapters of the thesis address some recent and relevant combinatorial optimization problems formulated on graphs, and present suitable metaheuristics used to attain near-optimal solutions. These problems constitute some new and intriguing research areas, and are able to model and describe many real-world problems.

The first problem addressed is the *minimum labelling spanning tree problem*, discussed and examined in Chapter 3 and based on (Consoli et al., 2008b) and (Consoli et al., 2008c). In this chapter, some new metaheuristics for the problem are proposed. Some nonparametric statistical tests are performed to compare the performance of the proposed heuristics with that of the other algorithms recommended in the literature. A comparison with the results provided by an exact approach is also presented.

A similar study is presented in Chapter 4 for the *minimum labelling Steiner tree problem*, another graph problem related to the minimum labelling spanning tree problem and to the well-known Steiner tree problem. This chapter is based on (Consoli et al., 2008d), (Consoli et al., 2008e), and (Consoli et al., 2008f). Several effective metaheuristics are proposed, evaluated, and compared to the best performing approaches in the literature, with respect to the quality of their solutions and the computational running times.

Finally, Chapter 5 deals with the *quartet method* of hierarchical clustering. This chapter is based on (Consoli et al., 2008a). Because the quartet method is based on an NP-hard graph optimization problem, called *minimum quartet tree cost problem*, any practical approach to obtain or approximate the optimal solutions requires heuristics. Thus, some new metaheuristic approaches are proposed and discussed in depth, showing the importance and the potential of these approaches to deal with complex graph problems arising in real-world applications. Furthermore, the performance of the proposed algorithms is tested through extensive computational experiments and comparison with other approaches in the literature.

This thesis is intended to provide the communities of both researchers and practitioners with a broadly applicable, up to date coverage of metaheuristic

methodologies that have proven to be successful in a wide variety of graph theoretic models, and that hold particular promise for success in the future. The metaheuristics used to solve the graph problems reported in this thesis serve as illustrations in showing the importance and the potential of metaheuristic approaches to deal with these classes of problems. In addition, thorough analysis of the implementation of these methods provides insights for the implementation of metaheuristic strategies for other complex graph problems. With this thesis, the author hopes to encourage an even wider adoption of metaheuristic methods for solving graph problems, and to stimulate research that may lead to additional innovations in metaheuristic procedures.

Consider your origin; you were not born to live like brutes, but to follow virtue and knowledge.

DANTE ALIGHIERI

Chapter 2

Metaheuristics: foundations and classification

Since the early years of operational research (OR), there has been much interest in combinatorial optimization (CO) problems formulated on graphs and their practical applications (Avis et al., 2005). Most of these problems are NP-hard (Appendix A). Thus there is a need for heuristics and approximate solution approaches with performance guarantees. Indeed, the goal of approximate methods is to find "quickly" (reasonable run-times), with "high" probability, provable "good" solutions (low error from the real optimal solution). This chapter briefly outlines the components, foundations, advantages and disadvantages of different metaheuristic approaches from a conceptual point of view, in order to analyse their similarities and differences. In Section 2.1 the basic concepts of metaheuristics are outlined, allowing different kinds of classification between them. The two very significant forces of intensification and diversification, that mainly determine the behaviour of a metaheuristic, are also pointed out. In Section 2.2 and Section 2.3, the most important single-solution and population-based metaheuristics are presented in order to analyse their components, similarities and differences, from conceptual and practical point of views. Section 2.4 concludes by exploring the importance of hybridization and integration of metaheuristics. For a survey on the basic concepts of metaheuristics and combinatorial optimization, the reader is referred to (Voß et al., 1999; Glover and Kochenberger, 2003; Gendreau and Potvin, 2005).

2.1 Main concepts on metaheuristics

In recent years, there have been significant advances in the theory and application of metaheuristics to the approximate solution of hard optimization problems. The term *metaheuristic* derives from the composition of two Greek words: "Heuristic" (from the verb heuriskein) that means "to find"; and the suffix "Meta" that means "beyond, in an upper level". Before this term was largely adopted, metaheuristics were often called modern heuristics (V. J. Rayward-Smith, 1996). This family includes, but it is not limited to, Simulated Annealing (SA), Tabu Search (TS), Greedy Randomized Adaptive Search Procedure (GRASP), Iterated Local Search (ILS), Variable Neighbourhood Search (VNS), Guided Local Search (GLS), Genetic Algorithms (GAs), Quantum-inspired Genetic Algorithms (QGAs), Estimation of Distribution Algorithms (EDAs), Scatter Search (SS), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO).

As Voß et al. (1999) state, "A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single-solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method". Before going into the details of such a statement, it is important to clarify the concepts of *diversification* and *intensification* used in metaheuristics. The first term means the exploration of the search space while the latter one refers to the exploitation of the accumulated search-experience. When the search process starts, it needs to compute the value of different points in the search domain in order to find promising areas (diversification). Then the algorithm needs to investigate promising zones to find the local-optimum (intensification). The best local optimum found in the different areas will be the candidate solution, hoping to be as near as possible to the optimum that the algorithm is looking for. The terms "diversification" and "intensification" are mainly used in methods based on the concept of memory, such as Tabu Search. Conversely the terms "exploration" and "exploitation" are used in strategies that do not require explicit usage of memory, such as in GRASP. Finding a good balance between diversification (exploration) and intensification (exploitation) is

essential for a metaheuristic in order to quickly identify regions in the search space with high-quality solutions, without wasting too much time in regions with a low quality.

Intensification and diversification are not contradictory options, but each feature contains aspects of the other. Balancing properly these two strengths is a crucial issue in metaheuristics, and so many techniques have been proposed in recent years with this intent. These techniques are based sometimes on intuition and experience, and at other times on theoretically or empirically derived principles. In this context, both problem specific knowledge and a solid understanding of the properties and characteristics of the different metaheuristics are crucial for achieving peak performance and robustness.

Metaheuristics can be classified in different ways depending on the specific point of view of interest (Table 2.1). The main classification consists of considering *single-solution (or single-point) methods*, such as Tabu Search and Simulated Annealing, and *population-based methods*, such as Genetic Algorithms and Ant Colony Optimization. Often, single-solution methods are also called *trajectory methods*, because they work on a single solution at each time-step describing a curve (trajectory) in the search space during the progress of the search. On the other hand, population-based metaheuristics compute simultaneously a set of points at each time-step of the search process, describing the evolution of an entire population in the search domain.

A special class of single-point metaheuristics consists of *explorative methods*. Given a candidate solution, these strategies search for local minima by restricting the search process to a neighbourhood of the candidate solution. Here the neighbourhood of a current solution s is defined as a function $N(s) : S \rightarrow 2^s$, which assigns to every $s \in S$ a set of neighbourhoods $N(s) \subseteq S$, where S is the search space. With the introduction of a neighbourhood structure, it is possible to define the concept of *locally minimal solution* (or *local minimum*) with respect to a neighbourhood structure $N(\cdot)$, as a solution $\hat{s}$ such that $\forall s \in N(\hat{s}) \rightarrow f(\hat{s}) \leq f(s)$.

Table 2.1: Main classification of metaheuristics

		METAHEURISTICS										
		SA	TS	GRASP	ILS	VNS	GLS	GAs	QGAs	EDAs	SS	ACO	PSO
H	single-solution	✓	✓	✓	✓	✓	✓						
E	population-based							✓	✓	✓	✓	✓	✓
U	explorative: static neighbourhood			✓	✓		✓				✓		
C	explorative: dynamic neighbourhood		✓			✓							✓
R	memory-less	✓		✓		✓							
I	memory-usage		✓		✓		✓	✓	✓	✓	✓	✓	✓
S	nature-inspired							✓	✓			✓	✓
T	evolutionary							✓	✓	✓	✓		✓
I	static objective function	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
C	dynamic objective function						✓						
	local search	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓
	constructive search			✓								✓	

A further classification of explorative methods can be made by considering the *neighbourhood structure* that is "explored" during the search process. Some metaheuristics work on a *single (or static) neighbourhood structure*, meaning that the fitness landscape topology does not change in the course of the search process. Conversely, other metaheuristics use a set of different neighbourhood structures (*dynamic neighbourhood structures*). These methods diversify the search process by swapping between the neighbourhoods considered, allowing the exploration of different areas of the search space. A typical example is given by Iterative Local Search (ILS) and Variable Neighbourhood Search (VNS).

Another important feature in the classification of metaheuristics is the use of memory during the search history, because it is one of the fundamental elements of a powerful metaheuristic. In *memory-less algorithms* the next state depends only on the information accumulated in the current state of the search process, as a Markov process, while in *memory-usage algorithms* there is the use of a short-term and/or long-term memory. Usually, the first keeps track of recently visited solutions (moves), while the second is concerned with the storage of information about the entire search process.

Metaheuristics taking inspiration from nature and natural systems for the solution of complex problems are also called *nature-inspired algorithms*. Biological and natural processes have always been a source of inspiration for computer science and information technology in many real-world applications. Its well known that biological entities, from single cell organisms - like bacteria - to humans, often engage in a rich repertoire of social interaction that could range from altruistic cooperation to open conflict. One specific kind of social interaction is cooperative problem solving, where a group of autonomous entities work together to achieve a certain goal. Examples of nature-inspired algorithms include, but are not limited to, Particle Swarm Optimization and Ant Colony Optimization.

Population-based metaheuristics that are inspired by the Darwinian evolution theory (Darwin, 1859), belong to the class of *evolutionary algorithms*. The main idea consists of the survival of the best element in natural evolution processes. The field of natural evolution applied to optimization algorithms is at a stage of tremendous growth. There are currently three well-defined paradigms, which have served as the basis for much of the research in this field: Genetic Algorithms

(GAs), Evolution Strategies (ES), and Evolutionary Programming (EP). Each of these emphasizes a different aspect of natural evolution. In general, they have foundation on the following evolutionary operators: *recombination or crossover*, which recombines two or more individuals (ancestors) to produce new individuals (children); *modification or mutation*, which causes a self-adaptation of individuals; *selection of individuals based on their fitness*, where fitness is defined as a value of an objective function or some measure of the quality of solutions, which is the driving force in evolutionary algorithms. Individuals with a high fitness have a high probability to be chosen as members of the next population (or as parents for the population of new individuals). This is analogous to the principle of survival of the fittest in natural evolution, i.e. the capability of nature to adapt itself to a changing environment.

Metaheuristics can also be classified according to the way they make use of the objective function. If, during the search, the objective function is altered by trying to incorporate information collected during the search process (for example to escape from local minima), then the metaheuristic is said to have a *dynamic objective function*, as with the Guided Local Search (GLS). Techniques that keep the objective function as it is given by the problem belong to the class of metaheuristic with a *static objective function*.

Finally, metaheuristics can also be classified as *local search methods* and *constructive search methods*. A constructive search method builds a solution at each step, simply by adding components to the solution of the previous step, until the constraints are satisfied. GRASP is a typical example of a metaheuristic belonging to this class. These methods are usually faster than local search methods, but they also tend to be of lower quality. A local search method tries to replace the current solution at each step with a "better" one in a neighbourhood of the current solution. The concept in local search is simple: given a solution s and an objective function $f(\cdot)$, every "move" from the current solution s to a candidate solution s' is only performed if the objective function value $f(s')$ is smaller than the value given by the current solution $f(s)$ (in the case of a minimization problem). In this context, a move from the solution s is defined as the choice of a solution s' from the neighbourhood $N(s)$, that is $s' \in N(s)$.

The simplest local search method consists of an *iterative search*, and it is used often in conjunction with other metaheuristics. The algorithm is specified in Algorithm 2.1. Iterative search starts from a solution $s \in S$ (e.g. generated at ran-

Algorithm 2.1: Iterative search method

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define a static neighbourhood structure $N(\cdot)$;
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;

```
begin
    while termination conditions do
        s ← Generate-Solution();
        Find an improved solution in N(s): s ← Improve(N(s));
        if f(s) < f(s') then
            Move s' ← s;
        end
    end
    ⇒ Return(s').
end
```

dom). Then, the procedure *Improve*$(N(s))$ tries to find a better solution within the neighbourhood $N(s)$ of the current solution s. Therefore, iterative search belongs also to the class of explorative methods. The procedure *Improve*$(N(s))$ can be either a first improvement procedure or a best improvement procedure. In the first case, it scans the neighbourhood $N(s)$ and chooses the first solution that is better than the current solution s. In the second case, it exhaustively explores the neighbourhood $N(s)$ and returns the solution with the lowest objective function value. Both methods stop at local minima. Therefore, their performance strongly depends on the definition of the search space S, the objective function $f(\cdot)$, and the neighbourhood structure $N(\cdot)$. If the new improved solution s is better than the best solution to date s', the algorithm moves the current solution s to s' (i.e. $s' \leftarrow s$). The algorithm starts again with the same procedure, and it halts when some user termination conditions are satisfied. Possible termination conditions are the maximum allowed CPU time, the maximum number of iterations, or the maximum number of iterations without improvements. The best solution to date s' forms the output of the procedure. The effectiveness of iterative search tends to be highly unsatisfactory for many combinatorial optimization problems,

because it often becomes trapped in local minima. Therefore, rather than as stand-alone algorithm, iterative search is usually used as additional component in other metaheuristics.

Summarizing, metaheuristics are strategies, approximate and usually non deterministic, that guide the search process to efficiently explore the search space in order to find near-optimal solutions, using techniques which range from simple local search procedures to complex learning processes. They are not problem-specific, can incorporate mechanisms to avoid "traps" (local optima), may use domain-specific knowledge to explore the most promising areas, and finally they can memorize the search experience in order to guide the future search (long/short-time form of memory). A rigourous classification of metaheuristics can not be performed, because many methods may fit several classes at the same time, and also because in many cases it is not possible to clearly attribute an algorithm to one of the classes specified above. However, the classification of metaheuristics in single-solution and population-based methods permits a clear distinction between these kinds of algorithms. In the following sections, the most important single-point and population-based methods will be presented from a conceptual point of view, in order to analyse their components, similarities and differences, advantages and disadvantages.

2.2 Single-solution metaheuristics

Single-solution metaheuristics, also named trajectory methods, are so called because the search process designs a trajectory in the search space, starting from an initial state and dynamically adding a new solution to the curve in each discrete time-step. So, this process can be seen as the evolution in time of a discrete dynamical system in the state space. The generated trajectory is useful because it provides information about the behaviour of the algorithm and its dynamics in order to choose the most effective method to solve the problem instance under consideration.

The system dynamics are the result of the combination of algorithms (i.e. chosen strategy), problem representation (i.e. definition of the search landscape) and problem instance. Trajectory shape depends on the strategy used. Simple

algorithms generate a trajectory composed of a *transient phase* followed by an *attractor* (a fixed point, a cycle, or a complex attractor). Advanced algorithms generate more complex trajectories comprising more different phases, representing the dynamic tuning between diversification and intensification during the search process. These continuous oscillations provide alternate phases in the designed trajectory, trying to find an optimal balance between these fundamental strengths. The main single-solution metaheuristics are described below.

2.2.1 Simulated Annealing

Simulated Annealing (SA) is possibly the oldest probabilistic local search method for global optimization problems, and one of the first to clearly provide a way to escape from local traps. It was independently invented by Kirkpatrick et al. (1983) and by Cerny (1985). The SA metaheuristic performs a stochastic search of the neighbourhood space. In the case of a minimization problem, modifications to the current solution that increase the value of the objective function are allowed in SA, in contrast to classical descent methods where only modifications that decrease the objective value are possible.

The name and inspiration of this method come from the process of annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling provides an opportunity to find configurations with lower internal energy than the initial one. By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called *temperature*), that is gradually decreased during the process (cooling process).

The dependency is such that the current solution changes arbitrarily in the search domain when T is large, i.e. at the beginning of the algorithm, through uphill moves (or random walks) that saves the method from becoming trapped at

a local minimum. Afterwards, the temperature T is gradually decreased, intensifying the search process in the specific promising-zone of the domain (downhill moves). More precisely, the current solution is always replaced by a new one if this modification reduces the objective function value, while a modification increasing the objective function value by Δ is only accepted with a probability $\exp(-\Delta/T)$ (Boltzmann function), using the temperature T as a control parameter. At a high temperature T, the probability of accepting an increase to the objective value is high (uphill moves: high diversification and low intensification capabilities). Conversely, this probability gets lower as the temperature T is decreased (downhill moves: high intensification and low diversification capabilities). Therefore, according to the SA criterion, the value of T is initially high, which allows many worse moves to be accepted, and is gradually reduced following a so-called *cooling schedule* (or *cooling law*). Considering the iteration k and the temperature value T_k, the cooling schedule is a decreasing function which determines the temperature value at the successive iteration $k+1$, as follows:

$$T_{k+1} \leftarrow func(T_k, k), \tag{2.1}$$

The process described is memory-less because it follows a trajectory in the state space in which the successor state is chosen depending only on the incumbent one, without taking into account the history of the search process.

The details of the implementation of Simulated Annealing are specified in Algorithm 2.2. At the beginning, the initial temperature value (T_0), a static neighbourhood structure ($N(\cdot)$), the specific cooling schedule, and the user termination conditions need to be imposed. Possible termination conditions are the maximum allowed CPU time, the maximum number of iterations, or the maximum number of iterations without improvements. Then, the algorithm starts with an initial solution s, for example generated at random (*Generate-Initial-Solution()*), and selects at random a point s' within its neighbourhood $N(s)$ (*Pick-up-at-random*$(N(s))$). If s' produces an improvement in the objective function value with respect to s ($f(s') < f(s)$), then the current solution is replaced with the improved one ($s \leftarrow s'$). Otherwise, s' replaces s with probability $\exp(-(f(s') - f(s))/T)$. Specifically, a random number ξ with uniform distribution in $[0, 1]$ is independently generated ($\xi \leftarrow random[0, 1]$),

Algorithm 2.2: Simulated Annealing

Input: An objective function $f(\cdot)$, the search space S, a specific cooling schedule;
Output: A solution $s \in S$;
Initialization:
- Define a static neighbourhood structure $N(\cdot)$;
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Let T_0 be the initial temperature value;

begin
  $s \leftarrow$*Generate-Initial-Solution()*;
  Set the temperature to the initial value: $T \leftarrow T_0$;
  while *termination conditions* **do**
    $s' \leftarrow$*Pick-up-at-random(N(s))*;
    if $f(s') < f(s)$ **then**
      Move $s \leftarrow s'$;
    else
      Select at random a number between 0 and 1: $\xi \leftarrow random[0,1]$;
      if $\xi < \exp\left(-\frac{f(s')-f(s)}{T}\right)$ **then**
        Move $s \leftarrow s'$;
      end
    end
    Cooling schedule for the temperature: $T_{k+1} \leftarrow func(T_k, k)$;
    Continue with the next iteration: $k \leftarrow k+1$;
  end
  $\Rightarrow$ *Return(s).*
end

and if $\xi < \exp(-(f(s') - f(s))/T)$, the worse move s' is accepted ($s \leftarrow s'$). Then, the temperature T is updated according to the specific cooling schedule ($T_{k+1} \leftarrow func(T_k, k)$), and the algorithm stops when the termination conditions are satisfied.

Theoretical results on non-homogeneous Markov chains (Aarts and Korst, 1988; Aarts et al., 2005) state that under particular conditions on the cooling schedule, the algorithm converges in probability to a global minimum as $k \to +\infty$. More precisely, calling p_k the probability to find a global minimum after k steps, then there exists $\Gamma \in \Re$ such that $\sum_{k=1}^{+\infty} \exp \frac{\Gamma}{T_k} \to +\infty$ if and only if $\lim_{k\to\infty} p_k = 1$.

Different cooling schedules in Simulated Annealing, all satisfying this hypothesis of convergence, may be considered, such as a *logarithmic cooling law*:

$$T_{k+1} = \frac{\Gamma}{\lg(k + k_0)}, \tag{2.2}$$

where Γ and k_0 are arbitrary constant values that must be set experimentally by

the user. Logarithmic cooling schedule satisfies the hypothesis of convergence, as is shown in the following equation:

$$\sum_{k=0}^{+\infty} \exp \frac{\Gamma}{T_{k+1}} = \sum_{k=0}^{+\infty} \exp \frac{\Gamma}{\frac{\Gamma}{\lg(k+k_0)}} = \\ = \sum_{k=0}^{+\infty} \exp \lg(k+k_0) = \sum_{k=0}^{+\infty} (k+k_0) \to +\infty. \tag{2.3}$$

Sometimes, the logarithmic cooling law is too slow for practical purposes. Therefore, faster cooling schedule techniques may be adopted, such as a *geometric cooling law*, which is a cooling rule with an exponential decay of the temperature:

$$T_{k+1} = \alpha \cdot T_k, \tag{2.4}$$

where $\alpha \in [0, 1]$.

Other complex cooling techniques can be used in order to improve the performance of the SA algorithm. For example, to have an optimal balance between diversification and intensification, the cooling rule may be updated during the search process. At the beginning, T can be constant or linearly decreasing to have a high diversification factor for a larger exploration of the domain. Then, T can follow a fast rule, such as the geometric one, to converge quickly to a local optimum. Other successful variants are *non-monotonic cooling schedules* that alternate phases of cooling and reheating, providing an oscillating balance between diversification and intensification.

Simulated Annealing has been applied to several combinatorial problems with success (Gendreau and Potvin, 2005). Rather than as a stand-alone algorithm, it is nowadays used as a component in many hybrid metaheuristics to improve their performance in specific applications (Aarts et al., 1997).

2.2.2 Tabu Search

Tabu Search (TS) is a widely used metaheuristic introduced by Glover (1986). It shares with Simulated Annealing the ability to guide the search avoiding traps in poor local optima, but in a deterministic way rather than a stochastic one, modelling human memory processes. Memory is implemented by the implicit

recording of previously seen solutions using a simple data structure. This consists of a *tabu list* of moves which have been made in the recent past of the search, and which are forbidden (tabu) for a certain numbers of iterations. This helps to avoid cycling, and serves also to promote a diversified search of the solution, trying to escape from local minima.

The details of Tabu Search are specified in Algorithm 2.3. The procedure

Algorithm 2.3: Tabu Search

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define a neighbourhood rule $N(\cdot)$;
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Let *tabu list* be the set of forbidden solutions;
- Let *allowed set* be the set of admissible solutions;

begin
    tabu list $\leftarrow \emptyset$;
    $s \leftarrow$*Generate-Initial-Solution()*;
    Move $s' \leftarrow s$;
    Update the tabu list: *tabu list* $\leftarrow FIFO($*tabu list* $\cup\, s)$;
    while *termination conditions* **do**
        Update the allowed set: *allowed set* $\leftarrow N(s) -$ *tabu list*;
        Find the best solution within the allowed set: $s \leftarrow$*Improve(allowed set)*;
        if $f(s) < f(s')$ **then**
            Move $s' \leftarrow s$;
            Update the tabu list: *tabu list* $\leftarrow FIFO($*tabu list* $\cup\, s)$;
        end
    end
    $\Rightarrow$ *Return*(s').
end

starts with an initial solution s, for example generated at random (*Generate-Initial-Solution()*). Then, at each iteration, the procedure *Improve(·)* tries to find a better solution from the set of solutions that do not belong to the tabu list, referring to this set as the *allowed set*. This procedure can be either a first improvement procedure or a best improvement procedure. In the first case, it scans the *allowed set* and chooses the first solution that is better than s. In the second case, it exhaustively explores the *allowed set* and returns the solution with the lowest objective function value. If the new improved solution s is better than the best solution to date s', the algorithm moves the current solution s to s' (i.e. $s' \leftarrow s$), and the tabu list is updated by following a FIFO (First In First Out)

technique, i.e. the current solution s is added to the tabu list and the oldest element is removed ($tabu\ list \leftarrow FIFO(tabu\ list \cup s)$). Due to this dynamic restriction of allowed solutions in a neighbourhood, TS can be considered as an explorative method with a dynamic neighbourhood structure, and with a short-term memory implemented by the tabu list. The algorithm starts again with the same procedure, and it stops when a termination condition is met or the allowed set is empty. The best solution to date s' forms the output of the algorithm.

Usage of memory in metaheuristics can be described, generally, in terms of four "dimensions" in the search: recency, frequency, quality, and influence, in which the first two are the most important. *Recency* records the most recent iteration in which a solution was involved. In TS the most recent moves are forbidden and the length of the tabu list, called *tabu tenure*, represents the recency principle. The tabu tenure is either fixed or dynamically updated during the search process. If its value is small, there is a high exploitation of the domain, but not many uphill moves to differentiate the search. Otherwise, if the tabu tenure is large, the exploration of new areas is encouraged because it forbids revisiting a large number of solutions. A promising research direction in Tabu Search consists of creating advanced ways to adapt the tabu tenure dynamically (Glover, 1986). For example, the tabu tenure could be periodically re-initialized at random between a minimum value and a maximum value. Otherwise, it could be manually increased if there are many solution repetitions (i.e. a larger diversification factor is needed), while it could be decreased if no improvements are obtained and more intensification is required. It is often beneficial to focus on some components or *attributes* of a move rather than on the complete move itself, avoiding managing a list on entire solutions that could make TS inefficient and not practical. Attributes are stored in different tabu lists defining the *tabu conditions*, which are used to filter the neighbourhood of a solution and generate the allowed set. A neighbouring solution is considered forbidden, and deemed not admissible, if it has attributes on a tabu list. Storing attributes rather than complete solutions is much more efficient, but also it may cause some non-tabu solutions, because forbidding an attribute means assigning the tabu status to probably more than one solution. To correct such errors, some *aspiration criteria* are defined, enabling the introduction of a solution in the allowed set even if it is forbidden by tabu conditions. The

most commonly used aspiration criterion selects elements that are better than the current solution.

If recency simulates the short-term memory, a long-term memory can be implemented by the use of a variety of *frequency* measures, as "residence" measures and "transition" measures. The former is related to the number of times a particular attribute is observed, while the latter relates to the number of times an attribute changes from one value to another. In each case, the frequency measures are usually employed to generate penalties, which modify the objective function. Thereby, diversification is encouraged by the generation of solutions embodying combinations of attributes significantly different from those previously encountered. Conversely, intensification is promoted by incorporating attributes of solutions from selected subsets of elements, called *elite subsets*, implicitly focussing the search in sub-regions defined relative to these subsets.

After discussing the concepts of recency and frequency, it may be also helpful to provide a brief reiteration of the basic notions of quality and influence. *Quality* in TS usually refers to those solutions with good objective function values. A collection of such elite solutions may stimulate a more intensive search in the most promising regions of the search area. *Influence* is roughly a measure of the degree of change induced in solution structure, commonly expressed in terms of the distance of a move from one solution to the next. It is an important aspect of the use of aspiration criteria, and is also relevant to the development of candidate list strategies. Influence is a property regarding choices made during the search and can be used to indicate which choices have shown to be the most critical.

The Tabu Search heuristic is a rich source of ideas. Many of these ideas together with the corresponding strategies have been, and are currently, adopted by other metaheuristics. From a practical point of view, a recency-based approach with a simple neighbourhood structure, searched using a restricted candidate list strategy, will often provide very good results (Glover and Kochenberger, 2003).

2.2.3 Greedy Randomized Adaptive Search Procedure

The GRASP (Greedy Randomized Adaptive Search Procedure) methodology was developed in the late 1980s, and the acronym was coined by Feo and Re-

sende (1989). It was first used to solve set covering problems (Feo and Resende, 1995), but was then extended to a wide range of combinatorial optimization problems (Pitsoulis and Resende, 2002). Surely GRASP must have been implemented many times in an ad hoc way before anyone knew it was a specific metaheuristic. GRASP is a typical example of a constructive metaheuristic, belonging also to the class of explorative methods. It is basically a multi-start two-phase metaheuristic, consisting of a construction phase and a local search improvement phase (Algorithm 2.4).

Algorithm 2.4: Greedy Randomized Adaptive Search Procedure

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define a static neighbourhood structure $N(\cdot)$;
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
begin
  while *termination conditions* **do**
    Set $s \leftarrow \emptyset$;
    Construction phase(s);
    Local search$(N(s))$;
    if $f(s) < f(s')$ **then**
      Move $s' \leftarrow s$;
    end
  end
  $\Rightarrow$ *Return*(s').
end

The *solution construction* mechanism builds a feasible solution s by using a greedy randomized procedure, whose randomness allows solutions in different areas of the solution space to be obtained (*Construction phase(s)* procedure, see Algorithm 2.5). The greedy randomized procedure obtains a solution by iteratively creating a candidate list of elements that can be added to the partial

Algorithm 2.5: Procedure *Construction phase*$(\cdot)$

Procedure Construction phase(s):
Let $RCL_\alpha \leftarrow 0$ be the restricted candidate list of length α;
while s *is incomplete* **do**
  Update the restricted candidate list: $RCL_\alpha \leftarrow$*Greedy evaluation*(S, α);
  Select at random an element $x \in RCL_\alpha$;
  Add element x to the incomplete solution s: $s \leftarrow s \cup \{x\}$;
end

solution, and then randomly selecting an element from this list. The candidate list (RCL_α: Restricted Candidate List of length α) is created by evaluating the elements not yet included in the partial solution ($RCL_\alpha \leftarrow$*Greedy evaluation(S, α)*). A greedy function (or constructive heuristic), depending on the specifications of the problem, is used to perform this evaluation. Only the best elements, according to this greedy function, are included in RCL_α. In particular, the elements are ranked by means of the greedy function that gives them a score as a function of the benefit if inserted in the current partial solution. These scores can be either static values (fixed from the starting point to the end of the entire algorithm) or dynamic values (updated at each step depending on the current partial solution).

The size α of the candidate list is a very important parameter because it determines the strength of the heuristic bias, and also influences the sampling of the search space. It can be limited either by the number of elements, or by their quality with respect to the best candidate element. The simplest scheme to define α is updating it at each step, randomly or by means of greedy evaluation. The extreme cases for the size of the candidate list are: $\alpha = 1$ and $\alpha = n$, where n represents the number of elements to be evaluated. In the first case, only the best element is added to the restricted candidate list, and the construction mechanism is equivalent to a deterministic greedy heuristic. In the case of $\alpha = n$, the candidate list is filled with all the n elements, and the construction mechanism is equivalent to a random walk, because complete randomization is used to choose the next element to add to the partial solution.

The construction phase stops when a feasible solution is produced. The solution s obtained is not necessarily locally optimal, so a *local search* phase (such as Simulated Annealing or Tabu Search) is included to try to improve it (*Local search($N(s)$)* procedure). This phase uses a local search mechanism which, iteratively, tries to replace the current solution with a better neighbouring solution, until no better solution can be found. Different strategies may be used in order to evaluate the neighbourhood structure $N(\cdot)$. At each step, the best element found to date is memorized as s'. This two-phase process is iterative, continuing until the user termination condition such as the maximum allowed CPU time, the maximum number of iterations, or the maximum number of iterations between

two successive improvements, is reached. The final result of GRASP is the best solution found to date, s'.

The solutions obtained by GRASP are usually of good quality because it offers fast local convergence (high intensification capability) as a result of the greedy aspect of the procedure used in the construction phase, and of the local search mechanism; and also a wide exploration of the solution space (high diversification capability) for the randomization used in the selection of a new element from RCL_α. However, GRASP does not use history-memory of the search process and, for this reason, it can be outperformed by other metaheuristics in some applications.

GRASP can be effective if the solution construction mechanism samples the most promising regions of the domain (by using an effective constructive heuristic and an appropriate value of α), and if the resulting solutions from the constructive heuristic belong to regions associated with different local minima (by using an effective constructive heuristic and an appropriate local search with a good choice of the neighbourhood structure). Several new components, presented and discussed in (Resende and Ribeiro, 2003), have extended the scheme of GRASP (reactive GRASP, parameter variations, bias functions, memory and learning, improved local search, path relinking,...). For its characteristics of simplicity and high speed, GRASP is often used as a method for generating good starting points for other hybrid metaheuristics.

2.2.4 Iterated Local Search

Iterated Local Search (ILS) was proposed by Stützle (1999, 2006) for the quadratic assignment problem, and is probably the most general scheme among the explorative strategies. The aim of this heuristic is to prevent getting stuck in local optima of the objective function. Iterated Local Search mainly consists of two operators for generating new solutions (Lourenço et al., 2003). One is a local search, to reach local optima performing a walk in the search space, and the other is a perturbation operator, to efficiently escape from local optima. That is, when local search is trapped in a local optimum, the perturbation operator is applied to the local optimum to generate a new starting point for the local search.

It is desirable that the generated starting point should be in a promising area in the search space. Formally, Iterated Local Search is specified in Algorithm 2.6.

Algorithm 2.6: Iterated Local Search

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define the neighbourhood structure $N(\cdot)$;
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;

begin
  $s \leftarrow$*Generate-Initial-Solution()*;
  Set $s' \leftarrow s$;
  repeat
    $s \leftarrow$*Perturbation($N(s)$, history)*;
    Local search(s);
    if $f(s) < f(s')$ **then**
      Move $s' \leftarrow s$;
    else
      Apply an acceptance criterion to select the candidate solution:
      $s \leftarrow$*Acceptance criterion(s, s', history)*;
    end
  until *termination conditions* ;
  $\Rightarrow$ *Return(s').*
end

The algorithm initializes the search by selecting an initial candidate solution s. It is preferable to start from a good initial solution s, but also the construction of s should be, computationally, not too expensive. The fastest way is to generate randomly the initial solution. However constructive heuristics may also be adopted at this stage in order to quickly find high-quality starting points. The core of the overall algorithm consists of the following three phases:

1. A "perturbation" within the neighbourhood $N(\cdot)$ applied to the current candidate solution s ($s \leftarrow$*Perturbation($N(s)$, history)*);

2. A "local search" performed with respect to the perturbed solution s in order to find a local minimum (*Local search(s)*);

3. The application of an "acceptance criterion" to decide which of the two local optima, s or s', has to be chosen as the new candidate solution to continue the search process ($s \leftarrow$*Acceptance criterion(s, s', history)*).

The specific steps have to be properly designed and set to find a good trade-off between intensification and diversification of the search process, in order to achieve high performance of the algorithm and its efficacy to solve large and difficult instances of problems. Both the perturbation and the acceptance criterion mechanisms can use aspects of the search history (long-term or short-term memory). For example, stronger perturbation should be applied when the same local optima s are repeatedly encountered. The role of the perturbation *Perturbation($N(s)$, history)* (usually probabilistic to avoid cycling) is to modify the current candidate solution s within its neighbourhood $N(s)$ to help the search process to effectively escape from local minima, in order to eventually find different better points. Typically, the strength of the perturbation, given by the selected neighbourhood structure $N(\cdot)$, has a strong influence on the length of the subsequent local search phase. The neighbourhood structure can be either fixed independently of the problem size (static neighbourhood structure) or variable (dynamic neighbourhood structure). However, the latter one is in general more effective because the larger the problem size, the greater should be the strength. A more sophisticated adaptive strength scheme is also possible in which the perturbation strength is increased when more diversification is needed, and decreased when intensification seems preferable. Variable Neighbourhood Search and its variants belong to this category, as will be explained in the next section.

After the perturbation phase, a locally optimal solution s is achieved by applying the local search phase (*Local Search(s)* procedure), whose characteristics have a considerable influence on the performance of the entire algorithm. The local search considered is not restricted to $N(\cdot)$, but any neighbourhood structure can be used to try to improve, if possible, the current solution s.

The successive acceptance criterion (*Acceptance criterion(s, s′, history)*) has also a strong influence on the behaviour and the performances of ILS. The two extremes are:
- Accepting the new local minimum s as the new candidate solution only in case of improvement (i.e. only if $f(s) < f(s')$. This is the mechanism used in classic local search methods, which produces a strong intensification of the search process);

- Always accepting the new solution s as the new candidate solution (this corresponds to a random walk in the search space, which produces a high diversification of the search process).
Between these extremes, there are several intermediate choices. It is possible, for example, to adopt a kind of annealing schedule, consisting of accepting always the candidate solutions s which produce an improvement ($f(s) < f(s')$), and also the candidate solutions s which do not produce an improvement ($f(s) \geq f(s')$) with a probability that is a function of the temperature parameter T and the difference of objective function values ($\Delta = f(s) - f(s')$), as follows:

$$\exp(-\Delta/T) = \exp(-(f(s) - f(s'))/T). \tag{2.5}$$

As in Simulated Annealing, the cooling schedule for the temperature T can be either monotonic (non-increasing in time) or non-monotonic (adapted to tune the balance between diversification and intensification capabilities). The non-monotonic schedule is particularly effective if it exploits the history of the search process: instead of constantly decreasing the temperature, it is increased when more diversification seems to be required.

Iterated Local Search is often used as a framework for other metaheuristics or can be easily incorporated as a subcomponent in some of them to build effective hybrid methods. Successful applications of Iterated Local Search are the travelling salesman problem, the single-machine total weighted tardiness problem, and the quadratic assignment problem (Lourenço et al., 2003; Stützle, 2006).

2.2.5 Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is a relatively new and widely applicable metaheuristic based on dynamically changing neighbourhood structures during the search process (Hansen and Mladenović, 1997, 2001, 2003). VNS doesn't follow just a trajectory, but it searches for new solutions in increasingly distant neighbourhoods of the current solution, jumping only if a better solution than the current best solution is found. This single-point metaheuristic belongs also to the class of explorative methods which consider dynamic neighbourhood structures, and it can be considered a special case of Iterated Local Search.

The VNS approach can be summarized as: *"One Operator, One Landscape"*, meaning that promising zones of the search space given by a specific neighbourhood may not be promising for other neighbourhoods (landscape). Nevertheless, a local optimum with respect to a given neighbourhood may not be locally optimal with respect to another neighbourhood.

The basic VNS procedure is specified in Algorithm 2.7, and illustrated in Figure 2.1. At the starting point, it is required to define arbitrarily a suitable

Algorithm 2.7: Variable Neighbourhood Search

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define the neighbourhood structure $N_k(\cdot)$, with $k \leftarrow 1, 2, ..., k_{max}$, where k_{max} represents the size of the neighbourhood structure (e.g. increasingly distant neighbourhoods: $|N_1(\cdot)| < ... < |N_{k_{max}}(\cdot)|$);
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;

```
begin
    s' ← Generate-Initial-Solution();
    repeat
        Set k ← 1;
        while k < k_max do
            s ← Shaking phase(N_k(s'));
            Local search(s);
            if f(s) < f(s') then
                Move s' ← s;
                Set k ← 1;
            else
                Increase the size of the neighbourhood structure: k ← k + 1;
            end
        end
    until termination conditions ;
    ⇒ Return(s').
end
```

neighbourhood structure of size k_{max} (user parameter to be set), where $N_k(\cdot)$ defines a neighbourhood of size k, and $|N_k(\cdot)|$ its cardinality. The simplest and most common choice is a structure in which the neighbourhoods have increasing cardinality: $|N_1(\cdot)| < |N_2(\cdot)| < ... < |N_{k_{max}}(\cdot)|$ (nevertheless, with this sequence a large number of solutions could be revisited, at the cost of increased computational time. Today, attempts to improve the scanning of the landscape are made through more complex neighbourhood structures). The process of changing neighbourhoods when no improvement occurs diversifies the search. In particular, the choice of neighbourhoods of increasing cardinality yields a progressive

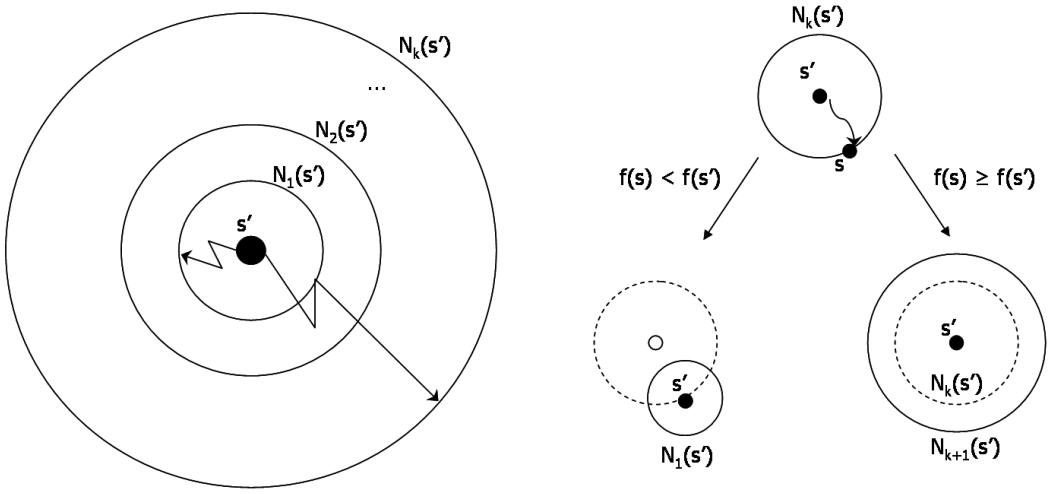


Figure 2.1: Basic schema of Variable Neighbourhood Search.

diversification.

VNS starts from an initial solution s' (e.g. generated at random) with k increasing from 1 up to k_{max} during the progressive execution. The basic idea of VNS to change the neighbourhood structure, when the search is trapped at a local minimum, is implemented by the Shaking phase (*Shaking phase*($N_k(s')$) procedure). It consists of the random selection of a point s in the neighbourhood $N_k(s')$ of the current solution s', which may provide a better starting point for the successive local search phase. The random point s is generated in order to avoid cycling, which might occur if any deterministic rule was used. The successive local search phase (*Local Search(s)* procedure) is not restricted to $N_k(\cdot)$, but any neighbourhood structure can be used to try to improve, if possible, the current solution s. Afterwards, if no improvements are obtained ($f(s) \geq f(s')$) in the move phase, the neighbourhood structure is increased ($k \leftarrow k + 1$) giving a progressive diversification ($|N_1(C)| < |N_2(C)| < ... < |N_{k_{max}}(C)|$). Otherwise, if an improved solution s is obtained ($f(s) < f(s')$), it becomes the best solution to date ($s' \leftarrow s$) and the algorithm restarts from the first neighbourhood ($k \leftarrow 1$) of the best solution to date ($N_1(s')$). The algorithm proceeds until the user termination conditions (maximum allowed CPU time, maximum number of iterations, or maximum number of iterations between two successive improvements) are satisfied.

VNS provides a general framework and many variants exist for specific requirements. Experimentally, VNS performance can be improved if s is not just picked at random from $N_k(s')$, but it is achieved by performing an iterative search in the shaking phase between a random selection of points. Moreover, setting $k \leftarrow k + k_{step}$ instead of $k \leftarrow k + 1$, and $k \leftarrow k_{min}$ instead of $k \leftarrow 1$, gives an easy and natural way to drive the intensification and diversification of the search. It is also possible to remove the local search step for very large problem instances for which it is costly, making it similar to the classic Monte-Carlo method. This variant of VNS is called *Reduced Variable Neighbourhood Search* (RVNS). Another important variant of VNS is the *Variable Neighbourhood Descent* (VND) algorithm. For some problems, the local search strategy may be time-consuming. Since the properties of a neighbourhood are in general different from those of other neighbourhoods, a local search strategy may perform differently on them (Hansen and Mladenović, 2003). From this consideration, VND is used to try to reduce the computational running times. VND orders the neighbourhood structures in a sequential way, and applies a local search by changing neighbourhoods deterministically.

The choice of the neighbourhood structures is the critical point in VNS and VND, because the neighbourhoods should exploit different properties and characteristics of the search space. Thus, another important variant of VNS, called *Variable Neighbourhood Decomposition Search* (VNDS), selects the neighbourhoods by producing a decomposition of the problem instance (Hansen and Mladenović, 2003). VNDS follows the same scheme of the basic VNS, but the neighbourhood structures and the local search are defined on sub-problems of each solution. All attributes (variables) of the current solution are kept fixed with the exception of k of them, which define a neighbourhood structure $N_k(\cdot)$. Local search only regards changes on the variables belonging to the sub-problem it is applied to. VNDS procedure can be obtained by substituting the inner loop of the VNS algorithm, as specified in Algorithm 2.8.

In the shaking phase, the current solution s' and the incumbent one s differ only in k attributes (variables). In the local search phase, the improved solution is obtained by just allowing movements involving these k attributes of the solution s (*Local search*(s, k variables)). If a better solution s is reached, then the current

Algorithm 2.8: Variable Neighbourhood Decomposition Search

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define the neighbourhood structure $N_k(\cdot)$, with $k \leftarrow 1, 2, ..., k_{max}$, where k_{max} represents the size of the neighbourhood structure (e.g. increasingly distant neighbourhoods: $|N_1(\cdot)| < ... < |N_{k_{max}}(\cdot)|$);
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;

```
begin
    s' ← Generate-Initial-Solution();
    repeat
        Set k ← 1;
        while k < k_max do
            s ← Shaking phase(N_k(s'));
            Local search(s, k variables);
            if f(s) < f(s') then
                Move s' ← s;
                Set k ← 1;
            else
                Increase the size of the neighbourhood structure: k ← k + 1;
            end
        end
    until termination conditions ;
    ⇒ Return(s').
end
```

solution is replaced with the improved one ($s' \leftarrow s$), and the algorithm will start again with the first neighbourhood by setting $k \leftarrow 1$. Conversely, if no improved solutions are reached ($f(s) \geq f(s')$), it means that the current solution s' is a local minimum for k variables, and the algorithm will increase the number of the variables to explore ($k \leftarrow k+1$). The algorithm proceeds iteratively and will stop if the usual stopping conditions are satisfied.

VNS, RVNS, VND, and VNDS are steepest descent-oriented algorithms and, often, they are unsuitable to effectively explore the search space. Another variant has been developed called *Skewed Variable Neighbourhood Search* (SVNS), which extends the basic VNS by providing a more flexible acceptance criterion (Hansen and Mladenović, 2003). As an alternative to only accepting solution improvements, worse solutions s can be accepted if they differ from the current one (s') by less than the value of $\alpha \cdot \rho(s', s)$, where $\rho(s', s)$ is the distance between s' and s, and $\alpha \in [0, 1]$ is a weight parameter in the acceptance criterion. The distance measure ρ is defined by the user with respect to the characteristics of the specific problem, and it may be, for example, the Hamming distance, the Manhattan

distance, or others. The SVNS procedure is specified in Algorithm 2.9.

Algorithm 2.9: Skewed Variable Neighbourhood Search

Input: An objective function $f(\cdot)$ and the search space S;
Output: A solution $s' \in S$;
Initialization:
- Define the neighbourhood structure $N_k(\cdot)$, with $k \leftarrow 1, 2, ..., k_{max}$, where k_{max} represents the size of the neighbourhood structure (e.g. increasingly distant neighbourhoods: $|N_1(\cdot)| < ... < |N_{k_{max}}(\cdot)|$);
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
begin
  $s' \leftarrow$*Generate-Initial-Solution()*;
  repeat
    Set $k \leftarrow 1$;
    while $k < k_{max}$ **do**
      $s \leftarrow$*Shaking phase*$(N_k(s'))$;
      Local search(s);
      if $(f(s) - f(s') < \alpha \cdot \rho(s', s))$ **then**
        Move $s' \leftarrow s$;
        Set $k \leftarrow 1$;
      else
        Increase the size of the neighbourhood structure: $k \leftarrow k + 1$;
      end
    end
  until *termination conditions* ;
  $\Rightarrow$ *Return(s')*.
end

Variable Neighbourhood Search and its variants have been successfully applied to many combinatorial optimization problems (Hansen and Mladenović, 2001, 2003), such as, for example, travelling salesman problem, vehicle routing problem, location and clustering problems, job shop scheduling. Current research activity in VNS is huge. A systematic study of moves and neighbourhood structures for whole classes of problems, together with the data-structures for their implementation, is one promising research direction. Another one is trying to consider more sophisticated distributions of neighbourhoods. Introduction of memory, parallel VNS, and hybridizing VNS within exact algorithms, are also interesting research areas (Hansen and Mladenović, 2003).

2.2.6 Guided Local Search

Guided Local Search (GLS) is an explorative metaheuristic based on penalties and was introduced in (Voudouris, 1997; Voudouris and Tsang, 1999). The Guided

Local Search approach gradually moves (to guide the search) away from local minima by changing the search landscape. In contrast to other explorative strategies such as Tabu Search and Variable Neighbourhood Search, the set of solutions and the neighbourhood structure are kept fixed (single neighbourhood structure) while the objective function $f(\cdot)$ is dynamically changed (dynamic objective function), in order to make the current local optimum less desirable and trying to escape from it.

Guided Local Search is an algorithm for modifying classic local search heuristics. This strategy is based on the definition of *solution features*, which may be any kind of properties or characteristics that can be used to discriminate between solutions (e.g. in travelling salesman problem they are the arcs between pairs of cities (Voudouris and Tsang, 1999)). An indicator function $I_i(s)$ is defined to show whether the feature i is present in a specific solution s, that is:

$$I_i(s) = \begin{cases} 1 & \text{if feature } i \text{ is present in solution } s \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

The GLS procedure is specified in Algorithm 2.10, and illustrated in Figure 2.2.

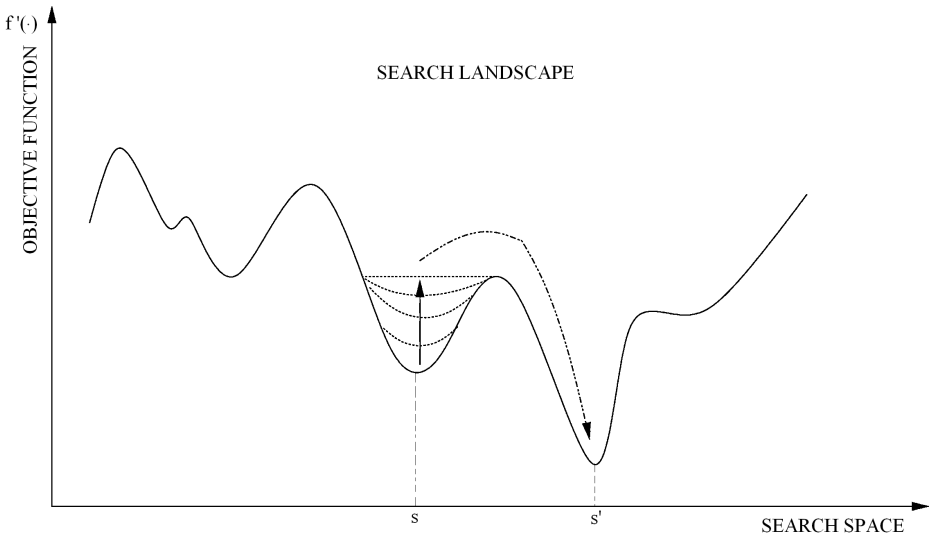


Figure 2.2: Guided Local Search strategy.

Algorithm 2.10: Guided Local Search

Input: An objective function $f(\cdot)$, the search space S, m solution features, the regulation parameter λ for the solution features;
Output: A solution $s' \in S$;
Initialization:
- Define a static neighbourhood structure $N(\cdot)$;
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Let p_i, with $i \leftarrow 1 \dots m$, be the penalty parameters for the m solution features considered;
- Let c_i, with $i \leftarrow 1 \dots m$, be the costs assigned to the m solution features considered;

begin
  $s \leftarrow$*Generate-Initial-Solution()*;
  Set $s' \leftarrow s$;
  Initialize the penalty parameters to 0: $p_i \leftarrow 0$, $\forall i \leftarrow 1 \dots m$;
  while *termination conditions* **do**
    Modify the objective function: $f'(s) = f(s) + \lambda \cdot \sum_{i=1}^{m} p_i \cdot I_i(s)$, where $I_i(s) = \begin{cases} 1 & \text{if feature } i \text{ is present in solution } s \\ 0 & \text{otherwise} \end{cases}$
    Local search($f'(\cdot)$, $N(s)$);
    if $f'(s) < f'(s')$ **then**
      Move $s' \leftarrow s$;
    end
    Calculate the utility function $Util(s, i)$ for each solution feature $i, \forall i \leftarrow 1 \dots m$, of the current candidate solution s: $Util(s, i) = \begin{cases} I_i(s) \cdot \frac{c_i}{1+p_i} & \text{if feature } i \text{ is present in solution } s \\ 0 & \text{otherwise} \end{cases}$
    foreach *solution feature i with max $Util(s, i)$* **do**
      Penalize the solution feature i: $p_i \leftarrow p_i + 1$;
    end
  end
  $\Rightarrow$ *Return(s').*
end

Consider a candidate solution s and a total of m features, the new objective function $f'(s)$ is equal to the sum of the current objective function $f(s)$ and a term depending on the m features:

$$f'(s) = f(s) + \lambda \cdot \sum_{i=1}^{m} p_i \cdot I_i(s), \tag{2.7}$$

where λ is the user-defined *regulation parameter* balancing the importance of the influence of all the features i with respect to the original objective function $f(s)$, and p_i are the *penalty parameters* weighting the importance of the specific feature i. At the beginning, the algorithm initializes all the penalty parameters to zero, and assigns the variables uniformly at random. Then, the local search (*Local search($f'(\cdot)$, $N(s)$)*) tries to find a better solution within the neighbourhood $N(s)$

of the current solution s. The local search is computed with respect to the new objective function $f'(s)$, and it may be either a first improvement procedure or a best improvement procedure, as in standard local search procedures. After the local search phase, the penalty parameters are updated by means of a *penalties update rule*. The most common choice is to use an incrementing rule: the penalties of all features with maximal *utility* are incremented by one ($p_i \leftarrow p_i + 1$), where the utility of solution s under feature i is a function defined as:

$$Util(s,i) = \begin{cases} I_i(s) \cdot \dfrac{c_i}{1+p_i} & \text{if feature } i \text{ is present in solution } s \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

where c_i is the *cost* assigned to feature i, obtained from an user-defined heuristic evaluation of the relative importance of each feature with respect to the others. The intention is to penalize "bad features", or features which "matter most", when a local search settles in a local optimum, by incrementing the penalty values of the features i with the greatest $Util(s,i)$ value. Besides, the more times that a local minimum s has been penalized, the greater $(p_i + 1)$ becomes, and therefore, the lower the utility of penalizing it again. The higher the cost of this feature, the greater the utility of penalizing it. In other words, if a feature is not exhibited in the local optimum, then the utility of penalizing it is 0. The feature which has high cost affects the overall cost more. Therefore, the cost is scaled by the penalty parameter p_i to prevent the algorithm from being totally biased toward the cost, and also to make the algorithm sensitive to the search history (memory-usage algorithm). The procedure continues iteratively and halts when the user termination conditions are satisfied. The best solution to date (s') is produced as output of the method.

A variant to the classic GLS scheme consists of modifying the incrementing update rule for the penalties with a multiplicative rule (Voudouris and Tsang, 1999). The multiplicative rule has the form: $p_i = \alpha \cdot p_i$, where $\alpha \in [0,1]$ is a user-defined parameter. This rule is applied with a lower frequency than the incrementing one (for example every few hundreds of iterations) in order to smooth the weights of penalized features and to prevent the landscape from becoming too rugged. The penalty update rules are often very sensitive to the problem instance. Another extension of GLS uses an additional mechanism for bounding

the range of the penalties: if after the updating process, the maximum penalty exceeds a given max threshold, all penalties are uniformly decayed, improving the performance of the algorithm and its efficacy to solve large and difficult problem instances.

2.3 Population-based metaheuristics

Population-based methods deal at each step with a set of solutions (or a population) rather than with a single one, providing a natural and intrinsic way to explore the search space. Their performance strongly depends on the way the populations are manipulated. The main population-based methods in combinatorial optimization are divided in evolutionary algorithms, such as Genetic Algorithms (GAs), Quantum-inspired Genetic Algorithms (QGAs), Estimation of Distribution Algorithms (EDAs), and Scatter Search (SS), and in nature-inspired algorithms, such as Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO).

Evolutionary algorithms are inspired by the Darwinian evolution theory (Darwin, 1859). The main idea is that populations evolve over the course of generations through a process of natural selection. In evolutionary algorithms, this idea consists of applying iteratively specific genetic operators to modify individuals, which are solutions to the problem, within a set (population). At the end of the iterations, the best individual among the population of survivors represents, hopefully, a near-solution to the problem. It is generally accepted that any evolutionary algorithm must have the following basic components: a genetic representation (or data structure) of problem solutions; a way to create the initial population; an evaluation function rating the solutions in terms of their fitness; some genetic operators, such as recombination (or crossover) and modification (or mutation); and a set of values for the specific parameters, such as population size and probabilities of applying genetic operators. The data structure used to represent the solutions and the set of genetic operators, constitute the skeleton of each evolutionary algorithm. There are currently three well-defined paradigms in evolutionary algorithms, characterized by different components. They are Genetic

Algorithms (GAs), Evolution Strategies (ES), and Evolutionary Programming (EP).

Genetic Algorithms consider a population of individuals, or generation of chromosomes, that are feasible solutions of the problem, represented in most of the cases by binary strings. Crossover and mutation operations are then applied in order to build one generation from the previous one. After a number of generations, the algorithm converges and the best individual, hopefully, represents a near-optimal solution.

Evolution Strategies were developed mainly to build systems capable of solving real-valued parameter optimization problems. Their natural representation of the individuals consists of a vector of real numbers in order to help mutation operators and manipulation of the candidate solutions. Generally, Evolution Strategies emphasize behavioural changes by mutation at the level of the individual.

Evolutionary Programming stresses behavioural change at the level of the species. The phenotypes of individuals are represented as finite state machines capable of reacting to environmental stimulation, and to develop operators (primarily mutation) for reflecting structural and behavioural change over time. They are mainly used to build predictive systems.

Nature-inspired algorithms are influenced by the social behaviour of biological organisms inside swarms occurring in nature and natural systems, such as bacteria, colonies of ants, flocks of birds, or schools of fish. It is well known that biological entities often engage in a rich repertoire of social interaction that could range from altruistic cooperation to open conflict. One specific kind of social interaction is cooperative problem solving, where a group of autonomous entities work together to achieve a goal. In nature-inspired algorithms, the principles of natural evolution are applied to optimization procedures for the solution of complex problems. For example, in Ant Colony Optimization, a colony of artificial ants is used to construct solutions guided by the pheromone trails and by heuristic information, as specified in Section 2.3.5.

The classification of population-based metaheuristics in evolutionary algorithms and nature-inspired algorithms is not a rigourous classification, because the two classes share common properties. This means that many methods may fit both classes at the same time, because the Darwinian evolution theory used

in evolutionary algorithms is also a nature-inspired process. This is the case, for example, with Genetic Algorithms.

2.3.1 Genetic Algorithms

Genetic Algorithms (GAs) have their origins from the studies of cellular automata conducted by Holland (1975), but only recently their potential for solving combinatorial optimization, linear, and non-linear problems has been exploited (Goldberg et al., 1991; Holland, 1992), becoming the most used evolutionary algorithms. The original motivation for GAs resulted from a biological analogy: if the natural process of selecting the best individuals for reproduction and for the creation of new individuals managed to develop strong species adapted to their environments, would it manage to find good solutions also for optimization problems? In the selective breeding of plants and animals, offspring are sought to receive certain desirable characteristics, determined by the genetic combination of the parents' chromosomes. In the case of GAs, a population of strings (usually referred to, in the literature of evolutionary algorithms, as *chromosomes*) is used in order to obtain genetic recombination. Genetic Algorithms work on finite populations, called also *generations*, and the chromosomes represent candidate solutions to the problem. The elements of a chromosome are called *genes*, and the values that these elements can take *alleles*. In general, this is defined as the "phenotype - genotype" mapping, and it is one of the central points in the development of a good GA (usually this mapping is constituted by a bijection). In most of the cases, the chromosomes are represented by fixed strings with binary values. In this case, an allele is the 0 or 1 value in the bit string, while the position at which the 0 or 1 value is placed in the chromosome is called the *locus*. Each generation evolves under a selective pressure that helps the survival of the fittest individual. Based on the evaluation of a specified criterion of goodness, not only dependent on the value of the objective function, and defined as *fitness*, the strings have a lower or higher probability of being selected for reproduction. Chromosomes are evaluated according to the fitness, and are selectively interbred in pairs to produce offspring, through the genetic operators. The resulting offspring inherit properties directly from their parents. The fitter a chromosome is, the more likely

it is to produce offspring. The offspring are evaluated and placed in the new population, replacing the weaker members. Fitness is a central key to GAs and it is usually defined in order to avoid too flat search space, by creating "valleys" and "mountains", so as to guide properly the search process. The term "Genetic Algorithms" is due to these genetic concepts of representation and manipulation of individuals.

Summarizing, the GA mechanism consists of three phases: evaluation of the fitness of each chromosome, selection of the parent chromosomes, and applications of the genetic operators to the parent chromosomes. When two or more parents are selected for reproduction, GAs use the genetic operators of *crossover* and *mutation*. Crossover is a matter of replacing some of the genes in one parent, with some other genes of the other parent, consequently producing offspring. Mutation is instead applied to a single chromosome, where some of the genes are randomly selected and the corresponding allele values are changed. The evolution process is repeated until the system ceases to improve. The survival of the fittest ensures that the overall solution quality increases as the algorithm proceeds from one generation to the next one. The main steps of the GA approach are specified in Algorithm 2.11.

After the definition of the encoding of an individual, the first relevant point to consider is the size, n_P, and the composition of the initial population P. Concerning the size, it is important to find a good compromise between efficiency (in the terms of computational complexity and time) and efficacy (in terms of quality of the solutions achieved) of the GA. The size of the population can remain unchanged in the following generations (steady state), as in Algorithm 2.11, or vary according to different criteria. As to how the population is chosen ($P = (p[0], p[1], \ldots, p[n_P - 1]) \leftarrow$*Initialize-Population(*$S$*,* n_P*)*), a random creation is commonly assumed, but there are several approaches that use heuristic techniques in order to produce a first population containing already solutions of good quality.

The successive step consists of evaluating the fitness of each individual in the population, and selecting the parents for the genetic operators of crossover and mutation (*Select(*P*,* $f(\cdot)$*)* procedure). The basic idea for the selection of the parents to be mated is that it should be related to fitness, and the original scheme for its implementation is commonly known as the *roulette-wheel method*.

Algorithm 2.11: Genetic Algorithm

Input: A fitness function $f(\cdot)$, the search space S;
Output: A solution $s' \in S$;
Initialisation:
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Set the size n_P of the population $P = (p[0], p[1], \ldots, p[n_P - 1])$;
- Let $P' = (p'[0], p'[1], \ldots, p'[n_P - 1])$ be the next generation;
- Set the probability of crossover: $\mu_c \in [0, 1)$;
- Set the probability of mutation: $\mu_m \in [0, 1) | (\mu_c + \mu_m) < 1$;

begin
  Generate the initial population: $P = (p[0], p[1], \ldots, p[n_P - 1]) \leftarrow$*Initialize-Population(S, n_P)*;
  Evaluate the fitness of each individual in the population:
  $(f(p[0]), f(p[1]), \ldots, f(p[n_P - 1])) \leftarrow$*Evaluate($p[0], p[1], \ldots, p[n_P - 1]$)*;
  repeat
    Set $P' \leftarrow \emptyset$;
    for $i \leftarrow 1$ *to* n_P **do**
      According to the fitness evaluation, select at random an individual from the population:
      $s \leftarrow$*Select(P, $f(\cdot)$)*;
      Select at random a number between 0 and 1: $\xi \leftarrow random[0, 1)$;
      if $\xi \in [0, \mu_c)$ **then**
        Let s be the first parent for the crossover: $p_{c1} \leftarrow s$;
        According to the fitness evaluation, select at random the second parent for the crossover: $p_{c2} \leftarrow$*Select(P, $f(\cdot)$)*;
        Perform the crossover operation: $p_{crossovered} \leftarrow$*Crossover($p_{c1}$, p_{c2})*;
        Add the crossover offspring to the next generation: $P' \leftarrow P' \cup \{p_{crossovered}\}$;
      else if $\xi \in [\mu_c, (\mu_c + \mu_m))$ **then**
        Let s be the parent for the mutation: $p_m \leftarrow s$;
        Perform the mutation operation: $p_{mutated} \leftarrow$*Mutation(p_m)*;
        Add the mutation offspring to the next generation: $P' \leftarrow P' \cup \{p_{mutated}\}$;
      else if $\xi \in [(\mu_c + \mu_m), 1)$ **then**
        Add the selected individual to the next generation: $P' \leftarrow P' \cup \{s\}$;
      end
    end
    Update the generation: $P \leftarrow P'$;
  until *termination conditions* ;
  Select the best individual to date: $s' \leftarrow$*Extract-the-Best(P, $f(\cdot)$)*;
  $\Rightarrow$ *Return(s')*.
end

It uses a probability distribution for selection, in which the selection probability of a given string is proportional to its fitness. Another method, known as the *stochastic universal selection*, proved to be particularly effective for reducing the high stochastic variability of the roulette-wheel method. Another approach is the *tournament selection*, in which a subset of parents is randomly chosen and the best among them is used for parent selection (for details on these approaches see (Glover and Kochenberger, 2003)). Regarding the selection process, GAs can

deal with an *unstructured population*, in which any individual may be recombined with any other one to create offspring, or with a *structured population*, if any individual can be recombined with only those included in a particular set, as is the case of Parallel Genetic Algorithms (Glover and Kochenberger, 2003).

The selection process is followed by the reproduction of the individuals by means of the genetic operators of crossover and mutation. In Algorithm 2.11, crossover is applied with a probability μ_c, while mutation with probability μ_m. For the remaining probability, the selected individuals are simply duplicated in the next generation P' (*generational replacement evolution*). However, there exist other possibilities for the selection process. For example, it is possible to use crossover and mutation at the same time, or use only one of them, or taking into account other tailored mechanisms dependent on the problem addressed. Usually, crossover is always applied and mutation has just a low probability of being selected, since empirical studies show that, with higher probabilities, mutation has the negative effect of reducing the average solution value of the population and disallowing the achievement of new good solutions (Holland, 1992).

Crossover replaces some of the genes in one parent (p_{c1}), with some other genes of the other parent (p_{c2}), consequently producing offspring ($p_{crossovered}$). If the information sources for the crossover operations are just a couple of individuals, as in Algorithm 2.11, it is a case of a *two-parents crossover* scheme. Otherwise, if the offspring are produced by some recombination of more than two parents, it is the case of a *multi-parents crossover*. Recently clever crossover schemes were developed, such as Gene Pool Recombination (using population statistics to generate the individuals of the next population), or the Bit-Simulated Crossover (using a probability distribution over the search space given by the current population to generate the next one).

A problem to avoid in GAs is the premature convergence toward sub-optimal solutions. A correct use of the mutation operator is fundamental to balance the diversification capability of the Genetic Algorithm, trying to avoid premature convergence. Mutation is a simple mechanism which just performs a small random perturbation on the selected individual (noise). Considering the chromosome p_m, the mutation operator consists of randomly selecting some of the genes of p_m and changing the corresponding allele values, producing the new individual $p_{mutated}$.

An alternative approach is to use an *immigration theory*, that operates by including in the new generations individuals either randomly created, or coming from areas not frequently searched during the execution of the algorithm (the history of the evolution has thus to be memorized).

The application of the genetic operators can produce infeasible solutions. There are three different ways to handle infeasible solutions. Infeasible individuals could be simply "rejected", "penalized" (by assigning them an additional poor fitness value, so that they will have difficulty in being reselected in the succeeding steps to create offspring), or just "repaired" (but this is not always possible).

When the next generation of individuals P' is completed, it becomes the new current population ($P \leftarrow P'$), and the algorithm continues with the same procedure until some user termination conditions are satisfied. Then, the best individual s' within the survivors represents the output of the Genetic Algorithm.

Some Genetic Algorithms can include mechanisms to improve the intensification capability of the search process (Glover and Kochenberger, 2003). These mechanisms consist of including local search procedures by means of hybridization with other metaheuristics. Hybridization of GAs with other metaheuristics proved to be very useful, if not necessary, for efficiently addressing many optimization problems. While the use of a population ensures the diversification of the search, the use of local search techniques may improve the intensification factor on the promising zones. This is the case, for example, of Memetic Algorithms (MAs). Memetic Algorithms were first introduced by Moscato (1989) and represent a broad class of evolutionary algorithms. The main idea of MAs is to combine the effective search method of Genetic Algorithms, with the use of specific information related to the optimization problem addressed. As for the evolutionary approaches, MAs use a population of solutions that are combined together through crossover and mutation in order to produce new solutions. The intensification phase is obtained by incorporating heuristics, approximation algorithms, local search, truncated exact methods, and other techniques tailored to the solution of the specific problem, and aimed to quickly identify promising areas in the search space. The term MA is particularly used when the intensification phase is performed with the use of another nested metaheuristic, applied to each

individual of the population. Typically, a MA consists of a GA in which a nested Tabu Search or Simulated Annealing is used (Moscato, 1989).

2.3.2 Quantum-inspired Genetic Algorithms

Quantum-inspired Genetic Algorithms (QGAs) are a family of novel evolutionary algorithms proposed by Narayanan and Moore (1996). They are based on concepts and principles of quantum mechanics, such as standing waves, interference, and coherence, applied to Genetic Algorithms in order to increase their performance. A quantum-inspired computational method generates candidate solutions to the problem instance, and a classical algorithm checks if these solutions are in fact feasible. In order to understand QGAs further, it is necessary to underline some basic principles of quantum mechanics (Feynman and Hibbs, 1965).

An atom consists of a nucleus (containing particles called protons (positive charges) and neutrons (neutral charges)) and electrons (negative charges), surrounding the nucleus through wave orbits (not-planar). There are different types of orbit, depending on two factors: angular momentum and energy level. An electron around a nucleus jumps states in discrete quanta by absorbing photons (from a low energy orbit to an higher energy one) or releasing it (high level to a lower one): the term "quantum" means that in-between states or orbits do not exist, while a "photon" is the smallest unit of energy.

A quantum particle's *location* can be described by a *quantum state vector* $\mid \Psi >$, representing a *linear superposition* (i.e. a weighted sum) of the particle given individual quantum state vectors $\mid A >$, $\mid B >$, $\mid C >$, ..., respectively of the possible positions A, B, C, ..., as follows (Feynman and Hibbs, 1965):

$$\mid \Psi >= \alpha \cdot \mid A > + \beta \cdot \mid B > + \gamma \cdot \mid C > + \ldots, \tag{2.9}$$

where the weighting factors α, β, γ, ..., are complex numbers, which represent the probabilities that the particle is in a specific location ($prob_A = |\alpha|^2$, $prob_B = |\beta|^2$, $prob_C = |\gamma|^2$, ..., respectively). From Heisenberg's uncertainly principle (Feynman and Hibbs, 1965), both the position and momentum of a particle cannot be simultaneously known at any particular instant. Thus, if there are n locations given by n state vectors, the particle is said to be at all n locations at the same

time. However, in the act of observing a quantum state (or wave function), it collapses to a single one. This is a consequence of the *many-universes interpretation* by Everett (1957): since all quantum systems exist in parallel universes, it is not possible to view a quantum system in all these universes but only in a single one. For example, in the case of two universes, the probability P_{12} of arrival of the particle in a specific point is the square of the height of its *quantum amplitude* a_{12} (Narayanan, 1999):

$$P_{12} = a_{12}^2. \tag{2.10}$$

From the analogy with water waves theory, the total amplitude a_{12} is the sum of the wave amplitude of each single universe (Narayanan, 1999):

$$a_{12} = a_1 + a_2. \tag{2.11}$$

Thus, the probability P_{12} is given by:

$$P_{12} = a_{12}^2 = (a_1 + a_2)^2 = a_1^2 + a_2^2 + 2a_1a_2 = P_1 + P_2 + 2a_1a_2; \tag{2.12}$$

that is, the probability P_{12} of arrival of the particle in a specific point is the sum of the probability of having the particle in each single universe and adding an *interference factor*, $2a_1a_2$, due by the scrambling between the universes.

Recently, it was proved that a quantum system could be used to perform computations and to simulate quantum processes, impossible to compute efficiently on a conventional calculator (Narayanan, 1999). The "many universes" interpretation was used by Shor (1994) in his quantum computing method for extracting prime factors of very large integers. This result was used to deal with cryptography algorithms, in which key production methods are based on the seeming intractability of finding the prime factors of very large integers.

Quantum principles were applied to Genetics Algorithms, giving an initial basic methodology to design quantum computational algorithms (Narayanan and Moore, 1996). The following guidelines explain how to develop a Quantum-inspired Genetic Algorithm:

1. Express the problem in a numerical form through specific conversion methods;

2. Determine the initial configuration;
3. Define the terminating conditions;
4. Divide the problem instance into smaller sub-problems;
5. Identify the number of required universes;
6. Assign an universe to each sub-problem;
7. Compute in parallel in the different universes;
8. There must be a form of interaction (interference) between all the universes, which yields a solution or new useful information for the universes.

An important difference between the classical GAs and QGAs is in the representation of the elementary information unit. If GAs are based on bits, QGAs are based on *QuBits*, derived by the superposition principle of quantum mechanics. The QuBit does not represent only the value 0 or 1, but a superposition of the two bits. Its state is represented as follows:

$$| \Psi >= \alpha \cdot | 0 > + \beta \cdot | 1 >, \tag{2.13}$$

where $| 0 >$ and $| 1 >$ are the classical bit values 0 and 1, and α and β are complex numbers whose square values, $|\alpha|^2$ and $|\beta|^2$, stand respectively for the probability to measure the value 0 for the QuBit ($prob_0 = |\alpha|^2$), and that to measure 1 ($prob_1 = |\beta|^2$). That is:

$$|\alpha|^2 + |\beta|^2 = 1. \tag{2.14}$$

In the case of multiple QuBits, as in a quantum system, the resulting state space grows exponentially with respect to the number of particles. For example, in the case of ρ QuBits, the state space has 2^ρ dimensions, and its representation is defined as follows (Narayanan, 1999):

$$\left[\begin{array}{c|c|c|c} \alpha_1 & \alpha_2 & \cdots & \alpha_\rho \\ \beta_1 & \beta_2 & \cdots & \beta_\rho \end{array}\right], \tag{2.15}$$

where $|\alpha_i|^2 + |\beta_i|^2 = 1$, with $i = 1, \ldots, \rho$. Each chromosome in such a QGA is encoded as a matrix of 2 x ρ QuBits. This allows a chromosome to encode

not only one solution, but all the possible solutions by using the superposition principle. Again, $|\alpha_i|^2$ and $|\beta_i|^2$ are the probabilities to measure respectively the value $\mid 0 >$ and the value $\mid 1 >$ for the QuBit i of a certain chromosome. Each quantum operation regards in parallel all the states present within the superposition (characteristic of diversity (Narayanan, 1999)). Only one QuBit chromosome is enough to represent c states, while in a classical bit representation at least c chromosomes are needed. This means that the QuBit representation possesses simultaneously the two characteristics of exploration and exploitation. If $|\alpha_i|^2$ or $|\beta_i|^2$ converges to 1 or 0, the QuBit i of the chromosome considered stretches to a single state (0 or 1 value), and the property of diversity disappears

Algorithm 2.12: Quantum-inspired Genetic Algorithm

Input: A fitness function $f(\cdot)$, the search space S;
Output: A solution $s' \in S$;
Initialisation:
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Set the number ρ of QuBits for each chromosome;
- Set the size n_P of the population $P = (p[0]^{2x\rho}, p[1]^{2x\rho}, \ldots, p[n_P - 1]^{2x\rho})$;
- Let $P' = (p'[0]^{2x\rho}, p'[1]^{2x\rho}, \ldots, p'[n_P - 1]^{2x\rho})$ be the next generation;
- Set the probability of crossover: $\mu_c \in [0, 1)$;
- Set the probability of mutation: $\mu_m \in [0, 1) | (\mu_c + \mu_m) < 1$;

begin
  Generate the initial population:
  $P = (p[0]^{2x\rho}, p[1]^{2x\rho}, \ldots, p[n_P - 1]^{2x\rho}) \leftarrow$*Initialize-Population(S, n_P, ρ)*;
  Evaluate the fitness of each individual in the population:
  $(f(p[0]), f(p[1]), \ldots, f(p[n_P - 1])) \leftarrow$*Evaluate(*$p[0]^{2x\rho}, p[1]^{2x\rho}, \ldots, p[n_P - 1]^{2x\rho}$*)*;
  According to the fitness evaluation, select the best individual in the population:
  $s \leftarrow$*Extract-the-Best(P, $f(\cdot)$)*;
  Set $s' \leftarrow s$;
  repeat
    Set $P' \leftarrow \emptyset$;
    Perform the quantum interference operation: $P \leftarrow$*Interference(P, s')*;
    According to the probability μ_c, perform the crossover operation: $P' \leftarrow$*Crossover(P, μ_c)*;
    According to the probability μ_m, perform the mutation operation: $P' \leftarrow$*Mutation(P, μ_m)*;
    Perform at random the shifting operation: $P' \leftarrow$*Shifting(P)*;
    Update the generation: $P \leftarrow P'$;
    According to the fitness evaluation, select the best individual in the population:
    $s \leftarrow$*Extract-the-Best(P, $f(\cdot)$)*;
    if $f(s) < f(s')$ **then**
      Move $s' \leftarrow s$;
    end
  until *termination conditions* ;
  $\Rightarrow$ *Return(s').*
end

gradually.

The procedure for a general Quantum-inspired Genetic Algorithm, starting from an initial population, applies four quantum operators (*quantum interference*, *crossover*, *mutation*, *shifting*), and an evaluation. The evaluation is a special kind of measurement applied to the resulting solutions in order to extract the corresponding fitness values. Formally, the procedure can be specified in Algorithm 2.12. At the beginning, the initial population P, composed of n_P "quantum chromosomes", each one containing ρ QuBits, is randomly generated ($P \leftarrow$*Initialize-Population*$(S,\ n_P,\ \rho)$). The most common choice is to generate P at random. After, the four operators of quantum interference, crossover, mutation, and shifting are applied.

The first operator is the quantum interference that allows a shift of each QuBit of the chromosomes in P in the direction of the corresponding bit value in the best solution. That is performed by rotating the specific QuBit by an angle, $\pm\delta$, which is a function of the value of the corresponding bit in the best solution, called the *reference bit*. Consider a QuBit i, with a probability to measure the value 0 equal to α_i, and a probability to measure the value 1 equal to β_i. Table 2.2 gives the value of the rotation angle, $\pm\delta$, in function of the current probability values α_i and β_i, and of the corresponding bit in the best solution (reference bit). Figure 2.3 shows the results of the rotation of the QuBit i, performed by the quantum interference operator according to the corresponding reference bit.

Table 2.2: Example of the rotation angle of a QuBit i in function of the current probability to measure the value 0 (α_i), of the current probability to measure the value 1 (β_i), and of the corresponding bit value of the best solution (reference bit)

α_i	β_i	Reference bit	Rotation angle
> 0	> 0	1	$+\delta$
> 0	> 0	0	$-\delta$
> 0	< 0	1	$-\delta$
> 0	< 0	0	$+\delta$
< 0	> 0	1	$-\delta$
< 0	> 0	0	$+\delta$
< 0	< 0	1	$+\delta$
< 0	< 0	0	$-\delta$

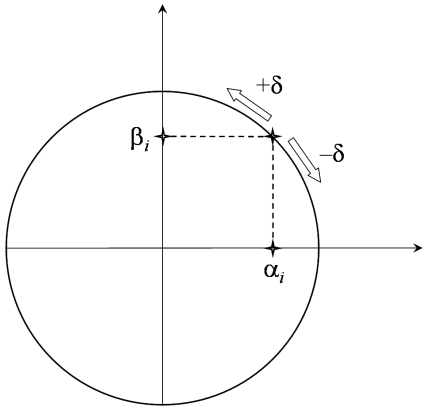


Figure 2.3: Rotation of the QuBit i performed by the quantum interference operator according to the corresponding reference bit in a Quantum-inspired Genetic Algorithm.

The second operation is a classical GA crossover performed, with probability μ_c, between pairs of chromosomes, within the population P, selected at random positions. The resulting crossover offspring constitutes part of the next generation ($P' \leftarrow$*Crossover(P, μ_c)*). According to the probability μ_m, the successive mutation operator is applied at random over some chromosomes ($P' \leftarrow$*Mutation(P, μ_m)*). Note that the probability μ_m depends on the probability of applying crossover μ_c, that is $\mu_m \in [0, 1) | (\mu_c + \mu_m) < 1$.

The fourth operation ($P' \leftarrow$*Shifting(P)*) consists of a random shifting of some chromosomes, in order to further increase the diversification of the search process. The shifting is obtained by permuting the columns of each chromosome with other columns. After these four quantum operators, an evaluation of the fitness of each chromosome within the population is applied in order to select the best individual to date. The evaluation is a special kind of measurement applied to the resulting solutions within the population to extract their corresponding fitness values. In quantum mechanics, only states containing exactly one QuBit with the value 1 in each line, and exactly one QuBit having the value 1 in each column (coherent solutions) are possible. Conversely, in QGA, the final measurement does not destroy the states superposition, keeping all the possible solutions for the following iterations. After the evaluation of the solutions, a new population for the next iteration is selected ($P \leftarrow P'$). The population P will consist of the

best $(n_P - 1)$ chromosomes from the generation P' obtained by the operators, plus one chromosome randomly selected among the other ones (in order to maintain a good diversity). The algorithm continues iteratively until the user termination conditions are satisfied. Then, the best solution to date (s') is produced as output of the algorithm.

The increased performance of Quantum-inspired Genetic Algorithms with respect to classical Genetic Algorithms may be attributed mainly to the interference operation and to the multiple superpositions of individuals, obtained by representing the chromosomes with QuBits. The quantum interference operator provides a larger number of chromosomes to choose for the next generation, while the multiple superpositions of individuals allow losing less good solutions during each step.

If progress continues at this rate, future computer circuits will be based on nanotechnology and the behaviour of such circuits will have to be given in quantum mechanical terms rather than in terms of classical physics (since on the atomic scale matter obeys the laws of quantum mechanics). Such compilers will require less translation to machine language than the classical ones, so carrying efficiency benefits. Although it is currently not clear how true quantum computation algorithms will be related to quantum hardware (e.g. quantum logic gates), quantum-inspired computing could help quantum hardware platforms to be feasible. The increased performance of Quantum-inspired Genetic Algorithms with respect to classical Genetic Algorithms has been recently demonstrated for some classical combinatorial optimization problems, such as the travelling salesman problem (Talbi et al., 2004).

2.3.3 Estimation of Distribution Algorithms

Genetic Algorithms are optimization techniques based on selection and recombination of promising solutions. Their behaviour depends on the setting of the genetic operators of selection, crossover, and mutation, and on the choice of many parameters, such as population size, probabilities of crossover and mutation, rate of generational reproduction, and number of iterations. However, interactions

among the variables of the search space are not explicitly considered. Furthermore, the fixed two-parents crossover and mutation sometimes provide low quality solutions in the next generations. Two-parents crossover can been replaced by generating new solutions according to a probability distribution associated with the variables of the search space. This new approach was introduced by Mühlenbein and Paaß (1996) and used in the so called Estimation of Distribution Algorithms (EDAs).

In EDAs, interactions among the variables of the individuals are explicitly expressed through the joint probability distribution associated to the variables that are present in a database of individuals selected from the previous generation. The estimation of this joint probability distribution is not an easy task, and different methods can be used. The method determines the form of Estimation of Distribution Algorithm. Afterwards, the offspring for the next generation are created by sampling the joint probability distribution. The evaluation of the individuals used by EDAs is based on fitness measurement, as with Genetic Algorithms, but neither crossover nor mutation is applied. Formally, the EDA approach can be summarized in Algorithm 2.13.

The algorithm starts by generating an initial population P of n_P individuals. Then, $n'_P < n_P$ individuals are selected to form the next generation ($P' \leftarrow$*Select(P, n'_P)*). Using one of the EDA methods, the successive step calculates the joint probability distribution, $prob(x|P')$, of the variables x that are present in the selected individuals P' (*Estimation distribution(P, n'_P)*). Offspring are generated by just sampling the probability distribution, and replacing the old population ($P \leftarrow$*Sampling(P', $prob(x|P')$)*). The algorithm is repeated iteratively until the termination conditions are satisfied, producing the best solution to date (s') as output.

Different methods can be used to estimate the joint probability distribution $prob(x|P')$, determining different Estimation of Distribution Algorithms. In specific problems, the particular method is selected according to the dependencies among the variables of the search space. Univariate Marginal Distribution Algorithm, Population Based Incremental Learning, and Compact Genetic Algorithm are different EDAs which do not consider interaction among variables (univariate variables). In this case, the joint probability distribution can be simply calculated

Algorithm 2.13: Estimation of Distribution Algorithm

Input: A fitness function $f(\cdot)$, the search space S;
Output: A solution $s' \in S$;
Initialisation:
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Set the size n_P of the population $P = (p[0], p[1], \ldots, p[n_P - 1])$;
- Set the size n'_P of the next generation $P' = (p'[0], p'[1], \ldots, p'[n'_P - 1])$, where $n'_P < n_P$;
- Let $prob(x|P')$ be the joint probability distribution of the variables x that are present in the individuals in P';

begin
  Generate the initial population: $P = (p[0], p[1], \ldots, p[n_P - 1]) \leftarrow$*Initialize-Population(S, n_P)*;
  Evaluate the fitness of each individual in the population:
  $(f(p[0]), f(p[1]), \ldots, f(p[n_P - 1])) \leftarrow$*Evaluate($p[0], p[1], \ldots, p[n_P - 1]$)*;
  According to the fitness evaluation, select the best individual in the population:
  $s \leftarrow$*Extract-the-Best(P, $f(\cdot)$)*;
  Set $s' \leftarrow s$;
  repeat
    Select n'_P individuals for the next generation: $P' \leftarrow$*Select(P, n'_P)*;
    Estimate the joint probability distribution of the variables x that are present in the individuals in P': $prob(x|P') \leftarrow$*Estimation distribution(P, n'_P)*;
    Generate offspring by sampling the joint probability distribution:
    $P \leftarrow$*Sampling(P', $prob(x|P')$)*;
    According to the fitness evaluation, select the best individual in the population:
    $s \leftarrow$*Extract-the-Best(P, $f(\cdot)$)*;
    if $f(s) < f(s')$ **then**
      Move $s' \leftarrow s$;
    end
  until *termination conditions* ;
  $\Rightarrow$ *Return(s').*
end

as the product of the marginal probabilities of each single variable (Larrañaga and Lozano, 2001). In *Univariate Marginal Distribution Algorithm*, the joint probability distribution is factorized as a product of independent univariate marginal distributions, estimated from marginal frequencies. In the case of *Population Based Incremental Learning*, the joint probability distribution is represented by a vector of probability distributions: $(prob_g(x_0|P'), prob_g(x_1|P'), \ldots, prob_g(x_i|P'), \ldots)$, where $prob_g(x_i|P')$ refers to the probability of obtaining a 1 in the i-th variable of the search space in the g-th generation. At each step, the next generation of individuals is obtained by sampling the vector of probability distributions. At each iteration, a number of best individuals in the current generation are selected in order to update the probability vector by a rule, which shifts the vector towards the best individuals. *Compact Genetic Algorithm* considers also a vector

of probability distributions, as Population Based Incremental Learning. In Compact Genetic Algorithm, the probability for each variable is initialized to a 0.5 value. Then, using this vector of probabilities, the method randomly generates new individuals. An evaluation of their objective function values provides a ranking of individuals. The probability distributions are shifted toward the generated solution vector(s) with highest quality. The distance that the probability distributions are shifted depends on a learning rate parameter. At this step, a mutation operator may be further applied to the probability distributions. This procedure is repeated iteratively until the vector of probability distributions converges to a local optimum. The vector of probability distributions can be regarded as a prototype vector for generating high-quality solution vectors with respect to the available knowledge about the search space. The drawback of this method is the fact that it does not automatically provide a way to deal with constrained problems (Larrañaga and Lozano, 2001).

To solve problems with variables with pairwise interactions (bivariate dependencies), other Estimation of Distribution Algorithms exist, such as Mutual Information Maximizing Input Clustering, Combining Optimizers with Mutual Information Trees, and Bivariate Marginal Distribution Algorithm (Larrañaga and Lozano, 2001). For real-world problems, where multiple interactions occur, the followed EDAs are used: Factorized Distribution Algorithm, Extended Compact Genetic Algorithm, Bayesian Optimization Algorithm, Estimation of Bayesian Network Algorithm (Larrañaga and Lozano, 2001). Bayesian Optimization Algorithm, for example, estimates the joint probability distributions of selected individuals using modelling data from Bayesian Networks. The Bayesian metric, used to measure the goodness of each structure, has the property that structures reflecting the same conditional dependency or independency have the same scores (Larrañaga and Lozano, 2001). In order to reduce the cardinality of the search space, the algorithm imposes restrictions on the number of parents a node may have (for problems where a node may have more than 2 parents, the situation is complicated to solve).

The field of EDAs is still quite young, and nowadays much of the research effort is focused on methodology rather than high-performance applications.

2.3.4 Scatter Search

Scatter Search (SS) is a novel evolutionary algorithm compatible with randomized implementations, but not based on randomization as in the case of the other evolutionary approaches (Glover et al., 2000). It joins solutions by generalized path constructions (in both Euclidean and neighbourhood spaces) and utilizing strategic designs, instead of exclusively using randomization. Scatter Search embodies strategies still not emulated by other evolutionary methods. The approach has been shown to be advantageous for solving a variety of complex optimization problems (Glover et al., 2003).

Scatter Search captures information not separately contained in the original vectors. It takes advantage of auxiliary heuristic methods both for selecting the elements to be combined and for generating new vectors. It linearly combines solutions from a set, called the *reference set*, in order to create new ones. In the example specified in Figure 2.4, the original reference set consists of the solutions labelled A, B and C (Glover et al., 2000). After a non-convex combination of the reference solutions A and B, a number of new solutions in the line segment defined by A and B are created; in the example only solution 1 is introduced into the reference set. In a similar way, other convex and non-convex combinations

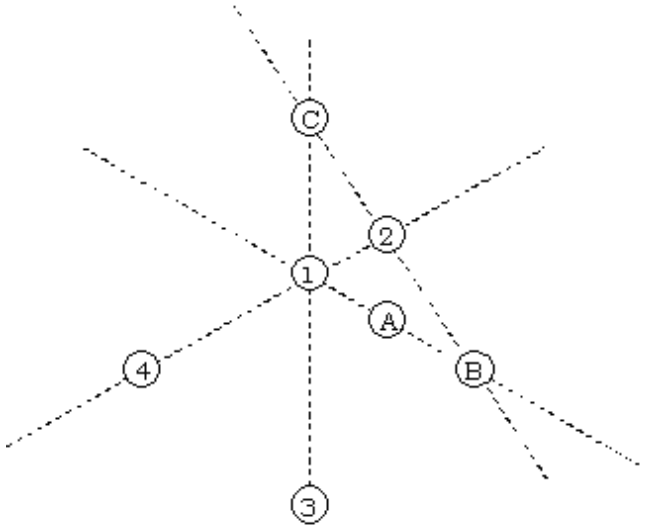


Figure 2.4: Example of reference set in Scatter Search.

between original and newly created reference solutions, produce points 2, 3, and 4. Finally, the resulting reference set is composed of seven solutions (or elements).

Scatter Search does not leave solutions in a raw form after the combination mechanism, but applies heuristic improvements to the candidates for entry into the reference set. Unlike a "population" in Genetic Algorithms, the reference set of solutions in Scatter Search is relatively small. A typical GA population size consists of 100 elements, which are randomly sampled to create combinations. In contrast, Scatter Search systematically chooses two or more elements of the reference set to create new solutions. If the reference set consists of b solutions, experimentally the procedure will examine around $(3b - 7) \cdot b/2$ elements, and so there is a practical need for keeping the cardinality of the reference set small. Typically, the reference set in Scatter Search has 20 solutions or less. Moreover, Genetic Algorithms need large populations to maintain a good level of diversification (for the random sampling embedded in its search mechanisms), while Scatter Search systematically injects diversity to the reference set. To limit the scope of the search to a selective group, a mechanism for controlling the number of possible combinations in a given reference set can be used. The reference set is divided into "tiers" and combined solutions must include at least one of the elements from each of them.

The Scatter Search approach may be outlined as follows (Glover et al., 2000):

1) Generate a starting set of solution vectors to guarantee a critical level of diversity. Apply custom heuristic processes to try to improve these solution vectors. The reference solutions will be a subset of the best vectors. A solution may be added to the reference set if the diversification factor of the set improves, even if its objective value is inferior to other solutions competing for admission into the set.

2) Create new solutions consisting of structured combinations of subsets of the current reference solutions. These combinations are chosen to produce points both inside and outside the convex regions spanned by the reference solutions, and they are modified to become acceptable solutions.

3) Apply the heuristic processes (already used to generate the reference set) to improve the solutions created. These heuristic processes must be able also to operate on infeasible solutions to restore feasibility if possible.

4) Extract a collection of the "best" improved solutions from the last step and add them to the reference set. The notion of "best" is once again broad, as in the step 1. Steps 2, 3, and 4 are repeated until the reference set does not change. Moreover, the reference set is periodically diversified restarting from step 1. When reaching a specified iteration limit the algorithm will stop.

The goal of structured combinations in Scatter Search is to create weighted centres of the selected sub-regions. Another important feature relates to the construction of new solutions "within" and "across" clusters of points. Finally, Scatter Search employs subordinate mechanisms to improve infeasible solutions, in order to make it possible for them to be included into the reference set.

The main general behaviour of Scatter Search is specified in the following routines (Blum and Roli, 2003):

- *Seed-Generation*: one or more seed trial solutions are created to initialize the algorithm;
- *Diversification-Generator*: a collection of diverse trial solutions are generated from an arbitrary seed as input;
- *Improvement*: a local search method transforms a trial solution into one or more enhanced ones (neither the input nor the output solutions are required to be feasible);
- *Reference-Set-Update*: the reference set, consisting of the "best" found solutions (typically small values, e.g. no more than 20 elements), is produced. Solutions gain membership of the reference set according to their quality or their diversity values;
- *Subset-Generation*: a subset of solutions from the reference set is generated as a basis for creating combined solutions;
- *Solution-Combination*: the solutions within the subset obtained from the reference set are transformed into one or more combined solution vectors.

From a spatial orientation, in Scatter Search new solutions are created by linear combinations of reference solutions using both positive and negative weights.

The resulting points can be both inside and outside the convex region spanned by the reference set. By natural extension, such combinations may be paths, generated between and beyond selected solutions in neighbourhood space rather than in Euclidean space. This SS extension is called *Path Relinking*. A path between solutions in a neighbourhood space will produce new solutions sharing a subset of attributes contained in the parent solutions. The attributes vary according to the path selected and the location on the path. Such paths are specified by the solution attributes that are added, dropped or modified by the moves executed in neighbourhood space. To generate the desired paths starting from an initiating solution, the moves must progressively introduce (or subtract) attributes by a guiding solution. This step consists of the incorporation of attributes from elite parents in partially or fully constructed solutions by means of heuristic methods. It is carried out by isolating assignments occurring frequently or influentially in high quality solutions, and then introducing them into other solutions (implicit form of frequency-based memory). Moreover, the possibilities of multi-parent path generation emerge in Path Relinking. Typically, the generation of such paths "relinks" previous points in the neighbourhood space in ways not achieved from the search history (hence giving the approach its name). Path Relinking is often used as a hybrid component in metaheuristics, such as Tabu Search and Greedy Randomized Adaptive Search Procedure.

The evolutionary Scatter Search and Path Relinking have proved unusually effectiveness for solving diverse optimization problems, from both classical and real-world settings (Glover et al., 2003). For example, they have been applied with success to the multi-objective quadratic assignment problem, the vehicle routing problem, job shop scheduling, and mixed integer programming.

2.3.5 Ant Colony Optimization

Ant Colony Optimization (ACO) is a recent nature-inspired metaheuristic for solving combinatorial optimization problems, proposed in the early 90's by Marco Dorigo and colleagues (see for example (Colorni et al., 1992)). As Dorigo and Stützle (2004) state, its inspiring source is the foraging behaviour of real ants. When searching for food, ants initially explore the area surrounding their nest in

a random manner. As soon as an ant finds a food source, it evaluates quantity and quality of the food and carries some of the found food to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone deposited, which may depend on the quantity and quality of the food, will guide other ants to the food source. The indirect communication between the ants via the pheromone trails allows them to find shortest paths between their nest and food sources (Figure 2.5). This functionality of real ant colonies is exploited in artificial models in order to solve discrete optimization problems.

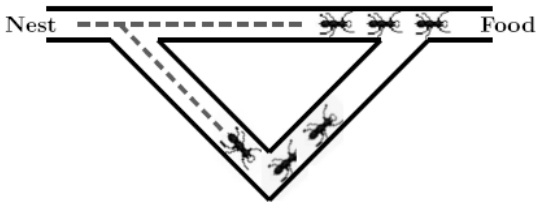


Figure 2.5: Foraging behaviour of real ants.

As an analogy to the biological example, ACO is based on the indirect communication of a colony of simple agents, called *artificial ants*, mediated by *artificial pheromone trials*. The pheromone trial in ACO is distributed numerical information, which is used by the ants for probabilistically constructing solutions to the problem being solved, and updated, by the same ants, during the execution of the process. Thus, Ant Colony Optimization may be also associated to the class of constructive metaheuristics.

The parameterized probabilistic mathematical model used by ACO is called *pheromone model*. The artificial ants perform randomized walks on a completely connected *construction (or decision) graph* $G = (C, L)$, whose vertices, c_i, are appropriately defined solution components, C, and the set of edges L constitutes the connections $\ell_{i,j}$ between these components (Figure 2.6).

The artificial ants incrementally construct solutions by adding solution components c_i to a partial solution under consideration. *Pheromone trail parameters*, τ_i and $\tau_{i,j}$, are associated, respectively, with every node c_i and each arc $\ell_{i,j}$,

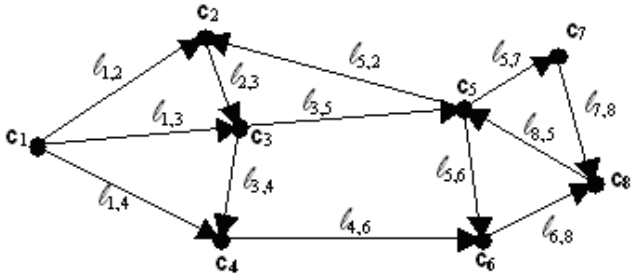


Figure 2.6: Example of a decision (or construction) graph.

which also has assigned *a priori or run time heuristic values*, respectively η_i and $\eta_{i,j}$ (Dorigo and Stützle, 2004). It is possible to define the following useful sets:

- $T = \{\tau_i, \tau_{i,j}\} \Rightarrow$ set of *pheromone trail parameters*;
- $H = \{\eta_i, \eta_{i,j}\} \Rightarrow$ set of *heuristic values*.

Ants do not move arbitrarily on the graph, but rather follow a construction policy which is a function of the problem constraints. The values T and H are used by the ants to take probabilistic decisions on how to move on the decision graph. The probabilities involved in moving on the construction graph are commonly called *transition probabilities*. The set of pheromone trail parameters T is associated with components and connections, and is iteratively updated. This set encodes a long term memory concerning the whole process. It is important to note that ants move independently one from the other. A single ant has a low probability of finding the global optimum, but the collection of the number of ants composing the colony, generally large, has an overall stronger probability. The approach obtained by the movements of the ants through adjacent states of the graph, allow the ants to construct solutions. The evaluation of this (eventually) partial solution is used in order to update the pheromone. The use of a colony of ants gives the algorithm increased robustness, and in many ACO applications the collective interaction of a population of agents is needed to efficiently solve a problem.

The simplest ACO algorithm is the *Ant System* (AS), which is based on the pheromone trail parameters T and the set of heuristic values H. Ant System is

specified in Algorithm 2.14. Given a set of artificial ants A, the algorithm first initializes the pheromone trail parameters T and the heuristic values H. The most common choice is to assign a positive constant number to these values, i.e. $\eta_i = \eta_{i,j} = \tau_i = \tau_{i,j} = const > 0$. Then, each ant $a \in A$ iteratively constructs a solution s_a to the problem ($s_a \leftarrow$*Construction(*T*,* H*)*). In this phase, an ant incrementally builds a solution by adding probabilistic-chosen components (by means of transition probabilities) to the partial solution constructed so far. Only feasible solution components can be added to the current partial solution. However, in particular circumstances where it is necessary or desirable, ants can also construct infeasible solutions. Considering a single ant $a \in A$ and the incomplete solution s_a constructed by a, the transition probability associated with a

Algorithm 2.14: Ant System

Input: An objective function $f(\cdot)$, a quality function $F(\cdot)$, the search space S;
Output: A solution $s' \in S$;
Initialisation:
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Let A be the set of n_a artificial ants;
- Let $T = \{\tau_i, \tau_{i,j}\}$ be the set of pheromone trail parameters;
- Let $H = \{\eta_i, \eta_{i,j}\}$ be the set heuristic values;
- Let ρ be the pheromone evaporation rate;

begin
  Initialize the pheromone trail parameters: $T = \{\tau_i, \tau_{i,j}\} \leftarrow$*Initialize-Pheromone(*$S$*)*;
  Initialize the heuristic values: $H = \{\eta_i, \eta_{i,j}\} \leftarrow$*Initialize-Heuristic-Values(*S*)*;
  Set $s' \leftarrow \emptyset$;
  repeat
    for $a \leftarrow 1$ *to* n_a **do**
      Set $s_a \leftarrow \emptyset$;
      $s_a \leftarrow$*Construction(*T*,* H*)*;
    end
    Apply the online delayed pheromone update rule to each solution component c_j:
    $\tau_j = (1-\rho) \cdot \tau_j + \sum_{a=1}^{n_a} \Delta\tau_j^{s_a}$; where $\Delta\tau_j^{s_a} = \begin{cases} F(s_a) & \text{if } c_j \text{ is included in } s_a \\ 0 & \text{otherwise} \end{cases}$
    According to the objective function, select the best solution obtained by the ants:
    $s \leftarrow$*Extract-the-Best(*A*,* $f(\cdot)$*)*;
    if $f(s) < f(s')$ **then**
      Move $s' \leftarrow s$;
    end
  until *termination conditions* ;
  $\Rightarrow$ *Return(*s'*)*.
end

component c_j is given by the following *state transition rule*:

$$prob(c_j|s_a[c_k]) = \begin{cases} \dfrac{[\eta_j]^\alpha \cdot [\tau_j]^\beta}{\sum_{c_u \in J(s_a[c_k])} [\eta_u]^\alpha \cdot [\tau_u]^\beta}, & \text{if } c_j \in J(s_a[c_k]) \\ 0 & \text{otherwise,} \end{cases} \tag{2.16}$$

where the above parameters have the following meaning:

- $prob(c_j|s_a[c_k])$ is the probability of adding the component c_j to the partial solution, $s_a[c_k]$, constructed by the ant a so far (c_k is the last node added to s_a);
- α and β are positive constant weights which adjust, respectively, the relative importance of the heuristic value H and of the pheromone trail parameters T;
- $J(s_a[c_k])$ is the set of solution components allowed to be added to the partial solution $s_a[c_k]$.

Once all ants have constructed a solution, the pheromone trail parameter of each solution component, c_j, is updated according to the following *online delayed pheromone update rule*:

$$\tau_j = (1-\rho) \cdot \tau_j + \sum_{a=1}^{n_a} \Delta\tau_j^{s_a},$$

$$\text{with } \Delta\tau_j^{s_a} = \begin{cases} F(s_a) & \text{if } c_j \text{ is included in } s_a \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

and where $\rho \in]0,1]$ is the *pheromone evaporation rate*, which is a parameter balancing the strength of the pheromone update rule, and $F(\cdot)$ is the *quality function* satisfying: $\forall s_1 \in S$, $s_2 \in S$ such that $s_1 \neq s_2$, if $f(s_1) < f(s_2)$ then $F(s_1) > F(s_2)$.

In the online delayed pheromone update rule, each ant iteratively ("online") retraces the path backwards ("delayed") and updates the pheromone trail parameters, according to the degree of excellence of the solution considered. The result of the update rule, in practice, consists of increasing the pheromone trail parameters of solution components that have been found in high-quality solutions.

In this way, the ants' experience accumulated during the search process is used to influence the solution construction in future iterations of the algorithm. The use of heuristic information guides the ants towards the most promising areas of the search space, while the stochastic component allows the ants to build also a variety of different solutions. At each iteration the best solution so far is selected, and the algorithm continues until the user termination conditions are satisfied. The output of the method is the best solution to date (s').

The general Ant Colony Optimization (Algorithm 2.15) can be obtained as an extension of Ant System. The construction method followed by ACO is the same of that used by AS. Each ant builds a solution moving through the decision graph G, following the same state transition rule pointed out in AS. This mechanism

Algorithm 2.15: Ant Colony Optimization

Input: An objective function $f(\cdot)$, a quality function $F(\cdot)$, the search space S;
Output: A solution $s' \in S$;
Initialisation:
- Let $s \in S$ be a generic solution;
- Let $s' \in S$ be the best solution to date;
- Let A be the set of n_a artificial ants;
- Let $T = \{\tau_i, \tau_{i,j}\}$ be the set of pheromone trail parameters;
- Let $H = \{\eta_i, \eta_{i,j}\}$ be the set heuristic values;
- Let ρ be the pheromone evaporation rate;

begin
  Initialize the pheromone trail parameters: $T = \{\tau_i, \tau_{i,j}\} \leftarrow$*Initialize-Pheromone(S)*;
  Initialize the heuristic values: $H = \{\eta_i, \eta_{i,j}\} \leftarrow$*Initialize-Heuristic-Values(S)*;
  Set $s' \leftarrow \emptyset$;
  repeat
    for $a \leftarrow 1$ *to* n_a **do**
      Set $s_a \leftarrow \emptyset$;
      $s_a \leftarrow$*Construction(T, H)*;
      Apply the online step-by-step pheromone update rule: *Step-by-step update(s_a, T, H)*;
    end
    Apply the online delayed pheromone update rule to each solution component c_j:
    $\tau_j = (1-\rho)\cdot\tau_j + \sum_{a=1}^{n_a} \Delta\tau_j^{s_a}$; where $\Delta\tau_j^{s_a} = \begin{cases} F(s_a) & \text{if } c_j \text{ is included in } s_a \\ 0 & \text{otherwise} \end{cases}$
    According to the objective function, select the best solution obtained by the ants:
    $s \leftarrow$*Extract-the-Best(A, $f(\cdot)$)*;
    if $f(s) < f(s')$ **then**
      Move $s' \leftarrow s$;
    end
    Apply the pheromone evaporation: *Evaporation(T)*;
    Apply the daemon offline pheromone updates: *Daemon-updates(T, H, ρ)*;
  until *termination conditions* ;
  $\Rightarrow$ *Return(s').*
end

makes use of a sort of memory because each ant keeps the partial solution it has built in terms of path, but with an ability to retrace backwards. Ant Colony Optimization extends Ant System by adding three components, which are designed and synchronized in relation to the requirements of the specific problem. The first consists of an improvement in the pheromone update method. Besides the online delayed pheromone update rule, another real time update rule is used, called *online step-by-step pheromone update rule*. This consists of updating step-by-step the pheromone trail parameters T during the construction phase, when an ant a is walking on connection $\ell_{i,j}$ in order to reach a component to add to its current partial solution s_a.

The second additive component is the mechanism of *pheromone evaporation*. The pheromone values, τ_i, decrease with time to avoid rapid convergence to local minima, due to the nature of the delayed and step-by-step pheromone update rules. This mechanism represents a form of "forgetting" which increases the diversification capability of the search process, by allowing new areas of the search domain to be explored. The third component consists of the (optional) application of *daemon offline pheromone updates*. For example, a daemon entity may collect global information about the path found by each ant, and can decide whether to apply additional weight (pheromone bias) to the pheromone trail parameters of the components used by the ant that built the best solution. The application of such centralized action on the algorithm is aimed at increasing the intensification capability of the search process.

There exist different ACO implementations in the literature. Currently, the best performing are *Ant Colony System* and *MAX-MIN Ant System* (Dorigo and Stützle, 2004). Ant Colony System extends the basic Ant System by adding an online step-by-step pheromone update rule and daemon offline pheromone updates, already explained in the general ACO. However, the mechanism used by Ant Colony System in the online delayed and step-by-step pheromone update rules is different than the mechanism used by ACO (Dorigo and Stützle, 2004). Ant Colony System uses "pseudo random-proportional update rules" to decide where each ant in the decision graph should be moved. The first mechanism involves deterministic moves (in a greedy manner) to intensify the search around

high-quality solutions, while the second includes random movements, as the usual online pheromone update rules, to diversify the search process.

In contrast, MAX-MIN Ant System extends the basic Ant System by considering an alternative strategy (Dorigo and Stützle, 2004). First, it adds daemon offline pheromone updates. Then, it considers "bounded values" for the pheromone trail parameters T. These values are bounded in a finite interval $[\tau_{min}, \tau_{max}]$, after being initialized to τ_{max}. In this way, the probability of constructing a solution can not exceed a minimum threshold value (a lower bound ≥ 0), previously fixed. Thus, solutions apparently of medium/low-quality have the chance to find a global optimum, by increasing the diversification factor of the search. In addition, MAX-MIN Ant System periodically re-initializes the values of the pheromone trail parameters in order to further encourage the diversification factor of the search.

In spite of many cases in which ACO could not reach the results obtained by other metaheuristics, the approach is still being used to address several optimization problems (Dorigo and Stützle, 2004), among which quadratic assignment, vehicle routing, sequential ordering and scheduling. Current research intent is concerned with the use of ACO with other metaheuristics in order to create efficient hybrids. Similarities between ACO and probabilistic learning algorithms have been found, such as with Estimation of Distribution Algorithms. For more details see (Blum and Roli, 2003).

2.3.6 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a nature-inspired algorithm first proposed by Kennedy and Eberhart (1995). It has been applied with success in many areas and appears to be a suitable approach for several optimization problems (Kennedy and Eberhart, 2001). Particle Swarm Optimization is a population-based technique, inspired by the social behaviour of individuals (or particles) inside swarms in nature, such as flocks of birds or schools of fish. Solutions of the problem are modelled as members of the swarm which fly in the solution space. Evolution of the swarm is obtained from the continuous movement of the particles that constitute the swarm submitted to the effect of inertia and the attraction of the

members who lead the swarm. Thus, Particle Swarm Optimization also belongs to the class of evolutionary algorithms. However, unlike classic evolutionary approaches as Genetic Algorithms, it has no crossover and mutation operators and is easy to implement, requiring few parameter settings and low computational memory.

The standard Particle Swarm Optimization considers a swarm SW containing n_{sw} particles ($SW = 1, 2, \ldots, n_{sw}$) in a d-dimensional continuous solution space (Kennedy and Eberhart, 2001). Each i-th particle of the swarm has a position $x_i = (x_{i1}, x_{i2}, \ldots, x_{ij}, \ldots, x_{id})$, and a velocity $v_i = (v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots, v_{id})$. The position x_i represents a solution to the problem, while the velocity v_i gives the rate of change for the position of particle i at the next iteration. Indeed, considering iteration k, the position of particle i is adjusted according to the following *update position equation*:

$$x_i^k = x_i^{k-1} + v_i^k. \tag{2.18}$$

Each particle i of the swarm communicates with a social environment or neighbourhood, $N(i) \subseteq S$, representing the group of particles with which it communicates, and which could change dynamically. In nature, a bird adjusts its position in order to find a better position, according to its own experience and the experience of its companions. In the same manner, considering iteration k of the PSO algorithm, each particle i updates its velocity reflecting the attractiveness of its best position so far (b_i), and the best position (g_i) of its social neighbourhood $N(i)$, according to the following equation:

$$v_i^k = c_1 \xi v_i^{k-1} + c_2 \xi (b_i - x_i^{k-1}) + c_3 \xi (g_i - x_i^{k-1}). \tag{2.19}$$

In particular, the parameter $c_1 \in [0, 1]$ represents the effect of inertia, whose mission is to control the magnitude of the velocity avoiding an indefinite growth. The parameters $c_2 \in [0, 1]$ and $c_3 \in [0, 1]$ are positive constant weights representing the degrees of confidence of particle i in the different positions (b_i and g_i) that influence its dynamics (either $c_2 = c_3$ or $c_1 + c_2 + c_3 = 1$ in many versions of PSO). The term ξ refers to a random number with uniform distribution $[0, 1]$ that is independently generated at each iteration. The current position x_i, the best

position so far b_i, and the best position of the social neighbourhood g_i, behave, with different weights c_1, c_2, and c_3, like centres of attraction for each particle i. Thus, the update position equation becomes (Kennedy and Eberhart, 1995):

$$x_i^k = x_i^{k-1} + c_1\xi v_i^{k-1} + c_2\xi(b_i - x_i^{k-1}) + c_3\xi(g_i - x_i^{k-1}). \tag{2.20}$$

Further details of the implementation of Particle Swarm Optimization are specified in Algorithm 2.16. The initial position x_i and velocity v_i for each particle i in the swarm SW are usually obtained at random. The position of a particle

Algorithm 2.16: Particle Swarm Optimization

Input: A fitness function $f(\cdot)$, the search space S, the positive constant weights $c_1 \in [0,1]$, $c_2 \in [0,1]$, $c_3 \in [0,1]$ that influence the dynamics of the swarm;
Output: A solution $g^* \in S$;
Initialisation:
- Let $s \in S$ be a generic solution;
- Let $g^* \in S$ be the best position to date;
- Define a neighbourhood structure $N(\cdot)$;
- Set the size n_{sw} of the swarm $SW = (1, 2 \ldots, n_{sw})$;

begin
Generate the initial swarm SW with positions at random:
$X = [x_1, x_2, \ldots, x_{n_{sw}}] \leftarrow$*Generate-Swarm-At-Random(S)*;
Initialize the velocity of each particle at random: $\forall i \in SW$, $v_i \leftarrow random()$;
Evaluate the fitness function of each individual in the swarm:
$(f(x_1), f(x_2), \ldots, f(x_{n_{sw}})) \leftarrow$*Evaluate*$(x_1, x_2, \ldots, x_{n_{sw}})$;
Initialize the best position so far for each particle: $\forall i \in SW$, $b_i \leftarrow x_i$;
Extract the best position of the social neighbourhood of each particle: $\forall i \in SW$,
$g_i \leftarrow$*Extract-the-Best(SW, N(i), f(·))*;
Extract the best position to date among all the particles: $g^* \leftarrow$*Extract-the-Best(SW, X, f(·))*;
repeat
Select at random a number between 0 and 1: $\xi \leftarrow random[0,1)$;
for $i \leftarrow 1$ *to* n_{sw} **do**
Update the velocity of particle i: $v_i^k = c_1\xi v_i^{k-1} + c_2\xi(b_i - x_i^{k-1}) + c_3\xi(g_i - x_i^{k-1})$;
Update the position of particle i: $x_i^k = x_i^{k-1} + v_i^k$;
if $f(x_i) < f(b_i)$ **then**
Update the best position so far for the given particle i: $b_i \leftarrow x_i$;
end
end
Extract the best position of the social neighbourhood of each particle: $\forall i \in SW$,
$g_i \leftarrow$*Extract-the-Best(SW, N(i), f(·))*;
Extract the best position among all the particles: $s \leftarrow$ *Extract-the-Best(SW, X, f(·))*;
if $f(s) < f(g^*)$ **then**
Update the best position to date among all the particles: $g^* \leftarrow s$;
end
until *termination conditions* ;
$\Rightarrow$ *Return*(g^*).
end

in the swarm is encoded as a feasible solution to the specific problem. At each iteration, each particle updates its best position so far (b_i), and the best position of its social neighbourhood (g_i). In order to update b_i, different neighbourhood structures $N(\cdot)$ can be selected (Kennedy and Eberhart, 2001). In the original PSO implementation by Kennedy and Eberhart (1995), the particles that constitute the neighbourhood of another particle are chosen at random. The neighbourhoods are newly generated at each iteration, when the best global position g^* does not improve. Another possibility is to associate a given probability to each particle to constitute the neighbourhood of another particle. In addition to the random selection of neighbourhoods, two other common topologies are the ring and the star neighbourhood structures (Kennedy and Eberhart, 2001). In the ring neighbourhood structure, each particle interacts just with the previous and the following particles (cyclic arrangement of the particles), i.e. $N(i) = \{i-1, i, i+1\}$, $\forall i \in SW$. In the star neighbourhood structure, each particle interacts with all the particles of the swarm, i.e. $N(i) = SW = \{1, 2 \ldots, i, \ldots, n_{sw}\}$, $\forall i \in SW$.

In the successive step, the random number ξ is selected ($\xi \leftarrow random[0, 1)$), and the position of each particle iteratively updated according to Equation 2.20. The particle with the best position in the new swarm is extracted ($s \leftarrow$ *Extract-the-Best(SW, X, $f(\cdot)$)*), and it is compared to the particle with best position to date (g^*). If $f(s) < f(g^*)$ then the best position to date is updated ($g^* \leftarrow s$). The attractors b_i and g_i are updated again, and the same procedure is repeated iteratively. The entire algorithm continues until the user termination conditions are satisfied, producing the best position to date as output (g^*).

Since, in the words of the inventors of PSO, it is not possible to "throw to fly" particles in a discrete space (Kennedy and Eberhart, 1995), several *Discrete Particle Swarm Optimization* (DPSO) methods have been proposed for combinatorial optimization problems. For example, in the DPSO proposed by Kennedy and Eberhart (1997) for problems with binary variables, the position of each particle is a vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{ij}, \ldots, x_{id})$ of the d-dimensional binary solution space, $x_i \in \{0, 1\}^d$, but the velocity is still a vector v_i of the d-dimensional continuous space, $v_i \in \Re^d$. A DPSO whose particles at each iteration are affected alternatively by its best position and the best position among its neighbours was proposed by Al-kazemi and Mohan (2002). Pampara et al. (2005) solved binary

problems by combining continuous PSO and Angle Modulation with only four parameters. Furthermore, several PSO variants applied to problems where the solutions are permutations were considered in (Onwubolu and Clerc, 2004; Pang et al., 2004; Secrest, 2001). The multi-valued PSO proposed by Pugh and Martinoli (2006) deals with variables with multiple discrete values. The position of each particle is a mono-dimensional array in the case of a continuous PSO, a 2-dimensional array in the case of a DPSO, and a 3-dimensional array for a multi-valued PSO. Indeed, the position of particle i in the multi-valued PSO is expressed by the term x_{ijk}, representing the probability that the i-th particle, in the j-th iteration, takes the k-th value. Another DPSO was proposed in (Correa et al., 2006) for feature selection problems, which are problems whose solutions are sets of items. In this DPSO, the velocity vectors consist of positive numbers representing the relative likelihood of the corresponding binary component of the positions of the particles. The position of each particle is updated by randomly generating changes according to these likelihoods, and then continuing in similar way to the standard PSO. A new DPSO proposed in (Moreno-Pérez et al., 2007) and (Martínez-García and Moreno-Pérez, 2008) does not consider any velocity since, from the lack of continuity of the movement in a discrete space, the notion of velocity loses sense; however they kept the attraction of the best positions. They interpret the weights of the updating equation as probabilities that, at each iteration, each particle has a random behaviour, or acts in a way guided by the effect of an attraction. The moves in a discrete or combinatorial space are jumps from one solution to another. The attraction causes the given particle to move towards this attractor if it results in an improved solution. An inspiration from the nature for this process is found in frogs, which jump from a lily pad to a pad in a pool. Thus, this new discrete PSO is called also Jumping Particle Swarm Optimization.

Particle Swarm Optimization and its variants are an extremely interesting areas of research. Particle Swarm Optimization is a nature-inspired algorithm which uses also the concept of fitness, as do all evolutionary computation paradigms. Unique to the concept of Particle Swarm Optimization is flying potential solutions through hyperspace, accelerating toward "better" solutions. Much of the success of PSO seems to lie in the agents' tendency to hurtle past their target.

The stochastic factors allow thorough search of spaces between regions that have been found to be relatively good, and the momentum effect caused by modifying the extant velocities rather than replacing them results in overshooting, or exploration of unknown regions of the problem domain. Another reason that makes PSO attractive is that there are few parameters to adjust. One version, with slight variations, works well in a wide variety of applications. Particle Swarm Optimization has been used for approaches that can be used across a wide range of applications, as well as for specific applications focused on a specific requirement. In recent years, it has been successfully applied in many research and application areas. In several cases, it is demonstrated that PSO gets better results in a faster and cheaper way, compared with other optimization methods (Kennedy and Eberhart, 2001).

2.4 Hybrid metaheuristics

A current trend in metaheuristics is the integration of single-solution methods with population-based methods. In this section, a brief description of the most important hybrid approaches is given. Generally, it is possible to divide hybrid methods into the three following classes (Blum and Roli, 2003).

1) The first type of hybrid metaheuristics, called *components exchange among metaheuristics*, consists of methods that include components from different metaheuristics, usually from a single-point method and a population-based one. The strength of population-based methods is the concept of recombining solutions, explicitly in most of evolutionary algorithms through recombination operators, implicitly in Ant Colony Optimization and Estimation of Distribution Algorithms for the nature of their mechanisms. The recombination follows a criterion of mixing high-quality solutions in the hope of finding better solutions, on the followed direction. The recombination in population-based methods allows "big" guided steps in the search space, usually larger than the ones performed by single-solution methods. Some single-solution methods, such as Iterated Local Search and Variable Neighbourhood Search, also perform "big" steps, but resulting from random mechanisms called "kick moves" or "perturbations", indicating the absence of guidance. Instead, the strength of single-solution methods is generally based

on embedded local search mechanisms, to strictly explore promising regions of the search space. In this way, the danger of being close to good solutions but "missing" them is not as high as in population-based methods. Many successful applications of evolutionary algorithms and nature-inspired algorithms also make use of local search procedures. Summarizing, population-based methods are better at identifying promising areas in the search space, whereas trajectory methods are superior in exploring specific zones of the domain. Thus, hybrid metaheuristics, combining the advantages of population-based methods with the power of single-solution methods, are often very successful (Blum and Roli, 2003).

2) The second form of hybridization, called *cooperative search*, consists of a search carried out by different algorithms, approximate or complete ones, exchanging information about states, models, entire sub-problems, solutions or other search space characteristics. Cooperative algorithms can be either different search techniques, or instances of the same algorithm with different settings of the model or the parameters, or algorithms in parallel execution with a variable level of communication. Cooperative Search also receives much attention as a result of the rapid growth of parallel implementations of metaheuristics.

3) The last class of hybridization is called *integration of metaheuristics and systematic (or complete) search methods*. This class includes very effective hybrids for real-world applications. There are three main approaches for the integration of metaheuristics, especially single-solution methods, and systematic techniques, such as constraint programming and tree search methods. The first approach consists of their sequential application and/or their interleaved execution. For example, the metaheuristic may produce some solutions which are then improved by systematic search (or vice-versa, the systematic algorithm may generate partial solutions which are completed by the metaheuristic). This procedure can also be viewed as a loose form of cooperative search. The second approach uses a complete method to efficiently explore a defined neighbourhood structure, instead of randomly sampling it or simply enumerating all the neighbours. This approach is particularly effective when the neighbourhood to explore is very large, because it combines the advantages of a fast exploration, by using a metaheuristic, with an efficient neighbourhood exploitation, performed by a systematic method. The third possibility consists of introducing concepts or strategies from classes of

algorithms into others. A typical case is a probabilistic backtracking instead of a deterministic one into a search-tree algorithm. This is carried out through the introduction of the concepts of tabu list from Tabu Search, and aspiration criteria from Simulated Annealing, into a search-tree algorithm, in order to manage the list of open nodes to explore.

Successful examples of hybridization are the introduction of the concept of memory in Simulated Annealing and Variable Neighbourhood Search, which, in their standard form, are memory-less methods, through the integration with other usage-memory metaheuristics, such as Tabu Search (Aarts et al., 1997). Tabu Search, in general, is a rich source of ideas, which have been and are currently adopted by other metaheuristics. Another way to produce hybrid metaheuristics with Variable Neighbourhood Search is through the integration with exact algorithms, in order to increase the intensification capability of its local search phase (Hansen and Mladenović, 2003). GRASP may be successfully integrated into other search techniques, due to its simplicity and, generally, high speed. However, a basic GRASP does not use the history of the search process. The only memory it requires is for storing the problem instance and for keeping the best solution to date. This is one of the reasons why GRASP is often outperformed by other metaheuristics. Thus, another promising research direction is trying to introduce concepts of memory in GRASP through other usage-memory methods (Blum and Roli, 2003). An example of recent success of hybridization in evolutionary algorithms includes the integration of Path Relinking as a component for Tabu Search and GRASP (Glover and Kochenberger, 2003). Regarding the hybridization of nature-inspired algorithms, researchers have been recently dealing with finding similarities between Ant Colony Optimization and probabilistic learning algorithms such as Estimation of Distribution Algorithms. Furthermore, connections of Ant Colony Optimization to Stochastic gradient descent algorithms represent a research area of growing interest. In conclusion, there is a need for the hybridization of metaheuristics to be examined in detail in order to be able to produce hybrid metaheuristics that perform better than their "pure" parents in specific circumstances.

I have been impressed with the urgency of doing. Knowing is not enough; we must apply. Being willing is not enough; we must do.

LEONARDO DA VINCI

Chapter 3

Minimum labelling spanning tree problem

In this chapter, heuristics for the minimum labelling spanning tree (MLST) problem are studied. The problem is to find a spanning tree using edges that are as similar as possible. Given an undirected labelled connected graph, the minimum labelling spanning tree problem seeks a spanning tree whose edges have the smallest number of distinct labels. This problem has been shown to be NP-hard (Chang and Leu, 1997). A Greedy Randomized Adaptive Search Procedure (GRASP), a Variable Neighbourhood Search (VNS), and a hybrid local search method (HYBRID) are proposed in this chapter. HYBRID is obtained by combining Variable Neighbourhood Search with another classic metaheuristic: Simulated Annealing (SA). The proposed methods are compared to other algorithms recommended in the literature: Modified Genetic Algorithm (MGA) and Pilot Method (PILOT). Nonparametric statistical tests show that the proposed heuristics outperform the other algorithms tested. Furthermore, a comparison with the results provided by an exact approach shows that these heuristics quickly obtain optimal or near-optimal solutions.

3.1 Description of the problem

The *minimum labelling spanning tree* (MLST) problem is an NP-hard problem in which, given a graph with labelled (or coloured) edges, one seeks a spanning tree

with the least number of labels (or colours). Such a model can represent many real-world problems in telecommunications networks, power networks, and multimodal transportation networks. For example, in telecommunications networks, there are many different types of communications media, such as optical fibre, coaxial cable, microwave, and telephone line (Tanenbaum, 1989). A communications node may communicate with different nodes by choosing different types of communications media. Given a set of communications network nodes, the problem is to find a spanning tree (a connected communications network) that uses as few communications types as possible. This spanning tree will reduce the construction cost and the complexity of the network.

The MLST problem can be formulated as a network or graph problem. Consider a labelled connected undirected graph $G = (V, E, L)$, where V is the set of n nodes, E is the set of m edges, and L is the set of ℓ labels. In the telecommunications example (Tanenbaum, 1989), the vertices represent communications nodes, the edges communications links, and the labels communications types. Each edge in E has a label in a finite set L that identifies the communications type. The objective is to find a spanning tree that uses the smallest number of different types of edges. Define L_T to be the set of different labels of the edges in a spanning tree T. The labelling can be represented by a function $f_L : E \rightarrow L$ for all edges $e \in E$ or by a partition P_L of the edge set; the sets of the partitions are those consisting of the edges with the same label.

Another example is given by multimodal transportation networks (Van-Nes, 2002). In such problems, it is desirable to provide a complete service using the minimum number of companies. The multimodal transportation network is represented by a graph where each edge is assigned a label, denoting a different company managing that edge. The aim is to find a spanning tree of the graph using the minimum number of labels. The interpretation is that all nodes representing termini are connected without cycles, using the minimum number of companies.

The minimum labelling spanning tree problem is formally defined as follows:

> *MLST problem:* Given a labelled graph $G = (V, E, L)$, where V is the set of n nodes, E is the set of m edges, and L is the set of ℓ labels, find a spanning tree T of G such that $|L_T|$ is minimized, where L_T is the set of labels used in T.

Although a solution to the MLST problem is a spanning tree, it is helpful to first consider connected subgraphs. A feasible solution is defined as a set of labels $C \subseteq L$, such that all the edges with labels in C represent a connected subgraph of G and span all the nodes in G. If C is a feasible solution, then any spanning tree of C has at most $|C|$ labels. Moreover, if C is an optimal solution, then any spanning tree of C is a minimum labelling spanning tree. Thus, in order to solve the MLST problem, it is preferable to seek a feasible solution with the least number of labels (Xiong et al., 2005b).

The upper left graph of Figure 3.1 is an example of an input graph with the optimal solution shown on the upper right. The lower part of Figure 3.1 shows examples of feasible solutions.

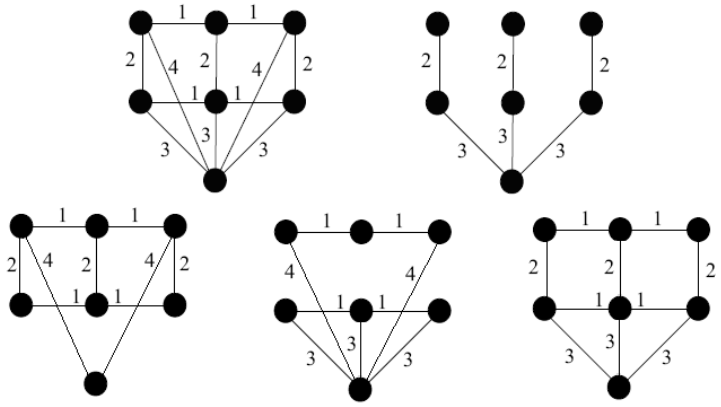


Figure 3.1: The top two graphs show a sample graph and its MLST solution. The bottom three graphs show some feasible solutions.

The rest of the chapter is organised as follows. In the next section, the literature of the problem is reviewed. In Section 3.3 the details of the heuristics considered in this chapter are presented: ones recommended in the literature (the *Modified Genetic Algorithm* by Xiong et al. (2006), and the *Pilot Method*

by Cerulli et al. (2005)), and some new approaches to the MLST problem proposed in this chapter (a *Greedy Randomized Adaptive Search Procedure*, a basic *Variable Neighbourhood Search*, and a hybrid local search method obtained by combining *Variable Neighbourhood Search* and *Simulated Annealing* metaheuristics). Section 3.4 includes the experimental analysis of the comparison of these metaheuristics, and the chapter ends with some conclusions in Section 3.5. The basic concepts of metaheuristics and combinatorial optimization were presented in Chapter 2, but, for further information, the reader is referred to (Voß et al., 1999; Glover and Kochenberger, 2003; Gendreau and Potvin, 2005).

3.2 Literature review

In communications network design, it is often desirable to obtain a tree that is "most uniform" in some specified sense. Motivated by this observation, Chang and Leu (1997) introduced the minimum labelling spanning tree problem. They also proved that it is an NP-hard problem and provided a polynomial-time heuristic, the *maximum vertex covering algorithm* (MVCA), to find (possibly suboptimal) solutions. This heuristic begins with an empty graph. It then adds the label whose edges cover as many isolated vertices as possible until there are no isolated vertices. The heuristic solution is an arbitrary spanning tree of the resulting graph. However, with this version of MVCA, it is possible that, although all the nodes of the graph are visited, it does not yield a connected graph and thus fails.

Krumke and Wirth (1998) proposed a corrected version of MVCA, depicted in Algorithm 3.1. This begins with an empty graph and successively adds at random one label from those labels that result in the least number of connected components. The procedure continues until only one connected component is left, i.e. when only a connected subgraph is obtained.

Figure 3.2 shows how this version of MVCA works on the graph of Figure 3.1. In the initial step, label 1 is added because it gives the least number of connected components (3 components). In the second step, all the three remaining labels (2, 3 and 4) produce the same number of components (2). In this case, the algorithm selects at random and, for example, adds label 3. At this time, all the

Algorithm 3.1: Revised MVCA (Krumke and Wirth, 1998)

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialisation:
- Let $C \leftarrow 0$ be the initially empty set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $Comp(C)$ be the number of connected components of $H = (V, E(C))$;

begin
  while $Comp(C) > 1$ **do**
    Select the unused label $c \in (L - C)$ that minimizes $Comp(C \cup \{c\})$;
    Add label c to the set of used labels: $C \leftarrow C \cup \{c\}$;
    Update $H = (V, E(C))$ and $Comp(C)$;
  end
  $\Rightarrow$ Take any arbitrary spanning tree T of $H = (V, E(C))$.
end

nodes of the graph are visited, but the subgraph is still disconnected. The old version of Chang and Leu (1997) would stop here, resulting in an error. However, the MVCA version of Krumke and Wirth (1998) finally adds label 2 to get only one connected component (equivalently, label 4 could have been added instead of label 2). Summarizing, the final solution is $\{1, 2, 3\}$, which is worse than the optimal solution $\{2, 3\}$ of Figure 3.1.

1st ITERATION				
LABEL	1	2	3	4
NUMBER OF COMPONENTS	3	4	4	5
2nd ITERATION				
LABEL	1	2	3	4
NUMBER OF COMPONENTS	-	2	2	2
3rd ITERATION				
LABEL	1	2	3	4
NUMBER OF COMPONENTS	-	1	-	1

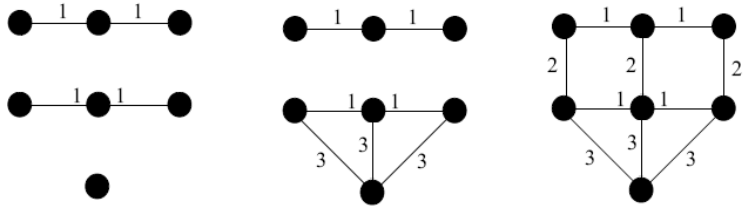


Figure 3.2: Example illustrating the steps of the revised MVCA.

Krumke and Wirth (1998) proved that MVCA can yield a solution no greater than $(1+2\log n)$ times optimal, where n is the total number of nodes. Later, Wan et al. (2002) obtained a better bound for the greedy algorithm introduced by Krumke and Wirth (1998). The algorithm was shown to be a $(1+\log(n-1))$-approximation for any graph with n nodes ($n > 1$).

Brüggemann et al. (2003) used a different approach; they applied local search techniques based on the concept of j-switch neighbourhoods to a restricted version of the MLST problem. In addition, they proved a number of complexity results and showed that if each label appears at most twice in the input graph, the MLST problem is solvable in polynomial-time.

Xiong et al. (2005a) derived tighter bounds than those proposed by Wan et al. (2002). For any graph with label frequency bounded by b, they showed that the worst-case bound of MVCA is the b-th-harmonic number

$$H_b = \sum_{i=1}^{b} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{b}; \tag{3.1}$$

Later, they constructed a worst-case family of graphs such that the MVCA solution is exactly H_b times the optimal solution. Since $b \leq (n-1)$ (since otherwise the subgraph induced by the labels of maximum frequency contains a cycle and one can safely remove edges from the cycle) and $H_b < (1+\log(n-1))$, the tight bound H_b obtained is, therefore, an improvement on the previously known performance bound of $(1+\log(n-1))$ given by Wan et al. (2002).

The usual rule of Krumke and Wirth (1998) to select the label that minimizes the total number of connected components at each step, results in fast and high-quality solutions. The problem with this classic approach occurs when more than one label with same resulting minimum number of connected components is detected, in a specific step. Since, frequently, there are many labels reaching this minimum value, the results mainly depend on the rule chosen to select a candidate from this set of ties. If the initial label encountered from this set is chosen, the results are affected by the sorting of the labels. Therefore, different executions of the algorithm may result in different solutions, with a slightly different number of labels.

Other heuristic approaches to the MLST problem are proposed in the literature. For example, Xiong et al. (2005b) presented a *Genetic Algorithm* (GA) to solve the MLST problem, outperforming MVCA in most cases.

Subsequently, Cerulli et al. (2005) applied the *Pilot Method*, a greedy heuristic developed by Duin and Voß (1999) and subsequently extended in (Voß et al., 2004), to the MLST problem. Considering different sets of instances of the MLST problem, Cerulli et al. (2005) compared this method with other metaheuristics (Reactive Tabu Search, Simulated Annealing, and an ad-hoc implementation of Variable Neighbourhood Search). Their Pilot Method obtained the best results in most of the cases. It generates high-quality solutions to the MLST problem, but running times are quite large (especially if the number of labels is high).

Xiong et al. (2006) implemented modified versions of MVCA focusing on the initial label added. For example, after the labels have been sorted according to their frequencies, from highest to lowest, the modified version tries only the most promising 10% of the labels at the initial step. Afterwards, it runs MVCA to determine the remaining labels and then it selects the best of the $|L|/10$ resulting solutions (where L is the set of possible labels for all edges). Compared to the Pilot Method of Cerulli et al. (2005), this version can potentially reduce the computational running time by about 90%. However, since a higher frequency label may not always be the best place to start, it may not perform as well as the Pilot Method. Another modified version by Xiong et al. (2006) is similar to the previous one, except that it tries the most promising 30% of the labels at the initial step. Then it runs MVCA to determine the remaining labels. Moreover, Xiong et al. (2006) proposed another way to modify MVCA. They consider at each step the three most promising labels, and assign a different probability of selection that is proportional to their frequencies. Then, they randomly select one of these candidates, and add it to the incomplete solution. In addition, Xiong et al. (2006) presented a *Modified Genetic Algorithm* (MGA) that was shown to have the best performance for the MLST problem in terms of solution quality and running time.

3.3 Exploited metaheuristics

In this section, the details of the heuristics considered for the MLST problem are specified. First, those that are reported in the literature to be the best performing are considered, followed by some new approaches.

Xiong et al. (2005b) presented two slightly different Genetic Algorithms to solve the MLST problem. They both were shown to be simple, fast, and effective. In most cases, they also outperformed MVCA, the most popular MLST heuristic in the literature at that time. Later, a *Modified Genetic Algorithm* (MGA) was proposed in (Xiong et al., 2006). It outperformed the first two Genetic Algorithms with respect to solution quality and running time. MGA is the first metaheuristic that is considered.

Cerulli et al. (2005) applied the *Pilot Method* (PILOT) to the MLST problem. Comparing it with some other metaheuristic implementations (Reactive Tabu Search, Simulated Annealing, and an ad-hoc implementation of Variable Neighbourhood Search), it was the best performing in most of the test problems. The Pilot Method of Cerulli et al. (2005) is the second metaheuristic considered in this chapter. Then, some new approaches to the problem are presented. A new heuristic for the problem based on *Greedy Randomized Adaptive Search Procedure* (GRASP) is proposed. Basically, GRASP is a metaheuristic combining the power of greedy local search with randomisation. A survey on GRASP is presented in Section 2.2.3, but the reader is also referred to (Feo and Resende, 1995; Resende and Ribeiro, 2003).

The remaining algorithms in this chapter are a basic *Variable Neighbourhood Search* (VNS), and a hybrid local search method (HYBRID) obtained by combining *Variable Neighbourhood Search* and *Simulated Annealing* (SA) metaheuristics. For more details on the general implementations of Variable Neighbourhood Search and Simulated Annealing see, respectively, Section 2.2.5 and Section 2.2.1.

3.3.1 Modified Genetic Algorithm

Genetic Algorithms are based on the principle of evolution, operations such as crossover and mutation, and the concept of fitness (Holland, 1992). For a survey on the basic concepts of Genetic Algorithms, the reader is referred to Section 2.3.1.

In the MLST problem, fitness is defined as the number of distinct labels in the candidate solution. After a number of generations, the algorithm converges and the best individual, hopefully, represents a near-optimal solution. The details of the Modified Genetic Algorithm for the MLST problem are specified in Algorithm 3.2.

Algorithm 3.2: The MGA for the MLST problem (Xiong et al., 2006)

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialisation:
- Let $C \leftarrow 0$ be the initially empty set of used labels for each iteration;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Set the size n_P of the population;

```
begin
    (s[0], s[1], ..., s[n_P − 1]) ←Initialize-Population(G, n_P);
    repeat
        for i ← 1 to n_P/2 do
            for j ← 1 to n_P/2 do
                t[1] ← s[j];
                t[2] ← s[mod((j + i), n_P)];
                t_crossovered ←Crossover(t[1], t[2]);
                t_mutated ←Mutation(t_crossovered);
                if t_mutated < t[1] then
                    t[1] ← t_mutated;
                end
            end
        end
    until termination conditions ;
    C ←Extract-the-Best(s[0], s[1], ..., s[n_P − 1]);
    Update H = (V, E(C));
    ⇒ Take any arbitrary spanning tree T of H = (V, E(C)).
end
```

An individual (or a chromosome) in a population is a feasible solution. Each label in a feasible solution can be viewed as a gene. The initial population is generated by adding labels randomly to empty sets, until feasible solutions emerge. Crossover and mutation operations are then applied in order to build one generation from the previous one. Crossover and mutation probability values are set to 100%, which is at odds with an usual GA implementation. This means that all the individuals within the current generation are selected as offsprings for the crossover operation, and also as offsprings for the mutation operation. The overall number of generations is chosen to be half of the initial population value.

Therefore, in the Genetic Algorithm of Xiong et al. (2006) the only parameter to tune is the population size.

The crossover operation builds one offspring from two parents which are feasible solutions. Given the parents $P_1 \subset L$ and $P_2 \subset L$, it begins by forming their union $P \leftarrow P_1 \cup P_2$. Then it adds labels from the subgraph P to the initially empty offspring until a feasible solution is obtained, by applying the revised MVCA of Krumke and Wirth (1998) to the subgraph with labels in P, node set V, and the edge set associated with P. On the other hand, the mutation operation consists of adding a new label at random, and next trying to remove the labels (i.e., the associated edges), from the least frequently occurring label to the most frequently occurring one, whilst retaining feasibility.

3.3.2 Pilot Method

The Pilot Method is a metaheuristic proposed by Duin and Voß (1999) and Voß et al. (2004). It uses a *basic heuristic* as a building block or *application process*, and then it tentatively performs iterations of the application process with respect to a so-called *master solution*. The iterations of the basic heuristic are performed until all the possible local choices (or moves) with respect to the master solution are evaluated. At the end of all the iterations, the new master solution is obtained by extending the current master solution with the move that corresponds to the best result produced.

Considering a master solution M, for each element $i \notin M$, the Pilot Method extends tentatively a copy of M to a (fully grown) solution including i, built through the application of the basic heuristic. Let $f(i)$ denote the objective function value of the solution obtained by including each element $i \notin M$, and let i^* be the most promising of such elements, i.e. $f(i^*) \leq f(i), \forall i \notin M$. The element i^*, representing the best local move with respect to M, is included in the master solution by changing it in a minimal fashion, leading to a new master solution $M \leftarrow M \cup \{i^*\}$. On the basis of this new master solution M, new iterations of the Pilot Method are started $\forall i \notin M$, providing a new solution element i^*, and so on. This *look-ahead* mechanism is repeated for all the successive stages of the Pilot Method, until no further moves need to be added to the master solution,

or until some termination conditions imposed by the user are satisfied. The last master solution corresponds to the best solution to date and forms the output of the procedure.

The details of the Pilot Method proposed by Cerulli et al. (2005) for the MLST problem are specified in Algorithm 3.3. It starts from the null solution

Algorithm 3.3: The Pilot Method for the MLST problem (Cerulli et al., 2005)

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialisation:
- Let $M \leftarrow 0$ be the initially empty master solution,
- Let $H = (V, E(M))$ be the subgraph of G restricted to V and edges with labels in M, where $E(M) = \{e \in E : L(e) \in M\}$;
- Let $Comp(M)$ be the number of connected components of $H = (V, E(M))$;
- Let $M^* \leftarrow L$ be a set of labels;
- Let $H^* = (V, E(M^*))$ be the subgraph of G restricted to V and edges with labels in M^*, where $E(M^*) = \{e \in E : L(e) \in M^*\}$;
- Let i^* be the best candidate move;

```
begin
    while (not termination conditions) OR (Comp(M) > 1) do
        foreach i ∈ (L − M) do
            Add label i to the master solution: M ← M ∪ {i};
            Update H = (V, E(M)) and Comp(M);
            while Comp(M) > 1 do
                Let S be the set of unused labels which minimize the number of connected
                components, i.e. S = {e ∈ (L − M) : min Comp(M ∪ {e})};
                Select at random a label u ∈ S;
                Add label u to the solution: M ← M ∪ {u};
                Update H = (V, E(M)) and Comp(M);
            end
            Local search(M);
            if |M| < |M*| then
                Update the best candidate move i* ← i;
                Keep the solution produced by the best move: M* ← M;
            end
            Delete label i from the master solution: M ← M − {i};
            Update H = (V, E(M)) and Comp(M);
        end
        Update the master solution with the best move: M ← M ∪ {i*};
    end
    while Comp(M) > 1 do
        Let S be the set of unused labels which minimize the number of connected components, i.e.
        S = {e ∈ (L − M) : min Comp(M ∪ {e})};
        Select at random a label u ∈ S;
        Add label u to the solution: M ← M ∪ {u};
        Update H = (V, E(M)) and Comp(M);
    end
    ⇒ Take any arbitrary spanning tree T of H = (V, E(M)).
end
```

(an empty set of labels) as master solution, uses the revised MVCA of Krumke and Wirth (1998) as the application process, and evaluates the quality of a feasible solution by choosing the number of labels included in the solution as the objective function.

The method computes all the possible local choices from the master solution, performing a series of iterations of the application process to the master solution. This means that, at each step, it alternatively tries to add to the master solution each label not yet included, and then applies MVCA in a greedy fashion from then on (i.e. by adding at each successive step the label that minimizes the total number of connected components), stopping when the resulting subgraph is connected (note that, when the MVCA heuristic is applied to complete a partial solution, in case of ties in the minimum number of connected components, a label is selected at random within the set of labels producing the minimum number of components). The Pilot Method successively chooses the best local move, that is the label that, if included to the current master solution, produces the feasible solution with the minimum objective function value (number of labels). In case of ties, it selects one label at random within the set of labels with the minimum objective function value. This label is then included in the master solution, leading to a new master solution. If the new master solution is still infeasible, the Pilot Method proceeds with the same strategy in this new step, by alternatively adding to the master solution each label not yet included, and then applying the MVCA heuristic to produce feasible solutions for each of these candidate labels. Again, the best move is selected to be added to the master solution, producing a new master solution, and so on. The procedure continues with the same mechanism until a feasible master solution is produced, that is one

Algorithm 3.4: Procedure *Local search(·)*

Procedure Local search(*M*):
for $j \leftarrow 1$ *to* $|M|$ **do**
  Delete label j from the set M, i.e. $M \leftarrow M - \{j\}$;
  Update $H = (V, E(M))$ and $Comp(M)$;
  if $Comp(M) > 1$ **then**
    Add label j to the set M, i.e. $M \leftarrow M \cup \{j\}$;
    Update $H = (V, E(M))$ and $Comp(M)$;
  end
end

representing a connected subgraph, or until the user termination conditions are satisfied. The last master solution represents the output of the method. A local search mechanism is further included at the end of the computation in order to try to greedily drop labels whilst retaining feasibility (see Algorithm 3.4).

Since up to ℓ master solutions can be considered by this procedure, and up to ℓ local choices can be evaluated for each master solution, the overall computational running time of the Pilot Method is $O(\ell^2)$ times the computational time of the application process (i.e. the MVCA heuristic), leading to an overall complexity $O(\ell^3 n)$.

3.3.3 Greedy Randomized Adaptive Search Procedure

The difficulty with the classical version of MVCA is when it finds more than one label with the same number of connected components. A question arises on the label to be chosen. To find the best MVCA solution, alternatively each of these labels should be added, continuing the same strategy in successive steps. In this way, every possible local choice is computed, because all the solutions that MVCA can produce are visited. But the execution time increases dramatically, especially for low-density graphs with a high number of nodes and labels.

The Pilot Method is able to achieve a greater diversification of the search process than the MVCA approach. This is because it alternatively tries every label at each step, and not only labels within the set that minimize the total number of connected components, as is the case with MVCA. Thus, the Pilot Method is able to reach some solutions that MVCA would never consider. Instead, to increase the intensification of the basic MVCA, it is possible to perform multiple repetitions of the MVCA heuristic. In this way, more solutions that MVCA would have produced are visited.

In this section a Greedy Randomized Adaptive Search Procedure (GRASP) for the MLST problem is proposed, trying to unify multiple repetitions of the MVCA heuristic with the Pilot Method strategy in order to obtain an optimal balance between intensification and diversification capabilities.

GRASP is a recently exploited method combining the power of greedy heuristics, randomisation, and local search. It is a multi-start two-phase metaheuristic

for combinatorial optimization proposed by Feo and Resende (1995), basically consisting of a construction phase and a local search improvement phase (for a survey on GRASP see Section 2.2.3).

The *solution construction* mechanism builds an initial solution using a greedy randomized procedure, whose randomness allows solutions in different areas of the solution space to be obtained. Each solution is randomly produced step-by-step by uniformly adding one new element from a candidate list (RCL_α: restricted candidate list of length α) to the current solution. Subsequently, a *local search* phase is applied (such as Simulated Annealing, Tabu Search) to try to improve the current best solution. This two-phase process is iterative, continuing until the user termination condition such as the maximum allowed CPU time, the maximum number of iterations, or the maximum number of iterations between two successive improvements, is reached. Several new components have extended the scheme of GRASP (reactive GRASP, parameter variations, bias functions, memory and learning, improved local search, path relinking, hybrids,...). These components are presented and discussed in (Resende and Ribeiro, 2003).

The proposed GRASP implementation for the MLST problem is specified in

Algorithm 3.5: Greedy Randomized Adaptive Search Procedure for the MLST problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialisation:
- Let $C \leftarrow 0$ be the initially empty set of used labels for each iteration;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $C' \leftarrow L$ be the global set of used labels;
- Let $H' = (V, E(C'))$ be the subgraph of G restricted to V and edges with labels in C', where $E(C') = \{e \in E : L(e) \in C'\}$;
- Let $Comp(C)$ be the number of connected components of $H = (V, E(C))$;

```
begin
    repeat
        Set C ← 0 and update H = (V, E(C));
        Construction phase(C);
        Local search(C);
        if |C| < |C'| then
            Move C' ← C;
            Update H' = (V, E(C'));
        end
    until termination conditions ;
    ⇒ Take any arbitrary spanning tree T of H' = (V, E(C')).
end
```

Algorithm 3.5. The greedy criterion of the construction phase of GRASP (*Construction phase()* procedure) is based on the number of connected components produced by the labels, and a *value-based* restricted candidate list is used (Resende and Ribeiro, 2003). This involves placing in the list only the candidate labels having a greedy value (number of connected components) not greater than a user-defined threshold, α, whose values can vary dynamically during the search process. The value of the threshold α and its tuning during the iterations of the algorithm need to be chosen in an appropriate way. Indeed, a small value of α results in few labels in the restricted candidate list, giving a large intensification capability and a small diversification capability. This means that the resulting algorithm is very fast, but it can easily become trapped at a local optimum. Conversely, a large value of α produces an algorithm with a large diversification capability, but a short intensification capability, because many candidate labels are included in the restricted candidate list. In this implementation for the MLST problem (Algorithm 3.6), an adequate trade-off between intensification and diversification capabilities has been found by considering the following scheme. In order to fill the restricted candidate list, the threshold is set equal to the minimum number of connected components produced by the candidate labels. This means that only the labels producing the least number of connected components constitute the restricted candidate list. Furthermore, after two iterations, complete randomisation is used to choose the initial label to add, taking inspiration from the Pilot Method. This corresponds to setting the threshold to $+\infty$, and

Algorithm 3.6: Procedure *Construction phase(·)*

Procedure Construction phase(*C*):
Let $RCL_\alpha \leftarrow 0$ be the restricted candidate list of length α;
if *Number of iterations* > 2 **then**
  Set $RCL_\alpha \leftarrow L$ and $\alpha \leftarrow \ell$;
  Select at random a label $c \in RCL_\alpha$;
  Add label c to the set of used labels: $C \leftarrow C \cup \{c\}$;
  Update $H = (V, E(C))$ and $Comp(C)$;
end
while $Comp(C) > 1$ **do**
  Set $RCL_\alpha \leftarrow \{c \in L | c$ minimizes $Comp(C \cup \{c\})\}$;
  Select at random a label $c \in RCL_\alpha$;
  Add label c to the set of used labels: $C \leftarrow C \cup \{c\}$;
  Update $H = (V, E(C))$ and $Comp(C)$;
end

all the labels of the graph are present within the restricted candidate list (length α = total number of labels, ℓ). To intensify the search for the remaining labels to add, the list is filled considering only the labels leading to the minimum total number of connected components, as in the previous iterations.

At the end of the construction phase, a local search phase is included (*Local search(C)* procedure). It simply consists of trying to drop some labels, one by one, from the current solution C whilst retaining feasibility (see Algorithm 3.7). Local search gives a further improvement to the intensification phase of the algorithm. The entire algorithm proceeds until the user termination conditions are satisfied.

Algorithm 3.7: Procedure *Local search(·)*

Procedure Local search(C):
for $i \leftarrow 1$ *to* $|C|$ **do**
  Delete label i from the set C, i.e. $C \leftarrow C - \{i\}$;
  Update $H = (V, E(C))$ and $Comp(C)$;
  if $Comp(C) > 1$ **then**
    Add label i to the set C, i.e. $C \leftarrow C \cup \{i\}$;
  end
  Update $H = (V, E(C))$ and $Comp(C)$;
end

3.3.4 Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is a new and widely applicable metaheuristic based on dynamically changing neighbourhood structures during the search process. VNS does not follow a trajectory, but it searches for new solutions in increasingly distant neighbourhoods of the current solution, jumping only if a better solution than the current best solution is found (Hansen and Mladenović, 1997, 2001, 2003). For a survey on VNS see Section 2.2.5.

At the starting point, it is required to define a suitable neighbourhood structure. The simplest and most common choice is a structure in which the neighbourhoods have increasing cardinality: $|N_1(C)| < |N_2(C)| < ... < |N_{k_{max}}(C)|$. The process of changing neighbourhoods when no improvement occurs diversifies the search. In particular the choice of neighbourhoods of increasing cardinality yields a progressive diversification. The VNS approach can be summarized as: *"One Operator, One Landscape"*, meaning that promising zones of the search

space given by a specific neighbourhood may not be promising for other neighbourhoods (landscape). A local optimum with respect to a given neighbourhood may not be locally optimal with respect to another neighbourhood.

VNS provides a general framework and many variants exist for specific requirements. The proposed implementation for the MLST is described in Algorithm 3.8.

Algorithm 3.8: Variable Neighbourhood Search for the MLST problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialisation:
- Let $C \leftarrow 0$ be the global set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let C' be a set of labels;
- Let $H' = (V, E(C'))$ be the subgraph of G restricted to V and edges with labels in C', where $E(C') = \{e \in E : L(e) \in C'\}$;
- Let $Comp(C')$ be the number of connected components of $H' = (V, E(C'))$;

```
begin
    C ← Generate-Initial-Solution-At-Random();
    repeat
        Set k ← 1 and k_max ← (|C| + |C|/3);
        while k < k_max do
            C' ← Shaking phase(N_k(C));
            Local search(C');
            if |C'| < |C| then
                Move C ← C';
                Restart with the first neighbour: k ← 1;
            else
                Increase the size of the neighbourhood structure: k ← k + 1;
            end
        end
    until termination conditions ;
    Update H = (V, E(C));
    ⇒ Take any arbitrary spanning tree T of H = (V, E(C)).
end
```

Given a labelled graph $G = (V, E, L)$ with n vertices, m edges, and ℓ labels, each solution is encoded by a binary string, i.e. $C = (c_1, c_2, \ldots, c_\ell)$ where

$$c_i = \begin{cases} 1 & \text{if label } i \text{ is in solution } C \\ 0 & \text{otherwise} \end{cases} \qquad (\forall i = 1, \ldots, \ell). \tag{3.2}$$

The algorithm starts from an initial feasible solution C generated at random and lets parameter k vary during the execution. The successive shaking phase (*Shaking phase*$(N_k(C))$ procedure) represents the core idea of VNS: it changes the neighbourhood structure when the local search is trapped at a local minimum.

This is implemented by the random selection of a point C' within the neighbourhood $N_k(C)$ of the current solution C. The random point C' is generated in order to avoid cycling, which might occur if a deterministic rule is used.

In the shaking phase, in order to impose a neighbourhood structure on the solution space S, comprising all possible solutions, the distance considered between any two such solutions $C_1, C_2 \in S$, is the Hamming distance:

$$\rho(C_1, C_2) = |C_1 - C_2| = \sum_{i=1}^{\ell} \lambda_i \tag{3.3}$$

where $\lambda_i = 1$ if label i is included in one of the solutions but not in the other, and 0 otherwise, $\forall i = 1, ..., \ell$. Then, given a solution C, its k-th neighbourhood, $N_k(C)$, is considered as all the different sets having a Hamming distance from C equal to k labels, where $k = 1, 2, \ldots, k_{max}$, and k_{max} represents the size of the shaking. In a more formal way, the k-th neighbourhood of a solution C is defined as $N_k(C) = \{S \subset L : (\rho(C, S)) = k\}$, where $k = 1, ..., k_{max}$.

The value of k_{max} is an important parameter to tune in order to obtain an optimal balance between intensification and diversification capabilities. Choosing a small value for k_{max} produces a high intensification capability and a small diversification capability, resulting in a fast algorithm, but with a high probability of being trapped at a local minimum. Conversely, a large value for k_{max} decreases the intensification capability and increases the diversification capability, resulting in a slower algorithm, but able to escape from local minima. Computational experience indicates that the value $k_{max} \leftarrow (|C| + |C|/3)$ gives a good trade-off between these two factors.

In the shaking phase considered for the MLST problem (see Algorithm 3.9), in order to select a solution in the k-th neighbourhood of a solution C, the algorithm randomly adds further labels to C, or removes labels from C, until the resulting solution has a Hamming distance equal to k with respect to C. Addition and deletion of labels at this stage have the same probability of being chosen. For this purpose, a random number is selected between 0 and 1 ($rnd \leftarrow random[0, 1]$). If this number is smaller than 0.5, the algorithm proceeds with the deletion of a label from C. Otherwise, an additional label is included at random in C from

Algorithm 3.9: Procedure *Shaking phase(·)*

Procedure Shaking phase$(N_k(C))$***:***
Set $C' \leftarrow C$;
for $i \leftarrow 1$ *to* k **do**
  Select at random a number between 0 and 1: $rnd \leftarrow random[0,1]$;
  if $rnd \leq 0.5$ **then**
    Delete at random a label $c' \in C'$ from C', i.e. $C' \leftarrow C' - \{c'\}$;
  else
    Add at random a label $c' \in (L - C)$ to C', i.e. $C' \leftarrow C' \cup \{c'\}$;
  end
  Update $H' = (V, E(C'))$ and $Comp(C')$;
end

the set of unused labels $(L - C)$. The procedure is repeated until the number of addition/deletion operations is exactly equal to k.

The successive local search (*Local search(C′)* procedure, see Algorithm 3.10) consists of two steps. In the first step, since deletion of labels often gives an infeasible incomplete solution, additional labels may be added in order to restore feasibility. In this case, addition of labels follows the MVCA criterion of adding the label with the minimum number of connected components. Note that in case of ties in the minimum number of connected components, a label not yet included in the partial solution is chosen at random within the set of labels producing the minimum number of components (i.e. $u \in S$ where $S = \{e \in (L - C') : \min Comp(C' \cup \{e\})\}$). Then, the second step of the local search tries to delete labels one by one from the specific solution, whilst maintaining feasibility.

Algorithm 3.10: Procedure *Local search(·)*

Procedure Local search(C')***:***
while $Comp(C') > 1$ **do**
  Let S be the set of unused labels which minimize the number of connected components, i.e.
  $S = \{e \in (L - C') : \min Comp(C' \cup \{e\})\}$;
  Select at random a label $u \in S$;
  Add label u to the set of used labels: $C' \leftarrow C' \cup \{u\}$;
  Update $H' = (V, E(C'))$ and $Comp(C')$;
end
for $i \leftarrow 1$ *to* $|C'|$ **do**
  Delete label i from the set C', i.e. $C' \leftarrow C' - \{i\}$;
  Update $H' = (V, E(C'))$ and $Comp(C')$;
  if $Comp(C') > 1$ **then**
    Add label i to the set C', i.e. $C' \leftarrow C' \cup \{i\}$;
  end
  Update $H' = (V, E(C'))$ and $Comp(C')$;
end

After the local search phase, if no improvements are obtained ($|C'| \geq |C|$), the neighbourhood structure is increased ($k \leftarrow k+1$) giving a progressive diversification ($|N_1(C)| < |N_2(C)| < ... < |N_{k_{max}}(C)|$). Otherwise, the algorithm moves to the improved solution ($C \leftarrow C'$) and sets the first neighbourhood structure ($k \leftarrow 1$). Then the procedure restarts with the shaking and the local search phases, continuing iteratively until the user termination conditions (maximum allowed CPU time, maximum number of iterations, or maximum number of iterations between two successive improvements) are satisfied.

3.3.5 Hybrid local search

A current trend in the area of combinatorial optimization is the integration of good characteristics from one or more metaheuristics within the implementation of another "pure" one, in order to improve its performance. Often, this produces new methods that cannot be classified within a defined heuristic class, but are referred to as *hybrid metaheuristics* (Glover and Kochenberger, 2003; Gendreau and Potvin, 2005).

For example, a current trend is the integration of trajectory methods within population-based ones. The strength of population-based methods is the concept of recombining solutions. It allows the population-based methods to perform "big" guided steps in the search space, usually larger than the ones performed by trajectory methods. The strength of trajectory methods is based on a local search procedure which is able to strictly explore a promising region in the search space. In this way, the danger of being close to good solutions but "missing" them is not as high as in population-based methods. Summarizing, population-based methods tend to be better at identifying promising areas in the search space, whereas trajectory methods tend to be superior in exploring specific zones of the domain. Thus, hybrid local search methods, combining the advantages of population-based methods with the power of trajectory methods, are often very successful (Gendreau and Potvin, 2005). For a survey on hybridization of metaheuristics, see Section 2.4.

In many cases, hybrid algorithms are more complex to implement compared to pure ones. Thus, the application of hybrid local search to a combinatorial

optimization problem must be justified by establishing its effective performance with respect to that problem.

In this section a hybrid local search method obtained by combining Variable Neighbourhood Search and Simulated Annealing is considered, with a view to obtaining improved results for the MLST problem. Simulated Annealing has been applied to several combinatorial problems with success, such as the Quadratic Assignment problem and the Job Shop Scheduling problem (Gendreau and Potvin, 2005). Rather than as a stand-alone algorithm, it is nowadays used as a component in hybrid metaheuristics to improve performance in specific applications, as in the case of the MLST problem.

To obtain this hybrid method, a new local search mechanism for the MLST problem is first introduced. This local search is based on Variable Neighbourhood Search and is named *Complementary Local Search*. Then, Complementary Local Search is modified by adding another mechanism, the *Probabilistic MVCA* heuristic, that is inspired by Simulated Annealing. The resulting algorithm represents the hybrid local search method proposed in this section.

- Complementary Local Search

The first variant with respect to the basic Variable Neighbourhood Search proposed in this section consists of introducing a new local search mechanism, named *Complementary Local Search*. As in the previous section, given a labelled graph $G = (V, E, L)$, with n vertices, m edges, and ℓ labels, each solution is encoded by a binary string, i.e. $C = (c_1, c_2, \ldots, c_\ell)$ where

$$c_i = \begin{cases} 1 & \text{if label } i \text{ is in solution } C \\ 0 & \text{otherwise} \end{cases} \qquad (\forall i = 1, \ldots, |L|). \tag{3.4}$$

In order to impose a neighbourhood structure on the solution space S, comprising all possible solutions, the distance considered between any two such solutions $C_1, C_2 \in S$, is the Hamming distance:

$$\rho(C_1, C_2) = |C_1 - C_2| = \sum_{i=1}^{\ell} \lambda_i \tag{3.5}$$

where $\lambda_i = 1$ if label i is included in one of the solutions but not in the other, and 0 otherwise, $\forall i = 1, ..., \ell$.

Given a solution C, Complementary Local Search extracts a solution from the *complementary space* of C, and then replaces the current solution with the solution extracted. The complementary space of a solution C is defined as the set of all the labels that are not contained in C, that is $(L - C)$. To yield the solution, Complementary Local Search applies a constructive heuristic, such as the MVCA, to the subgraph of G with labels in the complementary space of the current solution. Then, the basic Variable Neighbourhood Search is applied in order to improve the resulting solution. Given C, its k-th neighbourhood, $N_k(C)$, is considered as all the different sets having a Hamming distance from C equal to k labels, where $k = 1, 2, \ldots, k_{max}$, and where k_{max} represents the size of the shaking phase. In order to select a solution in the k-th neighbourhood of a solution C, the algorithm randomly adds further labels to C, or removes labels from C, until the resulting solution has a Hamming distance equal to k with respect to C. Addition and deletion of labels at this stage have the same probability of being chosen. In a more formal way, the k-th neighbourhood of a solution C is defined as $N_k(C) = \{S \subset L : (\rho(C, S)) = k\}$, where $k = 1, ..., k_{max}$. Note that Complementary Local Search stops if either a feasible solution C is obtained, or the set of unused colours contained in the complementary space is empty, (i.e. $(\textit{complementary space} - C) \leftarrow 0$), producing a final infeasible solution. In this case, several mechanisms may be imposed by the user to deal with infeasibility.

In order to illustrate the Complementary Local Search, consider the example shown in Figure 3.3. Given an initial random solution X_0, the algorithm searches for new solutions in increasingly distant neighbourhoods of X_0. In this example, no better solutions are detected, and the current solution is still X_0. Now, the Complementary Local Search extracts a solution from the complementary space of X_0, defined as $(L - X_0)$. Let the new solution be X_0^{compl}. Then, the algorithm searches for new solutions in the neighbourhoods of X_0^{compl}. In this example, a better solution X_1 is found. The algorithm continues with this procedure until the termination conditions are satisfied. In the example, the final solution is denoted by X_2.

Complementary Local Search is proposed in order to improve the diversification of the basic Variable Neighbourhood Search for the MLST problem. Complementary Local Search has been compared to the previous algorithms, resulting in

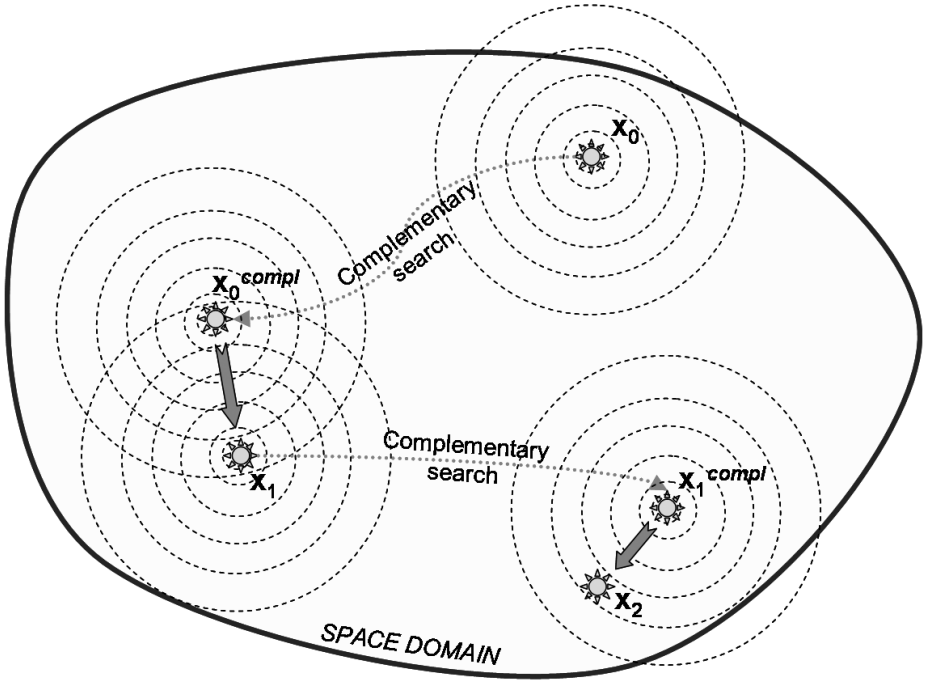


Figure 3.3: Example illustrating the steps of Complementary Local Search.

good performance. However, in order to seek further improvements, Complementary Local Search is modified by introducing concepts of the Simulated Annealing metaheuristic, resulting in the hybrid local search method that follows.

- The hybrid local search method

Variable Neighbourhood Search provides a general framework and many variants have been proposed in the literature to try to improve its performance in some circumstances (Hansen and Mladenović, 2003). For example, Pérez-Pérez et al. (2007) proposed a hybridization between VNS and a path-relinking metaheuristic to solve the p-hub median problem, while Pacheco et al. (2007) proposed mixed VNS and Tabu Search for variable selection and the determination of the coefficients for these variables that provide the best linear discrimination function, with the objective of obtaining a high classification success rate.

Although hybridizing a metaheuristic may increase the complexity of the implementation, a more advanced VNS version for the MLST problem is considered,

obtained by introducing the main concepts of Simulated Annealing within Complementary Local Search.

In particular, another heuristic is proposed to yield solutions from the complementary space of the current solution, in order to further improve the diversification by allowing worse components to be added to incomplete solutions. This heuristic is called *Probabilistic MVCA*. The introduction of a probabilistic element within the Probabilistic MVCA heuristic is inspired by Simulated Annealing (SA). However, the Probabilistic MVCA does not work with complete solutions but with partial solutions created with components added at each step. The resulting algorithm that combines Complementary Local Search and Probabilistic MVCA, represents a hybridization between VNS and SA metaheuristics.

The Probabilistic MVCA heuristic could be classified as another version of MVCA, but with a probabilistic choice of the next label. It extends basic greedy construction heuristic by allowing moves to worse solutions. Starting from an initial solution, successively a candidate move is randomly selected; this move is accepted if it leads to a solution with a better objective function value than the current solution, otherwise the move is accepted with a probability that depends on the deterioration Δ of the objective function value.

Following the SA criterion, the acceptance probability is computed according to the Boltzmann function as $\exp(-\Delta/T)$, using the temperature (T) as control parameter. The value of T is initially high, which allows many worse moves to be accepted, and is gradually reduced following a specific cooling schedule. The aim is to allow, with a specified probability, worse components with a higher number of connected components to be added to incomplete solutions.

Probability values assigned to each label are inversely proportional to the number of components they give. So the labels with a lower number of connected components will have a higher probability of being chosen. Conversely, labels with a higher number of connected components will have a lower probability of being chosen. Thus, the possibility of choosing less promising labels is allowed.

Summarizing, at each step the probabilities of selecting labels giving a smaller number of components will be higher than the probabilities of selecting labels with a higher number of components. Moreover, these differences in probabilities increase step by step as a result of the reduction of the temperature for the

cooling schedule. It means that the difference between the probabilities of two labels giving different numbers of components is higher as the algorithm proceeds. The probability of a label with a high number of components will decrease as the algorithm proceeds and will tend to zero. In this sense, the search becomes MVCA-like.

As an example, consider a graph with four labels a, b, c, and d. Starting from an empty incomplete solution, the first label is added. The numbers of connected components the labels give are evaluated. Suppose they give $a \Rightarrow 8$, $b \Rightarrow 4$, $c \Rightarrow 6$, $d \Rightarrow 2$ components. The smallest number of components is 2 given by d. Call this label s. To select the next label to add, it is necessary to compute the probabilities for each label. For a generic candidate label k, to evaluate the probability of it being added to the current solution C, the Boltzmann function $\exp(-\Delta/T)$ needs to be computed, that is $\exp\left(-\frac{Comp(C\cup k)-Comp(C\cup s)}{T}\right)$, where $Comp(C \cup k)$ is the number of connected components given by adding the label k, and $Comp(C \cup s)$ is the minimum number of connected components, given by adding the label s.

For simplicity, consider a linear cooling law for the temperature T, that is $T_{|C|} = \frac{1}{|C|+1}$, where C is the current incomplete solution. The temperature T will have value $1/1 = 1$ in the initial step (that is when the initial label needs to be added), $1/2 = 0.5$ in the second step, $1/3 = 0.33$ in the third step, and so on. Therefore, in the initial step the Boltzmann values for each label are: $a \Rightarrow 0.0024$, $b \Rightarrow$, $c \Rightarrow 0.018$, $d \Rightarrow 1$. After having evaluated the Boltzmann values, they are normalized to lie in the interval $[0, 1]$, giving the probabilities for each label to be selected. Thus, the probabilities (expressed as percentages) are: $a \Rightarrow 0.2\%$, $b \Rightarrow 11.7\%$, $c \Rightarrow 1.6\%$, $d \Rightarrow 86.5\%$. One label is selected at random according to these probabilities. Suppose label c is selected.

As the current solution is not a single connected component, a second label needs to be added. In this second step the probabilities are computed again, but with a temperature equal to 0.5. Suppose the numbers of connected components that the remaining labels give are: $a \Rightarrow 3$, $b \Rightarrow 2$, $d \Rightarrow 2$. The smaller number of components is 2 given by both b and d. Thus, in this second step ($T = 0.5$), the Boltzmann function for a generic candidate label k to be added is given by $\exp(-\Delta/T) = \exp\left(-\frac{Comp(C\cup k)-2)}{0.5}\right)$, resulting in the following values: $a \Rightarrow 0.135$,

$b \Rightarrow 1$, $d \Rightarrow 1$. They are normalized to lie in the interval $[0, 1]$, and resulting in the probabilities (expressed as percentages): $a \Rightarrow 6.3\%$, $b \Rightarrow 46.8\%$, $d \Rightarrow 46.8\%$. One label is selected at random according to these probabilities, and so on. The algorithm proceeds until only one single connected component is obtained.

Obviously, in a complex problem such as the MLST problem, the linear cooling law $T_{|C|} = \frac{1}{|C|+1}$ for the temperature is not satisfactory. After having tested different cooling laws, the best performance was obtained by using a geometric cooling schedule: $T_{k+1} = \alpha \cdot T_k = \alpha^k \cdot T_0$, where $\alpha \in [0, 1]$. This cooling law is very fast for the MLST problem, yielding a good balance between intensification and diversification. The initial temperature value T_0 and the value of α need to be evaluated experimentally.

A VNS implementation using the Probabilistic MVCA as a constructive heuristic has been tested. However, the best results were obtained by combining Complementary Local Search with the Probabilistic MVCA, obtaining the hybrid metaheuristic proposed in this section. The Probabilistic MVCA is applied both in the local search phase, to restore feasibility by adding labels to incomplete solutions, and in Complementary Local Search, to obtain a solution from the complementary space of the current solution.

The details of the implementation of this hybrid local search method are specified in Algorithm 3.11. It starts from an initial feasible solution generated at random, denoted by $Best_C$. Then the *Complementary(·)* procedure is applied to $Best_C$, as shown in Algorithm 3.12, to obtain a solution C from the complementary space of $Best_C$ by means of the Probabilistic MVCA constructive heuristic. For the geometric schedule in this procedure, computational experiments have shown that $T_0 = |Best_C|$ and $\alpha = 1/|Best_C|$, where $Best_C$ is the current best solution, are values that performed well. So, the resulting cooling law is

$$T^{Complementary}_{(|C|+1)} = \frac{T^{Complementary}_{(0)}}{\alpha^{|C|}} = \frac{1}{|Best_C|^{(|C|-1)}}. \tag{3.6}$$

The Complementary procedure stops if either a feasible solution C is obtained, or the set of unused colours contained in the complementary space is empty (i.e. $(Compl\_Space - C) = 0$), producing a final infeasible solution. Subsequently, the same shaking phase used for the basic VNS (Section 3.3.4) is ap-

Algorithm 3.11: Hybrid local search method for the MLST problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$, with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialization:
- Let $Best_C \leftarrow 0$ be the global set of labels;
- Let $H^{BEST} = (V, E(Best_C))$ be the subgraph of G restricted to V and edges with labels in $Best_C$, where $E(Best_C) = \{e \in E : L(e) \in Best_C\}$;
- Let $C \leftarrow 0$ be the set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $Comp(C)$ be the number of connected components of of $H = (V, E(C))$;
- Let C' be a set of labels;
- Let $H' = (V, E(C'))$ be the subgraph of G restricted to V and edges with labels in C', where $E(C') = \{e \in E : L(e) \in C'\}$;
- Let $Comp(C')$ be the number of connected components of $H' = (V, E(C'))$;
- Let $Compl\_Space = (L - Best_C)$ the complementary space of the best solution $Best_C$;

begin
  $Best_C \leftarrow$*Generate-Initial-Solution-At-Random()*;
  Local search($Best_C$);
  repeat
    Extract a solution from the complementary space of $Best_C$: $C \leftarrow$*Complementary*($Best_C$);
    while ($|C| < |Best_C|$) *AND (C is a feasible solution)* **do**
      Move $Best_C \leftarrow C$;
      Extract another complementary solution: $C \leftarrow$*Complementary*($Best_C$);
    end
    Set $k \leftarrow 1$ and $k_{max} \leftarrow |C| + |C|/3$;
    while $k < k_{max}$ **do**
      $C' \leftarrow$*Shaking phase*($N_k(C)$);
      Local search(C');
      if $|C'| < |C|$ **then**
        Move $C \leftarrow C'$;
        Restart with the first neighbour: $k \leftarrow 1$;
      else
        Increase the size of the neighbourhood structure: $k \leftarrow k + 1$;
      end
    end
    if $|C| < |Best_C|$ **then**
      Move $Best_C \leftarrow C$;
    end
  until *termination conditions* ;
  Update $H^{BEST} = (V, E(Best_C))$;
  $\Rightarrow$ Take any arbitrary spanning tree T of $H^{BEST} = (V, E(Best_C))$.
end

plied to the resulting solution C (*Shaking phase*($N_k(C)$) procedure, see Algorithm 3.9). It consists of the random selection of a point C' in the neighbourhood $N_k(C)$ of the current solution C ($N_k(C) = \{S \subset L : (\rho(C, S)) = k\}$, where $k = 1, 2, ..., k_{max}$). For the MLST problem, computational experience indicates that the value $k_{max} \leftarrow (|C| + |C|/3)$ gives a good trade-off between intensification

Algorithm 3.12: Procedure *Complementary(·)*

Procedure Complementary($Best_C$ ***):***
Set $C \leftarrow 0$;
while $(Comp(C) > 1) AND ((Compl\_Space - C) \neq 0)$ **do**
Let $s \in (Compl\_Space - C)$ be the label that minimizes $Comp(C \cup \{s\})$;
Geometric cooling schedule for the temperature:
$T^{Complementary}(|C|+1) = \frac{T^{Complementary}(0)}{\alpha^{|C|}}$ where $\begin{cases} T^{Complementary}(0) = |Best_C| \\ \alpha = 1/|Best_C| \end{cases}$;
foreach $c \in (Compl\_Space - C)$ **do**
Calculate the probabilities $P(c)$ for each label, normalizing the values given by the Boltzmann function: $\exp\left(-\frac{(Comp(C\cup\{c\})-Comp(C\cup\{s\}))}{T^{Complementary}(|C|+1)}\right)$ where $s \in (Compl\_Space - C)$ is the label which minimizes $Comp(C \cup \{s\})$;
end
Select at random an unused label $u \in (Compl\_Space - C)$ following the probabilities $P(\cdot)$;
Add label u to the set of used labels: $C \leftarrow C \cup \{u\}$;
Update $H = (V, E(C))$ and $Comp(C)$;
end

and diversification of the search process. At each iteration of the shaking phase, in order to select a solution in the k-th neighbourhood of a solution C, the algorithm randomly adds further labels to C, or removes labels from C, until the resulting solution has a Hamming distance equal to k with respect to C. Addition and deletion of labels at this stage have the same probability of being chosen. For this purpose, a random number is selected between 0 and 1 ($rnd \leftarrow random[0,1]$). If this number is smaller than 0.5, the algorithm proceeds with the deletion of a label from C. Otherwise, an additional label is included at random in C from the set of unused labels $(L - C)$. The procedure is repeated until the number of addition/deletion operations is exactly equal to k.

The successive local search (*Local search(·)* procedure, see Algorithm 3.13) is the same as that used in the previous VNS (Section 3.3.4). Since either the Complementary Local Search, or the deletion of labels in the shaking phase, can produce an infeasible incomplete solution, the first step of the local search consists of including additional labels in the current solution in order to restore feasibility, if needed. The addition of labels at this step is according to the Probabilistic MVCA constructive heuristic. For the geometric schedule in the local search, computational experiments have shown that $T_0 = |Best_C|^2$ and $\alpha = 1/|Best_C|$, where $Best_C$ is the current best solution, are values that performed well. The

Algorithm 3.13: Procedure *Local search(·)*

Procedure Local search(C'):
while $Comp(C') > 1$ **do**
  Let $s \in (L - C')$ be the label that minimizes $Comp(C' \cup \{s\})$;
  Geometric cooling schedule for the temperature:
  $T_{(|C'|+1)}^{Local\ search} = \frac{T_{(0)}^{Local\ search}}{\alpha^{|C'|}}$ where $\begin{cases} T^{Local\ search}(0) = |Best_C|^2 \\ \alpha = 1/|Best_C| \end{cases}$;
  foreach $c \in (L - C')$ **do**
    Calculate the probabilities $P(c)$ for each label, normalizing the values given by the Boltzmann function: $\exp\left(-\frac{(Comp(C' \cup \{c\}) - Comp(C' \cup \{s\}))}{T^{Local\ search}(|C'|+1)}\right)$ where $s \in (L - C')$ is the label which minimizes $Comp(C' \cup \{s\})$;
  end
  Select at random an unused label $u \in (L - C')$ following the probabilities $P(\cdot)$;
  Add label u to the set of used labels: $C' \leftarrow C' \cup \{u\}$;
  Update $H' = (V, E(C'))$ and $Comp(C')$;
end
for $i \leftarrow 1$ *to* $|C'|$ **do**
  Delete label i from the set C', i.e. $C' \leftarrow C' - \{i\}$;
  Update $H' = (V, E(C'))$ and $Comp(C')$;
  if $Comp(C') > 1$ **then**
    Add label i to the set C', i.e. $C' \leftarrow C' \cup \{i\}$;
  end
  Update $H' = (V, E(C'))$ and $Comp(C')$;
end

corresponding geometric cooling law is

$$T_{(|C'|+1)}^{Local\ search} = \frac{T_{(0)}^{Local\ search}}{\alpha^{|C'|}} = \frac{1}{|Best_C|^{(|C'|-2)}}. \tag{3.7}$$

Then, the second step of the local search tries to delete labels one by one from the specific solution, whilst maintaining feasibility.

Afterwards, if no improvements are obtained ($|C'| > |C|$), the neighbourhood structure is changed ($k \leftarrow k + 1$) giving a progressive diversification ($|N_1(C)| < |N_2(C)| < ... < |N_{k_{max}}(C)|$). Otherwise (i.e. if $|C'| < |C|$), the algorithm moves to the solution C' ($C \leftarrow C'$) restarting the search with the smallest neighbourhood ($k \leftarrow 1$). The algorithm proceeds with the same procedure until the user termination conditions (maximum allowed CPU time, maximum number of iterations, or maximum number of iterations between two successive improvements) are satisfied.

3.4 Computational results

In this section, the metaheuristics are compared in terms of solution quality and computational running time. The metaheuristics are identified with the abbreviations: PILOT (Pilot Method), MGA (Modified Genetic Algorithm), GRASP (Greedy Randomized Adaptive Search Procedure), VNS (Variable Neighbourhood Search), HYBRID (Hybrid local search method). All the algorithms have been implemented using the C++ programming language (Microsoft Visual C++ 2005).

Different sets of instances of the problem have been generated at random in order to evaluate how the algorithms are influenced by the parameters, the structure of the network, and the distribution of the labels on the edges. The parameters considered are the number of edges of the graph (m), the number of nodes of the graph (n), and the number of labels assigned to the edges (ℓ). Computational investigations of the behaviour of algorithms for graph theoretic problems generally use randomly generated test problems. Such problems have known statistical properties and the number of generated instances is under the control of the investigator. In addition, randomly generated test data is often publicly available and can therefore be used in computational studies.

The authors of (Cerulli et al., 2005), who kindly provided data for use in the experiments considered in this chapter, are strongly acknowledged. In the following computations, run on a Pentium Centrino microprocessor at 2.0 GHz with 512 MB RAM, different datasets are considered, each one containing 10 instances of the problem with the same set of values for the parameters n, ℓ, and m. For each dataset, solution quality is evaluated as the average objective function value for the 10 problem instances. A maximum allowed CPU time (*max-CPU-time*), determined with respect to the dimension of the problem instance, is chosen as the stopping condition for all the metaheuristics. For MGA, a variable number of iterations for each instance is used, determined such that the computations take approximately *max-CPU-time* for the specific dataset. Selection of the maximum allowed CPU time as the stopping criterion is made in order to have a direct comparison of all the metaheuristics with respect to the quality of their solutions.

All the heuristics run for *max-CPU-time* and, in each case, the best solution is recorded. The computational times reported in the tables are the average times at which the best solutions are obtained. The reported times have precision of ±5 ms. Where possible, the results of the metaheuristics are compared to the exact solution, identified with the label EXACT.

The *Exact Method* is an A* or backtracking procedure to test the subsets of L. This search method performs a branch and prune procedure in the partial solution space based on a recursive procedure *Test* that attempts to find a better solution from the current incomplete solution. The main program that solves the MLST problem calls the *Test* procedure with an empty set of labels. The details are specified in Algorithm 3.14.

Algorithm 3.14: Exact Method for the MLST problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels;
Output: A spanning tree T;
Initialisation:
- Let $C \leftarrow 0$ be the initially empty set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $C^* \leftarrow L$ be the global set of used labels;
- Let $H^* = (V, E(C^*))$ be the subgraph of G restricted to V and edges with labels in C^*, where $E(C^*) = \{e \in E : L(e) \in C^*\}$;
- Let $Comp(C)$ be the number of connected components of $H = (V, E(C))$;

```
begin
    Call Test(C);
    ⇒ Take any arbitrary spanning tree T of H* = (V, E(C*)).
end

Procedure Test(C):
if |C| < |C*| then
    Update Comp(C);
    if Comp(C) = 1 then
        Move C* ← C;
    else if |C| < |C*| − 1 then
        foreach c ∈ (L − C) do
            Try to add label c : Test(C ∪ {c});
        end
    end
end
```

In order to reduce the number of test sets, it is more convenient to use a good approximate solution for C^* in the initial step, instead of considering all the labels. Another improvement that avoids the examination of a large number of incomplete solutions consists of rejecting every incomplete solution that cannot

be completed to get only one connected component. Note that if an incomplete solution C' with a number of labels $|C'| = |C^*| - 2$ is evaluated, the algorithm should try to add the labels one by one to check if it is possible to find a better solution for C^* with a smaller dimension, that is $|C'| = |C^*| - 1$. To complete this solution C', a label with a frequency at least equal to the actual number of connected components minus 1 needs to be added. If this requirement is not satisfied, the incomplete solution can be rejected, speeding up the search process.

The running time of this Exact Method grows exponentially, but if either the problem size is small or the optimal objective function value is small, the running time is reasonable and the method obtains the exact solution. The complexity of the instances increases with the dimension of the graph (number of nodes and labels), and the reduction in the density of the graph. For the computational tests considered in this chapter, the optimal solution is reported unless a single instance requires more than 3 hours of CPU time. In such a case, not found (NF) is reported.

3.4.1 Experimental analysis

In the considered computations, two different groups of datasets have been computed, including instances with a number of vertices, n, and a number of labels, ℓ, from 20 up to 500. All these instances are available from the author (Consoli, 2007a). The number of edges, m, is obtained indirectly from the density d of edges whose values are chosen to be 0.8, 0.5, and 0.2. Analysing the performance of the algorithms considered, for a single dataset a metaheuristic should be considered *worse* than another one if either it obtains a larger average objective function value, or an equal average objective function value but in a greater computational time.

Group 1 examines small instances with the number of vertices equal to the number of labels. These values are chosen to be between 20 and 50 in steps of 10. Thus, the datasets considered are $n = \ell = 20$, 30, 40, 50, and $d = 0.8$, 0.5, 0.2, for a total of 12 datasets (120 instances). Computational results are presented in Table 3.1, which reports the average objective function values found

Table 3.1: Computational results for Group 1 (*max-CPU-time* for heuristics = 1000 ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
		0.8	*2.4*	2.4	2.4	2.4	2.4	2.4
20	20	0.5	*3.1*	3.2	3.1	3.1	3.1	3.1
		0.2	*6.7*	6.7	6.7	6.7	6.7	6.7
		0.8	*2.8*	2.8	2.8	2.8	2.8	2.8
30	30	0.5	*3.7*	3.7	3.7	3.7	3.7	3.7
		0.2	*7.4*	7.4	7.4	7.4	7.4	7.4
		0.8	*2.9*	2.9	2.9	2.9	2.9	2.9
40	40	0.5	*3.7*	3.7	3.7	3.7	3.7	3.7
		0.2	*7.4*	7.6	7.4	7.4	7.4	7.4
		0.8	*3*	3	3	3	3	3
50	50	0.5	*4*	4	4.1	4	4	4
		0.2	*8.6*	8.6	8.6	8.6	8.6	8.6
	TOTAL:		*55.7*	56	55.8	55.7	55.7	55.7
Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
		0.8	*0*	0	15.6	1.6	0	0
20	20	0.5	*0*	1.6	22	0	0	0
		0.2	*11*	3.1	23.4	0	1.6	0
		0.8	*0*	3	9.4	1.6	0	1.5
30	30	0.5	*0*	3.1	26.5	0	0	0
		0.2	*138*	4.7	45.4	1.5	5.2	3.1
		0.8	*2*	6.3	12.5	1.5	0	3.1
40	40	0.5	*3.2*	7.9	28.2	1.5	3.1	6.2
		0.2	*100.2*$*10^3$	10.8	120.3	15.6	9.6	1.6
		0.8	*3.1*	17.1	21.8	3	0	3.1
50	50	0.5	*21.9*	20.2	531.3	9.4	4.1	6.2
		0.2	*66.3*$*10^3$	17.2	93.6	3.2	11.9	8
	TOTAL:		*166.7*$*10^3$	95	950	38.9	35.5	32.7

by the heuristics for the datasets of Group 1, and the corresponding average computational times, with a *max-CPU-time* of 1 second.

Looking at this table, all the heuristics performed well for the Group 1 instances. However, MGA is considerably slower than the other metaheuristics, as a result of a poor intensification capability and an excessive diversification capa-

bility for these instances. PILOT is faster than MGA but it produces slightly worse solutions with respect to solution quality. It exhibits an opposite behaviour to that of MGA, being characterised by a limited diversification capability which sometimes does not allow the search process to escape from local optima. The performance of GRASP, VNS, and HYBRID are comparable for these trivial instances of the problem. They are able to obtain all the exact solutions in very short running times and are the best performing heuristics for Group 1 in terms of solution quality and computational running time.

Group 2 considers larger instances of the MLST problem with a fixed number of vertices, and a number of labels $\ell = 0.25 \cdot n$, $0.5 \cdot n$, n, $1.25 \cdot n$. Thus, the datasets of Group 2 are $n = 100$, 200, 500 vertices, $\ell = 0.25 \cdot n$, $0.5 \cdot n$, n, $1.25 \cdot n$ labels, and $d = 0.8$, 0.5, 0.2 density, for a total of 36 datasets (360 instances). Furthermore, a *max-CPU-time* of 20 seconds has been considered for Group 2 with $n = 100$; of 60 seconds for Group 2 with $n = 200$; and of 300 seconds for Group 2 with $n = 500$. Average objective function values and the corresponding average computational times are reported in Tables 3.2 - 3.3 - 3.4 respectively.

For all the Group 2 instances with $n = 100$, looking at Table 3.2, the best performance is obtained by VNS which produces the solutions with the best solution quality and the shortest running times. Next in performance ranking is HYBRID which produces solutions with the same quality of those produced by VNS, although for the instance $[n = 100, \ell = 125, d = 0.2]$ is quite slow. GRASP also performs well, obtaining the same solutions as VNS and HYBRID, with the exception for the instance $[n = \ell = 100, d = 0.2]$. As in Group 1, PILOT and MGA obtain worse solutions and their defects of excessive diversification and poor intensification for MGA and, conversely, of excessive intensification and poor diversification for PILOT are demonstrated.

Table 3.3 and Table 3.4, with larger instances of the problem (Group 2 with $n = 200$, and Group 2 with $n = 500$) show the same relative behaviour for all the metaheuristics considered. VNS, HYBRID, and GRASP are always the best performing methods, indicating an optimal tuning between intensification and diversification of the search process, which evidently is not obtained by PILOT and MGA which obtain the worst solution in terms of quality and computational running time. VNS and HYBRID always obtain the solutions with the best

Table 3.2: Computational results for Group 2 with $n = 100$ (*max-CPU-time* for heuristics = $20{*}10^3$ ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
100	25	0.8	*1.8*	1.8	1.8	1.8	1.8	1.8
		0.5	*2*	2	2	2	2	2
		0.2	*4.5*	4.5	4.5	4.5	4.5	4.5
	50	0.8	*2*	2	2	2	2	2
		0.5	*3*	3.1	3	3	3	3
		0.2	*6.7*	6.9	6.7	6.7	6.7	6.7
	100	0.8	*3*	3	3	3	3	3
		0.5	*4.7*	4.7	4.7	4.7	4.7	4.7
		0.2	*NF*	10.1	9.9	9.8	9.7	9.7
	125	0.8	*4*	4	4	4	4	4
		0.5	*5.2*	5.4	5.2	5.2	5.2	5.2
		0.2	*NF*	11.2	11.1	11	11	11
TOTAL:			-	58.7	57.9	57.7	57.6	57.6

Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
100	25	0.8	*9.4*	4.7	26.5	0	0	0
		0.5	*14*	12.6	29.7	4.6	0	4.5
		0.2	*34.3*	23.2	45.3	9.3	3.1	4.8
	50	0.8	*17.8*	67.3	23.5	6.4	7.7	12.6
		0.5	*23.5*	90.7	106.2	51.6	42.4	21.7
		0.2	*10.2*$*10^3$	103.2	148.3	57.8	49.7	26.5
	100	0.8	*142.8*	378.1	254.7	61	215	146.9
		0.5	*2.4*$*10^3$	376.2	300	28.2	114.7	75.9
		0.2	*NF*	399.9	$9.4{*}10^3$	$1.2{*}10^3$	414.8	514
	125	0.8	*496.9*	565.7	68.7	9.4	10.1	20.2
		0.5	*179.6*$*10^3$	576.3	759.4	595.4	551.1	345.4
		0.2	*NF*	634.5	$2{*}10^3$	562.9	420.4	$1.2{*}10^3$
TOTAL:			-	$3.2{*}10^3$	$13.2{*}10^3$	$2.6{*}10^3$	$1.8{*}10^3$	$2.4{*}10^3$

quality, but they lose a lot, sometimes, in terms of computational running time with respect to GRASP (see for example the instances [$n = \ell = 200$, $d = 0.2$], [$n = \ell = 500$, $d = 0.2$], and [$n = 500, \ell = 625$, $d = 0.2$]). From this analysis, perhaps GRASP is slightly lacking in terms of exploration of the search space with respect to the VNS and HYBRID approaches. HYBRID and VNS consis-

Table 3.3: Computational results for Group 2 with $n = 200$ (*max-CPU-time* for heuristics = $60{*}10^3$ ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
200	50	0.8	*2*	2	2	2	2	2
		0.5	*2.2*	2.2	2.2	2.2	2.2	2.2
		0.2	*5.2*	5.2	5.2	5.2	5.2	5.2
	100	0.8	*2.6*	2.6	2.6	2.6	2.6	2.6
		0.5	*3.4*	3.4	3.4	3.4	3.4	3.4
		0.2	*NF*	8.3	8.3	8.1	7.9	7.9
	200	0.8	*4*	4	4	4	4	4
		0.5	*NF*	5.5	5.4	5.4	5.4	5.4
		0.2	*NF*	12.4	12.4	12.2	12	12
	250	0.8	*4*	4	4	4.1	4	4
		0.5	*NF*	6.3	6.3	6.3	6.3	6.3
		0.2	*NF*	13.9	14	13.9	13.9	13.9
TOTAL:			-	69.8	69.8	69.4	68.9	68.9

Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
200	50	0.8	*29.7*	90.7	26.5	20.5	0	0
		0.5	*32.7*	164.1	68.8	14.2	17.2	34.4
		0.2	$5.4{*}10^3$	320.4	326.6	37.5	241.3	232.8
	100	0.8	*138.6*	876.5	139.3	45.3	123.2	140.8
		0.5	*807.8*	$1.2{*}10^3$	$1.6{*}10^3$	176.6	151.1	159.4
		0.2	*NF*	$1.3{*}10^3$	$2.2{*}10^3$	667.2	$1.7{*}10^3$	$2.9{*}10^3$
	200	0.8	$22.5{*}10^3$	$5.9{*}10^3$	204.6	43.6	32	79.7
		0.5	*NF*	$5.6{*}10^3$	$16.1{*}10^3$	885.6	971.9	876.1
		0.2	*NF*	$5{*}10^3$	$12.7{*}10^3$	$9.4{*}10^3$	$12.8{*}10^3$	$33.7{*}10^3$
	250	0.8	$20.6{*}10^3$	$9.1{*}10^3$	$2.2{*}10^3$	$4.9{*}10^3$	$1.1{*}10^3$	$1.5{*}10^3$
		0.5	*NF*	$8.4{*}10^3$	$17.6{*}10^3$	506	$3.4{*}10^3$	$2.3{*}10^3$
		0.2	*NF*	$8{*}10^3$	$26.4{*}10^3$	$1.4{*}10^3$	$3.2{*}10^3$	$1.5{*}10^3$
TOTAL:			-	$45.9{*}10^3$	$79.6{*}10^3$	$18.1{*}10^3$	$23.7{*}10^3$	$43.4{*}10^3$

tently produce equally good solutions. However, the motivation to introduce a high diversification capability in HYBRID is to obtain improved performance in large problem instances. Inspection of Table 3.4 shows that this aim is achieved: HYBRID is faster than VNS for large problem instances. Conversely, for smaller problem instances (see Tables 3.1 - 3.3), in general VNS obtains solutions with

Table 3.4: Computational results for Group 2 with $n = 500$ (*max-CPU-time* for heuristics = $300{*}10^3$ ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
500	125	0.8	*2*	2	2	2	2	2
		0.5	*2.6*	2.6	2.6	2.6	2.6	2.6
		0.2	*NF*	6.3	6.2	6.2	6.2	6.2
	250	0.8	*3*	3	3	3	3	3
		0.5	*NF*	4.2	4.3	4.2	4.1	4.1
		0.2	*NF*	9.9	10.1	9.9	9.9	9.9
	500	0.8	*NF*	4.8	4.7	4.7	4.7	4.7
		0.5	*NF*	6.7	7.1	6.5	6.5	6.5
		0.2	*NF*	15.9	16.6	15.9	15.8	15.8
	625	0.8	*NF*	5.1	5.4	5.1	5.1	5.1
		0.5	*NF*	8.1	8.3	7.9	7.9	7.9
		0.2	*NF*	18.5	19.1	18.4	18.3	18.3
TOTAL:			-	87.1	89.4	86.4	86.1	86.1

Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	MGA	GRASP	VNS	HYBRID
500	125	0.8	*370*	$3.4{*}10^3$	18	152	17.1	45
		0.5	*597*	$6.6{*}10^3$	$2.6{*}10^3$	455	$1.1{*}10^3$	560
		0.2	*NF*	$11.9{*}10^3$	$57.1{*}10^3$	$4{*}10^3$	$3.9{*}10^3$	$3.7{*}10^3$
	250	0.8	$5.3{*}10^3$	$35.49{*}10^3$	516	248	142.3	490
		0.5	*NF*	$65.3{*}10^3$	$28{*}10^3$	583	$84{*}10^3$	$26.9{*}10^3$
		0.2	*NF*	$156.4{*}10^3$	$181.2{*}10^3$	$3.3{*}10^3$	$5.1{*}10^3$	$10.2{*}10^3$
	500	0.8	*NF*	$200.5{*}10^3$	$117.5{*}10^3$	$28.1{*}10^3$	$22.3{*}10^3$	$8.6{*}10^3$
		0.5	*NF*	$190.1{*}10^3$	$170.9{*}10^3$	$90.9{*}10^3$	$32.3{*}10^3$	$110.2{*}10^3$
		0.2	*NF*	$300.6{*}10^3$	$241.8{*}10^3$	$20.2{*}10^3$	$139.7{*}10^3$	$50.3{*}10^3$
	625	0.8	*NF*	$184.3{*}10^3$	$51.9{*}10^3$	$4.9{*}10^3$	$16.1{*}10^3$	970
		0.5	*NF*	$200.9{*}10^3$	$222.2{*}10^3$	$35.7{*}10^3$	$44.7{*}10^3$	$33.9{*}10^3$
		0.2	*NF*	$289.9{*}10^3$	$297.8{*}10^3$	$53.1{*}10^3$	$155.5{*}10^3$	$60{*}10^3$
TOTAL:			-	$1645.3{*}10^3$	$1371.5{*}10^3$	$213.8{*}10^3$	$504.9{*}10^3$	$395.9{*}10^3$

shorter computational running than HYBRID.

Considering only solution quality, the average values of the objective function of the metaheuristics among all the considered datasets are: PILOT = 5.66, MGA = 5.68, GRASP = 5.61, VNS = 5.59, HYBRID = 5.59. Thus, the best ranking with respect to the solution quality (from the best to the worst) is: VNS and

HYBRID, followed respectively by GRASP, PILOT, and MGA.

3.4.2 Statistical analysis of the results

Computing only the average objective function values of the metaheuristics over multiple data does not provide a full comparison between them. Averages are susceptible to outliers: they can allow excellent performance on some datasets to compensate for an overall bad performance. There may be situations in which such behaviour is desired. However, in general, algorithms that behave well on as many problems as possible are preferred.

Tests to determine the statistical significance of differences between the performances of the metaheuristics have been carried out (Hollander and Wolfe, 1999). The issue of statistical tests for comparison of algorithms on multiple datasets was theoretically and empirically reviewed by Demśar (2006). The null-hypothesis being tested is that the metaheuristics have equal mean performance and the observed differences are merely random. The alternative hypothesis is that the algorithms have different mean performances of statistical significance.

The most common statistical method for testing differences between more than two algorithms is Analysis of Variance (ANOVA) (see Hollander and Wolfe (1999) and Demśar (2006) for more details). Since ANOVA is based on assumptions that are violated in this context, the *Friedman test* (Friedman, 1940), that is the non-parametric equivalent of ANOVA, and its corresponding *Nemenyi post-hoc test* (Nemenyi, 1963), are used.

According to the Friedman test, the statistical significance of differences between the metaheuristics is examined by testing whether the measured average ranks are significantly different from the overall mean rank. In particular, the version of the Friedman test developed by Iman and Davenport (1980) is used, which considers a powerful test statistic F_F (Appendix B). If the equivalence of the algorithms is rejected, the Nemenyi post-hoc test is applied in order to perform pairwise comparisons.

To perform the Friedman and Nemenyi tests, the ranks of the algorithms for each dataset are evaluated, with a rank of 1 assigned to the best performing algorithm, rank 2 to the second best one, and so on. The average ranks for

each metaheuristic among the 48 datasets are: PILOT = 4.23, MGA = 4.45, GRASP = 2.3, VNS = 2, HYBRID = 2.02. According to the ranking, VNS is the best performing algorithm, immediately followed by HYBRID and GRASP, then PILOT and MGA achieving the worst results.

Now, the statistical significance of differences between these ranks are analysed. Consider the version by Iman and Davenport (1980) for the Friedman test for $k = 5$ algorithms and $N = 48$ datasets. The value of the F_F test statistic, which is distributed according to the F-distribution with $(k-1, (k-1)(N-1)) = (4, 188)$ degrees of freedom, is computed. This value is 72.08, which is greater than the critical value (3.42 for $\alpha = 1\%$, where α is the significance level of the test expressed as percentage). Thus, a significant difference between the performance of the metaheuristics exists, according to the Friedman test.

As the equivalence of the algorithms is rejected, the Nemenyi post-hoc test is applied. Considering a significance level $\alpha = 1\%$, the critical value is $q_{0.01} \cong 3.26$. The critical difference (CD) for the Nemenyi test is

$$CD = 3.26 \cdot \sqrt{\frac{5 \cdot 6}{6 \cdot 48}} \cong 1.05; \tag{3.8}$$

The differences between the average ranks of the metaheuristics are reported in Table 3.5. From this table, two groups of metaheuristics are identified. The first group includes VNS, HYBRID, and GRASP, while the second group includes PILOT and MGA. Considering a significance level $\alpha = 1\%$, the algorithms within each group have comparable performance according to the Nemenyi test since, in

Table 3.5: Pairwise differences of the average ranks of the algorithms (Critical difference = 1.05 for a significance level $\alpha = 1\%$ for the Nemenyi test)

ALGORITHM (average rank)	VNS (2)	HYBRID (2.02)	GRASP (2.3)	PILOT (4.23)	MGA (4.45)
VNS (2)	-	0.02	0.3	**2.23**	**2.45**
HYBRID (2.02)	-	-	0.28	**2.21**	**2.43**
GRASP (2.3)	-	-	-	**1.93**	**2.15**
PILOT (4.23)	-	-	-	-	0.22
MGA (4.45)	-	-	-	-	-

each case, the value of the test statistic is less than the critical difference. Conversely, two algorithms belonging to different groups have significantly different performance according to the Nemenyi test. Summarizing, from the Friedman and Nemenyi statistical tests, VNS, HYBRID, and GRASP have comparable performance, and they are the best performing algorithms. On the other hand, PILOT and MGA have comparable performance, but worse than VNS, HYBRID, and GRASP.

Another way to compare the performance of the algorithms is to count the number of times they generate the optimal solution. In particular, counting the overall number of exact solutions obtained is a good approach to estimating the diversification capability of each metaheuristic. The Exact Method obtains the exact solution for all problem instances of 32 datasets, among the overall 48 datasets; for the remaining sets NF is reported. Therefore, the total number of instances in which the exact solution was obtained is: $32 \times 10 = 320$.

The percentages of the number of optimal solutions obtained by the metaheuristics among the 320 instances are (ranking from the best to the worst algorithm): VNS = 100, HYBRID = 100, GRASP = 99.7, MGA = 99.7, PILOT = 97.5.

VNS and HYBRID obtain all the optimal solutions, underlying a high exploration capability even for complex instances. In the same way, GRASP and MGA offer very good results, missing only 1 solution out of 320, although MGA is extremely time consuming. With 8 cases (out of 320), PILOT fails to find the global optimum and became trapped at a local optimum.

Furthermore, some optima reached by the metaheuristics require a greater computational time than required by the Exact Method, thus nullifying the purpose of the metaheuristics. In this sense the best performances are obtained again by VNS, HYBRID, and GRASP, all of which require less computational time than the Exact Method among the 32 datasets. In contrast, PILOT and MGA obtain the optimal solution but in a time that exceeds that of the Exact Method in 11 and 18 datasets, respectively. Although MGA reaches more exact solutions than PILOT, it is computationally more burdensome.

From this further analysis, the results reinforce the conclusion that VNS, HYBRID, and GRASP are effective metaheuristics for the MLST problem. Fur-

thermore, the algorithm which appears to be the most suitable for the proposed problem is VNS, thanks to the following features: ease of implementation, user-friendly code, high-quality of the solutions, and shorter computational running times.

3.5 Conclusions and further research

In this chapter, several metaheuristics for the minimum labelling spanning tree (MLST) problem have been studied. In particular, the metaheuristics recommended in the literature have been examined and implemented: the Modified Genetic Algorithm (MGA) by Xiong et al. (2006) and the Pilot Method (PILOT) by Cerulli et al. (2005). Furthermore, some new implementations for the MLST problem have been proposed: a Greedy Randomized Adaptive Search Procedure (GRASP), a basic Variable Neighbourhood Search (VNS), and a hybrid local search method (HYBRID) obtained by combining Variable Neighbourhood Search with Simulated Annealing (SA).

Computational experiments were performed using different instances of the MLST problem to evaluate how the algorithms are influenced by the parameters, the structure of the network, and the distribution of the labels on the edges. Applying the nonparametric statistical tests of Friedman (1940) and Nemenyi (1963), it has been concluded that VNS, HYBRID, and GRASP have significantly better performance than the other methods recommended in the literature with respect to solution quality and running time. Furthermore, this result has been reinforced by comparing the metaheuristics with an exact approach. VNS, HYBRID, and GRASP obtain a large number of optimal or near-optimal solutions, showing an enhanced diversification capability.

The results indicate that VNS, HYBRID, and GRASP are fast and extremely effective metaheuristics for the MLST problem. In addition, VNS is particularly recommended for the proposed problem because of its simplicity and its ability to obtain high-quality solutions in short computational running times.

Future research will consist of trying to further improve the performance of these procedures (for example through hybridization with other metaheuristics) particularly for large instances of the problem. For this purpose, an algorithm

based on Ant Colony Optimization (ACO) is currently under study in order to try to obtain a larger diversification capability by extending the current greedy MVCA local search. Indeed, a proper ACO implementation may allow moves to worse solutions by providing an alternative probabilistic solution construction mechanism.

You know more than you think you know, just as you know less than you want to know.

Oscar Wilde

Chapter 4

Minimum labelling Steiner tree problem

This chapter presents a study on heuristic solution approaches to the minimum labelling Steiner tree (MLSteiner) problem, an NP-hard graph problem related to the minimum labelling spanning tree problem. Given an undirected labelled connected graph, the aim is to find a spanning tree covering a given subset of nodes of the graph, whose edges have the smallest number of distinct labels. Such a model may be used to represent many real-world problems in telecommunications and multimodal transportation networks. Several metaheuristics are proposed and evaluated. They outperform the Pilot Method (PILOT), which is the heuristic recommended by the literature for the MLSteiner problem (Cerulli et al., 2006). Further experimental analysis shows that some of the proposed heuristics (a Greedy Randomized Adaptive Search Procedure (GRASP), a Variable Neighbourhood Search (VNS), and a hybrid local search method (HYBRID) obtained by combining Variable Neighbourhood Search with Simulated Annealing (SA)) are effective approaches for the MLSteiner problem, obtaining high-quality solutions in short computational running times.

4.1 Introduction

This chapter focuses on the *minimum labelling Steiner tree* (MLSteiner) problem, a generalization of the minimum labelling spanning tree (MLST) problem, already

discussed in Chapter 3, to the case where not necessarily all but only a subset of required nodes need to be spanned. In particular, given a graph with labelled (or coloured) edges, the MLSteiner problem seeks a subgraph which spans a subset of nodes (basic nodes) of the graph, and whose edges have the least number of distinct labels (or colours).

As with the MLST problem, the MLSteiner problem has many applications in real-world problems. For example, in telecommunications networks, a node may communicate with other nodes by means of different types of communications media. Considering a set of basic nodes that must be connected, the construction cost may be reduced, in some situations, by connecting the basic nodes with the smallest number of possible communications types (Tanenbaum, 1989).

Another example is given by multimodal transportation networks (Van-Nes, 2002). A multimodal transportation network can be represented by a graph where a label is assigned to each edge, denoting a different company managing that edge, and each node represents a different location. It is often desirable to provide a complete service between a basic set of locations, without cycles, using the minimum number of companies, in order to minimize the cost.

The minimum labelling Steiner tree problem is formally defined as a network or graph problem as follows:

> *MLSteiner problem:* Let $G = (V, E, L)$ be a labelled, connected, undirected graph, where V is the set of nodes, E is the set of edges, that are labelled on the set L of labels, and let $Q \subseteq V$ be a set of nodes that must be connected (basic nodes). The aim is to find a subgraph T connecting all the basic nodes Q such that $|L_T|$ is minimized, where L_T is the set of labels used in T.

The MLSteiner problem is NP-hard by reduction from the MLST problem, as the MLSteiner problem is the special case of the MLST problem when $Q = V$. Figure 4.1 shows an example of an input graph, where the solid vertices represent the basic nodes. The minimum labelling Steiner tree solution of this example is shown in Figure 4.2.

In this chapter, several new metaheuristics for the MLSteiner problem are proposed: a *Greedy Randomized Adaptive Search Procedure*, a *Discrete Particle Swarm Optimization*, a *Variable Neighbourhood Search*, and a hybrid local search

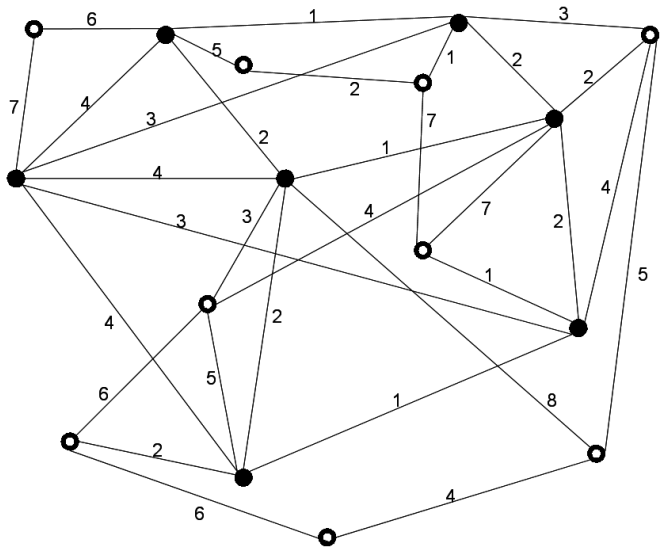


Figure 4.1: Example of an input graph of the MLSteiner problem.

method, which is a hybridization between *Variable Neighbourhood Search* and *Simulated Annealing* metaheuristics. Computational results for these approaches are compared to those from the *Pilot Method*, which is considered to be the best performing heuristic in the current literature (Cerulli et al., 2006), and with those

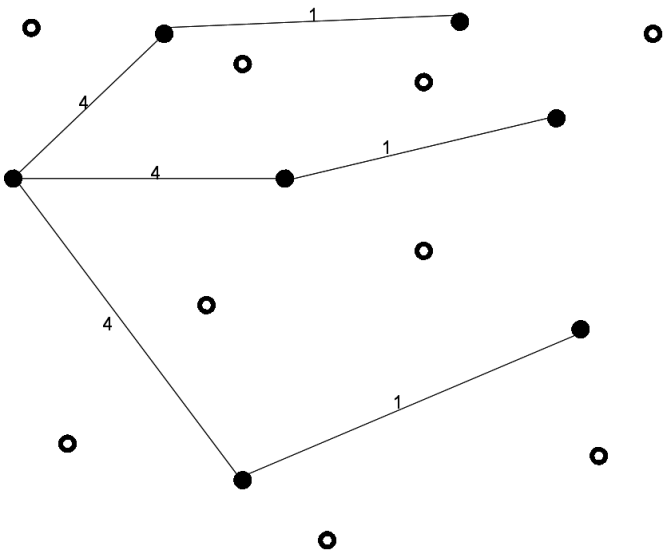


Figure 4.2: Minimum labelling Steiner tree solution for the graph of Figure 4.1.

from an Exact Method.

The structure of the chapter is as follows. First the problem and its origins are described, reviewing the associated literature. As the minimum labelling Steiner tree problem is a direct extension of the well-known Steiner tree problem and of the minimum labelling spanning tree problem, these basic problems are discussed. Details of the methods considered are presented in Section 4.3. Section 4.4 contains a computational analysis and evaluation and, finally, conclusions are described in Section 4.5. The basic concepts of metaheuristics and combinatorial optimization were presented in Chapter 2, but, for further information, the reader is referred to (Voß et al., 1999; Glover and Kochenberger, 2003; Gendreau and Potvin, 2005).

4.2 Origin of the problem

The minimum labelling Steiner tree problem was introduced by Cerulli et al. (2006). It is a graph combinatorial optimization problem extending the well-known Steiner tree problem and the minimum labelling spanning tree problem (already discussed in Chapter 3).

Given a graph with positive-weighted edges, and with a subset of basic nodes, the *Steiner tree* (Steiner) problem consists of finding a minimum-weight tree spanning all the basic nodes. This problem dates back to Fermat, who formulated it as a geometric problem: find a point p in the Euclidean plane minimizing the sum of the distances to three given points. This was solved before 1640 by Torricelli (Krarup and Vajda, 1997). Subsequently Steiner worked on the general problem for n points. More details appears in (Hwang et al., 1992). Expositions on the difficulty of the Steiner problem can be found in (Karp, 1975; Garey et al., 1977), while several heuristics for the Steiner problem in graphs are reported in (Grimwood, 1994; Voß, 2000).

A large number of real-world applications of the Steiner problem exist, most of them relate to network design (Winter, 1987) and telecommunications (Voß, 2006). Steiner problems arising in the layout of connection structures in networks, such as topological network design, location, and in VLSI (Very Large Scale Integrated) circuit design, are discussed in (Korte et al., 1990; Francis et al.,

1992). Furthermore, analogies can be drawn between minimum Steiner trees and minimum energy configurations in certain physical systems (Miehle, 1958).

The MLSteiner problem was first considered by Cerulli et al. (2006) as an extension of the Steiner problem and the MLST problem. They also compared their Pilot Method with some other metaheuristics for the MLSteiner problem: Tabu Search, Simulated Annealing, and some implementations of Variable Neighbourhood Search. From their analysis, the Pilot Method was shown to be the best performing heuristic for the problem (Cerulli et al., 2006).

The success of the heuristic solution approaches for the MLST problem proposed in Chapter 3 provided the motivation for considering the implementation of similar approaches for the MLSteiner problem, and this is the focus of the work reported in this chapter.

4.3 Description of the algorithms

This section introduces an Exact Method for the MLSteiner problem, and analyses the *Pilot Method* (PILOT) by Cerulli et al. (2006). It then describes the main features of other metaheuristics proposed for the MLSteiner problem: a *Greedy Randomized Adaptive Search Procedure* (GRASP), a *Discrete Particle Swarm Optimization* (DPSO), a *Variable Neighbourhood Search* (VNS), and a hybrid local search method (HYBRID) obtained by combining *Variable Neighbourhood Search* with *Simulated Annealing* (SA).

Before going into the details of these algorithms, it is useful to define the concept of a Steiner component (Cerulli et al., 2006). Given an undirected, connected, labelled input graph, a Steiner component is a connected subgraph of the input graph containing at least one basic node. This concept will be used throughout the section.

4.3.1 Exact Method

The Exact Method (EXACT) for the MLSteiner problem is based on a backtracking procedure, as with the Exact Method for the MLST problem (see Chapter 3). Given a labelled connected undirected graph $G = (V, E, L)$ with n vertices, m

edges, ℓ labels, and a subset $Q \subseteq V$ of basic nodes, EXACT performs a branch and prune procedure in the partial solution space based on a recursive procedure, *Test*. The details are specified in Algorithm 4.1.

Algorithm 4.1: Exact Method for the MLSteiner problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels, $Q \subseteq V$ basic nodes;
Output: A tree T;
Initialization:
- Let $C \leftarrow 0$ be the initially empty set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $C^* \leftarrow L$ be the global set of used labels;
- Let $H^* = (V, E(C^*))$ be the subgraph of G restricted to V and edges with labels in C^*, where $E(C^*) = \{e \in E : L(e) \in C^*\}$;
- Let $Comp(C)$ be the number of Steiner components of C, i.e. the number of connected components of the subgraph $(Q, E(C))$;

begin
  Call $Test(C)$;
  $\Rightarrow$ Take any arbitrary spanning tree T of $H^* = (V, E(C^*))$.
end

Procedure Test(C):
if $|C| < |C^*|$ **then**
  Update $Comp(C)$;
  if $Comp(C) \leftarrow 1$ **then**
    Move $C^* \leftarrow C$;
  else if $|C| < |C^*| - 1$ **then**
    foreach $c \in (L - C)$ **do**
      Try to add label c : $Test(C \cup \{c\})$;
    end
  end
end

The procedure *Test* starts from an empty set of labels and iteratively builds a solution by adding labels one by one until all the basic nodes, $Q \subseteq V$, are connected. In this method, all the possible combinations of labels are considered, and so its running time is computationally burdensome. The running time grows exponentially with the dimension of the graph (number of nodes and labels), and the reduction in the density of the graph.

In order to speed up this method, the following procedure is adopted. Let $C^* \subseteq L$ be a current solution, and $C' \subseteq L$ be an incomplete solution to evaluate. If the dimension of C' is equal to $|C^*| - 2$, the algorithm should try to add all the labels one by one to check if it is possible to find a better solution for C^* with

a smaller dimension, that is $|C^*| - 1$. Instead of trying to add all the labels one by one to complete C', the algorithm only considers the labels with a frequency at least equal to the actual number of connected components minus 1 (in other words only the candidate labels which may yield a connected graph if added to the incomplete solution C' are considered). If this requirement is not satisfied, the incomplete solution can be rejected, speeding up the search process.

If either the problem size is small or the optimal objective function value is small, the running time of this exact approach is acceptable and it is possible to obtain the exact solution.

4.3.2 Pilot Method

The Pilot Method (PILOT) metaheuristic was first introduced by Duin and Voß (1999) for the Steiner tree problem, and was applied with success to several combinatorial optimization problems (Voß et al., 2004). The core idea of this metaheuristic is to exhaust tentatively all the possible choices with respect to a reference solution, called the master solution, by means of a basic constructive heuristic. For each possible choice, the basic heuristic (or application process) works as a building block for the master solution, by adding components until a feasible solution is obtained. When all the possible choices have been evaluated, the master solution is updated with the best choice, and the procedure proceeds iteratively until the user termination conditions are reached. Further details are included in (Voß et al., 2004).

Cerulli et al. (2005) applied the Pilot Method to the MLST problem (see Chapter 3) and, following the same procedure, to the MLSteiner problem (Cerulli et al., 2006). They also performed a comparison between PILOT and other ad-hoc metaheuristics (Tabu Search, Simulated Annealing, and Variable Neighbourhood Search) for different instances of the MLSteiner problem (Cerulli et al., 2006). From their computational analysis, the Pilot Method obtained the best results.

The details of the Pilot Method proposed by Cerulli et al. (2006) for the MLSteiner problem are specified in Algorithm 4.2. PILOT starts from the null solution (an empty set of labels) as master solution, M. Then, for each element $i \notin M$, it tries to extend tentatively a copy of M to a (fully grown) feasible

Algorithm 4.2: The Pilot Method for the MLSteiner problem (Cerulli et al., 2006)

Input: A labelled, undirected, connected graph $G = (V, E, L)$, with n vertices, m edges, ℓ labels, $Q \subseteq V$ basic nodes;
Output: A tree T;
Initialization:
- Let $M \leftarrow 0$ be the initially empty master solution;
- Let $H = (V, E(M))$ be the subgraph of G restricted to V and edges with labels in M, where $E(M) = \{e \in E : L(e) \in M\}$;
- Let $Comp(M)$ be the number of Steiner components of $H = (V, E(M))$;
- Let $M^* \leftarrow L$ be a set of labels;
- Let $H^* = (V, E(M^*))$ be the subgraph of G restricted to V and edges with labels in M^*, where $E(M^*) = \{e \in E : L(e) \in M^*\}$;
- Let i^* be the best candidate move;

begin
  while (*not termination conditions*) *OR* $(Comp(M) > 1)$ **do**
    foreach $i \in (L - M)$ **do**
      Add label i to the master solution: $M \leftarrow M \cup \{i\}$;
      Update $H = (V, E(M))$ and $Comp(M)$;
      while $Comp(M) > 1$ **do**
        Let S be the set of unused labels which minimize the number of Steiner components, i.e. $S = \{e \in (L - M) : \min Comp(M \cup \{e\})\}$;
        Select at random a label $u \in S$;
        Add label u to the solution: $M \leftarrow M \cup \{u\}$;
        Update $H = (V, E(M))$ and $Comp(M)$;
      end
      Local search(M);
      if $|M| < |M^*|$ **then**
        Update the best candidate move $i^* \leftarrow i$;
        Keep the solution produced by the best move: $M^* \leftarrow M$;
      end
      Delete label i from the master solution: $M \leftarrow M - \{i\}$;
      Update $H = (V, E(M))$ and $Comp(M)$;
    end
    Update the master solution with the best move: $M \leftarrow M \cup \{i^*\}$;
  end
  while $Comp(M) > 1$ **do**
    Let S be the set of unused labels which minimize the number of Steiner components, i.e. $S = \{e \in (L - M) : \min Comp(M \cup \{e\})\}$;
    Select at random a label $u \in S$;
    Add label u to the solution: $M \leftarrow M \cup \{u\}$;
    Update $H = (V, E(M))$ and $Comp(M)$;
  end
  $\Rightarrow$ Take any arbitrary spanning tree T of $H = (V, E(M))$.
end

solution including i, built by the application process. The application process is a greedy procedure which, at each step, inserts in the partial solution the label producing the minimum number of Steiner components at that specific step, and stopping when a feasible solution is obtained. At the end of the execution of the

application process, a local search mechanism is included to try to greedily drop labels (i.e., the associated edges), from the least frequently occurring label to the most frequently occurring one, whilst retaining feasibility (see Algorithm 3.4). The number of labels produced by the feasible solution obtained from $M \leftarrow M \cup \{i\}$ is used as objective function for each candidate $i \notin M$. When all the possible candidate labels with respect to the master solution have been evaluated, a candidate i^* with minimum objective function value is added to the master solution ($M \leftarrow M \cup \{i^*\}$). On the basis of this new master solution M, new iterations of the Pilot Method are started $\forall i \notin M$, providing a new solution element i^*, and so on.

This mechanism is repeated for all the successive stages of the Pilot Method, until no further labels need to be added to the master solution (i.e., a feasible master solution is produced). Alternatively, some user termination conditions, such as the maximum allowed CPU time or the maximum number of iterations, may be imposed in order to allow the algorithm to proceed until these conditions are satisfied. The last master solution corresponds to the best solution to date and it is produced as the output of the method.

Note that, when the application process is applied to complete a partial solution, in case of ties in the minimum number of Steiner components, a label is selected at random within the set of labels producing the minimum number of components. Furthermore, note that no external parameters need to be tuned by the user for the Pilot Method.

4.3.3 Greedy Randomized Adaptive Search Procedure

GRASP (Greedy Randomized Adaptive Search Procedure) is an iterative metaheuristic consisting of two phases: a construction phase, followed by a local search phase (for a survey on GRASP see Section 2.2.3). The *construction* phase builds a feasible solution by applying a randomized greedy procedure. The randomized greedy procedure builds a solution by iteratively creating a candidate list of elements that can be added to the partial solution, and then randomly selecting an element from this list.

The candidate list (RCL_α: Restricted Candidate List of length α) is created by evaluating the elements not yet included in the partial solution. A greedy function, depending on the specifications of the problem, is used to perform this evaluation. Only the best elements, according to this greedy function, are included in RCL_α.

At each iteration one new element is randomly selected from RCL_α, added to the current solution, and the candidate list is updated. The construction phase stops when a feasible solution is obtained. The obtained solution is not necessarily locally optimal, so a *local search* phase is included to try to improve it. This two-phase process is iterative, continuing until the user termination condition such as the maximum allowed CPU time, the maximum number of iterations, or the maximum number of iterations between two successive improvements, is reached. The final result of GRASP is the best solution found to date.

The GRASP proposed for the MLSteiner problem takes inspiration from the GRASP proposed for the MLST problem (see Section 3.3.3). Its implementation is specified in Algorithm 4.3. For the construction phase of GRASP (*Construc-*

Algorithm 4.3: Greedy Randomized Adaptive Search Procedure for the MLSteiner problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels, $Q \subseteq V$ basic nodes;
Output: A tree T;
Initialization:
- Let $C \leftarrow 0$ be the initially empty set of used labels for each iteration;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $C' \leftarrow L$ be the global set of used labels;
- Let $H' = (V, E(C'))$ be the subgraph of G restricted to V and edges with labels in C', where $E(C') = \{e \in E : L(e) \in C'\}$;
- Let $Comp(C)$ be the number of Steiner components of C, i.e. the number of connected components of the subgraph $(Q, E(C))$;

```
begin
    repeat
        Set C ← 0 and update H = (V, E(C));
        Construction phase(C);
        Local search(C);
        if |C| < |C'| then
            Move C' ← C;
            Update H' = (V, E(C'));
        end
    until termination conditions ;
    ⇒ Take any arbitrary spanning tree T of H' = (V, E(C')).
end
```

tion phase() procedure, see Algorithm 4.4), a value-based restricted candidate list is used in order to select the labels to be placed in RCL_α. This is an extension of the classic greedy criterion used in GRASP, consisting of placing in the list only the candidate labels having a greedy value (the number of Steiner components in the case of the MLSteiner problem) not greater than a user-defined threshold (Resende and Ribeiro, 2003). In the proposed implementation, complete randomization is used to choose the initial label to add. This corresponds to setting the threshold to $+\infty$, meaning that the candidate list is filled with all the labels of the graph (length α = total number of labels). For the remaining labels to add, the list is formed by considering only the labels that result in the minimum number of Steiner components at the specific step, in order to further intensify the search process. This means fixing the threshold as the minimum number of Steiner components produced by the candidate labels at the specific step (i.e. only the labels producing the least number of Steiner components at that step constitute the candidate list).

Algorithm 4.4: Procedure *Construction phase(·)*

Procedure Construction phase(C):
Let $RCL_\alpha \leftarrow 0$ be the restricted candidate list of length α;
Set $RCL_\alpha \leftarrow L$ and $\alpha \leftarrow \ell$;
Select at random a label $c \in RCL_\alpha$;
Add label c to the set of used labels: $C \leftarrow C \cup \{c\}$;
Update $H = (V, E(C))$ and $Comp(C)$;
while $Comp(C) > 1$ **do**
    Set $RCL_\alpha \leftarrow \{c \in L | c$ minimizes $Comp(C \cup \{c\})\}$;
    Select at random a label $c \in RCL_\alpha$;
    Add label c to the set of used labels: $C \leftarrow C \cup \{c\}$;
    Update $H = (V, E(C))$ and $Comp(C)$;
end

At the end of the construction phase of GRASP, the successive local search phase (*Local search(C)* procedure, see Algorithm 3.7) consists of trying to greedily drop some labels (i.e. the associated edges) from the current solution, whilst retaining feasibility. It yields a further improvement to the intensification phase of the algorithm.

4.3.4 Discrete Particle Swarm Optimization

Over the years, evolutionary and nature-inspired algorithms have been widely used as robust techniques for solving hard combinatorial optimization problems. Their behaviour is directed by the evolution of a population searching for the optimum. Particle Swarm Optimization (PSO) is a a population-based metaheuristic proposed by Kennedy and Eberhart (1995). As is the case with Genetic Algorithms, PSO is an evolutionary algorithm, inspired by the social behaviour of individuals (or particles) inside swarms occurring in nature, such as flocks of birds or schools of fish. Being inspired by the principles of natural evolution, Particle Swarm Optimization is also a main representative of the class of nature-inspired algorithms. Unlike classic evolutionary approaches as Genetic Algorithms, it has no crossover and mutation operators, is easy to implement, and requires few parameter settings and low computational memory. For a survey on PSO, the reader is referred to Section 2.3.6.

The standard PSO (Kennedy and Eberhart, 2001) considers a swarm SW containing n_{sw} particles ($SW = 1, 2, \ldots, n_{sw}$) in a d-dimensional continuous solution space. Each i-th particle of the swarm has a position $x_i = (x_{i1}, x_{i2}, \ldots, x_{ij}, \ldots, x_{id})$ associated with it, and a velocity $v_i = (v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots, v_{id})$. The position x_i represents a solution for the problem, while the velocity v_i gives the change rate for the position of particle i in the next iteration. Indeed, considering an iteration k, the position of particle i is adjusted according to

$$x_i^k = x_i^{k-1} + v_i^k. \tag{4.1}$$

Each particle i of the swarm communicates with a social environment or neighbourhood $N(i) \subseteq SW$, which may change dynamically and represents the group of particles with which particle i communicates. In nature, a bird adjusts its position in order to find a better position, according to its own experience and the experience of its companions. In the same manner, consider an iteration k of the PSO algorithm. Each particle i updates its velocity reflecting the attraction of its best position so far (b_i) and the best position (g_i) of its social neighbourhood $N(i)$, following the equation:

$$v_i^k = c_1 \xi v_i^{k-1} + c_2 \xi (b_i - x_i^{k-1}) + c_3 \xi (g_i - x_i^{k-1}). \tag{4.2}$$

The parameters c_i are positive constant weights applied to the three factors that influence the velocity of the particle i, while the term ξ refers to a random number with uniform distribution in $[0, 1)$ that is independently generated at each iteration.

Since the original PSO is applicable to optimization problems with continuous variables, several adaptations of the method to discrete problems, known as Discrete Particle Swarm Optimization (DPSO), have been proposed (Kennedy and Eberhart, 1997). In this section the DPSO procedure introduced by Moreno-Pérez et al. (2007) is used. This DPSO considers a swarm SW containing n_{sw} particles ($SW = 1, 2, \ldots, n_{sw}$) whose positions x_i evolve in the discrete solution space, jumping from a solution to another. In such a case, the notion of velocity used in the standard PSO loses its meaning, and is not considered. Furthermore, the weights of the updating equation used in the standard PSO are interpreted as probabilities that, at each iteration, each particle has a random behaviour, or acts in a manner guided by the effect of attractors. The effect of the attraction of a position causes the given particle to jump towards this attractor. An inspiration from nature for this process is found in frogs, which jump from lily pad to lily pad in a pool.

Given a particle i, three attractors are considered: its own best position (b_i), the best position of its social neighbourhood (g_i), and the global best position (g^*). Indeed, considering a generic iteration k, the update equation for the position x_i of a particle i is:

$$x_i^k = c_1 x_i^{k-1} \oplus c_2 b_i \oplus c_3 g_i \oplus c_4 g^*. \tag{4.3}$$

The meaning of this equation is that, at the k-th iteration, the i-th particle with position x_i performs random jumps with respect to its current position with probability c_1, improving jumps approaching b_i with probability c_2, improving jumps approaching g_i with probability c_3, and improving jumps approaching g^* with probability c_4. Note that exactly one type of jump is performed at each iteration. In order to implement this operation, a random number ξ is generated in order to select the type of jump to be chosen. A jump approaching an attractor consists of modifying a feature of the current solution with the corresponding feature of the selected attractor (or giving an arbitrary value in the case of the

random jump). For the MLSteiner problem the features of a solution are the labels that are included in the solution, while the parameters c_1, c_2, c_3, and c_4, are set to 0.25.

Further details of the DPSO proposed for the MLSteiner problem are specified in Algorithm 4.5. The position of a particle in the swarm is encoded as a feasible

Algorithm 4.5: Discrete Particle Swarm Optimization for the MLSteiner problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$ with n vertices, m edges, ℓ labels, $Q \subseteq V$ basic nodes;
Output: A tree T;
Initialization:
- Let $C \leftarrow 0$ be a set of labels, initially empty;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Set the size n_{sw} of the swarm SW;
begin
  Generate the initial swarm SW with positions at random:
  $X = [x_1, x_2, \ldots, x_{n_{sw}}] \leftarrow$*Generate-Swarm-At-Random(G)*;
  Update the vector of the best positions $B = [b_1, b_2, \ldots, b_{n_{sw}}] \leftarrow X$;
  Extract the best position among all the particles: $g^* \leftarrow$ *Extract-the-Best(SW, X)*;
  repeat
    for $i \leftarrow 1$ *to* n_{sw} **do**
      if $i \leftarrow 1$ **then**
        Initialize the best position of the social neighbourhood of i: $g_i \leftarrow \ell$;
      else
        Update the best position of the social neighbourhood of i: $g_i \leftarrow g_{i-1}$;
      end
      Select at random a number between 0 and 1: $\xi \leftarrow random[0, 1)$;
      if $\xi \in [0, 0.25)$ **then** $selected \leftarrow x_i$;
      else if $\xi \in [0.25, 0.5)$ **then** $selected \leftarrow b_i$;
      else if $\xi \in [0.5, 0.75)$ **then** $selected \leftarrow g_i$;
      else if $\xi \in [0.75, 1)$ **then** $selected \leftarrow g^*$;
      Combine the given particle i and the selected particle: $x_i \leftarrow$ *Combine*(x_i, *selected*);
      Local search(i, x_i);
      if $|x_i| < |b_i|$ **then**
        Update the best position of the given particle i: $b_i \leftarrow x_i$;
      end
      if $|x_i| < |g_i|$ **then**
        Update the best position of the social neighbourhood of i: $g_i \leftarrow x_i$;
      end
      if $|x_i| < |g^*|$ **then**
        Update the global best position to date: $g^* \leftarrow x_i$;
      end
    end
  until *termination conditions* ;
  Set $C \leftarrow g^*$;
  Update $H = (V, E(C))$;
  $\Rightarrow$ Take any arbitrary spanning tree T of $H = (V, E(C))$.
end

solution to the MLSteiner problem. The initial positions $X = [x_1, x_2, \ldots, x_{n_{sw}}]$ of the swarm SW, containing n_{sw} particles, are generated by starting from empty sets of labels and adding, at random, labels until feasible solutions emerge. The position x_i of a particle i is a 0-1 vector denoting which labels are present in particle i. Then, for each particle of the swarm, a random number ξ between 0 and 1 is selected. Considering the i-th particle of the swarm, if ξ belongs to $[0, 0.25)$ the current position of the given particle is selected ($selected \leftarrow x_i$) in order to perform a random jump. Otherwise, if ξ is in $[0.25, 0.5)$ the best position to date (b_i) of the given particle is selected ($selected \leftarrow best\_s(p)$) as attractor for the movement of x_i. Instead, if $\xi \in [0.5, 0.75)$ the selected attractor is the best position g_i of the social neighbourhood, interpreted as the best position obtained within the swarm in the current iteration. For the remaining case, if $\xi \in [0.75, 1)$ the selected attractor is the best position to date obtained by all the particles, which is called the global best position to date (g^*).

Algorithm 4.6: Procedure *Combine(x_i, selected)*

```
Procedure Combine(x_i, selected):
Select a random integer between 0 and |x_i|: ψ ← Random(0, |x_i|);
for j ← 1 to ψ do
    Select at random a number between 0 and 1: ξ ←Random(0, 1);
    if ξ ≤ 0.5 then
        Select at random a label c' ∈ x_i;
        Delete label c' from the the position of the given particle: x_i ← x_i − {c'};
    else
        Select at random a label c' ∈ selected;
        Add label c' to the position of the given particle i: x_i ← x_i ∪ {c'};
    end
end
while Comp(x_i) > 1 do
    Select at random an unused label u ∈ (L − x_i);
    Add label u to the position of the given particle i: x_i ← x_i ∪ {u};
end
```

Afterwards, the i-th particle with current position x_i performs a jump approaching the selected attractor by means of the procedure *Combine* (Algorithm 4.6). This procedure first selects a random integer ψ between 0 and $|x_i|$. Successively, it either drops some labels from x_i, or randomly picks up some labels from the selected attractor and adds to x_i, until ψ labels have been added or deleted with respect to x_i. Note that if an infeasible x_i is obtained at this

stage, further labels are added at random to x_i in order to restore feasibility. At the end of the procedure *Combine*, a local search procedure is applied to the resulting particle (*Local-Search(i, x_i)*), in order to try to delete some labels from x_i whilst retaining the feasibility. Then all the attractors (b_i, g_i, g^*) are updated, and the same procedure is repeated for all the particles in the swarm. The entire algorithm continues until the user termination conditions are satisfied.

4.3.5 Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is an effective metaheuristic introduced by Hansen and Mladenović (1997) (for a survey on VNS see Section 2.2.5). The basic idea behind this method is to define a neighbourhood structure for the solution space, and to explore different increasingly distant neighbourhoods whenever a local optimum is reached by a prescribed local search.

At the starting point, a set of k_{max} (a parameter) neighbourhoods (N_k, with $k = 1, 2, ..., k_{max}$), is selected. A stopping condition is determined (either the maximum allowed CPU time, or the maximum number of iterations, or the maximum number of iterations between two successive improvements), and an initial feasible solution found (at random, in this case). Denoting by $N_k(C)$ the set of solutions in the k-th neighbourhood of the solution C, the simplest and most common choice is a structure in which the neighbourhoods have increasing cardinality: $|N_1(C)| < |N_2(C)| < ... < |N_{k_{max}}(C)|$. The process of changing neighbourhoods when no improvement occurs diversifies the search. In particular the choice of neighbourhoods of increasing cardinality yields a progressive diversification.

Although a VNS for the MLSteiner was implemented by Cerulli et al. (2006), our implementation is motivated by the successful VNS proposed for the MLST problem in Chapter 3. The two approaches mainly differ in the implementation of the neighbourhood structures, in the way the initial solution is obtained, and in the maximum size of the shaking phase k_{max}, among others. The VNS by Cerulli et al. (2006) uses three different neighbourhood structures (*k - Switch Neighbourhood, k - Covering Neighbourhood, k - Mixed Neighbourhood*, see (Cerulli et al., 2006) for more details), in order to check whether one neighbourhood is better

than another. For each neighbourhood, the procedure starts from an initial feasible solution provided by a greedy algorithm, and then tries to find an improved solution by selecting one of the neighbourhoods considered. After a specified number of iterations, another neighbourhood is chosen to be explored in subsequent iterations. For each neighbourhood, the parameter k_{max} varies during the execution, determined by $k_{max} \leftarrow min(|C|, \frac{|L|}{4})$, where C is the current feasible solution and L is the set of labels. In contrast, our VNS implementation for the MLSteiner is specified in Algorithm 4.7.

Before going into detail, consider the following notation. Given a labelled graph $G = (V, E, L)$, with n vertices, m edges, ℓ labels, and $Q \subseteq V$ basic nodes,

Algorithm 4.7: Variable Neighbourhood Search for the MLSteiner problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$, with n vertices, m edges, ℓ labels, $Q \subseteq V$ basic nodes;
Output: A tree T;
Initialization:
- Let $C \leftarrow 0$ be the global set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let C' be a set of labels;
- Let $H' = (V, E(C'))$ be the subgraph of G restricted to V and edges with labels in C', where $E(C') = \{e \in E : L(e) \in C'\}$;
- Let $Comp(C')$ be the number of Steiner components of C', i.e. the number of connected components of the subgraph $(Q, E(C'))$;

```
begin
    C ← Generate-Initial-Solution-At-Random();
    repeat
        Set k ← 1 and k_max ← (|C| + |C|/3);
        while k < k_max do
            C' ← Shaking phase(N_k(C));
            Local search(C');
            if |C'| < |C| then
                Move C ← C';
                Restart with the first neighbour: k ← 1;
            else
                Increase the size of the neighbourhood structure: k ← k + 1;
            end
        end
    until termination conditions ;
    Update H = (V, E(C));
    ⇒ Take any arbitrary spanning tree T of H = (V, E(C)).
end
```

each solution is encoded by a binary string, i.e. $C = (c_1, c_2, ..., c_\ell)$ where

$$c_i = \begin{cases} 1 & \text{if label } i \text{ is in solution } C \\ 0 & \text{otherwise} \end{cases} \qquad (\forall i = 1, \ldots, \ell). \tag{4.4}$$

Now, define the solution space, S, as the set of all the possible solutions, and let

$$\rho(C_1, C_2) = |C_1 - C_2| = \sum_{i=1}^{\ell} \lambda_i \tag{4.5}$$

be the Hamming distance between any two solutions C_1 and C_2, where $\lambda_i = 1$ if label i is included in one of the solutions but not in the other, and 0 otherwise, $\forall i = 1, ..., \ell$. The k-th neighbourhood induced by (S, ρ), of a given solution C, may be defined as

$$N_k(C) = \{S \subset L : (\rho(C, S)) = k\} \qquad (\forall k = 1, ..., k_{max}). \tag{4.6}$$

The parameter k_{max} represents the size of the neighbourhood structure and, according to our experience, the value $k_{max} \leftarrow (|C| + |C|/3)$ is the best choice for the MLSteiner problem.

Looking at Algorithm 4.7, after defining the neighbourhood structure and obtaining the initial random solution C, the algorithm applies a shaking phase (*Shaking phase($N_k(C)$)* procedure), letting parameter k vary throughout the execution. The shaking phase (see Algorithm 3.9) consists of the random selection of a solution C' in the neighbourhood $N_k(C)$ of the current solution C, with the intention of providing a better starting point for the successive local search phase.

In order to select a solution in the k-th neighbourhood of a solution C, the algorithm randomly adds further labels to C, or removes labels from C, until the resulting solution has a Hamming distance equal to k with respect to C. Addition and deletion of labels at this stage have the same probability of being chosen. For this purpose, a random number is selected between 0 and 1 ($rnd \leftarrow random[0, 1]$). If this number is smaller than 0.5, the algorithm proceeds with the deletion of a label from C. Otherwise, an additional label is included at random in C from the set of unused labels $(L - C)$. The procedure is repeated until the number of addition/deletion operations is exactly equal to k.

The shaking phase represents the core idea of VNS, that of changing the neighbourhood structure when the local search is trapped at a local minimum.

The successive local search (*Local search(C′)* procedure) is the same local search for the VNS implementation used for the MLST problem (see Algorithm 3.10). It basically consists of two steps. In the first step, since deletion of labels often gives an infeasible incomplete solution, additional labels may be added in order to restore feasibility. In this case, addition of labels follows the MVCA criterion of adding the label with the minimum number of connected components. Note that in case of ties in the minimum number of connected components, a label not yet included in the partial solution is chosen at random within the set of labels producing the minimum number of components (i.e. $u \in S$ where $S = \{e \in (L - C') : \min Comp(C' \cup \{e\})\}$). Then, the second step of the local search tries to delete labels one by one from the specific solution, whilst maintaining feasibility.

After the local search phase, if no improvements are obtained ($|C'| \geq |C|$), the neighbourhood is increased ($k \leftarrow k + 1$), resulting in a higher diversification of the search process. Otherwise, if $|C'| < |C|$, the algorithm moves to the improved solution ($C \leftarrow C'$), restarting the search with the smallest neighbourhood ($k \leftarrow 1$). The algorithm proceeds until the established stopping conditions are reached.

4.3.6 Hybrid local search

Although hybridizing a metaheuristic may increase the complexity of the implementation, a more advanced VNS version is considered for the MLSteiner problem, with a view to obtaining improved results. For this purpose, a hybrid local search method (HYBRID) is used, in order to improve the diversification of the search process. The motivation for introducing a high diversification capability is to obtain a better performance in large problem instances. HYBRID is a variant of the hybrid local search method proposed for the MLST problem in Chapter 3, that is a hybridization between Variable Neighbourhood Search and Simulated Annealing. The details of HYBRID are specified in Algorithm 4.8.

The algorithm starts from an initial feasible solution ($Best_C$) generated at random. Then the *Complementary Local Search*, already introduced for the MLST problem in Section 3.3.5, is applied (*Complementary(·)* procedure, see Algorithm 3.12). It consists of extracting a solution from the *complementary*

Algorithm 4.8: Hybrid local search method for the MLSteiner problem

Input: A labelled, undirected, connected graph $G = (V, E, L)$, with n vertices, m edges, ℓ labels, $Q \subseteq V$ basic nodes;
Output: A tree T;
Initialization:
- Let $Best_C \leftarrow 0$ be the global set of labels;
- Let $H^{BEST} = (V, E(Best_C))$ be the subgraph of G restricted to V and edges with labels in $Best_C$, where $E(Best_C) = \{e \in E : L(e) \in Best_C\}$;
- Let $C \leftarrow 0$ be the set of used labels;
- Let $H = (V, E(C))$ be the subgraph of G restricted to V and edges with labels in C, where $E(C) = \{e \in E : L(e) \in C\}$;
- Let $Comp(C)$ be the number of Steiner components of C, i.e. the number of connected components of the subgraph $(Q, E(C))$;
- Let C' be a set of labels;
- Let $H' = (V, E(C'))$ be the subgraph of G restricted to V and edges with labels in C', where $E(C') = \{e \in E : L(e) \in C'\}$;
- Let $Comp(C')$ be the number of Steiner components of C', i.e. the number of connected components of the subgraph $(V, E(C'))$;
- Let $Compl\_Space = (L - Best_C)$ the complementary space of the best solution $Best_C$;

```
begin
    Best_C ←Generate-Initial-Solution-At-Random();
    Local search(Best_C);
    repeat
        Extract a solution from the complementary space of Best_C: C ←Complementary(Best_C);
        while |C| < |Best_C| AND (C is a feasible solution) do
            Move Best_C ← C;
            Extract another complementary solution: C ←Complementary(Best_C);
        end
        Set k ← 1 and k_max ← |C| + |C|/3;
        while k < k_max do
            C' ←Shaking phase(N_k(C));
            Local search(C');
            if |C'| < |C| then
                Move C ← C';
                Restart with the first neighbour: k ← 1;
            else
                Increase the size of the neighbourhood structure: k ← k + 1;
            end
        end
        if |C| < |Best_C| then
            Move Best_C ← C;
        end
    until termination conditions ;
    Update H^BEST = (V, E(Best_C));
    ⇒ Take any arbitrary spanning tree T of H^BEST = (V, E(Best_C)).
end
```

space of the current solution. Given the solution $Best_C$, its complementary space ($Compl\_Space$) is defined as the set of all the labels that are not contained in $Best_C$, that is $(L - Best_C)$.

To yield the solution, Complementary Local Search applies a constructive heuristic to the subgraph of G with labels in $(Compl\_Space)$. In the proposed implementation, the *Probabilistic MVCA* heuristic, already introduced in Section 3.3.5 for the MLST problem, is used. The Probabilistic MVCA uses an idea similar to the basic one of the Simulated Annealing metaheuristic (Aarts et al., 2005): the introduction of probabilities for the choice of the next labels to add to incomplete solutions. Thus, it further improves the diversification of the search process because it allows the possibility of adding worse components at each iteration. The introduction of this probabilistic element makes HYBRID a hybridization between VNS and Simulated Annealing.

The Probabilistic MVCA begins from an initial solution, and successively selects a candidate move at random. This move is accepted if it leads to a solution with a better objective function value than the current solution, otherwise the move is accepted with a probability that depends on the deterioration Δ of the objective function value. Consider a label x. The deterioration Δ of the objective function value is $(Comp(x) - Comp_{min})$, where $Comp(x)$ represents the number of Steiner components obtained by inserting x in the partial solution, and $Comp_{min}$ is the minimum number of Steiner components at the specific step. Thus, following the criteria of Simulated Annealing, the acceptance probability is computed according to the Boltzmann function as $\exp(-\Delta/T)$, using a temperature T as control parameter (Kirkpatrick et al., 1983). Probability values assigned to each label are inversely proportional to the number of Steiner components they give. The labels with a lower number of Steiner components will have a higher probability of being chosen. Conversely, labels with a higher number of Steiner components will have a lower probability of being chosen. Thus the possibility of choosing less promising labels to be added to incomplete solutions is allowed.

The value of the parameter T is initially high, which allows many worse moves to be accepted, and is gradually reduced following a geometric cooling schedule:

$$T^{Complementary}_{(|C|+1)} \leftarrow \frac{T^{Complementary}_{(|C|)}}{\alpha} \leftarrow \frac{T^{Complementary}_{(0)}}{\alpha^{|C|}}, \tag{4.7}$$

where experimentally it was found that the values $T^{Complementary}(0) \leftarrow |Best_C|$ and $\alpha \leftarrow 1/|Best_C|$ produce good results. This cooling schedule is very fast for

the MLSteiner problem, yielding a good balance between intensification and diversification. At each step, the probabilities of selecting labels giving a smaller number of Steiner components will be higher than the probabilities of selecting labels with a higher number of Steiner components. Furthermore, these differences in probabilities increase step by step as a result of the reduction of the temperature given by the cooling schedule. It means that the difference between the probabilities of two labels giving different numbers of Steiner components is higher as the algorithm proceeds.

The Complementary procedure stops if either a feasible solution C is obtained, or the set of unused colours contained in the complementary space is empty (i.e. $(Compl\_Space - C) = 0$), producing a final infeasible solution. After the Complementary procedure, a shaking phase similar to the one used for the basic VNS is applied to the resulting solution, denoted by C (see Algorithm 3.9). It consists of the random selection of a point C' in the neighbourhood $N_k(C)$ of the current solution C. For the proposed implementation, given a solution C, its k-th neighbourhood $N_k(C)$ is considered as all the different sets of labels that are possible to obtain from C by randomly adding further labels to C, or by removing labels from C, until the resulting solution has a Hamming distance equal to k with respect to C, where $k = 1, 2, ..., k_{max}$. In a more formal way, the k-th neighbourhood of a solution C is defined as $N_k(C) = \{S \subset L : (\rho(C, S)) = k\}$, where $k = 1, 2, ..., k_{max}$. Computational experience indicates that the value $k_{max} \leftarrow (|C| + |C|/3)$ gives a good trade-off between intensification and diversification of the search process. Addition and deletion of labels at this stage have the same probability of being chosen. For this purpose, a random number is selected between 0 and 1 ($rnd \leftarrow random[0, 1]$). If this number is smaller than 0.5, the algorithm proceeds with the deletion of a label from C. Otherwise, an additional label is included at random in C from the set of unused labels $(L - C)$. The procedure is repeated until the number of addition/deletion operations is exactly equal to k.

Since either the Complementary procedure, or the deletion of labels in the shaking phase, can produce an infeasible solution, additional labels may be added in order to restore feasibility in the first step of the successive local search (*Local search(C′)* procedure, see Algorithm 3.13). Addition of labels at this step is according to the Probabilistic MVCA heuristic, as in the Complementary Local

Search. For the geometric schedule in the local search, computational experiments have shown that $T_0 \leftarrow |Best_C|^2$ and $\alpha \leftarrow 1/|Best_C|$, where $Best_C$ is the current best solution, are values that performed well. The corresponding geometric cooling law is

$$T^{Local\,search}_{(|C'|+1)} \leftarrow \frac{T^{Local\,search}_{(0)}}{\alpha^{|C'|}} \leftarrow \frac{1}{|Best_C|^{(|C'|-2)}}. \tag{4.8}$$

Afterwards, the second step of the local search tries to delete labels one by one from the specific solution, whilst maintaining feasibility.

At this stage, if no improvements are obtained, i.e. if $|C'| \geq |C|$, the neighbourhood structure is increased ($k \leftarrow k+1$), yielding a progressive diversification ($|N_1(C)| < |N_2(C)| < ... < |N_{k_{max}}(C)|$). Otherwise, the algorithm moves to the solution C' restarting the search with the smallest neighbourhood ($k \leftarrow 1$). After the entire shaking phase, the Complementary procedure is applied again to the actual best solution ($Best_C$) and the algorithm continues iteratively with the same procedure until the user termination conditions are satisfied.

4.4 Computational results

To test the performance and the efficiency of the algorithms presented in this section, several instances of the MLSteiner problem have been randomly generated with respect to the number of nodes (n), the density of the graph (d), the number of labels (ℓ), and the number of basic nodes (q). In the considered experiments, 48 different datasets have been computed, each one containing 10 instances of the problem (yielding a total of 480 instances), with $n = 100$, 500 nodes, $\ell = 0.25 \cdot n$, $0.5 \cdot n$, n, $1.25 \cdot n$ labels, and $q = 0.2 \cdot n$, $0.4 \cdot n$ basic nodes. The number of edges, m, is obtained indirectly from the density d, whose values are chosen to be 0.8, 0.5, and 0.2. The complexity of the instances increases with the dimension of the graph (number of nodes, number of basic nodes, and number of labels), and the reduction in the density of the graph. All the data considered are available from the author in (Consoli, 2007b).

For each dataset, solution quality is evaluated as the average objective function value among the 10 problem instances. A maximum allowed CPU time,

called *max-CPU-time*, is chosen as the stopping condition for all the metaheuristics, determined experimentally with respect to the dimension of the problem instance. For the Discrete Particle Swarm Optimization, a swarm of 100 particles is considered and a variable number of iterations for each instance is used, determined such that the computations take approximately *max-CPU-time* for the specific dataset. Selection of the maximum allowed CPU time as the stopping criterion is made in order to have a direct comparison of the metaheuristics with respect to the quality of their solutions.

Computational experiments are reported in Tables 4.1 - 4.4. All the computations have been made on a Pentium Centrino microprocessor at 2.0 GHz with 512 MB RAM. In each table, the first three columns show the parameters characterizing the different datasets (n, ℓ, d), while the values of q determine the different tables. The remaining columns give the computational results of the algorithms considered, identified with the abbreviations: EXACT (Exact Method), PILOT (Pilot Method), GRASP (Greedy Randomized Adaptive Search Procedure), DPSO (Discrete Particle Swarm Optimization), VNS (Variable Neighbourhood Search), HYBRID (hybrid local search method). All the algorithms have been implemented using the C++ programming language (Microsoft Visual C++ 2005).

All the metaheuristics run for the *max-CPU-time* specified in each table and, in each case, the best solution is recorded. The computational times reported in the tables are the average times at which the best solutions are obtained. For the Exact Method, a time limit of 3 hours is used. If an exact solution is not found within this time limit for any instance within a dataset, a not found status (NF) is reported. All the reported times have precision of ± 5 ms. It is interesting to note that in all the problem instances for which the Exact Method obtains the solution, also VNS, HYBRID, and DPSO yielded the exact solution.

For each dataset in the tables, the performance of an algorithm is considered *better* than another one if either it obtains a smaller average objective function value, or an equal average objective function value but in a shorter computational running time. Thus, according to this evaluation, the algorithms are ranked for each dataset, assigning a rank of 1 to the best performing algorithm, rank 2 to

Table 4.1: Computational results for $n = 100$ and $q = 0.2 \cdot n$ (*max-CPU-time* for heuristics = 5000 ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
		0.8	*1*	1	1	1	1	1
	25	0.5	*1.5*	1.5	1.5	1.5	1.5	1.5
		0.2	*2.1*	2.1	2.1	2.1	2.1	2.1
		0.8	*1.9*	1.9	1.9	1.9	1.9	1.9
	50	0.5	*2*	2	2	2	2	2
100		0.2	*3.2*	3.2	3.2	3.2	3.2	3.2
		0.8	*2*	2	2	2	2	2
	100	0.5	*3*	3	3	3	3	3
		0.2	*4.6*	4.6	4.6	4.6	4.6	4.6
		0.8	*2.8*	2.8	2.8	2.8	2.8	2.8
	125	0.5	*3.3*	3.3	3.3	3.3	3.3	3.3
		0.2	*5.2*	5.4	5.3	5.2	5.2	5.2
	TOTAL:		*32.6*	32.8	32.7	32.6	32.6	32.6
Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
		0.8	*14.7*	14.1	6.7	1.6	1.5	1.5
	25	0.5	*26.3*	20.3	6.3	3.2	4.7	4.8
		0.2	*16.2*	15.6	4.7	6.1	4.6	6.2
		0.8	*59.4*	56.1	9.4	6.4	1.6	7.9
	50	0.5	*66.3*	67.2	6.1	10.9	4.7	7.8
100		0.2	*40.6*	75.1	15.6	15.7	1.5	9.5
		0.8	*306.3*	270.3	40.6	75.1	28.2	43.8
	100	0.5	*251.6*	275.1	7.6	31.2	7.3	12.6
		0.2	*0.9*$*10^3$	314.1	32.8	45.3	32.9	40.4
		0.8	*78.2*	381.2	14.1	48.4	15.3	32.8
	125	0.5	*451.5*	443.9	93.8	157.7	96.9	218.8
		0.2	*4.7*$*10^3$	518.8	68.8	322	136	162.4
	TOTAL:		*6.9*$*10^3$	2.5$*10^3$	306.5	723.6	335.2	548.5

the second best one, and so on. Obviously, if the Exact Method records a NF for a dataset, the worst rank is assigned to it in the specified dataset.

The average ranks of the algorithms, among the datasets considered, are (from the best one to the worst one with respect to the average ranks): EXACT = 5.49, PILOT = 5.21, GRASP = 2.56, DPSO = 3.88, VNS = 1.38, HYBRID =

Table 4.2: Computational results for $n = 100$ and $q = 0.4 \cdot n$ (*max-CPU-time* for heuristics = 6000 ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
100	25	0.8	*1*	1	1	1	1	1
		0.5	*1.9*	1.9	1.9	1.9	1.9	1.9
		0.2	*3*	3	3	3	3	3
	50	0.8	*2*	2	2	2	2	2
		0.5	*2.2*	2.2	2.2	2.2	2.2	2.2
		0.2	*4.3*	4.4	4.3	4.3	4.3	4.3
	100	0.8	*3*	3	3	3	3	3
		0.5	*3.6*	3.6	3.6	3.6	3.6	3.6
		0.2	*NF*	6.5	6.4	6.4	6.4	6.4
	125	0.8	*3*	3	3	3	3	3
		0.5	*4*	4	4	4	4	4
		0.2	*NF*	7	6.9	6.9	6.9	6.9
TOTAL:			-	41.6	41.3	41.3	41.3	41.3

Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
100	25	0.8	*24.7*	15.6	6.3	9.3	1.6	4.6
		0.5	*29.7*	21.7	6.4	6.4	1.6	1.5
		0.2	*36.9*	29.8	3.2	23.6	3	9.3
	50	0.8	*60.9*	53	7.2	20.4	3.1	7.9
		0.5	*117.2*	76.6	15.1	34.3	17.2	23.4
		0.2	*314.1*	111	34.4	45.1	28.1	29.7
	100	0.8	*175*	260.9	10.9	39.2	9.4	17.4
		0.5	*389.1*	312.5	38.4	96.8	32.3	39.7
		0.2	*NF*	472	79.8	350	79.7	99.9
	125	0.8	*354.6*	440.7	18.7	57.6	23.4	20.3
		0.5	*479.6*	507.8	73.4	67.1	60.9	70.4
		0.2	*NF*	811	177.8	411	191.7	197
TOTAL:			-	$3.1{*}10^3$	471.6	$1.2{*}10^3$	459.8	521.1

2.48. According to the ranking, VNS is the best performing algorithm, followed respectively by HYBRID, GRASP, DPSO, PILOT, and finally EXACT. The motivation to introduce a high diversification capability in HYBRID is to obtain a better performance in large problem instances. Inspection of Table 4.4 shows that this aim is achieved.

Table 4.3: Computational results for $n = 500$ and $q = 0.2 \cdot n$ (*max-CPU-time* for heuristics = $500{*}10^3$ ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
		0.8	*1.1*	1.1	1.1	1.1	1.1	1.1
	125	0.5	*2*	2	2	2	2	2
		0.2	*3*	3	3	3	3	3
		0.8	*2*	2	2	2	2	2
	250	0.5	*2.9*	2.9	2.9	2.9	2.9	2.9
500		0.2	*NF*	4.4	4.3	4.3	4.3	4.3
		0.8	*3*	3	3	3	3	3
	500	0.5	*NF*	3.9	3.9	4	3.9	3.9
		0.2	*NF*	6.8	6.8	6.9	6.7	6.7
		0.8	*NF*	3.8	3.8	3.8	3.8	3.8
	625	0.5	*NF*	4.8	4.8	4.8	4.7	4.7
		0.2	*NF*	8	8	7.9	7.9	8
	TOTAL:		-	45.7	45.6	45.7	45.3	45.4

Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
		0.8	$1.5{*}10^3$	$1.2{*}10^3$	173.4	$3.4{*}10^3$	172.2	404.7
	125	0.5	$2.1{*}10^3$	$2.5{*}10^3$	149.8	575	26.5	104.8
		0.2	$4.1{*}10^3$	$7.1{*}10^3$	318.8	$5.9{*}10^3$	265.7	634.4
		0.8	$13.6{*}10^3$	$17.4{*}10^3$	270	$9.7{*}10^3$	115.6	859.4
	250	0.5	$37.3{*}10^3$	$46.8{*}10^3$	334.6	$8.8{*}10^3$	148.4	301.6
500		0.2	*NF*	$48.1{*}10^3$	$14.5{*}10^3$	$36.7{*}10^3$	$11.9{*}10^3$	$17{*}10^3$
		0.8	$300.8{*}10^3$	$304.4{*}10^3$	$2.3{*}10^3$	$22.1{*}10^3$	$1.8{*}10^3$	$1.9{*}10^3$
	500	0.5	*NF*	$325.8{*}10^3$	$109.7{*}10^3$	$106.5{*}10^3$	$85.7{*}10^3$	$388.6{*}10^3$
		0.2	*NF*	$425.2{*}10^3$	$17.9{*}10^3$	$170.4{*}10^3$	$27.7{*}10^3$	$29{*}10^3$
		0.8	*NF*	$465.6{*}10^3$	$36.9{*}10^3$	$180.2{*}10^3$	$32.8{*}10^3$	$51.9{*}10^3$
	625	0.5	*NF*	$403{*}10^3$	$2.5{*}10^3$	$110.4{*}10^3$	$6.7{*}10^3$	$9.4{*}10^3$
		0.2	*NF*	$399.3{*}10^3$	$36.7{*}10^3$	$285.7{*}10^3$	$79.5{*}10^3$	$36.2{*}10^3$
	TOTAL:		-	$2446.4{*}10^3$	$221.8{*}10^3$	$940.4{*}10^3$	$246.8{*}10^3$	$536.3{*}10^3$

To analyse the statistical significance of differences between these ranks, the same procedure was followed as that for the MLST problem in Section 3.4, which makes use of the *Friedman test* (Friedman, 1940) and its corresponding *Nemenyi post-hoc test* (Nemenyi, 1963), is applied. In particular, the version of the Friedman test developed by Iman and Davenport (1980) is used, which considers a

Table 4.4: Computational results for $n = 500$ and $q = 0.4 \cdot n$ (*max-CPU-time* for heuristics = $600{*}10^3$ ms)

Parameters			**Average objective function values**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
		0.8	*1.9*	1.9	1.9	1.9	1.9	1.9
	125	0.5	*2*	2	2	2	2	2
		0.2	*NF*	4.1	4.1	4.1	4.1	4.1
		0.8	*2*	2	2	2	2	2
	250	0.5	*3*	3	3	3	3	3
500		0.2	*NF*	6.2	6.1	6.3	6.1	6.1
		0.8	*NF*	3.7	3.7	3.7	3.7	3.7
	500	0.5	*NF*	5	5	5	5	5
		0.2	*NF*	9.9	9.9	9.9	9.8	9.8
		0.8	*NF*	4	4	4	4	4
	625	0.5	*NF*	5.8	5.8	5.7	5.7	5.7
		0.2	*NF*	11.5	11.5	11.4	11.2	11.3
	TOTAL:		-	59.1	59	59	58.5	58.6

Parameters			**Computational times (milliseconds)**					
n	ℓ	d	EXACT	PILOT	GRASP	DPSO	VNS	HYBRID
		0.8	*218.8*	$1.1{*}10^3$	231	778.2	187.5	93.9
	125	0.5	*2.8*$*10^3$	$2.6{*}10^3$	230	$4.3{*}10^3$	184.2	218.7
		0.2	*NF*	$8.3{*}10^3$	$1.1{*}10^3$	$8.8{*}10^3$	853	$3.3{*}10^3$
		0.8	*44.6*$*10^3$	$20.2{*}10^3$	615.7	$12.5{*}10^3$	393.7	$1.2{*}10^3$
	250	0.5	*48.8*$*10^3$	$49.8{*}10^3$	864.2	$13.4{*}10^3$	650	$3.1{*}10^3$
500		0.2	*NF*	$48.7{*}10^3$	$20.4{*}10^3$	$122.2{*}10^3$	$38.1{*}10^3$	$24.8{*}10^3$
		0.8	*NF*	$201.1{*}10^3$	$13.1{*}10^3$	$19.4{*}10^3$	$12.1{*}10^3$	$13.7{*}10^3$
	500	0.5	*NF*	$193.1{*}10^3$	$5.5{*}10^3$	$19.6{*}10^3$	$4.9{*}10^3$	$5{*}10^3$
		0.2	*NF*	$579.7{*}10^3$	$75.9{*}10^3$	$195.3{*}10^3$	$258.4{*}10^3$	$133.3{*}10^3$
		0.8	*NF*	$384{*}10^3$	$6.9{*}10^3$	$18.5{*}10^3$	$6.2{*}10^3$	$6.5{*}10^3$
	625	0.5	*NF*	$421.2{*}10^3$	$50.5{*}10^3$	$32.6{*}10^3$	$321.5{*}10^3$	$12.7{*}10^3$
		0.2	*NF*	$397.9{*}10^3$	$95.4{*}10^3$	$232.1{*}10^3$	$115.9{*}10^3$	$68.6{*}10^3$
	TOTAL:		-	$2307.7{*}10^3$	$270.7{*}10^3$	$679.5{*}10^3$	$739.3{*}10^3$	$272.5{*}10^3$

powerful test statistic F_F (Appendix B). For more details on the issue of statistical tests for comparison of algorithms over multiple datasets see (Hollander and Wolfe, 1999; Demśar, 2006).

According to the version by Iman and Davenport (1980) for the Friedman test (Appendix B), and considering a significance level $\alpha = 1\%$ for this test, a

significant difference between the performance of the metaheuristics, with respect to the evaluated ranks, exists. Since the equivalence of the algorithms is rejected, the Nemenyi post-hoc test is applied (Appendix B) in order to perform pairwise comparisons. It considers the performance of two algorithms significantly different if their corresponding average ranks differ by at least a specific threshold critical difference (CD). In this case, considering a significance level of the Nemenyi test of $\alpha = 1\%$, this critical difference is $CD = 1.29$. The differences between the average ranks of the algorithms are reported in Table 4.5.

Table 4.5: Pairwise differences of the average ranks of the algorithms (Critical difference = 1.29 for a significance level of $\alpha = 1\%$ for the Nemenyi test)

ALGORITHM (average rank)	VNS (1.38)	HYBRID (2.48)	GRASP (2.56)	DPSO (3.88)	PILOT (5.21)	EXACT (5.49)
VNS (1.38)	-	1.1	1.18	**2.5**	**3.83**	**4.11**
HYBRID (2.48)	-	-	0.08	**1.4**	**2.73**	**3.01**
GRASP (2.56)	-	-	-	**1.32**	**2.65**	**2.93**
DPSO (3.88)	-	-	-	-	**1.33**	**1.61**
PILOT (5.21)	-	-	-	-	-	0.28
EXACT (5.49)	-	-	-	-	-	-

From this table, it is possible to identify three groups of algorithms with different performance. The best performing group consists of VNS, HYBRID, and GRASP, because they obtain the smallest ranks which are significantly different from the ranks of the remaining algorithms. The remaining groups are, in order, DPSO, and then PILOT and EXACT.

Within the group with the best performance, VNS seems to outperform HYBRID and GRASP, because it has the best rank. Furthermore, its pairwise differences in the ranks with respect to HYBRID (i.e., 1.1) and GRASP (i.e., 1.18) are extremely close to the critical difference ($CD = 1.29$) considering a significance level of $\alpha = 1\%$ for the Nemenyi test. With a a significance level of $\alpha = 5\%$, the critical difference would be $CD = 1.09$, and the rank of VNS would be significantly different with respect to the ranks of HYBRID and GRASP (because their pairwise differences in the ranks are bigger than $CD = 1.09$).

Summarizing, from the Friedman and Nemenyi statistical tests, VNS, HYBRID, and GRASP have comparable performance, and they are the best perform-

ing heuristics for the MLSteiner problem. They are extremely effective, obtaining high-quality solutions in short computational running times. Furthermore, the algorithm which appears to be the most suitable for the proposed problem is VNS. Although a VNS for the MLSteiner, along with other heuristic approaches, was implemented by Cerulli et al. (2006), it has been shown that our VNS implementation is fast, simple, and particularly effective for the MLSteiner problem. The superiority of Variable Neighbourhood Search with respect to the other algorithms is further evidenced by its ease implementation and simplicity.

4.5 Conclusions

In this chapter the minimum labelling Steiner tree (MLSteiner) problem has been considered. It is an extension of the minimum labelling spanning tree problem to the case where only a subset of specified nodes, the basic nodes, need to be connected. The MLSteiner problem is NP-hard, and therefore heuristics and approximate solution approaches with performance guarantees are of interest.

Some metaheuristics for the problem have been presented: a Greedy Randomized Adaptive Search Procedure (GRASP), a Discrete Particle Swarm Optimization (DPSO), a Variable Neighbourhood Search (VNS), and a hybrid local search method (HYBRID) obtained by combining Variable Neighbourhood Search with Simulated Annealing (SA). Considering a wide range of problem instances, these metaheuristics have been compared to the Pilot Method (PILOT) by Cerulli et al. (2006), the most popular MLSteiner heuristic in the literature. Based on this experimental analysis, all the proposed procedures clearly outperformed PILOT and, in particular, the best performance was obtained by VNS, HYBRID, and GRASP. It was shown that the proposed metaheuristics are fast and extremely effective for the MLSteiner problem, obtaining high-quality solutions in short computational times. Furthermore, the algorithm which appears to be the most suitable for the proposed problem is VNS, thanks to the following features: ease of implementation, user-friendly code, high-quality of the solutions, and shorter computational running times. This analysis provides further evidence of the ability of VNS to deal with NP-hard combinatorial problems.

Measure what is measurable, and make measurable what is not so.

Galileo Galilei

Chapter 5

Quartet method of hierarchical clustering

Given a set of objects and their pairwise distances, we wish to determine a visual representation of the data. We use the quartet paradigm to compute a hierarchy of clusters of the objects. The method is based on an NP-hard graph optimization problem called the minimum quartet tree cost problem. This chapter presents and compares several metaheuristic approaches to approximate the optimal hierarchy. The performance of the algorithms is tested through extensive computational experiments and it is shown that the Reduced Variable Neighbourhood Search metaheuristic is the most effective approach to the problem, obtaining high quality solutions in short computational running times.

5.1 Introduction

The problem of grouping similar objects to produce a *classification* (or *clustering*) (Kaufman and Rousseeuw, 2005) goes back to primitive times when early humans realized that many individual objects shared certain properties such as being edible, or poisonous, or ferocious, etc. A classification scheme may simply represent a convenient method for organizing a large data set so that it can be more easily understood and information retrieved more efficiently. If the data can validly be summarized by a small number of groups of objects, referred to

as *clusters* or *classes*, then the group labels may provide a very concise description of patterns of similarities and differences in the data. In natural sciences such as biology and zoology, the practice of classifying organisms is generally known as *taxonomy*. Numerical techniques for deriving classifications, named as *cluster analysis* or *segmentation*, originated largely in these areas (Kaufman and Rousseeuw, 2005).

Clusters are groups of objects that are similar according to a specific metric. There are various ways to cluster. A major class of cluster analysis techniques is represented by *hierarchical clustering* methods (Kaufman and Rousseeuw, 2005). Conceptually simple, hierarchical clustering is among the best known methods in this setting, and the most natural way to represent relations among data sets. In a hierarchical clustering method the data are not partitioned into a particular number of classes at a single step. Instead, a series of partitions takes place, which may run from a single cluster containing all objects, to n clusters each containing a single object. Hierarchical clustering methods may be classified as *agglomerative methods*, which proceed by a series of fusions of the n objects into groups, and *divisive methods*, which separate the n objects successively into finer groupings. Hierarchical clustering techniques have been employed in many different disciplines, such as social science, engineering, medicine, biology, planning, management, and even literature (Kaufman and Rousseeuw, 2005). For example, hierarchical clustering methods are used by ecologists to determine which plots in a forest are similar with respect to the vegetation growing on them; by medical researchers to determine which diseases have similar patterns of incidence; by market researchers to determine which brands of products the public perceives similarly; by archeologists to investigate the relationship between various types of artefact; by industrial engineers to find the best layout for a factory's machines; by sociologists to build ontologies of famous individuals (politicians, artists, historical persons, and so on).

Hierarchical classifications produced by either the agglomerative or the divisive approach may be represented by a two dimensional diagram known as *dendrogram* (Diestel, 2000), which illustrates the fusions or divisions made at each successive stage of analysis. The dendrogram, or tree diagram, is a mathematical way to represent the complete clustering procedure by means of a tree structure.

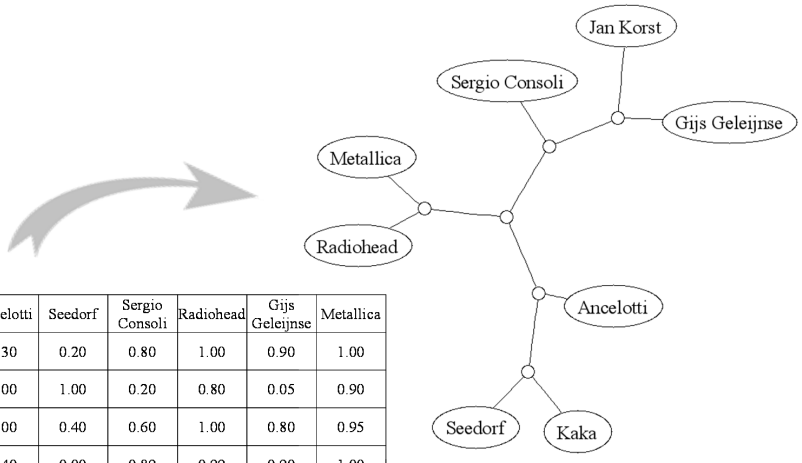
A dendrogram has the objects attached as *leaves* (i.e. nodes at the bottom-most level of the tree (degree = 1)), the *internal nodes* or *inner nodes* (i.e. nodes that are not leaves (degree > 1)) representing the structure of the clusters, and the length of the stems (*path lengths*) representing the distances among the clusters. The arrangement of leaves, internal nodes, and stems determines the *topology* of the dendrogram, whose branches show the relationships among the objects to be clustered. The clustering level of an object with respect to another is determined by the number of stems between the corresponding leaves. There are many different types of dendrograms (Diestel, 2000). In some there are limits placed on the degrees of the internal nodes. In others, additions are made to the structure, by labelling the nodes, or by orienting, ordering or assigning lengths to the edges. For example, a dendrogram is said to be *ordered* if the edges incident to each node are assigned a fixed order. Unordered trees are of dominant interest in clustering contexts because edge orderings have no effect on the path lengths between the nodes in the tree. A dendrogram is *directed* if each edge connecting two nodes has a direction, otherwise it is *undirected*. In directed dendrograms, a node is a *parent node* if it has an outgoing edge directed towards another node, called *child node*. Note that a node may be a parent with respect to a neighbouring node and, at the same time, a child with respect to another. A node which has only outgoing incident edges is referred to as a *root node* (i.e. a node that is never a child). A directed dendrogram is *rooted* if it contains exactly one root node. The root node can be used to further induce ancestry relations between nodes. Nodes near the root become "ancestors" of those reached from them via edges going "away from the root". A dendrogram without such a special internal node is said to be *unrooted* (i.e. there is no distinction between parent and child nodes) and is often displayed in a more free form.

Since all agglomerative hierarchical techniques ultimately reduce the data to a single cluster containing all the individuals, and the divisive techniques will finally split the entire set of data into n groups each containing a single individual, the investigator wishing to have a solution with an "optimal" number of clusters will need to decide when to stop. The tricky problem of deciding on the correct number of clusters represents a difficulty in most hierarchical clustering methods. Our aim is to analyse data sets for which the number of clusters is not

known a priori. Thus, this chapter focusses on the *quartet method* of hierarchical clustering (Cilibrasi and Vitányi, 2005, 2006) which, given a set of objects to be classified, does not require the number of clusters to be given as input, but produces a hierarchy of the objects according to a specific cost evaluation.

Given $n \geq 4$ objects to cluster, the quartet method of hierarchical clustering accepts as input a *distance matrix*, which is a matrix containing the distances, taken pairwise, among the n objects. It is therefore a symmetric $n \times n$ matrix containing non-negative reals, normalized between 0 and 1, as entries. The value 1 represents the largest distance between two objects. The quartet method produces a dendrogram with a special topology, called a *full unrooted binary tree* with $n \geq 4$ leaves. A dendrogram is a full unrooted binary tree if all the internal nodes have degree exactly three and there is no distinction between parent and child nodes (Furnas, 1984; Diestel, 2000). In order to visually represent the distance matrix as well as possible, the quartet method of hierarchical clustering places the n objects to be clustered as leaves of the full unrooted binary tree, such that objects with a short relative distance will be represented close to each other in the tree. A full unrooted binary tree with $n \geq 4$ leaves will have exactly $n - 2$ internal nodes, and consequently will have a total of $2n - 2$ nodes. This special dendrogram is sometimes called *boron tree* (or *ternary tree*), since such a tree, with $2n - 2$ total nodes, has $n - 2$ nodes of valency 3 (corresponding to boron atoms) and n nodes of valency 1 (corresponding to hydrogen atoms). Boron trees are of primary interest in clustering contexts because, of all trees with a fixed number of nodes, they have the richest internal structure (most differentiated paths between nodes). They are therefore the most sensitive for representing the structure of a set of objects (Furnas, 1984).

Figure 5.1 shows a simple example on how the quartet method by Cilibrasi and Vitányi (2005, 2006) classifies $n = 8$ objects from completely different domains by means of a full unrooted binary tree. The left part of Figure 5.1 is an example of an input distance matrix created arbitrarily by the authors. The right part of Figure 5.1 shows the boron tree of the optimal hierarchy of the $n = 8$ objects. The two famous bands Metallica and Radiohead form a cluster. Then Kaka, Seedorf, and Ancelotti, who belong to the same football club (A.C. Milan) form another cluster, with Kaka and Seedorf closer together as players, and coach Ancelotti



	Kaka	Jan Korst	Ancelotti	Seedorf	Sergio Consoli	Radiohead	Gijs Geleijnse	Metallica
Kaka	0.00	1.00	0.30	0.20	0.80	1.00	0.90	1.00
Jan Korst	1.00	0.00	1.00	1.00	0.20	0.80	0.05	0.90
Ancelotti	0.30	1.00	0.00	0.40	0.60	1.00	0.80	0.95
Seedorf	0.20	1.00	0.40	0.00	0.89	0.99	0.90	1.00
Sergio Consoli	0.80	0.20	0.60	0.89	0.00	0.93	0.30	1.00
Radiohead	1.00	0.80	1.00	0.99	0.93	0.00	0.90	0.05
Gijs Geleijnse	0.90	0.05	0.80	0.90	0.30	0.90	0.00	0.55
Metallica	1.00	0.90	0.95	1.00	1.00	0.05	0.55	0.00

Figure 5.1: The left part shows an example of a distance matrix in input to the quartet method of hierarchical clustering. The right part shows the boron tree representing the optimal hierarchy.

further away. The final cluster is that of Sergio Consoli, Gijs Geleijnse, and Jan Korst, who are research scientists, co-authors of (Consoli et al., 2008a).

The rest of the chapter is organised as follows. In Section 5.2, the quartet method and the related literature are described in depth. This method constructs the boron tree approximating the optimal hierarchy according to the input distance matrix. The quartet method of hierarchical clustering is based on an NP-hard graph optimization problem, called the *minimum quartet tree cost* (MQTC) problem (Cilibrasi and Vitányi, 2005, 2006). In Section 5.3, we present the details of several metaheuristics which find approximate solutions to the problem: the heuristic recommended in the literature (the *Randomized Hill Climbing* by Cilibrasi and Vitányi (2005, 2006)), and four new approaches to the quartet method (*Greedy Randomized Adaptive Search Procedure*, *Simulated Annealing*, *Variable Neighbourhood Search*, and *Reduced Variable Neighbourhood Search*). Section 5.4 includes the experimental analysis of the evaluation of these metaheuristics, and the chapter ends with some conclusions (Section 5.5). The basic concepts of

metaheuristics and combinatorial optimization were presented in Chapter 2, but, for further information, the reader is referred to (Voß et al., 1999; Glover and Kochenberger, 2003; Gendreau and Potvin, 2005).

5.2 The quartet method of hierarchical clustering

A fundamental problem in computational biology which has been widely studied in recent years is the reconstruction of evolutionary trees from biological data. An evolutionary tree, also called a phylogenetic tree, is a dendrogram which shows the evolutionary relationships between various biological species or other entities that are believed to have a common ancestor. In an evolutionary tree, each node with descendants represents the most recent common ancestor of the descendants, and the edge lengths in some trees correspond to time estimates. Each node is called a taxonomic unit or taxon. The compelling need for having efficient computational tools to solve this biological problem has attracted much attention to the analysis of the quartet paradigm for inferring evolutionary trees (Felsenstein, 1981). Quartet methods utilize topological information on sets of four objects, representing taxa, to infer an evolutionary tree. Given a set N of $n \geq 4$ objects, the number of sets of four objects from the set N is given by:

$$\binom{n}{4} = \frac{n!}{4!(n-4)!} = \frac{n(n-1)(n-2)(n-3)}{24}. \tag{5.1}$$

For each set of four objects $\{a, b, c, d\} \in N$, there exist exactly three different dendrograms with four leaves (i.e. two internal nodes), also known as *simple quartet topologies*: $ab|cd$, $ac|bd$, $ad|bc$ (Figure 5.2). The vertical bar in a simple quartet topology divides the two pairs of objects, where each pair is represented by two leaf nodes, labelled by the corresponding objects and attached to the same internal node. For example, in the simple quartet topology $ab|cd$, the objects a and b are connected to the same internal node, differently from c and d which are connected to another internal node. Thus, considering the set N of $n \geq 4$ objects, the total number of possible simple quartet topologies is:

$$3 \cdot \binom{n}{4} = \frac{n(n-1)(n-2)(n-3)}{8}. \tag{5.2}$$

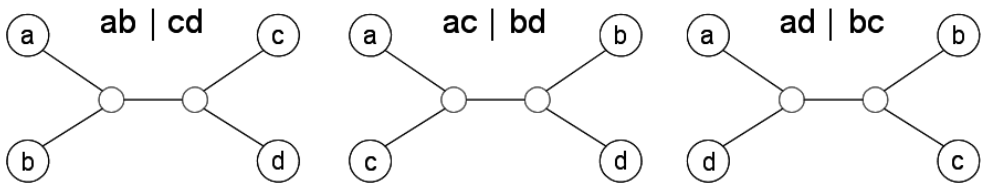


Figure 5.2: The three different simple quartet topologies of the generic set $\{a, b, c, d\}$ of objects.

The quartet methods proceed by first estimating the topology of each quartet of taxa and then recombining the inferred simple quartet topologies into an evolutionary tree. A major difficulty in this approach derives from the fact that quartet topology inference methods often make mistakes, and thus may result in a set Q of simple quartet topologies that is not consistent with any evolutionary tree. A full unrooted binary tree t is *consistent* with respect to a simple quartet topology $ab|cd$ *if and only if* the path from a to b does not cross the path from c to d (Felsenstein, 1981). We refer to $ab|cd$ as a simple quartet topology being *embedded* in the tree t. For example, the full unrooted binary tree in Figure 5.1 is consistent with the simple quartet topology *Seedorf, Radiohead | Sergio Consoli, Jan Korst*. However, it is not consistent with the quartet topology *Ancelotti, Sergio Consoli | Metallica, Jan Korst*.

The problem of recombining the quartet topologies of Q to form an estimate of the correct evolutionary tree is naturally formulated as an optimization problem that looks for an evolutionary tree t maximizing the number of consistent simple quartet topologies Q_t (i.e. max $Q \cap Q_t$). This problem, referred to as *maximum quartet consistency* (MQC) problem, has been shown to be NP-hard (Steel, 1992). Jiang et al. (2000) proved that the MQC problem admits a polynomial time approximation scheme by using the technique of smooth integer polynomial programming and by exploiting the natural denseness of the set Q. However, this scheme only guarantees an evolutionary tree that may deviate from Q by εn^4 quartet topologies for any small constant $\varepsilon > 0$, where n is the number of taxa.

Due to these results, most quartet methods are heuristics which attempt to solve the MQC problem, or some variants of the MQC problem with weaker op-

timization requirements. For example, Strimmer and von Haeseler (1996) formulated the MQC problem as what they call a "tree-puzzling problem" by providing the simple quartet topologies with a probability value to be inferred. Then, a set of simple quartet topologies is selected at random according to these probabilities to form the maximum-likelihood evolutionary tree. Berry et al. (1999) reported an interesting result. They presented two "quartet cleaning" algorithms for correcting bounded numbers of quartet errors (i.e. incorrect inferences of simple quartet topologies) for many popular quartet methods. Exact approaches to the MQC problem are presented in (Ben-Dor et al., 1998), where the problem is solved by using dynamic programming and a geometric algorithm, and in (Weyer-Menkhoff et al., 2005), where the problem is reformulated as an integer linear programming problem. However, these approaches are not able to solve problems with more than 15-20 taxa.

Cilibrasi et al. (2004) introduced a quartet method for hierarchically clustering data from different domains, not necessarily evolutionary data. This paper proposed a robust automatic music classification procedure consisting of two steps. The first step consists of extracting the *Normalized Compression Distances* (NCD) (Li and Vitányi, 1997) among some considered pieces of music. The Normalized Compression Distance is a similarity metric based on string compression which mimics the ideal performance of Kolmogorov complexity (Li and Vitányi, 1997). NCD is able to extract consistent pairwise distances among the pieces of music without considering numerical features related to pitch, rhythm, harmony, or other intrinsic information, as in other popular automatic music classification methods in the literature. The second step consists of creating an efficient visualization of the extracted pairwise distances by means of the quartet method of hierarchical clustering. To substantiate the claims of universality and robustness of this automatic classification method, evidence of other successful applications in areas as diverse as genomics, virology, languages, literature, handwriting, astronomy and combinations of objects from completely different domains, were reported in (Cilibrasi and Vitányi, 2005). In particular, Cilibrasi and Vitányi (2007) reported an interesting application of this theory, consisting of the automatic extraction of similarities among words and phrases from the World Wide Web (WWW) using Google page counts. The WWW is the largest information

source on earth, and the context information entered by millions of independent users provides automatic semantics of useful quality.

In (Cilibrasi and Vitányi, 2006), the authors presented the quartet method of hierarchical clustering in a more formal way. They showed the main concepts, components, advantages and disadvantages of the method, particularly underlining the similarities and differences with respect to other methods from biological phylogeny. Cilibrasi and Vitányi (2006) also showed that the quartet method of hierarchical clustering is based on the *minimum quartet tree cost* (MQTC) problem, and provided a Randomized Hill Climbing metaheuristic to obtain approximate solutions. Several experiments with natural data, like genomic and phylogenetic data, texts or music, and data of completely different types, were further presented. The Randomized Hill Climbing produced good approximate solutions for small sets of objects (up to 40-50 objects), but for larger sets the performance was poor.

5.2.1 Mathematical formulation

Given a set N of $n \geq 4$ objects as points in a space provided with a distance measure, the associated symmetric distance matrix $n \times n$ has as entries the pairwise distances between the objects, normalized between 0 and 1. To extract a hierarchy of clusters from the distance matrix, the quartet method by (Cilibrasi and Vitányi, 2005, 2006) determines a full unrooted binary tree that visually represents the symmetric distance matrix as well as possible according to a cost measure. This representation allows useful information to be extracted from the data and clusters of data to be related to each other.

Considering the set N of $n \geq 4$ objects, the quartet method of hierarchical clustering associates a real valued cost with each simple quartet topology by means of a *cost function* $C : Q \rightarrow \Re^+$, where Q is the set of simple quartet topologies. The cost assigned to each simple quartet topology is defined as the sum of the distances (taken from the distance matrix) between each pair of neighbouring leaves (Cilibrasi and Vitányi, 2005, 2006). For example, the cost associated with the simple quartet topology $ab|cd$ is

$$C_{ab|cd} = d(a, b) + d(c, d), \tag{5.3}$$

where $d(a, b)$ and $d(c, d)$ indicate, respectively, the distances between the two neighbouring objects (a and b) and (c and d), obtained from the distance matrix.

Consider the set Γ of full unrooted binary trees with $2n-2$ nodes (i.e. n leaves and $n-2$ internal nodes), obtained by placing the n objects to cluster as leaf nodes of the trees. For each boron tree $t \in \Gamma$, precisely one of the three possible simple quartet topologies for any set of four leaves is consistent (Cilibrasi and Vitányi, 2005, 2006). Thus, for each $t \in \Gamma$, there exist precisely $\binom{n}{4}$ consistent quartet topologies (one for each set of four objects) embedded in t (Cilibrasi and Vitányi, 2005, 2006). Let Q_t be the set of such $\binom{n}{4}$ quartet topologies embedded in t. Then, the *cost associated with a boron tree* $t \in \Gamma$ is defined as the sum of the costs of its $\binom{n}{4}$ consistent simple quartet topologies, that is

$$C_t = \sum_{\forall \{ab|cd\} \in Q_t} C_{ab|cd} \tag{5.4}$$

In most cases, it is not possible to create a boron tree which embeds all the simple quartet topologies with the minimum cost for all the sets of four objects (especially for a large number of objects n), due to inconsistency. Thus, it is a matter of making the most balanced choice of the quartet topologies to embed. This is the goal of the quartet method of hierarchical clustering: trying to find (or approximate as closely as possible) the boron tree $t \in \Gamma$ with the minimum total cost. This boron tree t will embed the combination of $\binom{n}{4}$ "possible" (consistent) simple quartet topologies of Q with the minimum costs, with respect to a full unrooted binary tree representation of the distance matrix. This optimization problem is called *minimum quartet tree cost* (MQTC) problem (Cilibrasi and Vitányi, 2005, 2006), and can be formally defined as follows:

> *MQTC problem:* Given a set N of $n \geq 4$ objects to be clustered, and a symmetric distance matrix $n \times n$ containing their pairwise distances, find the full unrooted binary tree $t \in \Gamma$ with the minimum total cost C_t, i.e. $\min C_t = \min \left(\sum_{\forall \{ab|cd\} \in Q_t} C_{ab|cd} \right)$.

In a hierarchical clustering context, we do not even have a priori knowledge that certain simple quartet topologies are objectively true and must be embedded. Thus, the quartet method by Cilibrasi and Vitányi (2005, 2006) assigns a cost value to each simple quartet topology, in order to express the relative importance

of the simple quartet topologies to be embedded in the full unrooted binary tree having the n objects as leaves. The boron tree $t \in \Gamma$ with the minimum cost C_t, produced by the quartet method, balances the importance of embedding different quartet topologies against others, leading to a boron tree that visually represents the symmetric distance matrix $n \times n$ as well as possible.

The MQTC may be normalized as follows (Cilibrasi and Vitányi, 2005, 2006). Consider the list of all possible four-tuples of $n \geq 4$ objects in N under consideration. For each set of four objects $\{a, b, c, d\} \in N$, among the three possible simple quartet topologies, extract the one with the minimum cost and that with the maximum cost. Denote these costs as, respectively, m_{abcd} and M_{abcd}, that is:

$$m_{abcd} = \min \{C_{ab|cd}, C_{ac|bd}, C_{ad|bc}\},$$

$$M_{abcd} = \max \{C_{ab|cd}, C_{ac|bd}, C_{ad|bc}\}. \tag{5.5}$$

The *best (minimal) total cost*, m, associated with $t \in \Gamma$ is calculated as the sum of the $\binom{n}{4}$ minimum costs m_{abcd} of each set of four objects $\{a, b, c, d\} \in N$, that is:

$$m = \sum_{\forall \{a,b,c,d\} \in N} m_{abcd}. \tag{5.6}$$

Similarly, the *worst (maximal) total cost*, M, associated with $t \in \Gamma$ is the sum of the $\binom{n}{4}$ maximum costs M_{abcd} of each set of four objects $\{a, b, c, d\} \in N$:

$$M = \sum_{\forall \{a,b,c,d\} \in N} M_{abcd}. \tag{5.7}$$

In most cases, these cost values m and M can not be really attained for any $t \in \Gamma$ (especially with a large number of objects n) and represent, respectively, a lower bound (m) and an upper bound (M) for the cost function C_t, that is $m \leq C_t \leq M, \forall t \in \Gamma$. For a better and more uniform comparison of the costs associated with different boron tree representations of different numbers of objects, the cost function is now rescaled linearly such that the best (minimal) cost maps to 1, and the worst (maximal) cost maps to 0. The rescaled cost function is called *normalized tree benefit score* S_t (Cilibrasi and Vitányi, 2005, 2006), and is defined as follows:

$$S_t = \frac{M - C_t}{M - m} \in [0, 1], \quad \forall t \in \Gamma. \tag{5.8}$$

The goal of the quartet method of hierarchical clustering is to find a boron tree $t \in \Gamma$ with a maximum value of S_t, which is to say, the lowest total cost C_t. In order to compare uniformly the solutions of instances of the quartet method with different sizes, the MQTC can be reformulated with respect to the normalized tree benefit score as follows (Cilibrasi and Vitányi, 2005, 2006):

> *MQTC problem:* Given a set N of $n \geq 4$ objects to be clustered, and a symmetric distance matrix $n \times n$ containing their pairwise distances, find the full unrooted binary tree $t \in \Gamma$ with the maximum normalized tree benefit score S_t (i.e. max S_t).

Considering a set N of $n \geq 4$ objects, all the possible representations of the distance matrix by means of a boron tree $t \in \Gamma$ will have a best normalized tree benefit score less than one in most of cases ($S_t < 1$, that is $C_t > m$), especially for a large number of objects n and noise in the distance matrix. The value $(1 - S_t)$ gives an estimation on how large is the distortion produced by a boron tree representation of the distance matrix, resulting from the quartet method of hierarchical clustering. Trying to find the boron tree $t \in \Gamma$ with the maximum S_t value (minimum C_t value) is the goal of the MQTC problem. This boron tree t will visually represent the distance matrix $n \times n$ as faithfully as possible by using the quartet method representation. As shown in (Cilibrasi and Vitányi, 2005, 2006), the minimum quartet tree cost problem is an NP-hard optimization problem by reduction from the maximum quartet consistency problem (Steel, 1992; Jiang et al., 2000). Therefore, any practical approach to obtain or approximate the optimal solution requires heuristics. In the next section, several metaheuristics for the problem considered are presented and discussed in detail.

5.3 Exploited metaheuristics

This section describes the main features of the metaheuristics considered in this chapter for the minimum quartet tree cost problem. First, the best performing method from the literature is reported, the *Randomized Hill Climbing* (RHC) by Cilibrasi and Vitányi (2005, 2006). The remaining heuristics are new approaches to the quartet method of hierarchical clustering. They are a *Greedy*

Randomized Adaptive Search Procedure (GRASP), a *Simulated Annealing* (SA) approach, a *Variable Neighbourhood Search* (VNS), and a *Reduced Variable Neighbourhood Search* (RVNS).

Before examining these methods in detail, it is useful to specify the notation used within the implementations of these algorithms. Given a full unrooted binary tree, its internal nodes can be classified as *terminal nodes*, which are internal nodes connected to two leaves and another internal node, *transition nodes*, which are internal nodes connected to one leaf node and two other internal nodes, and *cross nodes*, which are internal nodes connected to three other internal nodes (no attached leaf nodes). For example, the boron tree in Figure 5.1 has three terminal nodes which are connected to pairs of leaves with labels Seedorf and Kaka, Metallica and Radiohead, Gijs Geleijnse and Jan Korst, two transition nodes which are connected to the leaves with labels Sergio Consoli and Ancelotti, and one cross node which is not connected to any leaf. Furthermore, a *branch* of a full unrooted binary tree is defined as the subgraph, delimited between one terminal node and one cross node, containing only transition nodes. For example, the boron tree of Figure 5.1 contains three branches, each one rooted at the only cross node of the tree and finishing with one of the three terminal nodes. The first branch is attached to the leaves Metallica and Radiohead, another is attached to the leaves Sergio Consoli, Gijs Geleijnse, and Jan Korst, and the last branch is attached to the leaves Ancelotti, Seedorf, and Kaka.

5.3.1 Randomized Hill Climbing

The Randomized Hill Climbing (RHC) proposed by Cilibrasi and Vitányi (2005, 2006) for the quartet method of hierarchical clustering combines a basic Hill Climbing heuristic with randomization by using parallelized Genetic Programming (Glover and Kochenberger, 2003), where undirected trees evolve in a random walk driven by a prescribed fitness function (Cilibrasi and Vitányi, 2006). The details of this RHC for the quartet method are specified in Algorithm 5.1.

The algorithm starts by selecting at random a full unrooted binary trees $t \in \Gamma$ with $2n - 2$ nodes (i.e. n leaves and $n - 2$ internal nodes), obtained by placing the $n \geq 4$ objects to cluster as leaves. This boron tree t is used as basis

Algorithm 5.1: Randomized Hill Climbing for the quartet method of hierarchical clustering

Input: A symmetric distance matrix d containing the $n \times n$ pairwise distances among $n \geq 4$ objects;
Output: A full unrooted binary tree t with $2n-2$ nodes;
Initialisation:
- Let Γ be the class of full unrooted binary trees with $2n-2$ nodes (i.e. n leaves and $n-2$ internal nodes), obtained by placing the $n \geq 4$ objects to cluster as leaves;
- For each $x \in \Gamma$, let $S_x \in [0,1]$ be the normalized tree benefit score of x;
- Let $t' \in \Gamma$ be a full unrooted binary tree used as support solution at each iteration;

begin
  Generate the initial boron tree $t \in \Gamma$ at random: $t \leftarrow$*Generate-At-Random(Γ)*;
  Evaluate the normalized tree benefit score of t: $S_t \leftarrow$*Evaluate(t)*;
  repeat
    Set $t' \leftarrow t$;
    Select the number k of simple mutations with fat-tail probability distribution $p(k) = c/k(\log k)^2$ where $1/c = \sum_{k=1}^{\infty} 1/k(\log k)^2$;
    for $i \leftarrow 1$ *to* k **do**
      Apply a simple mutation to t': $t' \leftarrow$*Simple-Mutation(t')*;
      Increase i: $i \leftarrow i+1$;
    end
    Evaluate the normalized tree benefit score of t': $S_{t'} \leftarrow$*Evaluate(t')*;
    if $S_{t'} > S_t$ **then**
      Move $t \leftarrow t'$;
    end
  until *termination conditions* ;
  $\Rightarrow$ The full unrooted binary tree $t \in \Gamma$.
end

for further searching. The costs of the consistent quartet topologies embedded in t are calculated, and then the normalized tree benefit score S_t is computed ($S_t \leftarrow$*Evaluate(t)*). Afterwards, solution t is assigned to another boron tree t', which will be used as a support solution at each iteration of the search process. Then, a number k is picked up by a *fat-tail probability distribution* $p(k)$ (Cilibrasi and Vitányi, 2005, 2006):

$$p(k) = \frac{c}{k(\log k)^2}, \text{ where } \frac{1}{c} = \sum_{k=1}^{\infty} \frac{1}{k(\log k)^2}. \tag{5.9}$$

A fat tail probability distribution $p(k)$ with the fattest tail possible has been chosen, in order to concentrate maximal probability also on the larger values of k, trying to minimize the likelihood of being trapped at a local minimum. For more details see (Cilibrasi and Vitányi, 2005, 2006).

In order to search for a better solution, a *k-mutation* is applied to the support solution t'. A k-mutation is defined as a sequence of k *simple mutations*, where a

simple mutation, or *1-mutation*, is one of three possible transformations (Cilibrasi and Vitányi, 2005, 2006):

1. A *leaf swap*, which consists of randomly choosing two leaf nodes and swapping them;

2. A *subtree swap*, which consists of randomly choosing two internal nodes and swapping the subtrees rooted at those nodes;

3. A *subtree transfer*, whereby a randomly chosen subtree (possibly a transition node) is detached and reattached in another place, maintaining arity invariant.

Note that each of these simple mutations keeps the number of leaf nodes and internal nodes in the tree invariant. Only the structure of the boron tree and the positions of the nodes are changed. Considering the support boron tree t', a k-mutation is composed by choosing one of the three possible simple mutations with equal probability. Leaves and internal nodes for each simple mutation are selected completely at random. Boron trees which are close to t', in terms of number of simple mutation steps in between, are examined often, intensifying the search process, while boron trees that are far away from the original tree will eventually be examined, but not very frequently, diversifying the search process.

The normalized tree benefit score of the new solution t', obtained by the k-mutation, is evaluated ($S_{t'}$), and is compared to the normalized tree benefit score (S_t) of the best solution to date t. If an improved boron tree is obtained ($S_{t'} > S_t$), the best solution to date is updated with the new solution ($t \leftarrow t'$), otherwise the search restarts with the current t. This procedure continues iteratively until the termination conditions imposed by the user are satisfied and, at the end of the algorithm, the best boron tree to date $t \in \Gamma$ is produced as output of the procedure.

5.3.2 Greedy Randomized Adaptive Search Process

The GRASP (Greedy Randomized Adaptive Search Procedure) methodology was developed in the late 1980s, and the acronym was coined by Feo and Resende (1989). It was first used to solve set covering problems, but was then extended to a wide range of combinatorial optimization problems (Pitsoulis and Resende,

2002). GRASP is basically a multi-start two-phase metaheuristic, consisting of a *construction phase* and a *local search* phase (for a survey on GRASP see Section 2.2.3). The details are specified in Algorithm 5.2.

Algorithm 5.2: Greedy Randomized Adaptive Search Procedure for the quartet method of hierarchical clustering

Input: A symmetric distance matrix d containing the $n \times n$ pairwise distances among $n \geq 4$ objects;
Output: A full unrooted binary tree t with $2n-2$ nodes;
Initialisation:
- Let Γ be the class of full unrooted binary trees with $2n-2$ nodes (i.e. n leaves and $n-2$ internal nodes), obtained by placing the $n \geq 4$ objects to cluster as leaves;
- For each $x \in \Gamma$, let $S_x \in [0,1]$ be the normalized tree benefit score of x;
- Let $t' \in \Gamma$ be a full unrooted binary tree used as support solution at each iteration;
- Let RCL_α be the restricted candidate list of length α;

begin
  Generate the initial boron tree $t \in \Gamma$ at random: $t \leftarrow$*Generate-At-Random(Γ)*;
  Evaluate the normalized tree benefit score of t: $S_t \leftarrow$*Evaluate(t)*;
  repeat
    Set $t' \leftarrow \emptyset$;
    Construction phase(t', RCL_α);
    Local search(t');
    Evaluate the normalized tree benefit score of t': $S_{t'} \leftarrow$*Evaluate(t')*;
    if $S_{t'} > S_t$ **then**
      Move $t \leftarrow t'$;
    end
  until *termination conditions* ;
  $\Rightarrow$ The full unrooted binary tree $t \in \Gamma$.
end

The algorithm starts by selecting at random a full unrooted binary trees $t \in \Gamma$ with $2n-2$ nodes, obtained by placing the $n \geq 4$ objects to cluster as leaves. The costs of the consistent quartet topologies embedded in t are evaluated, and then the normalized tree benefit score S_t is computed ($S_t \leftarrow$*Evaluate(t)*). Then, the *Construction phase*(t', RCL_α) procedure builds another boron tree $t' \in \Gamma$ by using a greedy randomized mechanism, whose randomness allows solutions in different areas of the solution space to be obtained. Initially the partial solution consists of n vertices with no edges, where each object is assigned to a vertex as a rooted tree of size one. Then, the greedy randomized mechanism obtains a full unrooted binary tree t' by iteratively creating a candidate list of distances (RCL_α: Restricted Candidate List of length α), and then by randomly selecting a distance from this list and connecting the corresponding objects in t'. The connections are made by adding a path of length two between the roots of the

two subgraphs containing the objects, with the new vertex becoming the new root node. The candidate list is created by evaluating the distances between the objects that are not yet connected within the partial boron tree t', and then by including the shortest α of such distances in the list. At each iteration one new distance is randomly selected from RCL_α, the corresponding pair of objects are connected within the current boron tree t', and the candidate list is updated. The construction phase stops when a full unrooted binary tree t' is obtained.

GRASP is effective if the solution construction mechanism samples the most promising regions by using an appropriate value of α. In general, α can be limited either by the number of distances in the list, or by their quality with respect to the best candidate distance. The extreme cases for the size of the candidate list are: $\alpha = 1$ and $\alpha = (n-1)!$, the total number of relative distances between the objects. In the first case, only the best distance not yet included in the partial boron tree t' is added to the restricted candidate list, and the construction mechanism is equivalent to a deterministic greedy heuristic. In the case of $\alpha = (n-1)!$, the candidate list is filled with all the relative distances between the n objects, and the construction mechanism is equivalent to a random walk, because complete randomization is used to choose the next element to add to the partial solution. Thus, it is important to make a good tuning of α in order to obtain an optimal balance between the intensification and diversification capabilities of the search process. Our experience indicates that $5 \leq \alpha \leq 10$ produces good results for the quartet method of hierarchical clustering.

The construction phase stops when a full unrooted binary tree t' is obtained. The produced solution t' is not necessarily locally optimal, so the *Local search(t')* procedure tries to improve it. This phase uses a local search mechanism which, iteratively, tries to replace the current boron tree t' with a better neighbouring boron tree, until no better solution can be found. Different strategies may be used in order to evaluate the neighbourhood structure. In our implementation, we consider each internal node and the neighbouring nodes having Manhattan distance equals to one with respect to the node considered (*one-neighbourhood structure* with respect to the Manhattan distance), that is we consider the internal nodes which are directly connected to the internal node considered. Then, a transformation of each pair of selected internal nodes is performed, aimed at

producing small changes in the topology of the boron tree t' considered, checking whether these modifications improve the normalized tree benefit score of t'.

The internal nodes are selected following a specific order. First, all the terminal nodes are evaluated in order to improve each single branch of the current boron tree t'. After selecting a terminal node, all the successive transition nodes belonging to the corresponding branch of the tree are evaluated, starting from the ones that are closer to the terminal node and stopping when the cross node delimiting the current branch is reached. For each selected internal node, the algorithm tries to exchange its attached leaf (or leaves in case of the terminal node) with the leaves attached to the one-neighbouring internal nodes (according to the Manhattan distance). In Figure 5.3 is shown an example where the two leaves C and H attached to two one-neighbouring transition nodes are exchanged, while in Figure 5.4 the leaf node H attached to a transition node is exchanged with the leaf node L attached to the one-neighbouring terminal node.

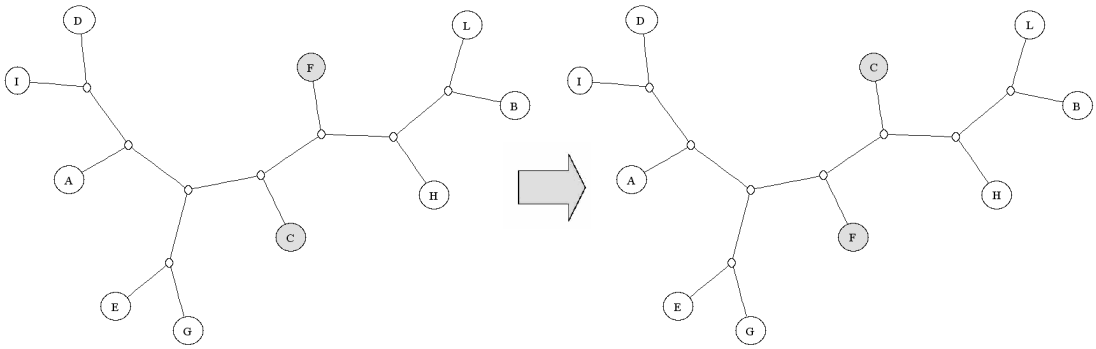


Figure 5.3: Example showing the exchange of two leaves attached to two one-neighbouring transition nodes.

The exchange of two leaves is retained if the normalized tree benefit score of t' improves. After selecting all the terminal nodes and trying to improve the corresponding branches, the algorithm selects all the remaining cross nodes. For each cross node, the algorithm tries to move each one-neighbouring transition node from the corresponding branch containing the transition node to the two other branches rooted at the selected cross node (see Figure 5.5). In the case of another neighbouring cross node, the algorithm alternatively swaps one branch

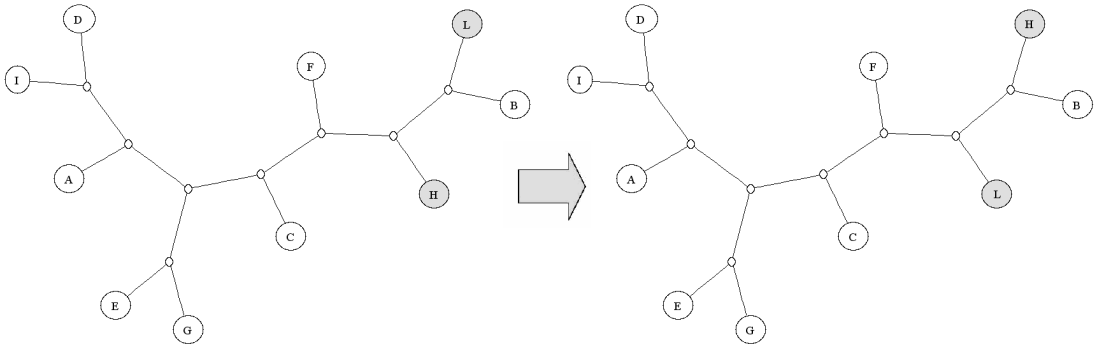


Figure 5.4: Example showing the exchange of two leaves attached to a transition node and to the one-neighbouring terminal node.

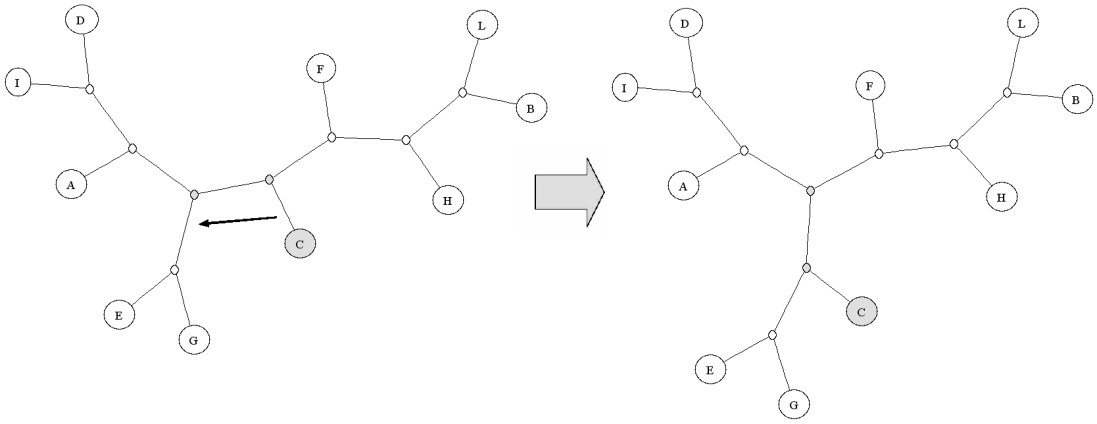


Figure 5.5: Example showing the move of a transition node to another branch of the one-neighbouring cross node.

of one cross node with another branch of the other cross node (see Figure 5.6). Again, each modification of the boron tree t' is retained if it produces a benefit in the normalized tree benefit score $S_{t'}$.

After exhausting all the cross nodes, the local search stops because all the internal nodes have been evaluated (best improvement strategy) and, hopefully, the obtained boron tree t' will represent an improved solution with respect to the boron tree previously obtained by the construction phase. Afterwards, if the normalized tree benefit score of t' is better than that of the best boron tree to date t (i.e. $S_{t'} > S_t$), the best boron tree to date is updated with the new solution

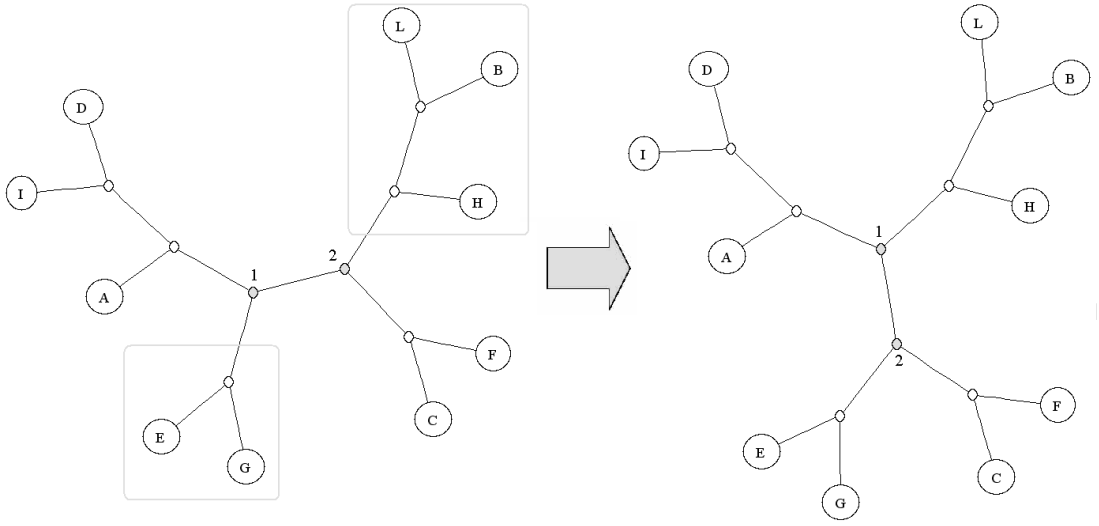


Figure 5.6: Example showing the exchange of two branches of two one-neighbouring cross nodes.

($t \leftarrow t'$). The entire algorithm proceeds iteratively until the user termination conditions are satisfied, and produces the best boron tree to date $t \in \Gamma$ as output of the procedure.

Success of a particular GRASP implementation depends on a number of different factors, such as the efficiency of the randomized greedy procedure used, the choice of the neighbourhood structure, and the implementation of the local search technique. The full unrooted binary trees obtained by our GRASP are usually of good quality because GRASP offers fast local convergence (high intensification capability) as a result of the greedy aspect of the procedure used in the construction phase, and of the local search mechanism; and also a large exploration of the solution space (high diversification capability) for the randomization used in the selection of a new element from RCL_α.

5.3.3 Simulated Annealing

Simulated Annealing (SA) is a descent heuristic with non-deterministic search developed by Kirkpatrick et al. (1983). In contrast to classical descent methods, where only modifications to the current solution that decrease the cost function

value are accepted, modifications that increase the value of the cost function are allowed in SA (for a survey on Simulated Annealing see Section 2.2.1).

SA exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system, forming the basis of an efficient optimisation technique for combinatorial and other problems. SA seeks to minimise an energy function (the cost function); free variables in SA are like particles in the metal, and "low energy" configurations correspond to high quality solutions of the problem, obtained by slowly reducing a *temperature parameter* (T) by means of a *cooling rule* (or *cooling schedule*). The dependency is such that the current solution is always replaced by a new one if this modification reduces the cost function value, while a modification increasing the cost function value by Δ is only accepted with a probability $\exp(-\Delta/T)$ (Boltzmann function), using the temperature T as a control parameter. At the beginning of the algorithm, at a high temperature T, the probability of accepting an increase in the cost function value is high (uphill moves), allowing many worse moves to be accepted. Conversely, this probability gets lower as the temperature T is decreased (downhill moves) during the search process by means of the cooling rule.

The details of the implementation of our Simulated Annealing for the quartet method are specified in Algorithm 5.3. For the problem considered, we implemented a *non-monotonic SA cooling schedule* (Osman, 1993), which requires specification of the following: (i) starting and final temperatures (T_s and T_f); (ii) decrement rule for updating the temperature T after each iteration; (iii) occasional increment rule for updating the temperature T every N_{reset} iterations with a reset temperature T_{reset} (in order to avoid the system being locked at local optima).

The algorithm starts by selecting at random a full unrooted binary tree $t \in \Gamma$ with $2n-2$ nodes, obtained by placing the $n \geq 4$ objects to cluster as leaves, with cost C_t and normalized tree benefit score S_t. Then, the starting and final temperatures, T_s and T_f, are set to the maximum and minimum estimated variations of the cost function, Δ_{max} and Δ_{min}, evaluated heuristically by means of the *Test-Cycle(t)* procedure. This procedure considers the *base moves* that each internal node of t can perform with its neighbouring internal nodes (*one-neighbourhood*

Algorithm 5.3: Simulated Annealing for the quartet method of hierarchical clustering

Input: A symmetric distance matrix d containing the $n \times n$ pairwise distances among $n \geq 4$ objects;
Output: A full unrooted binary tree t_{best} with $2n-2$ nodes;
Initialisation:
- Let Γ be the class of full unrooted binary trees with $2n-2$ nodes (i.e. n leaves and $n-2$ internal nodes), obtained by placing the $n \geq 4$ objects to cluster as leaves;
- For each $x \in \Gamma$, let C_x be the cost associated with x and $S_x \in [0,1]$ the corresponding normalized tree benefit score;
- Let Δ_{min} and Δ_{max} be the minimum and the maximum estimated variations of the cost function;
- Let T be the temperature parameter, T_s be the starting temperature value, T_f be the final temperature value, T_{best} be the best temperature value, T_{reset} be the reset temperature value, α be the geometric cooling rate;
- Let i be the number of iterations of the algorithm;
- Let $N_{reset} = \frac{2.5 \cdot 10^5}{n^2} + 200$ be the number of reset iterations;
- Let $t \in \Gamma$ be the full unrooted binary tree used at each iteration;
- Let $t' \in \Gamma$ be a full unrooted binary tree used as support solution at each iteration;

begin
  Generate the initial boron tree $t \in \Gamma$ at random: $t \leftarrow$*Generate-At-Random(Γ)*;
  Evaluate the cost of t and its normalized tree benefit score: $(C_t, S_t) \leftarrow$*Evaluate(t)*;
  Evaluate the minimum and the maximum estimated variations of the cost function: $(\Delta_{min}, \Delta_{max}) \leftarrow$*Test-Cycle(t)*;
  Move $t_{best} \leftarrow t$, and set $T \leftarrow T_{reset} \leftarrow T_s \leftarrow \Delta_{max}$, $T_f \leftarrow \Delta_{min}$, $i \leftarrow 1$;
  repeat
    Move $t' \leftarrow t$;
    Select at random an integer between 0 and $n-2$: $\lambda \leftarrow$ *Random(0, n-2)*;
    for $j \leftarrow 1$ *to* λ **do**
      Perform a base move with respect to t': $t' \leftarrow$*Base-Move(t')*;
      Increase j: $j \leftarrow j+1$;
    end
    Evaluate the cost of t' and its normalized tree benefit score: $(C_{t'}, S_{t'}) \leftarrow$*Evaluate($t'$)*;
    if $S_{t'} > S_t$ **then**
      Move $t \leftarrow t'$;
      if $S_t > S_{t_{best}}$ **then**
        Move $t_{best} \leftarrow t$ and set $T_{best} \leftarrow T$;
      end
    else
      Select at random a real number between 0 and 1: $\xi \leftarrow$ *Random(0, 1)*;
      if $\xi < \exp\left(-\frac{C_{t'} - C_t}{T}\right)$ **then**
        Move $t \leftarrow t'$;
      end
    end
    Geometric decrement rule for the temperature: $T = T_s \cdot \alpha^{(i \bmod N_{reset})/N_{reset}}$, where $\alpha = T_f / T_s$;
    if $(i \ mod \ N_{reset}) = 0$ **then**
      Occasional increment rule for the temperature: $T_{reset} \leftarrow \max(T_{reset}/2, T_{best})$;
      Set $T \leftarrow T_s \leftarrow T_{reset}$;
    end
    Increase the number of iterations: $i \leftarrow i+1$;
  until *termination conditions* ;
  $\Rightarrow$ The full unrooted binary tree $t_{best} \in \Gamma$.
end

structure with respect to the Manhattan distance). The alterations of the cost function corresponding to the performed base moves are evaluated, retaining the maximum and the minimum variations in Δ_{max} and Δ_{min}.

Given an internal node and its neighbouring internal nodes, the possible base moves that can be performed depend on the types of internal node pairs. In the case of:

1. two transition nodes → either the attached leaves are exchanged (see Figure 5.3), or they are transformed into one cross node and one terminal node connected to the corresponding leaves (see Figure 5.7);

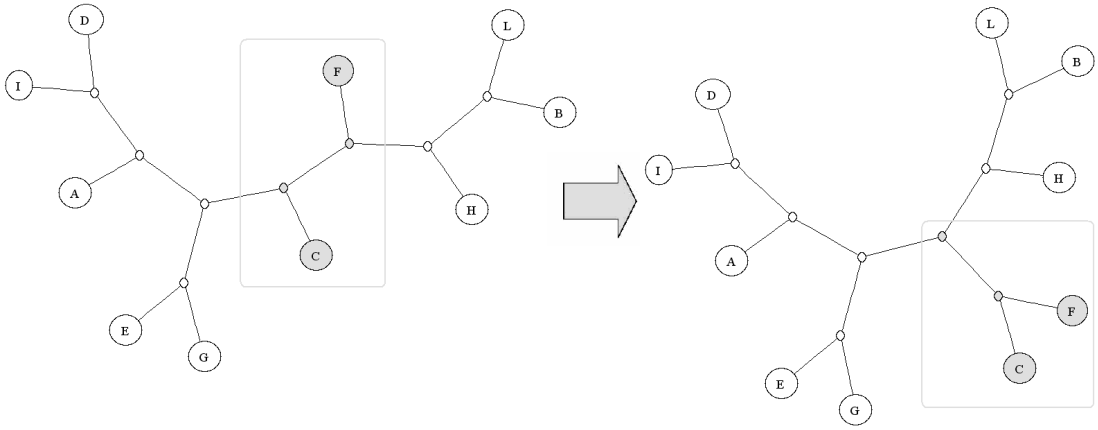


Figure 5.7: Transformation of two one-neighbouring transition nodes into one terminal node and one cross node.

2. one terminal node and one transition node → the leaf of the transition node is exchanged with one of the two leaves of the terminal node (see Figure 5.4);

3. one terminal node and one cross node → they are transformed into two transition nodes with the two leaves of the terminal node attached (see Figure 5.8);

4. one transition node and one cross node → the transition node is moved in one of the other two branches of the cross node (see Figure 5.5);

5. two cross nodes → one branch of one cross node is swapped with a branch of the other cross node (see Figure 5.6).

When T_f and T_s are evaluated, the algorithm continues by assigning the value of T_s to the current temperature T and to the reset temperature T_{reset}, and by

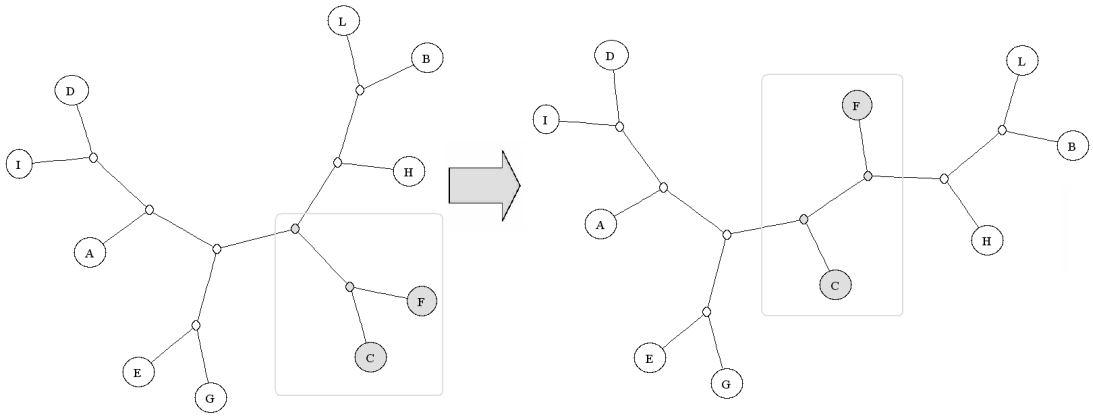


Figure 5.8: Transformation of one terminal node and the one-neighbouring cross node into two transition nodes.

making a copy of t to another boron tree t' that will be modified by means of a *random move*. A random move is defined as a set of consecutive base moves, whose number is a random integer λ selected between 0 and $n-2$. The random move starts by selecting a random internal node and one of its neighbouring internal nodes, and performing a base move with this pair of nodes. Then, to perform the successive base move, the algorithm selects one of the two internal nodes considered, and another neighbouring internal node that must be different from the two internal nodes already considered. The procedure continues until λ consecutive base moves are produced.

The cost and the normalized tree benefit score of the new boron tree t' are evaluated, $(C_{t'}, S_{t'}) \leftarrow$*Evaluate(t')*. If $S_{t'} > S_t$, the solution t is assigned to the boron tree t', storing the best solution to date in t_{best}, and the temperature at which this boron tree is obtained in T_{best}. Otherwise, if the new boron tree t' is worse than t ($S_{t'} < S_t$), the algorithm moves to t' with a probability that depends on the Boltzmann function $\exp(-\Delta/T) = \exp(-(C_{t'} - C_t)/T)$.

The non-monotonic SA cooling schedule that we use for the quartet method, decreases, at each iteration i of the algorithm, the temperature T according to the following geometric cooling rule:

$$T = T_s \cdot \alpha^{(i \bmod N_{reset})/N_{reset}}, \text{ where } \alpha = T_f/T_s < 1 \tag{5.10}$$

and where (i mod N_{reset}) represents the arithmetic remainder of the integer division between the number of iterations i and the number of reset iterations N_{reset}. Every N_{reset} iterations (i.e. when (i mod N_{reset}) $= 0$) the temperature T and the starting temperature T_s are reset to a larger value, T_{reset}, to allow the algorithm to escape from local optima ($T \leftarrow T_s \leftarrow T_{reset}$). T_{reset} is chosen as the maximum value between $T_{reset}/2$ and T_{best}, while N_{reset} is a user defined parameter (our experience indicates that the value $N_{reset} = (2.5 \cdot 10^5)/n^2 + 200$ produces good results). This cooling schedule and its implementation is in contrast to classical SA schemes. From our experience, the considered non-monotonic cooling schedule outperformed other different SA cooling schedules for the quartet method. Note that the importance of non-monotonic search has been widely discussed in (Glover, 1986) as a basic feature of Tabu Search methods.

Subsequently, the algorithm restarts with the same procedure by setting $t' \leftarrow t$, continuing iteratively until the user termination conditions are satisfied. At the end, the best boron tree to date, t_{best}, is produced as the output of the SA algorithm.

5.3.4 Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is a recent metaheuristic for solving combinatorial optimization problems based on dynamically changing neighbourhood structures during the search process (Hansen and Mladenović, 1997, 2003). VNS does not follow a trajectory, but it searches for new solutions in increasingly distant neighbourhoods of the current solution, jumping only if a better solution than the current best solution is found (for a survey on VNS see Section 2.2.5).

The proposed VNS for the quartet method of hierarchical clustering is specified in Algorithm 5.4. At the starting point, a full unrooted binary tree $t \in \Gamma$ with $2n - 2$ nodes, obtained by placing the $n \geq 4$ objects to cluster as leaves, is generated at random. Then, the *shaking phase*, which represents the core idea of VNS, is applied to t. A shaking phase of size k consists of the random selection of another boron tree t' within the neighbourhood $N_k(t)$ of the current solution t. To obtain t' from $N_k(t)$, the algorithm performs k consecutive base moves, already defined in Section 5.3.3. The first base move is performed to

Algorithm 5.4: Variable Neighbourhood Search for the quartet method of hierarchical clustering

Input: A symmetric distance matrix d containing the $n \times n$ pairwise distances among $n \geq 4$ objects;
Output: A full unrooted binary tree t with $2n-2$ nodes;
Initialisation:
- Let Γ be the class of full unrooted binary trees with $2n-2$ nodes (i.e. n leaves and $n-2$ internal nodes), obtained by placing the $n \geq 4$ objects to cluster as leaves;
- For each $x \in \Gamma$, let $S_x \in [0,1]$ be the normalized tree benefit score of x;
- Let $t' \in \Gamma$ be a full unrooted binary tree used as support solution at each iteration;
- Let k be the current size of the shaking phase, and k_{max} be the maximum size of the shaking phase;
- Let i be the number of iterations between two successive improvements;
- Let $i_{update} = \frac{1.25 \cdot 10^5}{n^2} + 50$ be the number of update iterations for k_{max};

begin
  Generate the initial boron tree $t \in \Gamma$ at random: $t \leftarrow$*Generate-At-Random(Γ)*;
  Evaluate the normalized tree benefit score of t: $S_t \leftarrow$*Evaluate(t)*;
  Set $i \leftarrow 0$ and $k_{max} \leftarrow 2$;
  repeat
    Set $k \leftarrow 1$;
    while $k < k_{max}$ **do**
      Move $t' \leftarrow t$;
      for $j \leftarrow 1$ *to* k **do**
        Shake t' by performing a base move: $t' \leftarrow$*Base-Move(t')*;
        Increase j: $j \leftarrow j + 1$;
      end
      Local search(t');
      Evaluate the normalized tree benefit score of t': $S_{t'} \leftarrow$*Evaluate(t')*;
      if $S_{t'} > S_t$ **then**
        Restart with the first neighbourhood structure: $k \leftarrow 1$;
        Move $t \leftarrow t'$;
        Set $i \leftarrow 0$;
      else
        Increase the current size of the shaking phase: $k \leftarrow k + 1$;
        Increase the number of iterations between two successive improvements: $i \leftarrow i + 1$;
      end
    end
    if $i >= i_{update}$ **then**
      Increase the maximum size of the shaking phase: $k_{max} \leftarrow k_{max} + 1$;
      Set $i \leftarrow 0$;
    end
  until *termination conditions* ;
  $\Rightarrow$ The full unrooted binary tree $t \in \Gamma$.
end

a randomly selected internal node and one of its neighbouring internal nodes (one-neighbourhood structure with respect to the Manhattan distance). Then, to perform the successive base move, the algorithm selects one of the two internal nodes considered, and another neighbouring internal node that must be different from the two internal nodes already considered, and so on. The procedure is repeated until k consecutive base moves are performed.

The shaking phase aims to change the neighbourhood structure when the algorithm is trapped at a local optimum. The solution t' is generated at random in order to avoid cycling, which might occur if a deterministic rule is used. Suitable neighbourhood structures need to be defined for the shaking phase. The simplest and most common choice consists of neighbourhoods with increasing cardinality: $|N_1(\cdot)| < |N_2(\cdot)| < ... < |N_{k_{max}}(\cdot)|$, where k_{max} represents the maximum size of the shaking phase. Let k be the current size of the shaking phase. The algorithm starts by selecting the first neighbourhood ($k \leftarrow 1$) and, at each iteration, it increases the parameter k if a better solution is not obtained ($k \leftarrow k + 1$), until the largest neighbourhood is reached ($k \leftarrow k_{max}$). The process of changing neighbourhoods when no improvement occurs diversifies the search. In particular, the choice of neighbourhoods of increasing cardinality yields a progressive diversification of the search process.

The boron tree t' produced by the shaking phase, represents the starting point for the successive *local search phase*, which tries to improve, if possible, the solution t'. The local search considered (*Local search(t')*) is a first improvement strategy. It considers each internal node of t' and each of its neighbouring internal nodes, and computes all the base moves that can be performed with the selected pair of nodes (as with the shaking phase, the local search phase uses a one-neighbourhood structure with respect to the Manhattan distance for the selection of the neighbouring internal nodes). The local search stops either when a base move which improves the normalized tree benefit cost of t', $S_{t'}$, is produced, or when all the internal nodes of t' have been evaluated without having improved $S_{t'}$.

If an improved boron tree t' is produced by the shaking and the local search phases ($S_{t'} > S_t$), it becomes the best solution to date ($t \leftarrow t'$) and the algorithm restarts from the first neighbourhood ($k \leftarrow 1$) of the best solution t. Otherwise, if no improvements are obtained ($S_{t'} < S_t$), the neighbourhood structure is increased ($k \leftarrow k + 1$) giving a progressive diversification of the search process. Parameter k is increased until the maximum size of the shaking phase, k_{max}, is reached. When this happen, k is re-initialized to the first neighbourhood ($k \leftarrow 1$). The correct setting of k_{max} is an important user task. For the quartet method, a simple reactive schema for the efficient tuning of k_{max} has been

implemented (Battiti et al., 2008). At the starting point, k_{max} is set to a small value ($k_{max} \leftarrow 2$) and is increased ($k_{max} \leftarrow k_{max} + 1$) every i_{update} iterations between two consecutive improvements. Our experience indicates that the value $i_{update} = (1.25 \cdot 10^5)/n^2 + 50$ produces good results. For more details on reactive search techniques, the reader is referred to (Battiti et al., 2008). The algorithm proceeds iteratively until the user termination conditions are satisfied, producing the best boron tree to date, t, as the output of the procedure.

5.3.5 Reduced Variable Neighbourhood Search

Reduced Variable Neighbourhood Search (RVNS) is a variant of the basic VNS algorithm, that has been shown to be successful for many combinatorial problems where local optima with respect to one or several neighbourhoods are relatively close to each other (Hansen and Mladenović, 2003).

The Reduced Variable Neighbourhood Search is obtained from VNS where random solutions are selected from the neighbourhoods $N_k(\cdot)$ of the current solution, without being followed by a local search phase. Therefore, it is a typical example of a pure stochastic heuristic. In practice, RVNS is akin to a classic Monte-Carlo method, but is a more systematic approach (Mladenović et al., 2003). It is useful especially for very large problem instances for which the local search of the basic VNS is costly, as in the case of the quartet method of hierarchical clustering. Hansen and Mladenović (2003) observed that, in RVNS, the best values for the maximum size of the shaking phase (i.e. parameter k_{max}) are often small values.

The details of the Reduced Variable Neighbourhood Search for the quartet method are specified in Algorithm 5.5. The algorithm starts by selecting at random a full unrooted binary trees $t \in \Gamma$ with $2n-2$ nodes, obtained by placing the $n \geq 4$ objects to cluster as leaves, with normalized tree benefit score S_t. Then, the same shaking phase of the VNS specified in Section 5.3.4 is applied. It selects at random another boron tree t' from the neighbourhood $N_k(t)$ of the current solution t, by performing k consecutive base moves. Again, a one-neighbourhood structure with respect to the Manhattan distance, for the selection of the internal nodes, is used. At the beginning, the first neighbourhood ($k \leftarrow 1$) is selected

Algorithm 5.5: Reduced Variable Neighbourhood Search for the quartet method of hierarchical clustering

Input: A symmetric distance matrix d containing the $n \times n$ pairwise distances among $n \geq 4$ objects;
Output: A full unrooted binary tree t with $2n-2$ nodes;
Initialisation:
- Let Γ be the class of full unrooted binary trees with $2n-2$ nodes (i.e. n leaves and $n-2$ internal nodes), obtained by placing the $n \geq 4$ objects to cluster as leaves;
- For each $x \in \Gamma$, let $S_x \in [0,1]$ be the normalized tree benefit score of x;
- Let $t' \in \Gamma$ be a full unrooted binary tree used as support solution at each iteration;
- Let k be the current size of the shaking phase, and k_{max} be the maximum size of the shaking phase;

```
begin
    Generate the initial boron tree t ∈ Γ at random: t ←Generate-At-Random(Γ);
    Evaluate the normalized tree benefit score of t: S_t ←Evaluate(t);
    Set k_max arbitrarily;
    repeat
        Set k ← 1;
        while k < k_max do
            Move t' ← t;
            for j ← 1 to k do
                Shake t' by performing a base move: t' ←Base-Move(t');
                Increase j: j ← j + 1;
            end
            Evaluate the normalized tree benefit score of t': S_t' ←Evaluate(t');
            if S_t' > S_t then
                Restart with the first neighbourhood structure: k ← 1;
                Move t ← t';
            else
                Increase the current size of the shaking phase: k ← k + 1;
            end
        end
    until termination conditions ;
    ⇒ The full unrooted binary tree t ∈ Γ.
end
```

and, at each iteration, the parameter k is increased ($k \leftarrow k+1$) whenever the solution obtained is not an improvement to the current best solution, until the maximum size of the shaking phase (k_{max}) is reached. Note that, in contrast to the VNS in the previous section, the setting of k_{max} in RVNS does not require any complicate schema, because its values are often small values (say $k_{max} = 2$ or 3). Thus, parameter k_{max} is set arbitrarily by the user through computational experience. As already stated, no local search phase is applied after the shaking phase. Throughout the execution of the algorithm, the best solution to date is stored as the boron tree t, which will be produced as output of the algorithm when the user termination conditions are met.

5.4 Experimental results

In this section, the metaheuristics proposed for the quartet method of hierarchical clustering are compared in terms of solution quality and computational running time. We identify the metaheuristics with the abbreviations: RHC (Randomized Hill Climbing), GRASP (Greedy Randomized Adaptive Search Procedure), SA (Simulated Annealing), VNS (Variable Neighbourhood Search), RVNS (Reduced Variable Neighbourhood Search). All the algorithms that we propose have been implemented using the C++ programming language (Microsoft Visual C++ 2005). For the Randomized Hill Climbing, we have used the open-source software released in the public domain by the authors (Cilibrasi, 2007b). All the computations have been made on a Pentium Centrino microprocessor at 2.0 GHz with 512 MB RAM.

In our experiments, we considered 26 different datasets with a number of objects to cluster (n) from 10 up to 224. Data from different fields have been considered in order to evaluate how the algorithms are influenced by the nature of the objects. First we considered data without inconsistency, that is data for which the exact solution is known and have a normalized tree benefit score equals to one, in order to test the accuracy of the quartet-based tree reconstruction. These data were produced artificially as described in Section 5.4.1. Then, in Section 5.4.2 we considered some examples from nature obtained from (Cilibrasi, 2007a,b), concerning a study in genomics with DNA sequences of different placental mammalian species. Section 5.4.3 contains data with real geographic distances between famous cities, while Section 5.4.4 contains data obtained by mining of the WWW through an automatic web information extraction method by Geleijnse et al. (2006). Specifically, we have focussed on data concerning musical artists. All the instances of the problem are available online from the authors (Consoli, 2008).

For each dataset, given a boron tree t produced by the quartet method, solution quality is evaluated by means of its normalized tree benefit score $S_t \in [0, 1]$. The quartet method of hierarchical clustering tries to find the solution which maximizes the S_t value, which is to say, the lowest total cost C_t. A maximum allowed CPU time (*max-CPU-time*), determined with respect to the dimension of

the problem instance, is chosen as the stopping condition for all the metaheuristics. Experimentally, for problem instances with a number of objects $n \leq 100$, we set *max-CPU-time* to one hour (3600 sec). For larger instances ($n \geq 100$), *max-CPU-time* is set to 10 hours (36000 sec). Selection of the maximum allowed CPU time as the stopping criterion is made in order to have a direct comparison of the metaheuristics with respect to the quality of their solutions.

Our results are reported in Tables 5.1 - 5.4. In each table, the first column shows the number n of objects of the datasets considered, while the kind of data determines the different tables. The last row shows the averages, respectively, of the normalized tree benefits score and of the computational running times among the group of data instances considered. All the metaheuristics run for *max-CPU-time* and, in each case, the normalized tree benefit score of the best solution is recorded. The computational times reported in the tables are the times at which the best solutions are obtained. The reported times have precision of ± 1 sec. Analysing the performance of the algorithms considered, for a single dataset a metaheuristic should be considered *better* than another if either it obtains a larger normalized tree benefit score, or an equal normalized tree benefit score but in a smaller computational running time.

5.4.1 Testing the quartet-based tree reconstruction

In this section, we test whether the quartet-based tree reconstruction heuristic is reliable and accurate on clean consistent data with known solutions. We used the same procedure by Cilibrasi and Vitányi (2005, 2006) to generate data instances with corresponding optimal boron trees t having normalized tree benefit score equal to one, $S_t = 1$. To obtain these data, we used the "rand" pseudo-random number generator from the C++ programming language (Microsoft Visual C++ 2005), and derived a metric from it by defining the distance, $d(x, y)$, between two objects x and y, as follows (Cilibrasi and Vitányi, 2005, 2006):

$$d(x,y) = \begin{cases} \dfrac{L(x,y)+1}{n} & \text{if } x \neq y, \\ 0 & \text{otherwise,} \end{cases} \tag{5.11}$$

where $L(x, y)$ is the length of the path from x to y, expressed by the number of edges which connect the leaves of the boron tree where the two objects are

assigned. Obviously, the entries in the diagonal of the distance matrix are all zeros, since $d(x, y) = 0$ if $x = y$. All the boron trees t constructed artificially with this procedure have optimal score $S_t = 1$. Figure 5.9 shows an example of a full unrooted binary tree t with 10 objects and $S_t = 1$ generated at random by means of this procedure.

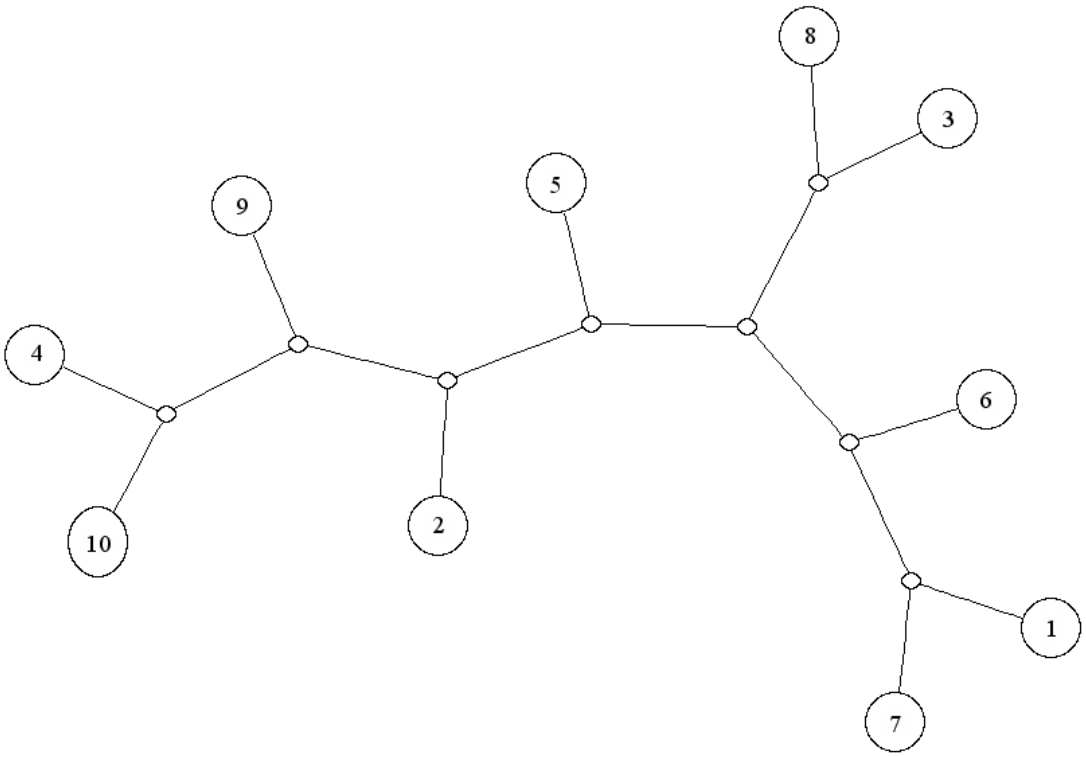


Figure 5.9: Randomly generated full unrooted binary tree t with 10 objects and $S_t = 1$.

We generated data instances with a number of objects n from 10 to 100, setting the *max-CPU-time* for the heuristics to one hour (3600 sec). Computational results, reporting the normalized tree benefit scores found by the heuristics and the corresponding computational times, are presented in Table 5.1. Looking at this table, for $n = 10$ and $n = 20$ all the heuristics obtained the exact solution ($S_t = 1$). However, RHC was considerably slower than the other metaheuristics. For $n > 20$, the performance of RHC was extremely poor, obtaining solutions with extremely low quality in very high computational running times. SA, VNS,

Table 5.1: Computational results for artificial data with optimal normalized tree benefit score equals to one (*max-CPU-time* for heuristics = 36000 sec)

Size	**Normalized tree benefit score**				
n	RHC	GRASP	SA	VNS	RVNS
10	1	1	1	1	1
20	1	1	1	1	1
30	0.99441	1	1	1	1
40	0.98297	0.99234	1	1	1
50	0.92642	0.99641	1	1	1
60	0.75907	0.99308	1	1	1
70	0.71672	0.99956	1	1	1
80	0.58044	0.99289	1	1	1
90	0.45588	0.98964	1	1	1
100	0.39074	0.98332	1	1	1
AVERAGE:	0.78066	0.99472	1	1	1
Size	**Computational times (seconds)**				
n	RHC	GRASP	SA	VNS	RVNS
10	4.81	0.21	0.31	0.24	0.04
20	666.37	11.18	1.71	7.48	0.42
30	2749.09	1.12	7.93	9.95	0.86
40	3272.73	100.32	24.66	39.01	8.41
50	3331.79	663.61	42.62	187.35	10.61
60	3411.07	517.13	181.30	180.9	38.72
70	3569.81	838.13	115.68	272.49	38.86
80	3524.89	266.56	248.79	723.59	66.88
90	3419.11	871.79	255.65	570.28	101.51
100	3492.53	978.05	3491.96	932.67	115.013
AVERAGE:	2744.22	424.81	437.06	292.396	38.13

and RVNS always produced the exact solutions ($S_t = 1$) for all the instances considered in Table 5.1, in very short computational times. In particular, RVNS was always faster than the other heuristics among all the datasets, indicating an optimal tuning between intensification and diversification of the search process,

while SA was extremely slow for the instance $n = 100$ with a time of 3491.96 sec. The performance of GRASP is between the poor performing RHC and the high performing SA, VNS, RVNS. The solution quality of GRASP decreases as the problem instance increases, but not as badly as for RHC, while the computational times are comparable with those of SA. Summarizing, the average values of the normalized tree benefit score of the metaheuristics among the instances of Table 5.1, ranking from the best to the worst performing algorithm, are: RVNS = 1, VNS = 1, SA = 1, GRASP = 0.99472, RHC = 0.78066 (in case of ties in the average normalized tree benefit scores, an algorithm is considered better than another if it has a smaller average computational time).

5.4.2 Testing on examples from nature

In evolutionary biology the timing and origin of the major extant *placental clades* (groups of organisms that have evolved from a common ancestor) continues to fuel debate and research (Rokas et al., 2003). As the complete genomes of various species become available, it has become possible to do whole genome phylogeny (Felsenstein, 1981; Ben-Dor et al., 1998). Traditional phylogenetic methods on individual genes depended on multiple alignment of the related proteins and on the model of evolution of individual amino acids. Neither of these is practically applicable to the genome level. In absence of such models, a method which can compute the shared information between two sequences is useful because biological sequences encode information, and the occurrence of evolutionary events (such as insertions, deletions, point mutations, rearrangements, and inversions) separating two sequences sharing a common ancestor will result in the loss of their shared information (Rokas et al., 2003).

This section considers a study in genomics with DNA sequences of different placental mammalian species, obtained from (Cilibrasi, 2007a,b). The distance matrices from the genomic data were computed as NCD distances by using the automated software method by Cilibrasi and Vitányi (2005, 2006), who downloaded the whole mitochondrial genomes of the placental mammalian species from the GenBank Database on the World Wide Web. Three sets of data with $n = 10$,

$n = 24$, and $n = 34$ were considered, with a *max-CPU-time* for the heuristics of one hour (3600 sec). Computational results are reported in Table 5.2.

Table 5.2: Computational results for examples from nature (DNA sequences of different placental mammalian species) (*max-CPU-time* for heuristics = 36000 sec)

Size	**Normalized tree benefit score**				
n	RHC	GRASP	SA	VNS	RVNS
10	0.99979	0.99979	0.99979	0.99979	0.99979
24	0.99575	0.99588	0.99588	0.99588	0.99588
34	0.98488	0.98782	0.98792	0.98792	0.98792
AVERAGE:	0.99347	0.99450	0.99453	0.99453	0.99453
Size	**Computational times (seconds)**				
n	RHC	GRASP	SA	VNS	RVNS
10	6.72	0.078	1.84	0.172	0.00
24	934.42	16.56	6.69	4.48	2.08
34	3352.01	228.702	32.78	65.28	10.61
AVERAGE:	1431.05	81.78	13.77	23.31	4.23

Looking at the table, all the heuristics obtained almost the same normalized tree benefit scores. However, as in the previous set of instances, RHC was considerably slower than the other metaheuristics, showing limited intensification and diversification capabilities of the search process. The average values of the normalized tree benefit scores, ranking from the best to the worst performing algorithm, are: RVNS = 0.99453, VNS = 0.99453, SA = 0.99453, GRASP = 0.99450, RHC = 0.99347 (again, in case of ties in the average normalized tree benefit scores, an algorithm is considered better than another if it has a shorter average computational time). RHC obtains the worst average normalized tree benefit score, and the worst average computational running time (1431.05 sec). The best performance in terms of solution quality and computational running time is obtained again by RVNS. Figure 5.10 shows the full unrooted binary tree t obtained by RVNS for the instance with $n = 24$ placental mammals, with a normalized tree benefit score of $S_t = 0.99588$ obtained in just 2.08 sec. The interpretation is that objects in a given subtree are pairwise closer (more similar) to

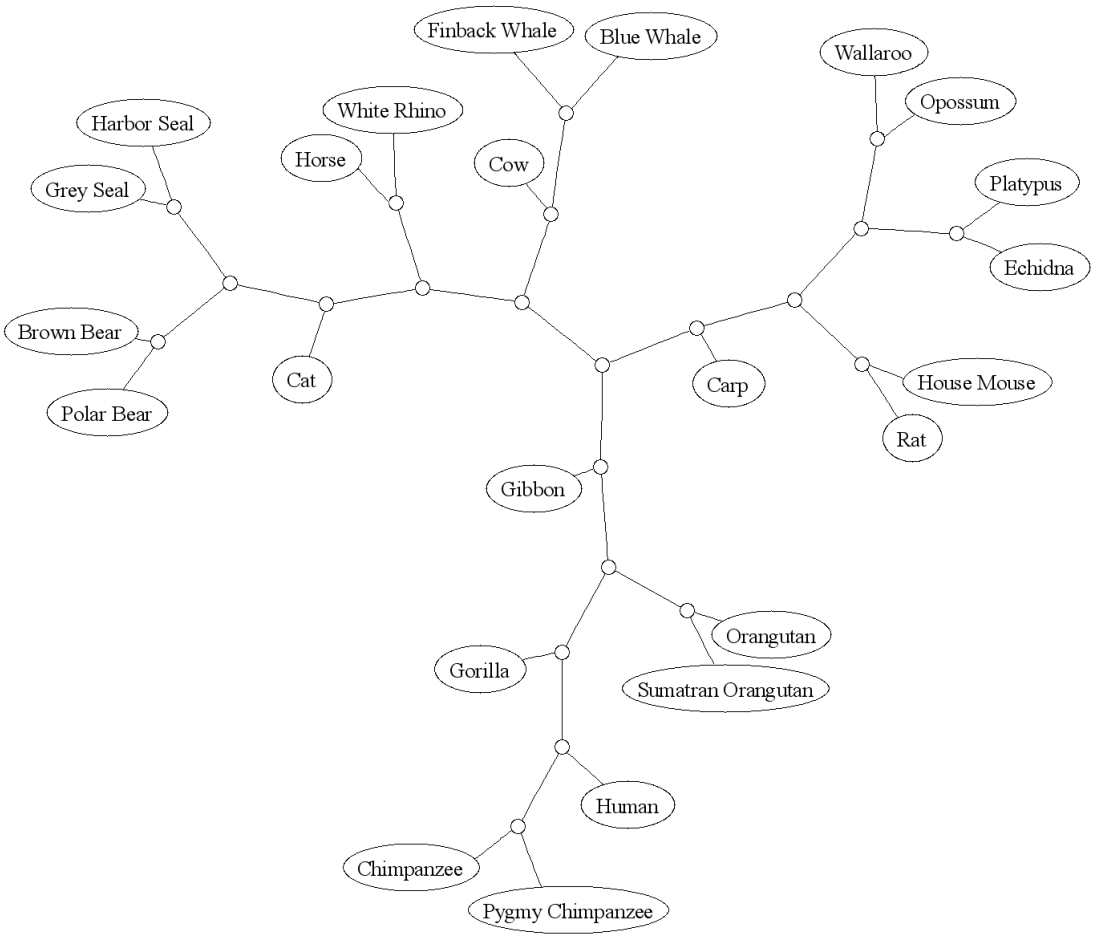


Figure 5.10: The full unrooted binary tree t obtained by RVNS for the instance with $n = 24$ mammals, with a normalized tree benefit score of $S_t = 0.99588$ obtained in 2.08 sec.

each other than any of those objects in a disjoint subtree. Roughly, it is possible to identify the following groups among the placental mammals considered: *Primates* (Chimpanzee, Pygmy Chimpanzee, Human, Gorilla, Orangutan, Sumatran Orangutan, Gibbon); *Ferungulates* (Grey Seal, Harbor Seal, Brown Bear, Polar Bear, Cat, Horse, White Rhino, Cow, Finback Whale, Blue Whale); *Marsupionta* (Wallaroo, Opossum, Platypus, Echidna, House Mouse, Rat, Carp).

5.4.3 Testing on geographic distances

In this section, the metaheuristics were compared by considering some famous cities as objects to cluster. Thus, the distances between the objects are real geographic distances between the cities considered, normalized in the interval $[0, 1]$. We considered data instances with a number of objects n from 13 to 37, setting the *max-CPU-time* for the heuristics to one hour (3600 sec). Computational results are presented in Table 5.3.

Table 5.3: Computational results for geographic distances between cities (*max-CPU-time* for heuristics = 36000 sec)

Size	**Normalized tree benefit score**				
n	RHC	GRASP	SA	VNS	RVNS
13	0.96843	0.96843	0.96843	0.96843	0.96843
22	0.93507	0.93507	0.93507	0.93507	0.93507
24	0.92459	0.92429	0.92459	0.92459	0.92459
25	0.98760	0.98760	0.98760	0.98760	0.98760
35	0.98203	0.94395	0.98367	0.98367	0.98367
37	0.90552	0.88094	0.91973	0.91973	0.91973
AVERAGE:	0.95054	0.94004	0.95318	0.95318	0.95318
Size	**Computational times (seconds)**				
n	RHC	GRASP	SA	VNS	RVNS
13	67.46	6.21	5.42	0.55	0.27
22	1365.22	198.49	15.42	17.26	3.14
24	803.61	311.80	15.46	17.61	3.29
25	1752.89	25.36	8.98	52.81	2.84
35	2686.73	996.63	89.72	43.27	10.75
37	3434.06	1480.82	74.94	53.53	32.94
AVERAGE:	1684.99	503.22	34.99	30.84	8.87

We observe that the normalized tree benefit score obtained by the heuristics deteriorates by increasing the size n of the problem instance to cluster, as a result of a higher inconsistency produced by the full unrooted binary tree representation of the distance matrices used by the quartet method. The average values of the

normalized tree benefit scores, ranking from the best to the worst performing algorithm, are: RVNS = 0.95318, VNS = 0.95318, SA = 0.95318, RHC = 0.95054, GRASP = 0.94004 (as in the previous sections, in case of ties in the average normalized tree benefit scores, an algorithm is considered better than another if it has a shorter average computational time). Again, the best performances are obtained by RVNS, VNS, and SA, which obtain the largest normalized tree benefit scores in the shortest computational running times. In particular, the best performing heuristic is again RVNS, which is considerable faster (average computational time: 8.87 sec) than VNS and SA (average computational times: 30.84 sec and 34.99 sec, respectively). The performance of RHC and GRASP are quite poor. RHC is considerably slower than all the other metaheuristics (average computational time: 1684.99 sec), but it produces slightly better solutions with respect to GRASP in terms of normalized tree benefit score. For these data instances, GRASP produces solutions of poor quality although being faster than RHC, as a result of a poor diversification capability and an excessive intensification capability which sometimes do not allow the search process to escape from local optima.

In Figure 5.11, the full unrooted binary tree t obtained by RVNS for the instance with $n = 37$, which contains the distances among some famous European cities, is illustrated. The normalized tree benefit score of this example is $S_t = 0.91973$, obtained by RVNS in 32.94 sec. Figure 5.11 represents an intuitive visual example of the way of clustering data hierarchically by means of the quartet method. Cities that have short relative distances are assigned to close positions of the boron tree. For instance, the Italian cities of Rome, Naples, Venice, Genoa, are placed in close positions of t, followed by Nice (that belongs to France but is extremely close to the Italian border) and Turin, and then Milan. Similarly, the Netherlands cities of Amsterdam, The Hague, Rotterdam, Antwerp, and Brussels belongs to the same group, and so on. Similarly, Figure 5.12 shows the full unrooted binary tree t obtained by RVNS for the instance with $n = 25$, which contains the distances among some famous Asian cities. The normalized tree benefit score of this example is $S_t = 0.98760$, obtained by RVNS in 2.84 sec.

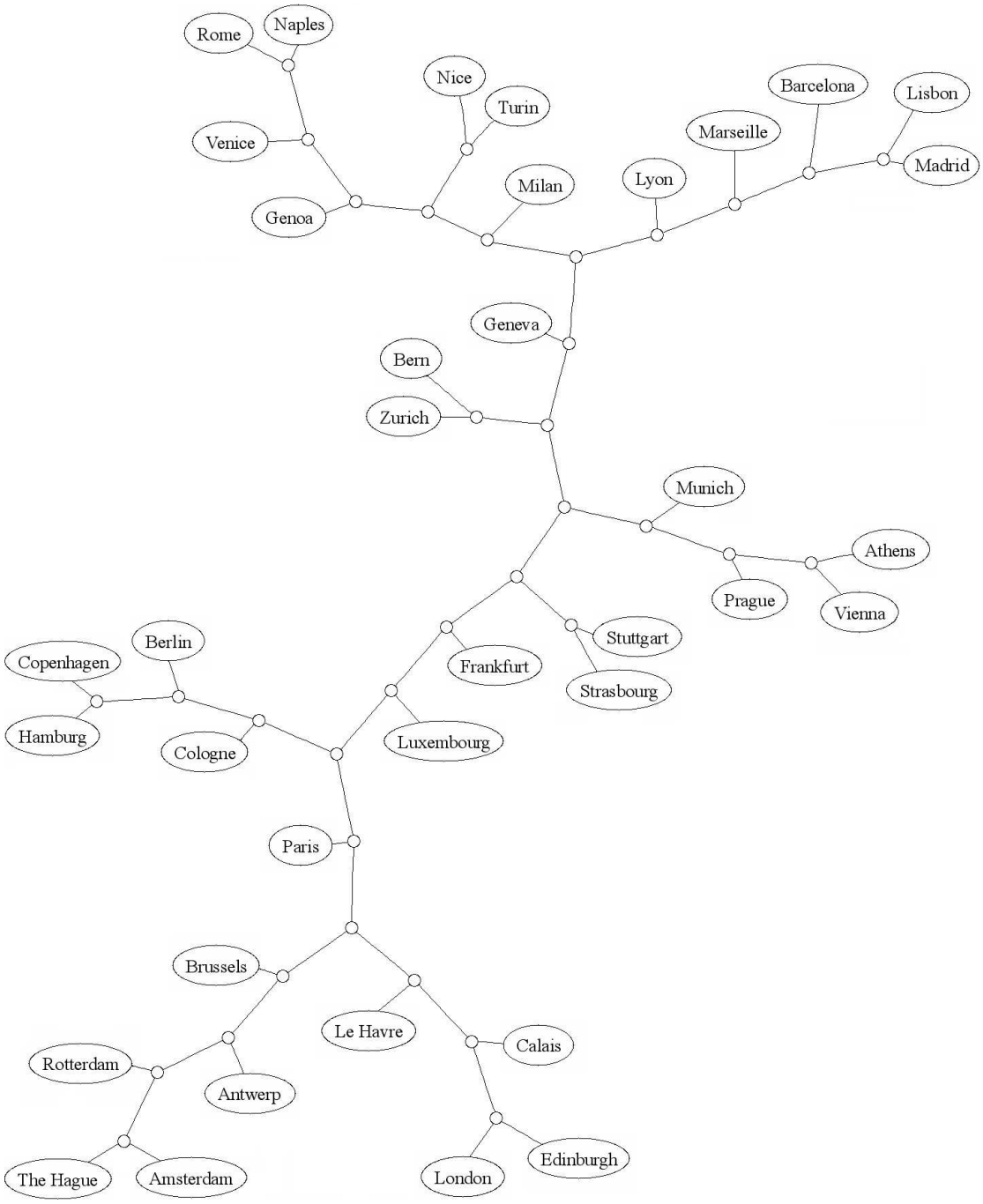


Figure 5.11: The full unrooted binary tree t with $S_t = 0.91973$ obtained by RVNS in 32.94 sec for the instance with $n = 37$ European cities.

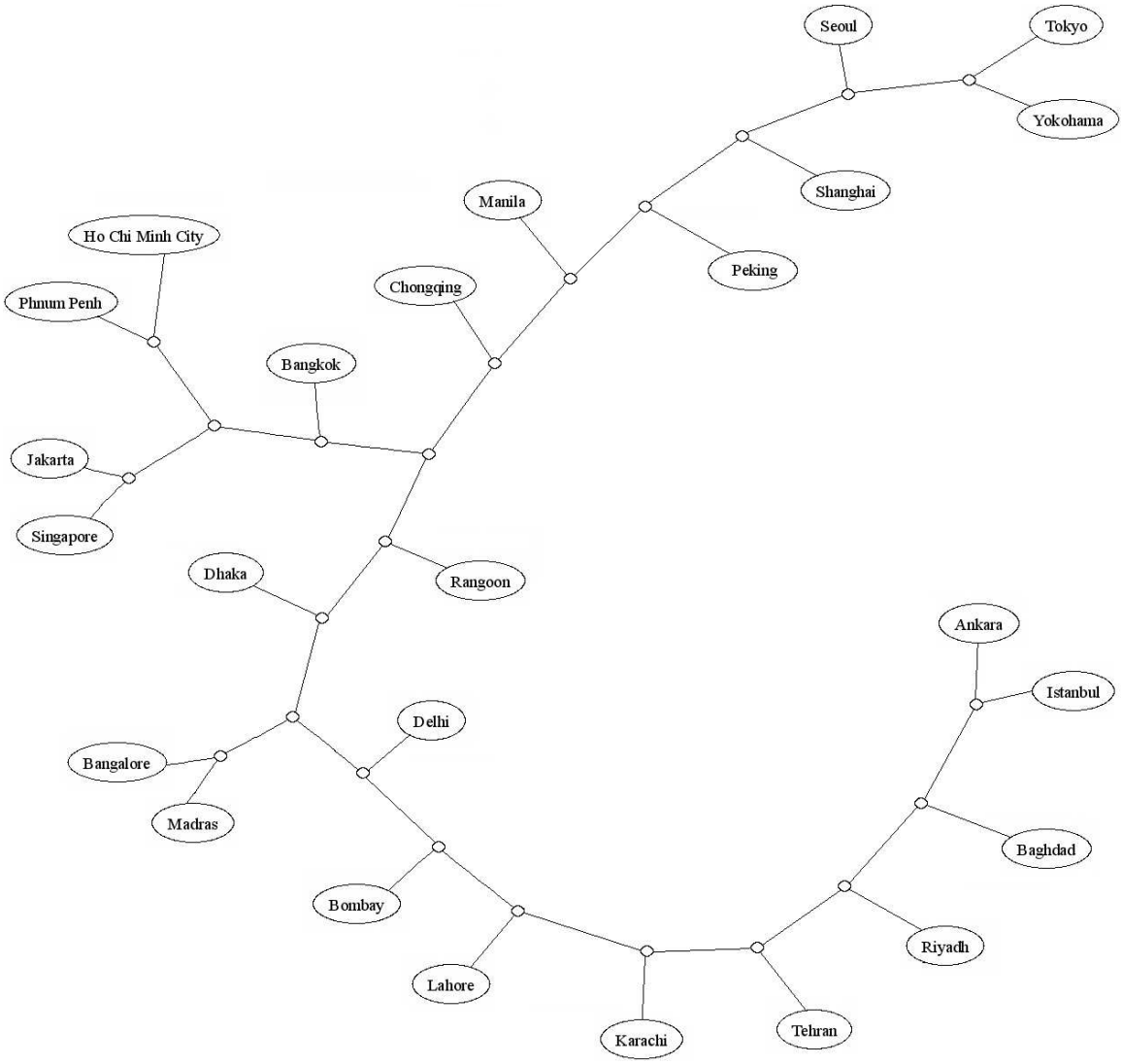


Figure 5.12: The full unrooted binary tree t with $S_t = 0.98760$ obtained by RVNS in 2.84 sec for the instance with $n = 25$ Asian cities.

5.4.4 Testing on data extracted from the World Wide Web

In this section, we consider data obtained by mining of the WWW through an automatic web information extraction method by Geleijnse et al. (2006). Specifically, we have focussed on data concerning musical artists, in order to easily show subjective artist categories such as genre of the music that they produce.

Geleijnse et al. (2006) use the assumption that related artists often share the same category (*working hypothesis*). Alternatively, if two artists are both known for the same category (e.g. romantic music), it is expected that they would occur often in the same context within the World Wide Web. To obtain a metric which expresses the similarity between each pair of artists a and b, selected from a given set of artists A, Geleijnse et al. (2006) count the number of *co-occurrences* of a and b, $co(a, b)$, within the WWW by means of, either a *Page-count-based mapping* (PCM), a *Pattern-based mapping* (PM), or a *Document-based mapping* (DM). In this chapter a PCM is used, where the number of co-occurrences of a pair of artists $(a, b) \in A$ is the number of Google hits for queries "a", "b". Note that the estimated numbers of Google hits can fluctuate which may lead to unexpected results (Geleijnse et al., 2006).

After having collected the number of co-occurrences for each pair of artists in A, Geleijnse et al. (2006) derive a similarity metric among the artists by defining a scoring function, $T(a, b)$, between two different artists $(a, b) \in A$, as follows:

$$T(a,b) = \frac{co(a,b)}{1 + \sum_{y \in A, y \neq a} co(a,y) \cdot \sum_{x \in A, x \neq b} co(x,b)}. \tag{5.12}$$

This similarity metric is inspired by the theory of "pointwise mutual information" (for more details see (Manning and Schütze, 1999)). Note that this similarity metric is symmetric in its arguments and that all the elements in the diagonal are forced to 1. As the metric between two objects approaches zero, the less the similarity between the two artists. For each $(a, b) \in A$, the similarity metric $T(a, b)$ is converted into a distance metric $d(a, b)$, as follows:

$$d(a,b) = 1 - T(a,b). \tag{5.13}$$

In this way, a symmetric distance matrix, suitable input for the quartet method, is produced.

Our results are presented in Table 5.4, which considers data instances with number of artists n from 15 to 224. For small problem instances ($n \leq 100$), *max-CPU-time* for the heuristics is set to one hour (3600 sec), while for the last two large instances with $n > 100$, (i.e. $n = 150$ and $n = 224$), a *max-CPU-time* of 10 hours (36000 sec) is imposed. The average values of the normalized

Table 5.4: Computational results for data concerning distances between musical artists extracted from the World Wide Web (*max-CPU-time* for heuristics = 36000 sec)

Size	**Normalized tree benefit score**				
n	RHC	GRASP	SA	VNS	RVNS
15	0.95273	0.95273	0.95273	0.95273	0.95273
25	0.92218	0.92080	0.92218	0.92190	0.92218
50	0.75077	0.90511	0.92244	0.92244	0.92252
100	0.43476	0.85988	0.88736	0.88212	0.88731
150	0.42591	0.74047	0.84214	0.84132	0.84614
224	0.40341	0.71262	0.80045	0.80080	0.80849
AVERAGE:	0.64829	0.84860	0.88788	0.88689	0.88990
Size	**Computational times (seconds)**				
n	RHC	GRASP	SA	VNS	RVNS
15	41.21	1.25	1.63	0.78	0.25
25	1107.81	12.68	13.13	19.13	2.34
50	3469.89	103.51	65.79	112.19	35.37
100	3525.28	94.91	3033.53	2735.55	884.31
150	34809.11	4580.45	34261.51	24425.52	17896.95
224	21652.34	34357.02	24292.81	35360.20	35299.22
AVERAGE:	10767.61	6524.97	10278.07	10442.23	9019.74

tree benefit scores, ranking from the best to the worst performing algorithm, are: RVNS = 0.88990, SA = 0.88788, VNS = 0.88689, GRASP = 0.84860, RHC = 0.64829; while the average computational running times, from the fastest to the slowest, are (in sec): GRASP = 6524.97, RVNS = 9019.74, SA = 10278.07, VNS = 10442.23, RHC = 10767.61.

As in the previous experimental analysis, the table shows approximately the same relative behaviour for all the metaheuristics considered. RVNS obtains the solutions with the best normalized tree benefit scores, followed by SA and VNS, then GRASP, and finally RHC, which produces extremely poor results (average normalized tree benefit score: 0.64829). In addition, the computational running times (average computational time: 10767.61 sec) are poor. For the data

instances considered in this section, GRASP is on average faster than the other algorithms, because it converges prematurely to local optima from where it is not able to escape, producing solutions of poor quality. SA and VNS produce results close to those of RVNS in terms of solution quality and computational running times, indicating an optimal tuning between intensification and diversification of the search process, which evidently is not obtained by GRASP and RHC. VNS obtains slightly worse solutions than those obtained by SA, perhaps lacking a bit in terms of exploration of the search space with respect to the SA approach. As in Section 5.4.3, it is interesting to note the effect of the data inconsistency in the normalized tree benefit score obtained by the heuristics as n becomes larger. For example, for $n = 224$, it is not possible to produce a solution having normalized tree benefit score larger than 0.80849, that is obtained by RVNS in a very high computational time (35299.22 sec)! This further analysis underlines the limit of the quartet method to process data instances larger than, approximately, $n = 100$ objects to cluster. For $n > 100$, the heuristics often produce results with inadequate normalized tree benefit scores in very high computational running times.

Summarizing, for all the problem instances considered containing objects to cluster of different nature, analysed in Sections 5.4.1 - 5.4.4, all the metaheuristics that we propose (RVNS, VNS, SA, GRASP) clearly outperformed the Randomized Hill Climbing by Cilibrasi and Vitányi (2005, 2006), the heuristic recommended in the literature for the quartet method of hierarchical clustering. In particular, the best performance in terms of normalized tree benefit score and computational running time were obtained by RVNS. This is the most effective heuristic for the minimum quartet tree cost problem. As shown in our experiments, RVNS is able to produce the most accurate full unrooted binary trees, capable of representing the symmetric distance matrices. From our analysis, it has been shown that our Reduced Variable Neighbourhood Search is fast and particularly effective for the quartet method of hierarchical clustering.

5.5 Conclusions

In this chapter we considered the quartet method of hierarchical clustering which, given a set of objects to be classified and a symmetric distance matrix containing their pairwise distances, produces the optimal hierarchy of the objects without knowing a priori the number of clusters to be produced. The optimal hierarchy produced by the quartet method is visualized by means of a special dendrogram topology, called full unrooted binary tree (or boron tree, or ternary tree), which visually represents the distance matrix as closely as possible, according to a specified cost evaluation.

In order to produce the optimal hierarchy through a boron tree, the quartet method of hierarchical clustering needs to solve a graph optimization problem, called the minimum quartet tree cost problem. A Greedy Randomized Adaptive Search Procedure, a Simulated Annealing approach, a Variable Neighbourhood Search, and a Reduced Variable Neighbourhood Search have been presented for this problem. Considering a wide range of problem instances, we compared these metaheuristics with the Randomized Hill Climbing by Cilibrasi and Vitányi (2005, 2006), the most popular heuristic in the literature for the quartet method of hierarchical clustering. Based on this experimental analysis, all the proposed procedures clearly outperformed the Randomized Hill Climbing and, in particular, the best performance was obtained by Reduced Variable Neighbourhood Search. Reduced Variable Neighbourhood Search was shown to be a fast, simple, and particularly effective metaheuristic for the quartet method of hierarchical clustering, obtaining high-quality solutions in short computational running times. This analysis provides further evidence of the ability of variable neighbourhood heuristics to deal with NP-hard combinatorial problems.

Future research will consist of trying to further improve the performance of these procedures (for example through hybridization with other metaheuristics) particularly for large instances of the problem. Furthermore, an exact approach to the minimum quartet tree cost problem is currently under study in order to produce the optimal hierarchy of the objects by means of the quartet method of hierarchical clustering. However, as the problem is NP-hard, an exact approach will be successful, in practise, just for very small instances of the problem.

I don't want to achieve immortality through my work. I want to achieve it through not dying.

Woody Allen

Chapter 6

Conclusions

The research reported in this thesis has focussed on the development and application of metaheuristics for problems in graph theory. The aim of this work is twofold. On the one hand, it has sought to bring together, in a systematic and consistent way, several features of different metaheuristic techniques. The most important and efficient metaheuristics, from classical to novel approaches, were presented in Chapter 2. This chapter covered many theoretical and practical aspects of metaheuristics, outlining their main concepts and components, similarities and differences, advantages and disadvantages. Different classes of metaheuristics were specified and, in particular, the most important single-solution and population-based metaheuristics were presented and extensively discussed. The two very significant forces of intensification and diversification that play an important role in the behaviour of a metaheuristic were highlighted. The importance of hybridization and integration of metaheuristics were discussed. In addition, the thesis addresses some recently proposed combinatorial optimization problems formulated on graphs, and presents appropriate metaheuristics to obtain near-optimal solutions. These problems constitute some new and interesting research areas, and are able to represent many real-world problems.

Several metaheuristics for the *minimum labelling spanning tree* (MLST) problem are presented in Chapter 3. Specifically, the metaheuristics recommended in the literature, the Modified Genetic Algorithm (MGA) by Xiong et al. (2006) and the Pilot Method (PILOT) by Cerulli et al. (2005), were examined and

implemented. Some new implementations of metaheuristics for the MLST problem were further proposed: a Greedy Randomized Adaptive Search Procedure (GRASP), a basic Variable Neighbourhood Search (VNS), and a hybrid local search method (HYBRID) obtained by combining Variable Neighbourhood Search with Simulated Annealing (SA). The nonparametric statistical tests of Friedman (1940) and Nemenyi (1963) were applied, in order to compare the performance of the algorithms considered on a wide range of problem instances. The results indicated that VNS, HYBRID, and GRASP have significantly better performance than the other methods recommended in the literature with respect to solution quality and running time. Furthermore, this result has been reinforced by comparing the metaheuristics with an exact approach. In addition, it was shown that VNS is particularly recommended for the proposed problem because of its simplicity and its ability to obtain high-quality solutions in short computational running times.

A similar study was presented in Chapter 4 for the *minimum labelling Steiner tree* (MLSteiner) problem, another graph problem related to the minimum labelling spanning tree problem and to the well-known Steiner tree problem. Some metaheuristics for the problem were presented: a Greedy Randomized Adaptive Search Procedure (GRASP), a Discrete Particle Swarm Optimization (DPSO), a Variable Neighbourhood Search (VNS), and a hybrid local search method (HYBRID) obtained by combining Variable Neighbourhood Search with Simulated Annealing (SA). Considering a wide range of problem instances, these metaheuristics were compared to the Pilot Method (PILOT) by Cerulli et al. (2006), the most popular MLSteiner heuristic in the literature. Based on this experimental analysis, all the proposed procedures clearly outperformed PILOT and, in particular, the best performance was obtained by VNS, HYBRID, and GRASP. In addition, it was shown that VNS is the most effective approach to the problem, thanks to the following features: ease of implementation, user-friendly code, high-quality of the solutions, and shorter computational running times.

Finally, Chapter 5 considered the *quartet method* of hierarchical clustering which, given a set of objects to be classified and a symmetric distance matrix containing their pairwise distances, produces the optimal hierarchy of the objects

without knowing a priori the number of clusters to be produced. The optimal hierarchy produced by the quartet method is visualized by means of a special dendrogram topology, called a full unrooted binary tree (or boron tree, or ternary tree), which visually represents the distance matrix as closely as possible, according to a specified cost evaluation. Because the quartet method is based on an NP-hard graph optimization problem, called *minimum quartet tree cost* (MQTC) problem, any practical approach to obtain or approximate the optimal solutions requires heuristics. Thus, a Greedy Randomized Adaptive Search Procedure (GRASP), a Simulated Annealing (SA) approach, a Variable Neighbourhood Search (VNS), and a Reduced Variable Neighbourhood Search (RVNS) were presented for the MQTC problem. The performance of the proposed algorithms was tested through extensive computational experiments and comparison with the Randomized Hill Climbing (RHC) by Cilibrasi and Vitányi (2005, 2006), the most popular heuristic in the literature for the quartet method of hierarchical clustering. Based on this experimental analysis, all the proposed procedures clearly outperformed RHC and, in particular, the best performance was obtained by RVNS. Reduced Variable Neighbourhood Search was shown to be a fast, simple, and particularly effective metaheuristic for the quartet method of hierarchical clustering, obtaining the best performance in terms of solution quality and computational running time.

This thesis is intended to provide both researchers and practitioners with a broadly applicable, up to date coverage of metaheuristic methodologies that have proven to be successful in a wide variety of graph theoretic models, and that hold particular promise for success in the future. The study of the graph problems considered in this thesis represent some new and relevant research areas in combinatorial optimization and metaheuristics. The metaheuristics used to solve these problems serve as illustrations in showing the importance and the potential of metaheuristic approaches to deal with these classes of problems. In addition, thorough analysis of the implementation of these methods provided insights into the implementation of metaheuristic strategies for other complex graph problems. With this thesis, the author hopes to encourage an even wider adoption of metaheuristic methods for solving graph problems, and to stimulate research that may lead to additional innovations in metaheuristic procedures.

All men by nature desire to know.

ARISTOTLE

Appendix A

Computational complexity

Most combinatorial optimization problems can be classified as problems in the complexity class **P** and **NP-hard** problems (Garey and Johnson, 1979). In computational complexity theory, the class **P** consists of all those decision problems that can be solved on a "deterministic sequential Turing-machine" in an amount of time that is bounded by a polynomial $p(|x|)$ in the size of the input x; the class **NP** consists of all those decision problems whose positive solutions x can be verified in polynomial time $p(|x|)$ given the right information, or equivalently, whose solution can be found in polynomial time p on a "non-deterministic Turing-machine".

We say that there is a "polynomial time many-one reduction" from a decision problem L_1 to a decision problem L_2, denoted by $L_1 \propto L_2$, if there exists a function f that is computable in polynomial time such that

$$x \in L_1 \Leftrightarrow f(x) \in L_2. \tag{A.1}$$

A problem L_1 in NP is said to be **NP-Complete** if for every $L_2 \in$ NP, $L_2 \propto L_1$. Because $\propto$ is transitive, to prove NP-Completeness of L_1 it is enough to show that some NP-Complete problem L_2 satisfies $L_2 \propto L_1$.

A version of reducibility that may be applied to problems that are not necessarily decision problems is now defined. A problem P_1 is "Turing reducible" to P_2, written $P_1 \propto_T P_2$ if the existence of a polynomial time algorithm for P_2 implies that there is a polynomial time algorithm for P_1. Given a subroutine for P_2 running in polynomial time, we can solve P_1 in polynomial time.

A problem P is **NP-hard** (Non-deterministic Polynomial-time hard) if $\exists L \in NP-Complete$ such that $L \propto P$. The notion of NP-hardness plays an important role in the discussion about the relationship between the complexity classes P and NP, because if it is possible to find an algorithm that solves one of these problems L_1 in polynomial time, it should be possible to construct a polynomial time algorithm for any problem $L_2 \in$ NP by first performing the reduction from L_2 to L_1 and then running the polynomial time algorithm. This would be equivalent stating "P = NP", and thus to solve the biggest open question in theoretical computer science concerning the relationship between these two classes. However it is widely suspected that there are no polynomial time algorithms for NP-hard problems, although this has never been proved (Garey and Johnson, 1979).

Figure A.1 illustrates the complexity classes NP, P and NP-Complete, assuming that P $\neq$ NP. The NP-Complete complexity class contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. For more details see (Garey and Johnson, 1979) in which many NP-Complete problems are classified.

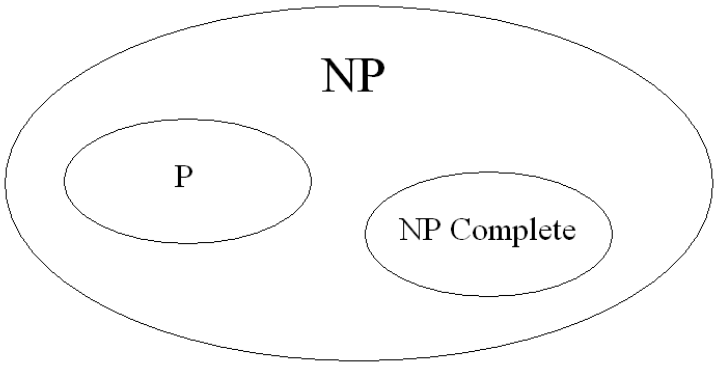


Figure A.1: Diagram of complexity classes.

If the facts don't fit the theory, change the facts.

ALBERT EINSTEIN

Appendix B

Statistical tests

Friedman test (Friedman, 1940): The Friedman test is a non-parametric statistical test that examines the existence of significant differences between the performance of multiple algorithms over different datasets. Given k algorithms and N datasets, it ranks the algorithms for each dataset separately, and tests whether the measured average ranks are significantly different from the mean rank. The statistic used by Friedman (1940) is

$$\chi_F^2 = \frac{12 \cdot N}{k \cdot (k+1)} \cdot \left[\sum_j R_j^2 - \frac{k \cdot (k+1)}{4} \right], \tag{B.1}$$

which follows a Chi-Square distribution with $(k-1)$ degrees of freedom.

Iman and Davenport (1980) developed a more powerful version of the Friedman test by considering the following statistic:

$$F_F^2 = \frac{(N-1) \cdot \chi_F^2}{N \cdot (k-1) - \chi_F^2}\ , \tag{B.2}$$

which is distributed according to the F-distribution with $(k-1)$ and $(k-1)\cdot(N-1)$ degrees of freedom. For more details, see (Demśar, 2006).

Nemenyi test (Nemenyi, 1963): The Nemenyi test is used to perform pairwise comparisons of multiple algorithms over different datasets (Nemenyi, 1963). The performance of two algorithms is considered significantly different if the corresponding average ranks differ by at least the critical difference (CD):

$$CD = q_\alpha \cdot \sqrt{\frac{k \cdot (k+1)}{6 \cdot N}}\ , \tag{B.3}$$

where k is the number of the metaheuristics, N the number of datasets, q_α the critical value, and α the significance level of the statistical test. For more details, see (Demśar, 2006).

References

NOTE: At the end of each reference is a list of page numbers of this thesis in which the references are cited.

E. Aarts and J. Korst (1988). *Simulated annealing and boltzmann machines: A stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Chichester. 17

E. Aarts, J. Korst, and W. Michiels (2005). Simulated annealing. In E. K. Burke and G. Kendall, editors, *Search methodologies: Introductory tutorials in optimization and decision support techniques*, pages 187–210. Springer Verlag. 17, 133

E. H. L. Aarts, J. H. M. Korst, and P. J. M. V. Laarhoven (1997). Simulated annealing. In E. H. L. Aarts and J. K. Lenstra, editors, *Local search in combinatorial optimization*, pages 91–120. John Wiley & Sons, Chichester. 18, 70

B. Al-kazemi and C. K. Mohan (2002). Multi-phase discrete particle swarm optimization. In *Fourth International Workshop on Frontiers in Evolutionary Algorithms*, Kinsale, Ireland. 66

D. Avis, A. Hertz, and O. Marcotte (2005). *Graph theory and combinatorial optimization*. Springer-Verlag, New York. 1, 7

R. Battiti, M. Brunato, and F. Mascia (2008). *Reactive search and intelligent optimization*, volume 45 of *Operations Research/Computer Science Interfaces Series*. Springer-Verlag, New York. 170

A. Ben-Dor, B. Chor, D. Graur, R. Ophir, and D. Pelleg (1998). Constructing phylogenies from quartets: Elucidation of eutherian superordinal relationships. *Journal of Computational Biology*, 5(3):377–390. 150, 176

V. Berry, T. Jiang, P. Kearney, M. Li, and T. Wareham (1999). Quartet cleaning: Improved algorithms and simulations. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Algorithms - Proceedings 7th European Symposium on Algorithms (ESA'99)*, volume 1643 of *Lecture Notes in Computer Science*, pages 313–324. Springer-Verlag, Berlin, Germany. 150

C. Blum and A. Roli (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308. 55, 63, 68, 69, 70

T. Brüggemann, J. Monnot, and G. J. Woeginger (2003). Local search for the minimum label spanning tree problem with bounded colour classes. *Operations Research Letters*, 31:195–201. 76

V. Cerny (1985). Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51. 15

R. Cerulli, A. Fink, M. Gentili, and S. Voß (2005). Metaheuristics comparison for the minimum labelling spanning tree problem. In B. L. Golden, S. Raghavan, and E. A. Wasil, editors, *The Next Wave on Computing, Optimization, and Decision Technologies*, pages 93–106. Springer-Verlag, New York. 74, 77, 78, 81, 100, 111, 119, 187

R. Cerulli, A. Fink, M. Gentili, and S. Voß (2006). Extensions of the minimum labelling spanning tree problem. *Journal of Telecommunications and Information Technology*, 4:39–45. 113, 115, 116, 117, 119, 120, 128, 142, 188

R. S. Chang and S. J. Leu (1997). The minimum labelling spanning trees. *Information Processing Letters*, 63(5):277–282. 71, 74, 75

R. Cilibrasi (2007a). *Statistical inference through data compression*. Ph.D. thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam, The Netherlands. 172, 176

R. Cilibrasi (2007b). The Complearn toolkit. [online]. URL `http://www.complearn.org/`. 172, 176

R. Cilibrasi and P. M. B. Vitányi (2005). Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545. 146, 147, 150, 151, 152, 153, 154, 155, 156, 157, 173, 176, 185, 186, 189

R. Cilibrasi and P. M. B. Vitányi (2006). A new quartet tree heuristic for hierarchical clustering. In D. V. Arnold, T. Jansen, M. D. Vose, and J. E. Rowe, editors, *Theory of Evolutionary Algorithms*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2006/598. 146, 147, 151, 152, 153, 154, 155, 156, 157, 173, 176, 185, 186, 189

R. Cilibrasi and P. M. B. Vitányi (2007). The google similarity distance. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):370–383. 150

R. Cilibrasi, P. M. B. Vitányi, and R. de Wolf (2004). Algorithmic clustering of music based on string compression. *Computer Music Journal*, 28(4):49–67. 150

A. Colorni, M. Dorigo, and V. Maniezzo (1992). Distributed optimization by ant colonies. In F. J. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 134–142. The MIT Press, Cambridge, MA. 56

S. Consoli (2007a). Test datasets for the minimum labelling spanning tree problem. [online]. URL http://www.sergioconsoli.com/MLSTP.htm. 102

S. Consoli (2008). Test datasets for the quartet method of hierarchical clustering. [online]. URL http://www.sergioconsoli.com/Quartet.htm. 172

S. Consoli (2007b). Test datasets for the minimum labelling Steiner tree problem. [online]. URL http://www.sergioconsoli.com/MLSteiner.htm. 135

S. Consoli, K. Darby-Dowman, G. Geleijnse, J. Korst, and S. Pauws (2008a). Heuristic approaches for the quartet method of hierarchical clustering. *IEEE Transactions on Knowledge and Data Engineering, submitted.* 5, 147

S. Consoli, K. Darby-Dowman, N. Mladenović, and J. A. Moreno-Pérez (2008b). Greedy randomized adaptive search and variable neighbourhood search for the minimum labelling spanning tree problem. *European Journal of Operational Research, accepted for publication.* doi: 10.1016/j.ejor.2008.03.014. 5

S. Consoli, K. Darby-Dowman, N. Mladenović, and J. A. Moreno-Pérez (2008c). Solving the minimum labelling spanning tree problem using hybrid local search. *Optimization Methods and Software, submitted.* Special Issue EURO XXII conference. 5

S. Consoli, K. Darby-Dowman, N. Mladenović, and J. A. Moreno-Pérez (2008d). Variable neighbourhood search for the minimum labelling Steiner tree problem. *Annals of Operations Research, accepted for publication.* 5

S. Consoli, J. A. Moreno-Pérez, K. Darby-Dowman, and N. Mladenović (2008e). Discrete particle swarm optimization for the minimum labelling Steiner tree problem. In N. Krasnogor, G. Nicosia, M. Pavone, and D. Pelta, editors, *Nature Inspired Cooperative Strategies for Optimization*, volume 129 of *Studies in Computational Intelligence*, pages 313–322. Springer-Verlag, New York. 5

S. Consoli, J. A. Moreno-Pérez, K. Darby-Dowman, and N. Mladenović (2008f). Discrete particle swarm optimization for the minimum labelling Steiner tree problem. *Natural Computing, submitted.* Special Issue NICSO conference. 5

E. S. Correa, A. A. Freitas, and C. G. Johnson (2006). A new discrete particle swarm algorithm applied to attribute selection in a bioinformatic data set. In *Proceedings of GECCO 2006*, pages 35–42. 67

C. Darwin (1859). *The origin of species by means of natural selection or the preservation of favoured races in the struggle for life.* John Murray, London, 6th edition. 11, 36

J. Demśar (2006). Statistical comparison of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30. 108, 140, 192, 193

R. Diestel (2000). *Graph theory.* Springer-Verlag, New York. 144, 145, 146

M. Dorigo and T. T. Stützle (2004). *Ant colony optimization*. The MIT Press, Cambridge, MA. 56, 58, 62, 63

C. Duin and S. Voß (1999). The pilot method: A strategy for heuristic repetition with applications to the Steiner problem in graphs. *Networks*, 34(3):181–191. 77, 80, 119

H. Everett (1957). "Relative state" formulation of quantum mechanics. *Reviews of Modern Physics*, 29(3):454–462. 44

J. Felsenstein (1981). Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of Molecular Evolution*, 17(6):368–376. 148, 149, 176

T. A. Feo and M. G. C. Resende (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71. 21, 157

T. A. Feo and M. G. C. Resende (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133. 22, 78, 84

R. P. Feynman and A. R. Hibbs (1965). *Quantum mechanics and path integrals*. McGraw-Hill Companies. 43

R. L. Francis, L. F. McGinnis, and J. A. White (1992). *Facility layout and location: an analytical approach*. Prentice-Hall, Englewood Cliffs, New Jersey. 116

M. Friedman (1940). A comparison of alternative tests of significance for the problem of m rankings. *Annals of Mathematical Statistics*, 11:86–92. 108, 111, 139, 188, 192

G. W. Furnas (1984). The generation of random, binary unordered trees. *Journal of Classification*, 1(1):187–233. 146

M. R. Garey and D. S. Johnson (1979). *Computers and intractability : A guide to the theory of NP-completeness*. W. H. Freeman, New York. 190, 191

M. R. Garey, R. L. Graham, and D. S. Johnson (1977). The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics*, 32: 835–859. 116

G. Geleijnse, J. Korst, and V. de Boer (2006). Instance classification using co-occurrences on the web. In *Proceedings of the ISWC 2006 workshop on Web Content Mining (WebConMine)*, Athens, GA. URL http://www.dse.nl/~gijsg/webconmine.pdf. 172, 182, 183

M. Gendreau and J.-Y. Potvin (2005). Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140(1):189–213. 7, 18, 74, 90, 91, 116, 148

F. Glover (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549. 18, 20, 167

F. Glover and G. A. Kochenberger (2003). *Handbook of metaheuristics*. Kluwer Academic Publishers, Norwell, MA. 7, 21, 40, 41, 42, 70, 74, 90, 116, 148, 155

F. Glover, M. Laguna, and R. Martí (2000). Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684. 53, 54

F. Glover, M. Laguna, and R. Martí (2003). Scatter search and path relinking: Advances and applications. In F. Glover and G. A. Kochenberger, editors, *Handbook of metaheuristics*, chapter 1, pages 1–36. Kluwer Academic Publishers, Norwell, MA. 53, 56

D. E. Goldberg, K. Deb, and B. Korb (1991). Don't worry, be messy. In *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 24–30, La Jolla, CA. Morgan-Kaufmann. 38

G. R. Grimwood (1994). The Euclidean Steiner tree problem: Simulated annealing and other heuristics. Master's thesis, Victoria University, Wellington, New Zealand. URL http://www.isor.vuw.ac.nz/~{}geoff/thesis.html. 116

P. Hansen and N. Mladenović (1997). Variable neighbourhood search. *Computers and Operations Research*, 24:1097–1100. 27, 86, 128, 167

P. Hansen and N. Mladenović (2001). Variable neighbourhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467. 27, 32, 86

P. Hansen and N. Mladenović (2003). Variable neighbourhood search. In F. Glover and G. A. Kochenberger, editors, *Handbook of metaheuristics*, chapter 6, pages 145–184. Kluwer Academic Publishers, Norwell, MA. 27, 30, 31, 32, 70, 86, 93, 167, 170

Y. C. Ho and D. L. Pepyne (2002). Simple explanation of the no-free-lunch theorem and its implications. *Journal of Optimization Theory and Applications*, 115(3):549–570. doi: 10.1023/A:1021251113462. 3

J. H. Holland (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, Ann Harbor. 38

J. H. Holland (1992). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. The MIT Press, Cambridge, MA. 38, 41, 78

M. Hollander and D. A. Wolfe (1999). *Nonparametric statistical methods*. John Wiley & Sons, New York, 2nd edition. 108, 140

F. K. Hwang, D. S. Richards, and P. Winter (1992). *The Steiner tree problem*. North-Holland, Amsterdam, Netherlands. 116

R. L. Iman and J. M. Davenport (1980). Approximations of the critical region of the Friedman statistic. *Communications in Statistics*, 9:571–595. 108, 109, 139, 140, 192

T. Jiang, P. Kearney, and M. Li (2000). A polynomial time approximation scheme for inferring evolutionary trees from quartet topologies and its application. *SIAM Journal on Computing*, 30(6):1942–1961. 149, 154

R. M. Karp (1975). On the computational complexity of combinatorial problems. *Networks*, 5:45–68. 116

L. Kaufman and P. J. Rousseeuw (2005). *Finding groups in data: An introduction to cluster analysis (Wiley Series in Probability and Statistics)*. John Wiley & Sons, Chichester. 143, 144

J. Kennedy and R. Eberhart (1997). A discrete binary version of the particle swarm algorithm. In *IEEE Conference on Systems, Man, and Cybernetics*, volume 5, pages 4104–4108. 66, 125

J. Kennedy and R. Eberhart (1995). Particle swarm optimization. In *Proceedings of the 4th IEEE International Conference on Neural Networks*, pages 1942–1948, Perth, Australia. 63, 65, 66, 124

J. Kennedy and R. Eberhart (2001). *Swarm Intelligence*. Morgan Kaufmann Publishers, San Francisco, CA. 63, 64, 66, 68, 124

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680. 15, 133, 162

B. Korte, H. J. Prömel, and A. Steger (1990). Steiner trees in VLSI-layout. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Paths, Flows, and VLSI-Layout*, pages 185–214. Springer-Verlag, Berlin, Germany. 116

J. Krarup and S. Vajda (1997). On Torricelli's geometrical solution to a problem of Fermat. *IMA Journal of Management Mathematics*, 8(3):215–224. 116

S. O. Krumke and H. C. Wirth (1998). On the minimum label spanning tree problem. *Information Processing Letters*, 66(2):81–85. 74, 75, 76, 80, 82

P. Larrañaga and J. A. Lozano (2001). *Estimation of distribution algorithms: A new tool for evolutionary optimization*. Kluwer Academic Publishers, Boston. 51, 52

M. Li and P. M. B. Vitányi (1997). *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag, New York, 2nd edition. 150

H. R. Lourenço, O. C. Martin, and T. Stützle (2003). Iterated local search. In F. Glover and G. A. Kochenberger, editors, *Handbook of metaheuristics*, volume 57, pages 320–353. Kluwer Academic Publishers, Norwell, MA. 24, 27

C. D. Manning and H. Schütze (1999). *Foundations of statistical natural language processing*. The MIT Press, Cambridge, MA. 183

F. J. Martínez-García and J. A. Moreno-Pérez (2008). Jumping Frogs Optimization: a new swarm method for discrete optimization. Tech. Rep. DEIOC 3/2008, Department of Statistics, O.R. and Computing, University of La Laguna, Tenerife, Spain. 67

H. Mühlenbein and G. Paaß (1996). From recombination of genes to the estimation of distributions i. binary parameters. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel problem solving from nature - PPSN IV*, volume 1141/1996 of *Lecture Notes in Computer Science*, pages 178–187. Springer-Verlag, Berlin, Germany. 50

W. Miehle (1958). Link-minimization in networks. *Operations Research*, 6:232–243. 117

N. Mladenović, J. Petrović, V. Kovačević-Vujčić, and M. Čangalović (2003). Solving spread spectrum radar polyphase code design problem by tabu search and variable neighbourhood search. *European Journal of Operational Research*, 151 (2):389–399. 170

J. A. Moreno-Pérez, J. P. Castro-Gutiérrez, F. J. Martínez-García, B. Melián, J. M. Moreno-Vega, and J. Ramos (2007). Discrete Particle Swarm Optimization for the p-median problem. In *Proceedings of the 7th Metaheuristics International Conference*, Montréal, Canada. 67, 125

P. Moscato (1989). On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Tech. Rep. 826, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, CA. 42, 43

A. Narayanan (1999). Quantum computing for beginners. In *Proceedings of the IEEE Congress on Evolutionary Computation*, volume 3, pages 2231–2238. 44, 45, 46

A. Narayanan and M. Moore (1996). Quantum-inspired genetic algorithms. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 61–66. 43, 44

P. B. Nemenyi (1963). *Distribution-free multiple comparisons*. Ph.D. thesis, Princeton University, New Jersey. 108, 111, 139, 188, 192

G. C. Onwubolu and M. Clerc (2004). Optimal operating path for automated drilling operations by a new heuristic approach using particle swarm optimisation. *International Journal of Production Research*, 42(3):473–491. 67

I. H. Osman (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41: 421–451. 163

J. Pacheco, S. Casado, and L. Nuñez (2007). Use of VNS and TS in classification: variable selection and determination of the linear discrimination function coefficients. *IMA Journal of Management Mathematics*, 18(2):191–206. 93

G. Pampara, N. Franken, and A. P. Engelbrecht (2005). Combining particle swarm optimisation with angle modulation to solve binary problems. In *Proceedings of the IEEE Congress on Evolutionary Computing*, volume 1, pages 89–96. 66

W. Pang, K. Wang, C. Zhou, and L. Dong (2004). Fuzzy discrete particle swarm optimization for solving traveling salesman problem. In *Proceedings of the 4th International Conference on Computer and Information Technology (CIT04)*, volume 1, pages 89–96. IEEE Computer Society. 67

L. S. Pitsoulis and M. G. C. Resende (2002). Greedy randomized adaptive search procedure. In P. Pardalos and M. G. C. Resende, editors, *Handbook of Applied Optimization*, pages 168–183. Oxford University Press. 22, 157

M. Pérez-Pérez, F. Almeida-Rodríguez, and J. M. Moreno-Vega (2007). A hybrid VNS-path relinking for the p-hub median problem. *IMA Journal of Management Mathematics*, 18(2):157–171. 93

J. Pugh and A. Martinoli (2006). Discrete multi-valued particle swarm optimization. In *Proceedings of IEEE Swarm Intelligence Symposium*, volume 1, pages 103–110. 67

M. G. C. Resende and C. C. Ribeiro (2003). Greedy randomized adaptive search procedure. In F. Glover and G. Kochenberger, editors, *Handbook of metaheuristics*, pages 219–249. Kluwer Academic Publishers, Norwell, MA. 24, 78, 84, 85, 123

A. Rokas, B. L. Williams, N. King, and S. B. Carroll (2003). Genome-scale approaches to resolving incongruence in molecular phylogenies. *Nature*, 425 (6960):798–804. 176

B. R. Secrest (2001). Traveling salesman problem for surveillance mission using particle swarm optimization. Master's thesis, School of Engineering and Management of the Air Force Institute of Technology, USA. 67

P. W. Shor (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 124–134. 44

M. A. Steel (1992). The complexity of reconstructiong trees from qualitative characters and subtrees. *Journal of Classification*, 9:91–116. 149, 154

K. Strimmer and A. von Haeseler (1996). Quartet puzzling: A quartet maximum-likelihood method for reconstructing tree topologies. *Moleculare Biology and Evolution*, 13(7):964–969. 150

T. Stützle (2006). Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539. doi: 10.1016/j.ejor.2005.01.066. 24, 27

T. Stützle (1999). Iterated local search for the quadratic assignment problem. Tech. Rep. AIDA-99-03, FG Intellektik, FB Informatik, TU Darmstadt, Germany. 24

H. Talbi, A. Draa, and M. Batouche (2004). A new quantum-inspired genetic algorithm for solving the travelling salesman problem. In *Proceedings of the IEEE International Conference on Industrial Technology*, volume 3, pages 1192–1197. 49

A. S. Tanenbaum (1989). *Computer networks*. Prentice-Hall, Englewood Cliffs, New Jersey. 72, 114

C. R. Reeves G. D. Smith V. J. Rayward-Smith, I. H. Osman (1996). *Modern heuristic search methods*. John Wiley & Sons, Chichester. 8

R. Van-Nes (2002). *Design of multimodal transport networks: A hierarchical approach*. Delft University Press. 72, 114

S. Voß (2000). Modern heuristic search methods for the Steiner tree problem in graphs. In D.-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner tree*, pages 283–323. Kluwer Academic Publishers, Boston. 116

S. Voß (2006). Steiner tree problems in telecommunications. In M. Resende and P.M. Pardalos, editors, *Handbook of optimization in telecommunications*, chapter 18, pages 459–492. Springer Science, New York. 116

S. Voß, S. Martello, I. H. Osman, and C. Roucairol (1999). *Meta-heuristics. Advanced and trends local search paradigms for optimization*. Kluwer Academic Publishers, Norwell, MA. 7, 8, 74, 116, 148

S. Voß, A. Fink, and C. Duin (2004). Looking ahead with the pilot method. *Annals of Operations Research*, 136:285–302. 77, 80, 119

C. Voudouris (1997). *Guided local search for combinatorial optimisation problems*. Ph.D. thesis, Department of Computer Science, University of Essex, United Kingdom. 32

C. Voudouris and E. Tsang (1999). Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113 (2):469–499. 32, 33, 35

Y. Wan, G. Chen, and Y. Xu (2002). A note on the minimum label spanning tree. *Information Processing Letters*, 84:99–101. 76

J. Weyer-Menkhoff, C. Devauchelle, A. Grossmann, and S. Grünewald (2005). Integer linear programming as a tool for constructing trees from quartet data. *Computational Biology and Chemistry*, 29(3):196–203. 150

P. Winter (1987). Steiner problem in networks: a survey. *Networks*, 17:129–167. 116

D. H. Wolpert and W. G. Macready (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82. 2

Y. Xiong, B. Golden, and E. Wasil (2005a). Worst case behavior of the mvca heuristic for the minimum labelling spanning tree problem. *Operations Research Letters*, 33(1):77–80. 76

Y. Xiong, B. Golden, and E. Wasil (2005b). A one-parameter genetic algorithm for the minimum labelling spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 9(1):55–60. 73, 77, 78

Y. Xiong, B. Golden, and E. Wasil (2006). Improved heuristics for the minimum labelling spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 10(6):700–703. 73, 77, 78, 79, 80, 111, 187