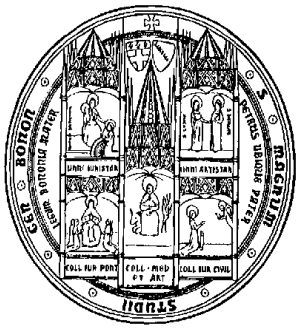

Università degli Studi di Bologna

Facoltà di Ingegneria

Dipartimento di Elettronica, Informatica e Sistemistica

Laboratorio d'Informatica Avanzata

Metaheuristics and Structure in Satisfiability Problems

Andrea Roli

Ph.D. Thesis

Tutor: Chiar.mo Prof. Maurelio Boari **Coordinator:** Chiar.mo Prof. Fabio Filicori

Supervisors: Chiar.ma Prof. Paola Mello, Chiar.ma Prof. Michela Milano

Metaheuristics and Structure in Satisfiability Problems

Andrea Roli

LIA - DEIS, Università di Bologna
Viale Risorgimento, 2 – 40136 Bologna, Italy

Abstract. The design and implementation of effective and efficient algorithms to tackle large real-world problems is nowadays a very active research field. Metaheuristics are approximate algorithms which have been proven effective in finding (near-)optimal solutions in a limited amount of time. In this thesis, we present and describe metaheuristics, focusing on their applications to the Satisfiability Problem (SAT) and the Maximum Satisfiability Problem (MAXSAT). Moreover, we discuss the impact of problem structure on algorithm behavior, by studying how some constraint graph properties affect the search performance.

We first present the state of the art in metaheuristics, underlining the importance of intensification and diversification, and showing that metaheuristics can be conceptually analyzed on their basis. We then provide an architecture that enables the design and implementation of metaheuristics in a component-based fashion, moving the focus from the algorithmic and conceptual viewpoint, to the software engineering approach. The issues of algorithm design and implementation are considered in the novel application of two metaheuristics to MAXSAT. We present the development and implementation of Ant Colony Optimization and of Iterated Local Search metaheuristics.

Aftewards, we capitalize the results achieved in the analysis and design of metaheuristics by investigating the impact of SAT/MAXSAT problem structure on metaheuristic behavior. We first define the structure on the basis of the constraint graph associated to the instances. This approach enables us to extract general properties of the problem structure. In particular, we study the impact of connectivity among variables on the parallelization of local search. We find empirical evidence for the presence of an optimal number of parallel local moves that enables the algorithm to achieve the highest effectiveness. We also discover that the optimal number of parallel moves is negatively correlated with the connectivity among variables. The results obtained can give insight into the Criticality and Parallelism phenomenon and into the behavior of trajectory methods. Finally, we also investigate the hardness of small-world SAT instances, finding results which may support the conjecture that small-world instances are among the hardest to solve for both approximate and complete algorithms.

Keywords: *Metaheuristics, Constraint Satisfaction Problems, Combinatorial Optimization Problems, Satisfiability Problem, Maximum Satisfiability Problem, Graph Theory*

To my friends.

Acknowledgments

Scientific research is fortunately a collaborative activity. I think am a very lucky man, as I could, and I can, collaborate with clever and nice people. This thesis is thus a work that has been possible through the help and support of researchers I work with.

First of all, I would like to thank my Tutor, Prof. Maurelio Boari, and my Supervisors, Prof. Paola Mello and Prof. Michela Milano, who always could find time, in their 'over-constrained' working plan, for help, advises and fruitful discussions.

Most of the time of my PhD, I have worked at LIA (*Laboratorio di Informatica Avanzata*) at the Dept. of Electronics, Computer Science and Systems (DEIS) in Bologna. The researchers I work with are not only colleagues, but also friends. I especially thank Silvia and Rebecca – with whom I shared for long time the 'open-space office', laughs and, sometimes, some frustrations –, Fabrizio, Paolo T., Paolo B., Rosy, Franco, Andrea O., Enrico, Cesare and Marco, who also deserves some credits for a suggestion about the title of this thesis. I also thank Andrea L., who contributed to make me widening the spectrum of my research. Moreover, I would like to thank all the members of the *Drunk Brains Group* for fruitful and interesting discussions.

A significant part of these three years has been spent in Bruxelles, at IRIDIA (*Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle* – Université Libre de Bruxelles), while I was involved in the European Project named *Metaheuristics Network*, under the supervision of Prof. Marco Dorigo. I am extremely grateful to him, since he has been both an excellent mentor and a friend. He taught me the guidelines for being a good researcher (curiosity, efficiency, continued effort, precision, enjoyment of your work, just to mention some), which I try to follow – with a lot of effort, though. At IRIDIA I also met and worked with very special persons, some visiting the lab because of the project, and I heartily thank them all.

Finally, I would like to thank two persons, without whom I surely could not have produced this work: Christian Blum and Michela Milano. Some topics of this thesis are the result of a joint work with either of them and my research has been greatly enriched by their contributions, discussions and ideas. I thank Christian for his honest and invaluable friendship and his polite way of making me realize the point when I am wrong (and this, surprisingly, happens quite often...). I thank Michela for her help and advises (and also her humor): if I am a researcher and I like this job, it is definitely also due to her constant support.

Contents

Chapter 1

Introduction

The space between
what's wrong and right
is where you'll find me, hiding, waiting for you.

Dave Matthews

In every facet of life, search is characterized by a mixture of *strategy*, *heuristics*, *experience* and *randomness*. Especially when we are looking for a solution to a problem, we apply – more or less consciously – a strategy, we use some general knowledge about the problem (the heuristic), we try to capitalize and exploit the experience made in the past and, when we have to choose among alternatives without any bit of information to guide us, we flip a coin and we hope to be lucky. We found what we are looking for in the space between our right and wrong suppositions, in the space between our right and wrong choices.

This thesis concerns algorithms which apply all the four above mentioned ingredients: strategy, heuristics, experience and randomness. These algorithms are commonly called *metaheuristics*, a very meaningful term that suggests the application of heuristics under the guidance of a general strategy. In this thesis we will consider the application of metaheuristics to Combinatorial Optimization Problems (COP) and Constraint Satisfaction Problems (CSP), in particular, the Satisfiability Problem (SAT) and the Maximum Satisfiability Problem (MAXSAT). Metaheuristics fall into the category of approximate algorithms, the ones that do not guarantee to find the optimal solution in bounded time. This characteristic enables them to be usually very effective in finding quickly optimal or near-optimal solutions for large problem instances.

Strategy is maybe the most important component in metaheuristics, since it controls the exploration of the search space. The goal of a good strategy is to drive the search toward promising regions of the search space and to explore them inten-

sively. Moreover, when an area has been explored, the strategy should move the search toward other, unexplored regions.

The *a priori* knowledge on a problem is provided by heuristics. Heuristics are the first source of information for attacking a problem, as they provide a (usually myopic) guidance for the search. Heuristics are often problem dependent, thus the role of metaheuristic strategies is to compose and integrate different heuristics, in order to efficiently explore the search space and find good solutions.

Metaheuristics not only include strategy and heuristic, but also they exploit the search history. This is achieved in diverse ways, from the introduction of short term memories to store the recent moves performed, to the accumulation of statistics on the visited states. This kind of *adaptation* is needed to metaheuristics to be effective in the exploration of the search space. Their search should be "clever" enough to both intensively explore areas of the search space with high quality solutions, and to move to unexplored areas of the search space when necessary.

Finally, randomness deserves to be discussed as well, since it has an important role in search. Randomness comes into play whenever a choice has to be taken without any guidance from the heuristic, nor the search experience. This happens, for instance, when we have to break ties. A random move is a cheap and usually effective action. In the same way as a prey running away from a predator can succeed if it behaves in an unforeseeable way, so metaheuristics can succeed in finding good solutions by introducing randomness. Furthermore, many choices are performed in metaheuristics by following a probability distribution: this enables the search to favor some choices, not excluding the possibility to follow the alternative ones.

The performance of a metaheuristic is strongly affected by the characteristics of the problem instance it is tackling. Every empirical scientist dealing with search algorithms could observe many times that, depending on the class of instances, an algorithm may perform differently. The typical example is that real-world problems are often more difficult to solve than random generated instances. However, despite the relevance of this topic, only in the past decade the study of the relations between problem structure and algorithm behavior has been explicitly and systematically started. Many are the questions that arise: What is *structure*? Which features characterize the difference between random, artificially generated and real-word problems? Which are the structural properties of a problem that have impact on the search behavior? How can we exploit them?

This thesis deals with structure in Satisfiability Problems and its impact on metaheuristic behavior. We present and discuss the structure modeling of Satisfiability Problems via *constraint graphs* and we investigate the influence of graph features on the behavior of some metaheuristics. The representation of structure as a graph enables us to extract general properties of the relations among problem variables, such as their connectivity, distance and clustering. We will see that the connectivity among variables affects local search, since it has direct impact on the number of parallel local moves that enables the algorithm to reach the best solutions. Moreover, we will discuss experimental results concerning the hardness of instances characterized

by *small-world* topology of the associated graph[1].

This thesis begins with a survey on the state of the art in metaheuristics, from the algorithmic viewpoint. It first discusses the possible classifications of metaheuristics, based on key concepts such as the number of solutions managed at each iteration (single vs. population based), the neighborhood structure, the objective function and the use of memory to exploit the search history. Then, the main metaheuristic algorithms are presented in their basic definition, along with variants and improvements. Chap. 2 concludes with an unifying view of metaheuristic concepts based on intensification (exploitation of the accumulated search experience) and diversification (exploration of the search space).

Chap. 3 presents an architecture suitable for the design and implementation of metaheuristics. The architecture is composed of four levels, each concerning a particular phase of metaheuristics. Moreover, the software components of the architecture can be seen as software agents and metaheuristics are the result of their coordination.

After a general introduction of metaheuristics, both from algorithmic and engineering perspectives, we consider the application of these algorithms to the Maximum Satisfiability Problem (MAXSAT) in Chap. 4. In this chapter, we present and discuss the design of two metaheuristics, namely Ant Colony Optimization and Iterated Local Search, which have not yet been applied to MAXSAT. Beside methodological contributions concerning the discussion of design choices, we also achieved very good results with the second metaheuristic.

The subsequent two chapters of the thesis represent the most investigative and empirical part of this work. In Chap. 5, we study the impact of connectivity among variables on the parallelization of local search. Starting from a phenomenon called *Criticality and Parallelism*, we deeply investigate the influence of node degree of the graph associated to the problem instances on the number of parallel local moves that leads to the best performance. We discover that the optimal number of parallel moves is negatively correlated with the average connectivity among variables. Furthermore, by analyzing the frequency of node degree in structured instances, we observe that the presence of a large number of variables with higher (resp. lower) node degree w.r.t. the average node degree strongly affects the optimal parallelism. We also face another related topic in Chap. 6, where we study the hardness of SAT instances characterized by a small-world topology. Those instances locate in-between random instances and instances associated to lattice graphs. We observe that small-world instances seem harder for local search than the ones at the two extremes.

Finally, we conclude with a chapter devoted to an overview of current work, which goes into the direction of the integration of metaheuristics and complete algorithms. In particular, we discuss possible ways of combining metaheuristics and tree search.

[1] An example of small-world graph is the network defined on the basis of friendship relationships.

Chapter 2

Metaheuristics: The State of the Art

2.1 Introduction

Many optimization problems of both practical and theoretical importance concern the choice of a "best" configuration of a set of variables to achieve some goals. They seem to divide naturally into two categories: those where solutions are encoded with *real-valued* variables, and those where solutions are encoded with *discrete* variables. Among the latter ones we find a class of problems called Combinatorial Optimization Problems (COP). According to [168], in COP, we are looking for an object from a finite – or possibly countably infinite – set. This object is typically an integer number, a subset, a permutation, or a graph structure.

Definition 1 *A* **Combinatorial Optimization** *problem $\mathcal{P} = (\mathcal{S}, f)$ can be defined by:*

- *a set of variables $X = \{x_1, \ldots, x_n\}$;*
- *variable domains $D_1, \ldots, D_n$;*
- *constraints among variables;*
- *an objective function to be minimized*[1] *$f : D_1 \times \ldots \times D_n \rightarrow \mathbb{R}^+$;*

The set of all possible feasible assignments is

$$\mathcal{S} = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i, s \text{ satisfies all the constraints}\}$$

$\mathcal{S}$ is usually called a search (or solution) space, as each element of the set can be seen as a candidate solution. To solve a combinatorial optimization problem one has to

[1] As maximizing an objective function f is the same as minimizing $-f$, in this work we will deal, without loss of generality, with minimization problems.

find a solution $s^* \in \mathcal{S}$ with minimum objective function value, that is, $f(s^*) \leq f(s)$ $\forall s \in S$. s^* is called a globally optimal solution of $(\mathcal{S}, f)$ and the set $\mathcal{S}^* \subseteq \mathcal{S}$ is called the set of globally optimal solutions.

Examples of COP are the Traveling Salesman problem (TSP), the Quadratic Assignment problem (QAP), Timetabling and Scheduling problems. Due to the practical importance of COP, many algorithms to tackle them have been developed. These algorithms can be classified as either *complete* or *approximate* algorithms. Complete algorithms are guaranteed to find, for every finite size instance of a COP, an optimal solution in bounded time (see [168, 162]). Yet, for COP that are $\mathcal{NP}$-hard [79], no polynomial time algorithm exists, assuming that $\mathcal{P} \neq \mathcal{NP}$. Therefore, complete methods might need exponential computation time in the worst-case. This often leads to computation times too high for practical purposes. Thus, the use of approximate methods to solve COP has received more and more attention in the last 30 years. In approximate methods we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time.

Among the basic approximate methods we usually distinguish between *constructive* methods and *local search* methods. Constructive algorithms generate solutions from scratch by adding components to an initially empty partial solution, until a solution is complete. They are typically the fastest approximate methods, yet they often return solutions of inferior quality when compared to local search algorithms. Local search algorithms start from some initial solution and iteratively try to replace the current solution by a better solution in an appropriately defined neighborhood of the current solution, where the neighborhood is formally defined as follows:

Definition 2 *A* **neighborhood structure** *is a function $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ that assigns to every $s \in \mathcal{S}$ a set of neighbors $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is called the neighborhood of s.*

The introduction of a neighborhood structure enables us to define the concept of *locally* minimal solutions.

Definition 3 *A* **locally minimal solution (or local minimum)** *with respect to a neighborhood structure $\mathcal{N}$ is a solution $\hat{s}$ such that $\forall\ s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$. We call $\hat{s}$ a strict locally minimal solution if $f(\hat{s}) < f(s)\ \forall\ s \in \mathcal{N}(\hat{s})$.*

In the last 20 years, a new kind of approximate algorithm has emerged which tries to combine basic heuristic methods in higher level frameworks aimed at efficiently and effectively exploring a search space. These methods are nowadays commonly called *metaheuristics*. The term *metaheuristic*, first introduced in [87], derives from the composition of two Greek words. *Heuristic* derives from the verb *heuriskein* ($\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$) which means "to find", while the suffix *meta* means "beyond, in an upper level". Before this term was widely adopted, metaheuristics were often called *modern heuristics* [181].

This class of algorithms includes[2], but is not restricted to, Ant Colony Optimization (ACO), Evolutionary Computation (EC) including Genetic Algorithms (GA),

[2] in alphabetical order

Iterated Local Search (ILS), Simulated Annealing (SA), and Tabu Search (TS). Up to now there is no commonly accepted definition for the term metaheuristic. Just in the last few years researchers in the field tried to propose a definition. In the following we quote some of them:

"A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions." [167].

"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." [225].

"Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristic, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descent by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in metaheuristic algorithms randomness is not used blindly but in an intelligent, biased form." [209].

"A metaheuristic is a set of concepts that can be used to define heuristic methods that can be applied to a wide set of different problems. In other words, a metaheuristic can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem." [145].

Summarizing, we outline fundamental properties which characterize metaheuristics:

- Metaheuristics are strategies that "guide" the search process.
- The goal is to efficiently explore the search space in order to find (near–)optimal solutions.
- Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes.

- Metaheuristic algorithms are approximate and usually non-deterministic.
- They may incorporate mechanisms to avoid getting trapped in confined areas of the search space.
- The basic concepts of metaheuristics permit an abstract level description.
- Metaheuristics are not problem-specific.
- Metaheuristics may make use of domain-specific knowledge as heuristic controlled by the upper level strategy.
- Todays more advanced metaheuristics use search experience (embodied in some form of memory) to guide the search.

In short we could say: Metaheuristics are high level strategies for exploring search spaces by using different methods. These strategies should be chosen in such a way to achieve a dynamic balance between *diversification* and *intensification*. The term diversification generally refers to the exploration of the search space, whereas the term intensification refers to the exploitation of the accumulated search experience. These terms stem from the Tabu Search field [90] and it is important to clarify that the terms *exploration* and *exploitation* are sometimes used instead, for example in the Evolutionary Computation field [57], with a more restricted meaning. In fact, the notions of exploitation and exploration often refer to rather short term strategies tied to randomness, whereas intensification and diversification also refer to medium and long term strategies based on the usage of memory. The use of the terms diversification and intensification in their initial meaning becomes more and more accepted by the whole field of metaheuristics. The balance between diversification and intensification as mentioned above is important, on one side to quickly identify regions in the search space with high quality solutions and on the other side not to waste too much time in regions of the search space which are either already explored or do not provide high quality solutions.

The structure of the strategies is highly dependent on the philosophy of the metaheuristic itself. There are several different philosophies apparent in the existing metaheuristics. Some of them can be seen as "intelligent" extensions of local search algorithms. The goal of this kind of metaheuristic is to escape from local minima to proceed in the exploration of the search space and to move on to find other hopefully better local minima. For example, to this category belong Tabu Search, Iterated Local Search, Variable Neighborhood Search, GRASP and Simulated Annealing. These metaheuristics (also called trajectory methods) work on one or diverse neighborhood structure(s) imposed on the solutions of the search space.

A second category is based on a different philosophy. Algorithms like Ant Colony Optimization and Evolutionary Computation incorporate a learning component, since they implicitly or explicitly try to learn correlations between decision variables to identify high quality areas in the search space. This kind of metaheuristic performs, in a

sense, a biased sampling of the search space. For instance, in Evolutionary Computation, this is achieved by recombination of solutions and in Ant Colony Optimization by sampling the search space according to a probability distribution.

In the following sections we first summarize different classification approaches of metaheuristics. Sec. 2.3 and Sec. 2.4 are devoted to a description of the most important metaheuristics nowadays. Section 2.3 describes the most relevant trajectory methods and in Section 2.4 we outline population-based methods. Finally, we discuss the use of intensification and diversification in metaheuristics, trying to evidence that these two concepts enable a uniform view of such algorithms.

For an in-deep survey on metaheuristics we forward the interested reader to [23].

2.2 Classification of metaheuristics

There are different ways to classify and describe metaheuristic algorithms. Depending on the characteristics selected to differentiate among them, several classifications are possible, each of them being the result of a specific viewpoint. We briefly summarize the most important ways of classifying metaheuristics.

Nature-inspired vs. non-nature inspired. Perhaps, the most intuitive way of classifying metaheuristics is based on the origins of the algorithm. There are nature-inspired algorithms, like Genetic Algorithms and Ant Algorithms, and non nature-inspired ones such as Tabu Search and Iterated Local Search. In our opinion this classification is not very meaningful for the following two reasons. First, many recent hybrid algorithms do not fit either class (or, in a sense, they fit both at the same time). Second, it is sometimes difficult to clearly attribute an algorithm to one of the two classes.

Population-based vs. single point search. Another characteristic that can be used for the classification of metaheuristics is the number of solutions used at the same time: Does the algorithm work on a population or on a single solution at any time? Algorithms working on single solutions are called *trajectory methods* and encompass local search-based metaheuristics, like Tabu Search, Iterated Local Search and Variable Neighborhood Search. They all share the property of describing a trajectory in the search space during the search process. Population-based metaheuristics, on the contrary, perform search processes which describe the evolution of a set of points in the search space.

Dynamic vs. static objective function. Metaheuristics can also be classified according to the way they make use of the objective function. While some algorithms keep the objective function given in the problem representation "as it is", some others, like Guided Local Search (GLS), modify it during the search. The idea behind this approach is to escape from local minima by modifying the search landscape. Accordingly, during the search the objective function is altered by trying to incorporate

information collected during the search process.

One vs. various neighborhood structures. Most metaheuristic algorithms work on one single neighborhood structure. In other words, the fitness landscape topology does not change in the course of the algorithm. Other metaheuristics, such as Variable Neighborhood Search (VNS), use a set of neighborhood structures which gives the possibility to diversify the search and tackle the problem swapping between different fitness landscapes.

Memory usage vs. memory-less methods. A very important feature to classify metaheuristics is the use they make of the search history, that is, whether they use memory[3] or not. Memory-less algorithms perform a Markov process, as the information they exclusively use to determine the next action is the current state of the search process. There are several different ways of making use of memory. Usually we differentiate between short term and long term memory usages. The first usually keeps track of recently performed moves, visited solutions or, in general, decisions taken. The second is usually an accumulation of synthetic parameters about the search. The use of memory is nowadays recognized as one of the fundamental elements of a powerful metaheuristic.

In the following we describe the most important metaheuristics according to the single point vs. population-based search classification, which divides metaheuristics into trajectory methods and population-based methods. This choice is motivated by the fact that this categorization permits a clearer description of the algorithms. Moreover, a current trend is the hybridization of methods in the direction of the integration of single point search algorithms in population-based ones.

In the following two sections we give a detailed description of the algorithms and their fundamental structures.

2.3 Trajectory Methods

In this section we outline metaheuristics called *trajectory methods*. The term trajectory methods is used because the search process performed by these methods is characterized by a trajectory in the search space. This trajectory can be continuous (i.e., the successor solution belongs to the neighborhood of the incumbent one) or not. In the following we will refer to a minimization problem $\mathcal{P}$ with an objective function f, a set of feasible solutions $\mathcal{S} = \{s_1, s_2, \ldots\}$ and a neighborhood structure $\mathcal{N}$ (see Definition 1).

The search process of trajectory methods can be seen as the evolution in (discrete) time of a discrete dynamical system [11, 42]. The algorithm starts from an initial state (the initial solution) and describes a trajectory in the state space. The

[3]Here we refer to the use of adaptive memory, in contrast to rather rigid memory, as used for instance in Branch & Bound.

system dynamics depends on the strategy used; simple algorithms generate a trajectory composed of two parts: a *transient* phase followed by an *attractor* (a fixed point, a cycle or a complex attractor). Algorithms with advanced strategies generate more complex trajectories which can not be subdivided in those two phases. The characteristics of the trajectory outline the behavior of the algorithm and its effectiveness with respect to the instance it is tackling. It is worth underlining that the dynamics is the result of the combination of algorithm, problem representation and instance. In fact, the problem representation (including the neighborhood structures) defines the search landscape; the algorithm describes the strategy used to explore the landscape and, finally, the actual search space characteristics are defined by the instance to be solved.

In the next section we will introduce the concept of *fitness landscape*, which represents an abstraction of the search space usually adopted in the case of metaheuristics.

2.3.1 Fitness Landscapes

The local search process can be viewed as an exploration of a landscape aimed at finding a global optimum, or, at least, a "good" local optimum.

A *Fitness Landscape* is defined by a triple:

$$\mathcal{L} = (S, \mathcal{N}, F)$$

where:

- S is the set of solutions (or states);
- $\mathcal{N}$ is the neighborhood function $\mathcal{N} : S \to 2^S$ that defines the neighborhood structure, by assigning to every $s \in S$ a set of states $\mathcal{N}(s) \subseteq S$.
- F is the objective function, in this specific case called *fitness function*, F: $S \to \mathbb{R}^+$.

The Fitness Landscape (hereafter referred to as FL) can be interpreted as a graph (see Figure 2.1) in which nodes are solutions (with their fitness value) and arcs represent the neighborhood relation between states.

The neighborhood function $\mathcal{N}$ implicitly defines an *operator* φ which takes a state s_1 and transforms it into another state $s_2 \in \mathcal{N}(s_1)$. Conversely, given an operator φ, it is possible to define a neighborhood of a variable $s_1 \in S$:

$$\mathcal{N}_\varphi(s_1) = \{s_2 \in S \setminus \{s_1\} \mid s_2 \text{ can be obtained by one application of } \varphi \text{ on } s_1\}$$

Usually, the operator is *symmetric*: if s_1 is a neighbor of s_2 then s_2 is a neighbor of s_1. In a graph representation (like the one depicted in Figure 2.1) undirected arcs represent symmetric neighborhood structures. A desirable property of the neighborhood structure is to allow a path from every pair of nodes (i.e., the neighborhood is strongly optimally connected) or at least from any node to an optimum (i.e., the neighborhood

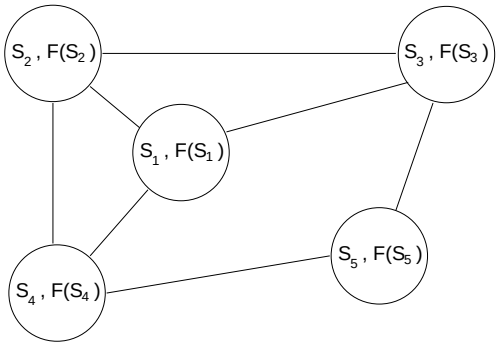

Figure 2.1: Example of undirected graph representing a fitness landscape. Each node is associated with a solution s_i and its corresponding fitness value $F(s_i)$. Arcs represent transition between states by means of φ. Undirected arcs correspond to symmetric neighborhood structure.

is weakly optimally connected). Nevertheless, there are some exceptions of effective neighborhood structures which do not enjoy this property [165].

The notion of FL and neighborhood enables to view local search algorithms as search processes in a graph. The search starts from an initial node and explores the graph moving from a node to one of its neighbors, until it reaches a termination condition.

There exists another definition of fitness landscape [122], which is more formal and general and can be used here as well. However, for the purpose of this contribution it is sufficient to know that this formal definition can deal with states (nodes of the search graph) composed of populations of solutions. The key point is the introduction of multi-sets, which are sets with possible repetitions of elements. A multi-set substitutes a single solution and the operator transforms a multi-set into another. Furthermore, the operator is defined as a function $\psi : M(S) \times M(S) \rightarrow [0, 1]$, which assigns a probability for each possible transition between states. This definition of FL enables to deal with population heuristics, like genetic algorithms, in the same way as simple local search algorithms.

There are some important design issues in developing a search algorithm over a FL: the solution representation, the neighborhood structure and the fitness function. Furthermore, there are some ways to cope with constraints. For example, it is possible to map a CSP or a COP into a Free Optimization Problem, where there are not constraints and infeasible solutions are penalized by the fitness function [56].

It is worth underlining that, given a fitness function, the choice of an operator determines the properties of the landscape. This is the "One Operator One Landscape" concept, introduced in [122, 123]. The algorithm performance is strongly affected by the model chosen and, in general, no best choice exists which leads to the best performance with every algorithm/problem combination. This empirical conjecture

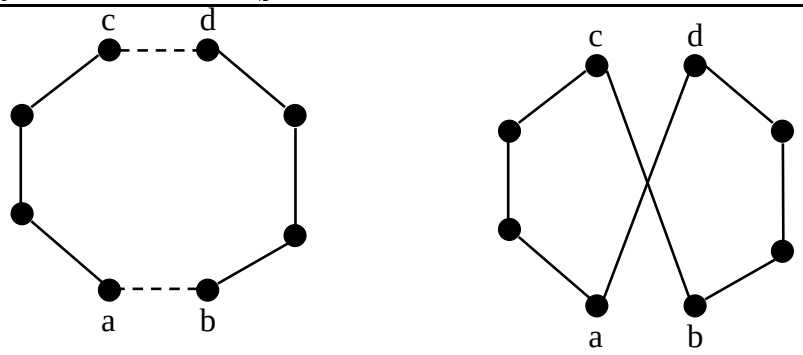

Figure 2.2: Example of 2-exchange move. The tour can be reconstructed in only one way.

is theoretically supported by the *No Free Lunch Theorem* [239].

Example: Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is defined as follows: given an undirected graph (supposed fully connected), with n nodes and each arc associated with a positive value, find the Hamiltonian tour with the minimum total cost. From the problem definition, several models are possible. We shall discuss two among the most common ones.

Model 1

A solution (i.e., an Hamiltonian tour) can be represented as a sequence of n arcs (v_i, v_j), where v_i and v_j are nodes of the graph. The solution cost is given by the sum of costs of the arcs in the tour.

Two very successful neighborhood structures which can be introduced in the above-given representation are called 2-exchange and 3-exchange which lead to 2 and 3-opt improvement algorithms [121]. Both algorithms need to start with a feasible solution. The 2-exchange operator deletes two arcs from the feasible tour and reconnects the two paths into a tour in the other way (see Figure 2.2). In Figure 2.3 the 3-exchange move is sketched: three arcs are removed and the feasible tour is then reconstructed by using different arcs (in this case there are six possible ways to reconstruct the cycle).

The FL generated by the two moves are different, as the defined neighborhoods differ in size and connectivity properties. 2-exchange neighborhood $\mathcal{N}_{2-ex}$ has a cardinality proportional to n^2, while for 3-exchange $|\mathcal{N}_{3-ex}|$ is proportional to n^3. Observe that the solutions defined in this model are feasible for the original problem as well, because moves preserve the Hamiltonian tour structure.

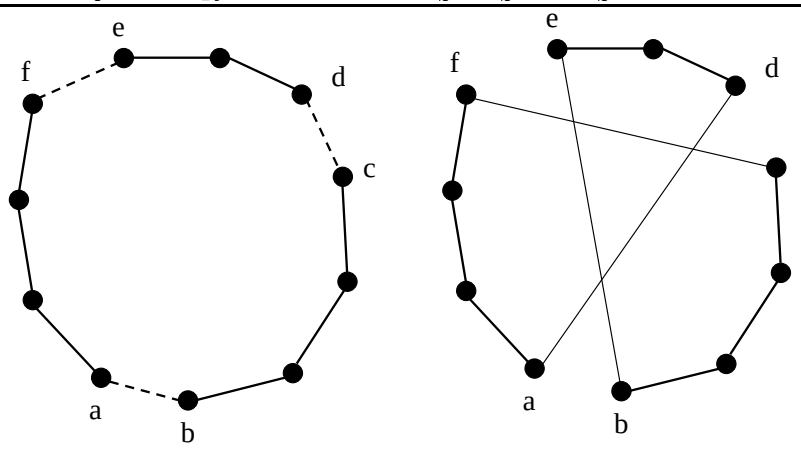

Figure 2.3: Example of 3-exchange move. After the removal of three arcs there are six possible ways to reconstruct the tour. In this figure just one of them is shown.

Model 2

A solution (i.e., an Hamiltonian tour) can be alternatively represented as a sequence of n values $\{v_1, v_2, \ldots, v_n\}$, where v_i are nodes of the graph and the solution represents the sequence of nodes in the tour. For example, $\{3, 5, 4, 2, 1\}$ represents the tour $3 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 1$. The solution cost is given by the sum of costs of the arcs in the tour.

A simple move operator may be the exchange of any pair nodes in the sequence. If the initial solution is a feasible tour, the result of a move is guaranteed to be feasible. Similarly, it is possible to define moves which pick up more than 2 nodes and exchange them, leading to the definition of neighborhoods $\mathcal{N}_{k-perm}$. Observe that these operators define neighborhoods of cardinality proportional to $\frac{n!}{k!(n-k)!}$.

It is worth to note that, if the representation chosen $\{v_1, v_2, \ldots, v_n\}$ was interpreted as "the position of node i is v_i", we should introduce an additional constraint to avoid subtours.

Example: Satisfiability Problem

In this example we discuss a possible representation of the Satisfiability Problem (SAT) when solved via Local Search.

SAT is defined as follows: given a set of clauses, each of which is the logical disjunction of k *literals* (a literal is a variable or its negation), the goal is to find an assignment to the variables that satisfies all the clauses. For example: the set of clauses $\{(x_1 \vee \neg x_2), (\neg x_1 \vee \neg x_3), (x_1 \vee x_3), (x_2 \vee \neg x_3)\}$ is satisfied by the assignments $\{(x_1 = 1, x_2 = 0, x_3 = 0), ((x_1 = 1, x_2 = 1, x_3 = 0))\}$.

Let us suppose to tackle SAT with local search. We model the problem by defining

as states the variable assignments. In this example we define the move operator as the flip of one variable. This move operator induces a neighborhood structure such that the neighbors of a state s are all the states at Hamming distance equal to 1 from s (this is the definition of neighborhood most often adopted in SAT problems). The SAT problem does not involve any optimization criteria since any solution satisfying all the constraints is accepted. However, solving SAT with LS, we need to introduce in the model an objective function which evaluates the assignments with respect to their "closeness" to the satisfying ones. A typical objective function is the number of satisfied clauses, but other choices are possible[4]. Thus, feasible solution for the SAT problem are mapped in optimal solutions in the model.

In the next sections, we will first describe the basic local search algorithms and then we will survey more complex strategies, ending with algorithms that define general strategies and can include other trajectory methods as components.

2.3.2 Basic Local Search: Iterative Improvement

The basic local search is usually called *iterative improvement*, since each move is only performed if the solution it produces is better than the current solution. The algorithm stops as soon as it finds a local minimum. The high level algorithm is sketched in Algorithm 1.

Algorithm 1 Iterative Improvement

```
s ← GenerateInitialSolution()
repeat
  s ← Improve(s,𝒩(s))
until no improvement is possible
```

The function $Improve(s, \mathcal{N}(s))$ can be in the extremes either a *first improvement*, or a *best improvement* function, or any intermediate option. The former scans the neighborhood $\mathcal{N}(s)$ and chooses the first solution that improves the objective function, the latter exhaustively explores the neighborhood and returns one of the solutions with the lowest objective function value. Both methods stop at local minima, therefore their performance strongly depends on the definition of S, f and $\mathcal{N}$. In general, the termination condition of metaheuristic algorithms is more complex than simply reaching a local minimum. Indeed, possible termination conditions include: maximum CPU time, a maximum number of iterations, a solution s with $f(s)$ less than a predefined threshold value is found, or the maximum number of iterations without improvements is reached.

The performance of iterative improvement procedures on COP is usually quite unsatisfactory, therefore several techniques have been developed to prevent algorithms from getting trapped in local minima or to escape from them. In the following we describe the most important and successful ones.

[4]See, for example, *non–oblivious functions* for MAXSAT problems in [14].

2.3.3 Simulated Annealing

Simulated Annealing (SA) is commonly said to be the oldest among the metaheuristics and surely one of the first algorithms that had an explicit strategy to avoid local minima. The origins of the algorithm are in statistical mechanics (Metropolis algorithm) and it was first presented as a search algorithm for COP in [132] and [30]. The fundamental idea is to allow moves resulting in solutions of worse quality than the current solution (uphill moves) in order to escape from local minima. The probability of doing such a move is decreased during the search. The high level algorithm is described in Algorithm 2.

Algorithm 2 Simulated Annealing (SA)

```
s ← GenerateInitialSolution()
T ← T_0
while termination conditions not met do
  s' ← PickAtRandom(N(s))
  if f(s') < f(s) then
    s ← s' {s' replaces s}
  else
    Accept s' as new solution with probability p(T, s', s)
  end if
  Update(T)
end while
```

The algorithm starts by generating an initial solution (either randomly or heuristically constructed) and by initializing the so-called temperature parameter T. Then this cycle is repeated until the termination condition is reached. The instructions in the inner cycle are very simple: a solution $s' \in \mathcal{N}(s)$ is randomly sampled and it is accepted as new current solution depending on $f(s)$, $f(s')$ and T. s' replaces s if $f(s') < f(s)$ or, in case $f(s') \geq f(s)$, with a probability which is a function of T and $f(s') - f(s)$. The probability is generally computed following the Boltzmann distribution $\exp(-\frac{f(s')-f(s)}{T})$.

The temperature T is decreased[5] during the search process, thus at the beginning of the search the probability of accepting uphill moves is high and it gradually decreases, converging to a simple iterative improvement algorithm. This process is analogous to the annealing process of metals and glass, which assume a low energy configuration when cooled with an appropriate cooling schedule. Regarding the search process, this means that the algorithm is the result of two combined strategies: random walk and iterative improvement. In the first phase of the search, the bias toward improvements is low and it permits the exploration of the search space; this erratic component is slowly decreased thus leading the search to converge to a (local) minimum. The probability of accepting uphill moves is controlled by two factors: the

[5] T is not necessarily decreased in a monotonic fashion. Elaborate cooling schemes also incorporate an occasional increase of the temperature.

difference of the objective functions and the temperature. On the one hand, at fixed temperature, the higher the difference $f(s') - f(s)$, the lower the probability to accept a move from s to s'. On the other hand, the higher T, the higher the probability of uphill moves.

The choice of an appropriate cooling schedule is crucial for the performance of the algorithm. The cooling schedule defines the value of T at each iteration k, $T_{k+1} = Q(T_k, k)$, where $Q(T_k, k)$ is a function of the temperature and of the iteration number. Theoretical results on non-homogeneous Markov chains [1] state that under particular conditions on the cooling schedule, the algorithm converges in probability to a global minimum for $k \to \infty$. More precisely:

$$\exists \Gamma \in I\!R \quad \text{s.t.} \quad \lim_{k \to \infty} Prob[\text{global minimum found after } k \text{ steps}] = 1$$

$$\text{iff} \quad \sum_{k=1}^{\infty} exp(\frac{\Gamma}{T_k}) = \infty$$

A particular cooling schedule that fulfills the hypothesis for the convergence is the one that follows a logarithmic law: $T_{k+1} = \frac{\Gamma}{\log(k+k_0)}$ (where k_0 is a constant). Unfortunately, cooling schedules which guarantee the convergence to a global optimum are not feasible in applications, because they are too slow for practical purposes. Therefore, faster cooling schedules are adopted in applications. One of the most used follows a geometric law: $T_{k+1} = \alpha T_k$, where $\alpha \in]0, 1[$, which corresponds to an exponential decay of the temperature.

The cooling rule can vary during the search, with the aim of tuning the balance between diversification and intensification. For example, at the beginning of the search, T might be constant or linearly decreasing, in order to sample the search space; then, T might follow a rule such as the geometric one, to converge to a local minimum at the end of the search. More successful variants are *non-monotonic* cooling (e.g., see [166, 138]). Non-monotonic cooling schedules are characterized by alternating phases of cooling and reheating, thus providing an oscillating balance between diversification and intensification.

The cooling schedule and the initial temperature should be adapted to the particular problem instance, since the cost of escaping from local minima depends on the structure of the search landscape. A simple way of empirically determining the starting temperature T_0 is to initially sample the search space with a random walk to roughly evaluate the average and the variance of objective function values. But also more elaborate schemes can be implemented [119].

The dynamic process described by SA is a *Markov chain* [58], as it follows a trajectory in the state space in which the successor state is chosen depending only on the incumbent one. Therefore this process of basic SA is memory-less. However, the use of memory can be beneficial for SA approaches (see for example [31]).

SA has been applied to several COP, such as the Quadratic Assignment Problem (QAP) [34] and the Job Shop Scheduling (JSS) problem [222]. References to other applications can be found in [2, 119, 64]. SA is nowadays used as a component in metaheuristics, rather than applied as stand-alone search algorithm. Variants of SA

called Threshold Accepting and The Great Deluge Algorithm were presented by Dueck et al. [54, 53].

2.3.4 Tabu Search

Tabu Search (TS) is among the most cited and used metaheuristics for COP. TS basic ideas were first introduced in [87], built on earlier ideas formulated in [86][6]. A description of the method and its concepts can be found in [90]. TS explicitly uses the history of the search, both to avoid local minima and to implement an explorative strategy. We will first describe a simple version of TS, to introduce the basic concepts. Then, we will explain a more applicable algorithm and finally we will discuss some improvements.

Algorithm 3 Simple Tabu Search (TS)

```
s ← GenerateInitialSolution()
TabuList ← ∅
while termination conditions not met do
  s ← ChooseBestOf(s,N(s) \ TabuList)
  Update(TabuList)
end while
```

The simple TS algorithm (see Algorithm 3) applies a best improvement local search as basic ingredient and uses a *short term memory* to escape from local minima and to avoid cycles. The short term memory is implemented as a *tabu list* that keeps track of the most recently visited solutions and forbids moves toward them. The neighborhood of the current solution is thus restricted to the solutions that do not belong to the tabu list. In the following we will refer to this set as *allowed set*. The new solution is chosen in the allowed set, which is exhaustively explored, then it is added to the tabu list and one of the solutions in the list is discarded (usually in a FIFO order). Due to this dynamic restriction of allowed solutions in a neighborhood, TS can be considered as a dynamic neighborhood search technique [209]. The algorithm stops when the termination condition is met. It might also terminate if the allowed set is empty, that is, if all the solutions in $\mathcal{N}(s)$ are forbidden by the tabu list[7].

The use of a tabu list prevents from returning to recently visited solutions, therefore it prevents from endless cycling[8] and forces the search to accept even uphill moves. The length l of the tabu list (*tabu tenure*) controls the memory of the search process. With small tabu tenures the search will concentrate on small areas of the search space. On the opposite, a large tabu tenure forces the search process to explore larger regions, because it forbids revisiting a higher number of solutions. The

[6] Related ideas were labeled steepest ascent/mildest descent method in [101].

[7] Other escaping strategies are possible, to avoid to stop when the allowed set is empty. For instance, one can choose the least recently visited solution, even if it is tabu.

[8] Cycles of higher period are possible, since the tabu list has a finite length l which is smaller than the cardinality of the search space.

tabu tenure can be varied during the search, leading to more robust algorithms. An example can be found in [215], where the tabu tenure is periodically reinitialized at random in the interval $[l_{min}, l_{max}]$. A more advanced use of dynamic tabu tenure is presented in [16, 13], where the tabu tenure is increased if there is evidence for repetitions of solutions (thus a higher diversification is needed), while it is decreased if there are no improvements (thus intensification should be boosted). More advanced ways to create dynamic tabu tenure are described in [88].

The implementation of short term memory as a list of visited solution is not practical, because managing a list of solutions is highly inefficient. Therefore, instead of the solutions themselves, solution *attributes* are stored[9]. Attributes are usually components of solutions, moves, or differences between two solutions. Since more than one attribute can be considered, a tabu list is introduced for each of them. The set of attributes and related tabu lists define the *tabu conditions* which are used to filter the neighborhood of a solution and generate the allowed set. Storing moves instead of complete solutions is much more efficient, but it introduces a loss of information, as forbidding a move means assigning the tabu status to probably more than one solution. Thus, it is possible that unvisited solutions of good quality are excluded from the allowed set. To overcome this problem, *aspiration criteria* are defined which allow to include a solution in the allowed set even if it is forbidden by tabu conditions. Aspiration criteria define the aspiration conditions that are used to construct the allowed set. The most commonly used aspiration criterion selects solutions which are better than the current best one. The complete algorithm, as described above, is reported in Algorithm 4.

Algorithm 4 Tabu Search (TS)

$s \leftarrow$ GenerateInitialSolution()
InitializeTabuLists($TL_1, \ldots, TL_r$)
$k \leftarrow 0$
while termination conditions not met **do**
  $AllowedSet(s,k) \leftarrow \{z \in \mathcal{N}(s) \mid$ no tabu condition is violated or at least one aspiration condition is satisfied$\}$
  $s \leftarrow$ ChooseBestOf(s,$AllowedSet(s,k)$)
  UpdateTabuListsAndAspirationConditions()
  $k \leftarrow k + 1$
end while

Tabu lists are only one of the possible ways of taking advantage of the history of the search. They are usually identified with the usage of short term memory. Information collected during the overall search process can also be very useful, especially for a strategic guidance of the algorithm. This kind of long term memory is usually added to TS by referring to four principles: *recency*, *frequency*, *quality* and *influence*. Recency-based memory records for each solution (or attribute) the most recent iteration it was

[9] In addition to storing attributes, some longer term TS strategies also keep complete solutions (e.g., elite solutions) in the memory.

involved in. Orthogonally, frequency-based memory keeps track of how many times each solution (attribute) has been visited. This information identifies the regions (or the subsets) of the solution space where the search was confined, or where it stayed for a high number of iterations. This kind of information about the past is usually exploited to diversify the search. The third principle, quality, is a guidance to learn and extract information from the search history in order to identify good solution components. This information can be usefully integrated in the solution construction. Other metaheuristics (e.g., Ant Colony Optimization) explicitly use this principle to learn about good combinations of solution components. Finally, 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. In general, the TS field is a rich source of ideas. Many of these ideas and strategies have been and are currently adopted by other metaheuristics.

TS has been applied to most COP; examples for successful applications are the Robust Tabu Search to the QAP [215], the Reactive Tabu Search to the MAXSAT problem [13], and to assignment problems [39]. TS approaches dominates the Job Shop Scheduling (JSS) problem area (see for example [165]) and the Vehicle Routing (VR) area [81]. Further current applications can be found at [214].

2.3.5 Explorative Local Search methods

In this section we present more recent trajectory methods. These are the Greedy Randomized Adaptive Search Procedure (GRASP), Variable Neighborhood Search (VNS), Guided Local Search (GLS) and Iterated Local Search (ILS).

GRASP

The Greedy Randomized Adaptive Search Procedure (GRASP), see [59, 173], is a simple algorithm that combines constructive heuristics and local search. Its structure is sketched in Algorithm 5. GRASP is an iterative procedure, composed of two phases: solution construction and solution improvement. The best found solution is returned at the end of the iterations, when the termination condition is reached.

Algorithm 5 Greedy Randomized Adaptive Search Procedure (GRASP)

```
while termination conditions not met do
  s ← ConstructGreedyRandomizedSolution(){see Algorithm 6}
  ApplyLocalSearch(s)
  MemorizeBestFoundSolution()
end while
```

The solution construction algorithm (see Algorithm 6) is characterized by two main ingredients: a dynamic constructive heuristic and randomization. Assuming that a solution s consists of a subset of a set of solution components (elements), the

Algorithm 6 Greedy randomized solution construction

```
s ← ∅{s denotes a partial solution in this case}
α ← CandidateListLength() {definition of the RCL length}
while solution not complete do
  RCL_α ← GenerateRestrictedCandidateList(s)
  x ← SelectElementAtRandom(RCL_α)
  s ← s ∪ {x}
  UpdateGreedyFunction(s){update the heuristic values (see text below)}
end while
```

solution is constructed one element at a time by randomly picking it from a candidate list. The elements are ranked by means of a heuristic criterion that gives them a score as a function of the (myopic) benefit if inserted in the solution. The candidate list, called *Restricted Candidate List* (RCL), is composed of the first α elements. The heuristic values are updated at each step, thus the score of elements changes during the construction phase, as the choice of an element might decrease/increase the desirability of another not yet inserted element. This constructive heuristic is called *dynamic*, in contrast to the *static* one which assigns a score to elements only before starting the construction. For instance, a static heuristic for TSP is based on the arc cost: the lower the cost, the higher its desirability. An example of a dynamic heuristic is cheapest insertion, where the desirability of an element is evaluated depending on the current partial solution.

The length α of the candidate list controls the heuristic bias. In the extreme case of $\alpha = 1$ the best element would be added, thus the construction would be fully greedy. On the opposite, for $\alpha = n$ the construction would be completely random (indeed, the choice of an element from the candidate list is done at random). Therefore, α is a critical parameter which influences the sampling of the search space. In [173] the most important schemes to define α are listed. The simplest scheme is, trivially, to keep α constant; it can also be changed every iteration, either randomly or by means of an adaptive scheme.

The second phase of the algorithm is a local search process. This can be implemented with a basic local search algorithm such as iterative improvement, or with more advanced techniques, such as SA and TS. GRASP can be effective if two conditions hold:

- the constructed solutions sample the most promising regions in the search space.
- the solutions constructed by the constructive heuristic belong to basins of attraction of different locally minimal solutions.

The first condition can be met by the choice of a good constructive heuristic and an appropriate length of the candidate list, whereas the second condition can be met by choosing the constructive heuristic and the local search in a way such that they fit well.

As can be noted by observing its basic structure, GRASP does not use the history of the search[10], since the only memory it needs is to store the current best solution. This is one of the reasons why GRASP is often outperformed by other metaheuristics. However, due to its simplicity, it is generally very fast and it is able to produce quite good solutions in a very short amount of computation time. Furthermore, it can be successfully integrated into other search techniques. Among the applications of GRASP we mention the JSS problem [20], the graph planarization problem [188] and assignment problems [174]. A detailed and annotated bibliography references many more applications [62].

Variable Neighborhood Search

Variable Neighborhood Search (VNS) [104, 105] is a metaheuristic that explicitly applies a strategy based on dynamically changing neighborhood structures. The algorithm is very general and many degrees of freedom exist for designing variants and particular instantiations[11].

Algorithm 7 Variable Neighborhood Search (VNS)

```
Select a set of neighborhood structures N_k, k = 1, ..., k_max
s ← GenerateInitialSolution()
while termination conditions not met do
  k ← 1
  while k < k_max do {Inner Loop}
    s' ← PickAtRandom(N_k(s)){Shaking phase}
    s'' ← LocalSearch(s')
    if f(s'') < f(s) then
      s ← s''
      k ← 1
    else
      k ← k + 1
    end if
  end while
end while
```

At the initialization step, a set of neighborhood structures has to be defined. These neighborhoods can be arbitrarily chosen, but often a sequence $|\mathcal{N}_1| < |\mathcal{N}_2| < \ldots < |\mathcal{N}_{k_{max}}|$ of neighborhoods with increasing cardinality is defined[12]. Then an initial solution is generated, the neighborhood index is initialized and the algorithm iterates

[10] Some extensions in this direction are cited in [173], and an example for a metaheuristic method using an adaptive greedy procedure depending on search history is Squeaky Wheel Optimization (SWO) [124].

[11] The variants described in the following are also described in [104, 105].

[12] In principle they could be one included in the other, $\mathcal{N}_1 \subset \mathcal{N}_2 \subset \ldots \subset \mathcal{N}_{k_{max}}$. Nevertheless, such a sequence might produce an inefficient search, because a large number of solutions could be revisited.

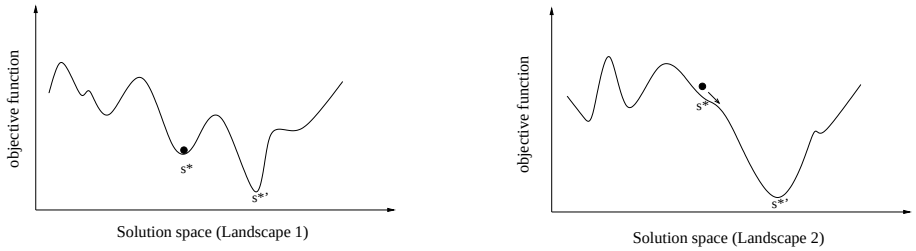

Figure 2.4: Two search landscapes defined by two different neighborhoods: the best improvement local search stops at s^* in the first, while it proceeds till a better optimum $s^{*'}$ in the second.

until a stopping condition is met (Algorithm 7). VNS' main cycle is composed of three phases: *shaking*, *local search* and *move*. In the *shaking* phase a solution s' in the k-th neighborhood of the current solution s is randomly selected. Then, s' becomes the local search starting point. The local search can use any neighborhood structure and it is not restricted to the set of neighborhood structures $\mathcal{N}_k$, $k = 1, \cdots, k_{max}$. At the end of the local search process (terminated as soon as a predefined termination condition is verified) the new solution s'' is compared with s and, if it is better, it replaces s and the algorithm starts again with $k = 1$. Otherwise, k is incremented and a new shaking phase starts using a different neighborhood.

The objective of the shaking phase is to perturb the solution so as to provide a good starting point for the local search. The starting point should belong to the basin of attraction of a different local minimum than the current one, but should not be "too far" from s, otherwise the algorithm would degenerate into a simple random multi-start. Moreover, choosing s' in the neighborhood of the current best solution is likely to produce a solution that maintains some good features of the current one.

The process of changing neighborhoods in case of no improvements corresponds to a diversification of the search. In particular the choice of neighborhoods of increasing cardinality yields a progressive diversification. The effectiveness of this dynamic neighborhood strategy can be explained by the fact that a "bad" place on the search landscape given by one neighborhood could be a "good" place on the search landscape given by another neighborhood[13]. Moreover, a solution that is locally optimal with respect to a neighborhood is probably not locally optimal with respect to another neighborhood. These concepts are known as "*One Operator, One Landscape*" and explained in [122, 123]. The core idea is that the neighborhood structure determines the topological properties of the search landscape and, for each neighborhood, one landscape is defined. The properties of a landscape are in general different from those of other landscapes, therefore the search strategy performs differently on them (see an example in Figure 2.4).

This property is directly exploited by a local search called Variable Neighborhood

[13] A "good" place in the search space is an area from which a good local minimum can be reached.

Descent (VND). In VND a *best improvement* local search (see section 2.2.1) is applied, and, in case a local minimum is found, the search proceeds with another neighborhood structure. The VND algorithm can be obtained by substituting the inner loop of the VNS algorithm (Algorithm 7) with the following pseudo-code:

$s' \leftarrow \mathsf{ChooseBestOf}(\mathcal{N}_k(s))$
if $f(s') < f(s)$ **then** {a better solution is found in the neighborhood}
  $s \leftarrow s'$
else {s is a local minimum}
  $k \leftarrow k + 1$
end if

As can be observed from the description given above, the choice of the neighborhood structures is the critical point of VNS and VND. The neighborhoods chosen should show and represent "different" properties and characteristics of the search space, that is, the neighborhood structures should give different *abstractions* of the search space. A variant of VNS is obtained by selecting the neighborhoods in such a way as to produce a problem decomposition (the algorithm is called Variable Neighborhood Decomposition Search – VNDS). VNDS follows the usual VNS scheme, but the neighborhood structures and the local search are defined on sub-problems. For each solution, all attributes (usually variables) are kept fixed except for k of them. For each k a neighborhood structure $\mathcal{N}_k$ is defined. Local search only regards changes on the variables belonging to the sub-problem it is applied to. The inner cycle of VNDS is the following:

$s' \leftarrow \mathsf{PickAtRandom}(\mathcal{N}_k(s))$ {s and s' differ in a set of k attributes}
$s'' \leftarrow \mathsf{LocalSearch}(s', \mathsf{Attributes})$ {only moves involving the k attributes are allowed}
if $f(s'') < f(s)$ **then**
  $s \leftarrow s''$
  $k \leftarrow 1$
else
  $k \leftarrow k + 1$
end if

The decision whether to perform a move can be varied as well. The acceptance criterion based on improvements is strongly steepest descent-oriented and it might not be suited to effectively explore the search space. For example, when local minima are clustered, VNS can quickly find the best optimum in a cluster, but it has no guidance to leave that cluster and find another one. Skewed VNS (SVNS) extends VNS by providing a more flexible acceptance criterion that takes also into account the distance from the current solution[14]. The new acceptance criterion is the following: besides always accepting improvements, worse solutions can be accepted if they are distant from the current one less than a value $\alpha\rho(s, s'')$. The function $\rho(s, s'')$ measures the distance between s and s'' and α is a parameter that weights the importance of the distance between the two solutions in the acceptance criterion. The inner cycle of SVNS can be sketched as follows:

[14] A distance measure between solutions has thus to be formally defined.

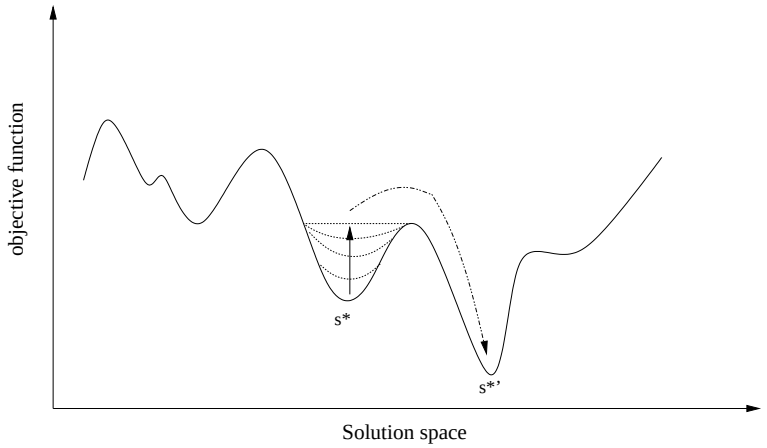

Figure 2.5: Basic GLS idea: escaping from a valley in the landscape by increasing the objective function value of its solutions.

if $f(s'') - \alpha\rho(s, s'') < f(s)$ **then**
  $s \leftarrow s''$
  $k \leftarrow 1$
else
  $k \leftarrow k + 1$
end if

VNS, and its variants, have been successfully applied to graph based COP such as the p-Median problem [103], the degree constrained minimum spanning tree problem [189], the Steiner tree problem [228] and the k-Cardinality Tree (KCT) problem [155]. References to more applications can be found in [105].

Guided Local Search

Tabu Search and Variable Neighborhood Search explicitly deal with dynamic neighborhoods with the aim of efficiently and effectively exploring the search space. A different approach for guiding the search is to dynamically change the objective function. Among the most general methods that use this approach is Guided Local Search (GLS) [227, 226].

The basic GLS principle is to help the search to move out gradually from local optima by changing the search landscape. In GLS the set of solutions and the neighborhood structure are kept fixed, while the objective function f is dynamically changed with the aim of making the current local optimum "less desirable". A pictorial description of this idea is given in Figure 2.5.

The mechanism used by GLS is based on *solution features*, which may be any kind of properties or characteristics that can be used to discriminate between solutions. For

example, solution features in the TSP could be arcs between pairs of cities, while in the MAXSAT problem they could be the number of unsatisfied clauses. An indicator function $I_i(s)$ indicates whether the feature i is present in solution s:

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

The objective function f is modified to yield a new objective function f' by adding a term that depends on the m features:

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

where p_i are called *penalty parameters* and λ is called the *regularization parameter*. The penalty parameters weight the importance of the features: the higher p_i, the higher the importance of feature i, thus the higher the cost of having that feature in the solution. The regularization parameter balances the relevance of features with respect to the original objective function.

The algorithm (see Algorithm 8) behaves as follows: it starts from an initial solution and applies a local search method until a local minimum is reached. Then the array $\mathbf{p} = (p_1, \ldots, p_m)$ of penalties is updated by incrementing some of the penalties and the local search is started again. The penalized features are those that have the maximum *utility*:

$$Util(s,i) = I_i(s) \cdot \frac{c_i}{1+p_i},$$

where c_i are *costs* assigned to every feature i giving a heuristic evaluation of the relative importance of features with respect to others. The higher the cost, the higher the utility of features. Nevertheless, the cost is scaled by the penalty parameter to prevent the algorithm from being totally biased toward the cost and to make it sensitive to the search history.

Algorithm 8 Guided Local Search (GLS)

```
s ← GenerateInitialSolution()
while termination conditions not met do
  s ← LocalSearch(s, f')
  for all feature i with maximum utility Util(s,i) do
    p_i ← p_i + 1
  end for
  Update(f',p){where p is the penalty vector}
end while
```

The penalties update procedure can be modified by adding a multiplicative rule to the simple incrementing rule (that is applied every iteration). The multiplicative rule has the form: $p_i \leftarrow p_i \cdot \alpha$, where $\alpha \in]0,1[$. This rule is applied with a lower frequency than the incrementing one (for example every few hundreds of iterations)

with the aim to decay the weights of penalized features so as to prevent the landscape from becoming too rugged. It is important to note that the penalties update rules are often very sensitive to the problem instance.

GLS has been successfully applied to the weighted MAXSAT [152], the VR problem [131], the TSP and the QAP [227].

Iterated Local Search

We conclude this presentation of explorative strategies with Iterated Local Search (ILS), the most general scheme among the explorative strategies. On the one hand, its generality makes it a framework for other metaheuristics (such as VNS); on the other hand, other metaheuristics can be easily included in it as sub-components. ILS is a simple but powerful metaheuristic algorithm [209, 208, 137, 136, 141]. It applies local search to an initial solution until it finds a local optimum; then it perturbs the solution and it restarts local search. The importance of the perturbation is obvious: too small a perturbation might not enable the system to escape from the basin of attraction of the local optimum just found. On the other side, too strong a perturbation would make the algorithm similar to a random restart local search.

A local search is effective if it is able to find good local minima, that is, if it can find the basin of attraction[15] of those states. When the search space is wide and/or when the basins of attraction of good local optima are small, a simple multi-start algorithm is almost useless. An effective search could be designed as a trajectory only in the set of local optima S^*, instead of in the set S of all the states. Unfortunately, the only way of introducing a neighborhood structure in S^* would be constructing (or at least, sampling) the basin of attraction of local optima involved in the search. As this is not tractable in practice, a trajectory along local optima $s_1^*, s_2^*, \ldots, s_t^*$ can be performed without explicitly introducing a neighborhood structure, by applying the following scheme:

1. Execute local search (LS) from an initial state s until a local optimum s^* is found.
2. *Perturb* s^* and obtain s'.
3. Execute LS from s' until it finds a local optimum $s^{*'}$.
4. On the basis of an *acceptance criterion* decide whether to set $s^* \leftarrow s^{*'}$.
5. Goto step 2.

The (ideal) task of the *perturbation* on s^* is to produce a starting point for local search such that it ends in a local optimum different from s^*, but *closer* than a local optimum reachable with a random restart. The *acceptance criterion* acts as a counterbalance, as it filters and gives feedbacks to the perturbation action, depending

[15] The basin of attraction size of a point s (in a finite space), is defined as the fraction of initial states of trajectories which converge to point s.

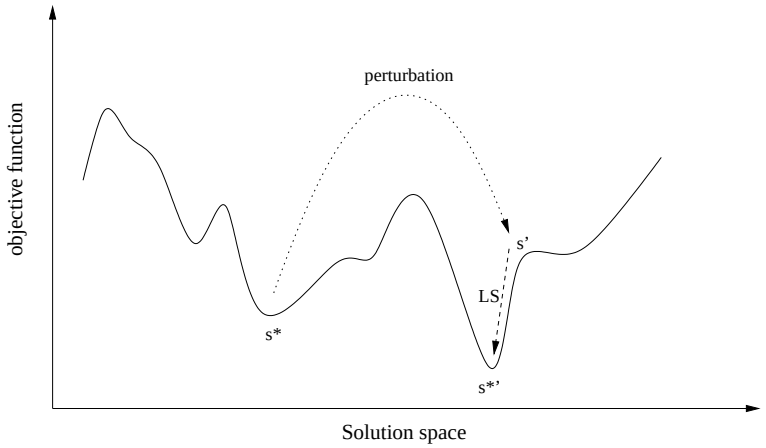

Figure 2.6: A desirable ILS step: the local optimum s^* is perturbed, then LS is applied and a new local minimum is found.

on the characteristics of the new local optimum. A high level description of ILS as it appears in [137] is reported in Algorithm 9. Figure 2.6 shows a possible (lucky) ILS step.

Algorithm 9 Iterated Local Search (ILS)

$s_0 \leftarrow$ GenerateInitialSolution()
$s^* \leftarrow$ LocalSearch(s_0)
while termination conditions not met **do**
  $s' \leftarrow$ Perturbation($s^*, history$)
  $s^{*'} \leftarrow$ LocalSearch(s')
  $s^* \leftarrow$ ApplyAcceptanceCriterion($s^*, s^{*'}, history$)
end while

The design of ILS algorithms has several degrees of freedom in the choice of the initial solution, perturbation and acceptance criteria. A key role is played by the *history* of the search which can be exploited both as short and long term memory.

The construction of initial solutions should be fast (computationally not expensive), and initial solutions should be a good starting point for local search. The fastest way of producing an initial solution is to generate it at random; however, this is the easiest way for problems where every possible assignment is a feasible solution, whilst in other cases the construction of a feasible solution requires also constraint checking. Constructive methods, guided by heuristics, can also be adopted. It is worth underlining that an initial solution is considered a good starting point depending on the particular LS applied and on the problem structure, thus the algorithm designer's goal is to find a trade-off between speed and quality of solutions.

The perturbation is usually non-deterministic, to avoid cycling. Its most important characteristic is the *strength*, roughly defined as the amount of changes made on the current solution. The strength can be either fixed or variable. In the first case, the distance between s^* and s' is kept constant, independently of the problem size. However, a dynamic strength is in general more effective, since it has been experimentally found that, in most of the problems, the bigger the problem size, the larger should be the strength. More sophisticated schemes are possible; for example, the strength can be adaptive: it increases when more diversification is needed and it decreases when intensification seems preferable. VNS and its variants belong to this category. A second choice is the mechanism to perform perturbations. This may be a random mechanism, or the perturbation may be produced by a (semi–)deterministic method (e.g., a LS different from the one used in the main algorithm).

The third important component is the acceptance criterion. There are two extreme cases: accept the new local optimum only in case of improvement or always accept the new state. In-between, there are several possibilities. For example, it is possible to adopt a kind of annealing schedule: accept all the improving new local optima and accept also the non-improving ones with a probability that is a function of the temperature T and the difference of objective function values. In formulas:

$$Prob[Accept(s^*, s', history)] = \begin{cases} 1 & : \text{ if } f(s^{*'}) < f(s^*) \\ \exp(-\frac{f(s^{*'}) - f(s^*)}{T} & : \text{ otherwise} \end{cases}$$

The cooling schedule can be either monotonic (non-increasing in time) or non-monotonic (adapted to tune the balance between diversification and intensification). The non-monotonic schedule is particularly effective if it exploits the history of the search, in a way similar to the Reactive Tabu Search [215] mentioned at the end of the section about Tabu Search. When intensification seems no longer effective, a diversification phase is needed and the temperature is increased.

Examples for successful applications of ILS are to the TSP [140, 121], to the QAP [137], and to the Single Machine Total Weighted Tardiness (SMTWT) problem [40]. References to other applications can be found in [137].

2.4 Population-based methods

Population-based methods deal with a set (a population) of solutions rather than with a single solution. Since they deal with a population of solutions, population-based algorithms provide a natural, intrinsic way for the exploration of the search space. Yet, the final performance depends strongly on the way the population is manipulated. The most studied population-based methods in combinatorial optimization are Evolutionary Computation (EC) and Ant Colony Optimization (ACO). In EC algorithms a population of individuals is modified by recombination and mutation operators, and in ACO a colony of artificial ants is used to construct solutions guided by the pheromone trails and heuristic information.

2.4.1 Evolutionary Computation (EC)

Evolutionary Computation (EC) algorithms are inspired by nature's capability to evolve living beings well adapted to their environment. EC algorithms can be succinctly characterized as computational models of evolutionary processes. In every iteration a number of operators is applied to the individuals of the current population to generate the individuals of the population of the next generation (iteration). Usually, EC algorithms use operators called *recombination* or *crossover* to recombine two or more individuals to produce new individuals. They also use *mutation* or *modification* operators which cause a self-adaptation of individuals. The driving force in evolutionary algorithms is the *selection* of individuals based on their *fitness* (this can be the value of an objective function or the result of a simulation experiment, or some other kind of quality measure).

Individuals with higher fitness have higher probability to be chosen as members of the population of the next iteration (or as parents for the generation of new individuals). This corresponds to the principle of *survival of the fittest* in natural evolution. It is the capability of nature to adapt itself to a changing environment, which gave the inspiration for EC algorithms.

EC algorithms heve been proposed in many variants over the years. They basically fall into three different categories, which have been developed independently from each other: Evolutionary Programming (EP) [69, 70], Evolutionary Strategies (ES) [180] and Genetic Algorithms [114] (see also [92], [154], [183] and [224] for further references). EP arose from the desire to generate machine intelligence. While EP originally was proposed to operate on discrete representations of finite state machines, most of the present variants are used for continuous optimization problems. The latter also holds for most present variants of ES, whereas GAs are mainly applied to solve combinatorial optimization problems. Overviews and surveys about EC methods can be found [7], [67], [204], [148] and [28].

In the following we concentrate on a "combinatorial optimization"-oriented introduction to EC algorithms. In this perspective, we follow an overview by Hertz et al. [109], which gives, in our opinion, a good overview of the different components of EC algorithms and of the possibilities to define them. Algorithm 10 shows the basic structure of every EC algorithm.

Algorithm 10 Evolutionary Computation

```
P ← GenerateInitialPopulation()
Evaluate(P)
while termination conditions not met do
  P' ← Recombine(P)
  P'' ← Mutate(P')
  Evaluate(P'')
  P ← Select(P'' ∪ P)
end while
```

In this algorithm, P denotes the population of individuals. A population of offspring is generated by *recombination* and *mutation* operators and the individuals for the next population are *selected* from the union of the old population and the offspring population. The main features of an EC algorithm are outlined in the following.

Description of the individuals: EC algorithms handle populations of individuals. These individuals are not necessarily solutions of the considered problem. They may be partial solutions, or sets of solutions, or any object which can be transformed into one or more solutions in a structured way. Most commonly used in combinatorial optimization is the representation of solutions as bit-strings or as permutations of n integer numbers. Tree-structures or other complex structures are also possible. In the context of Genetic Algorithms, individuals are called *genotypes*, whereas the solutions that are encoded by individuals are called *phenotypes*. This is to differentiate between the representation of solutions and solutions themselves. The choice of an appropriate representation is crucial for the success of an EC algorithm. Holland's schema analysis [114] and Radcliffe's generalization to formae [177] are examples of how theory can help to guide representation choices.

Evolution process: In each iteration it has to be decided which individuals will enter the population of the next iteration. This is done by a selection scheme. The strategy of only choosing among the offspring as individuals for the next population is called *generational replacement*. On the contrary, in the *steady state* evolution process, it is possible to transfer individuals of the current population into the next population.

Most EC algorithms work with populations of fixed size keeping at least the best individual always in the current population. It is also possible to have a variable population size. In case of a continuously shrinking population size, the situation where only one individual is left in the population (or no crossover partners can be found for any member of the population) might be one of the stopping conditions of the algorithm.

Neighborhood structure: A neighborhood function $\mathcal{N}_{\mathcal{EC}} : \mathcal{I} \to 2^{\mathcal{I}}$ on the set of individuals $\mathcal{I}$ assigns to every individual $i \in \mathcal{I}$ a set of individuals $\mathcal{N}_{\mathcal{EC}}(i) \subseteq \mathcal{I}$ which are permitted to act as recombination partners for i to create offspring. In the case of *unstructured* populations, an individual can be recombined with any other individual (e.g., in simple GA), whilst in *structured* populations neighborhoods are restricted. An example for an EC algorithm using structured populations is the Parallel Genetic Algorithm proposed by in [159].

Information sources: The most common form of information sources to create offspring (i.e., new individuals) is a couple of parents (two-parent crossover). But there are also recombination operators recombining more than two individuals to create a new individual (multi-parent crossover), see [55]. More recent developments even use population statistics for generating the individuals of the next population.

Examples are the recombination operators called Gene Pool Recombination [161] and Bit-Simulated Crossover [213] which make use of a distribution over the search space given by the current population to generate the next population.

Infeasibility: An important characteristic of an EC algorithm is the way it deals with infeasible individuals. When recombining individuals the offspring might be infeasible. There are mainly three different ways to handle such a situation. The most simple action is to *reject* infeasible individuals. Nevertheless, for many problems it might be very difficult to find feasible individuals. Therefore, the strategy of *penalizing* infeasible individuals in the function that measures the quality of an individual is sometimes more appropriate (or even unavoidable). The third possibility consists of trying to *repair* an infeasible solution (see [56] for an example).

Intensification strategy: In many applications it proved to be beneficial to use *improvement algorithms* to improve the fitness of individuals. EC algorithms using a local search algorithm on every individual of a population are often called Memetic Algorithms [157, 158]. While the use of a population ensures an exploration of the search space, the use of local search techniques helps to quickly identify "good" areas in the search space.

Another intensification strategy is the use of recombination operators that explicitly try to combine "good" parts of individuals (rather than, for example, a simple one-point crossover for bit-strings). This also concentrates the search performed by the EC algorithm to areas of individuals with certain "good" properties. Techniques of this kind are sometimes called *linkage learning* or *building block learning* (see [94, 221, 233, 106] as examples).

Diversification strategy: One of the major difficulties of EC algorithms (especially when applying local search) is the premature convergence toward sub-optimal solutions. The most simple mechanism to diversify the search process is the use of a mutation operator. The simple form of a mutation operator just performs a small random perturbation of an individual, introducing a kind of *noise*. In order to avoid premature convergence there are ways of maintaining the population diversity, such as *niching*, whereby the reproductive fitness allocated to an individual in a population is reduced proportionally to the number of other individuals that share the same region of the search space.

This concludes the list of the main features of EC algorithms. EC algorithms have been applied to nearly every combinatorial optimization problem. Recent successes were obtained in the rapidly growing bioinformatics area (see for example [68]), but also in multi-objective optimization [33], and in evolvable hardware [202]. For an extensive collection of references to EC applications we refer to [8].

Other populations-based methods, namely Scatter Search and Path Relinking [89, 91], are sometimes also regarded as being EC algorithms. Scatter Search and its generalized form called Path Relinking [89, 91] differ from EC algorithms mainly by

providing unifying principles for joining (or recombining) solutions based on generalized path constructions in Euclidean or neighborhood spaces. They also incorporate some ideas originating from Tabu Search methods, as, for example, the use of adaptive memory and associated memory-exploiting mechanisms.

In the last decade researchers tried to overcome the drawbacks of usual recombination operators of EC algorithms which are likely to break good building blocks. So, a number of algorithms (sometimes called Estimation of Distribution Algorithms (EDA) [160]) have been developed. These algorithms have a theoretical foundation in probability theory and are based on populations that evolve as the search progresses. EDAs use probabilistic modeling of promising solutions to estimate a distribution over the search space which is then used to produce the next generation by sampling the search space according to the estimated distribution. After every iteration the distribution is re-estimated. For a survey of EDAs see [170]. One of the first EDAs proposed for combinatorial optimization is called Population-Based Incremental Learning (PBIL) [9, 10]. The objective of this method is to create a real valued probability vector (each position corresponds to a binary decision variable) which, when used to sample the search space, generates high quality solutions with high probability. Initially, the values of the probability vector are initialized to 0.5 (for each variable there is equal probability to be assigned to 0 or 1). The goal of shifting the values of this probability vector in order to generate high quality solutions is accomplished as follows: a number of solution vectors are generated according to the probability vector. Then the probability vector is shifted toward the generated solution vector(s) with highest quality. The distance that the probability vector is pushed depends on the learning rate parameter. Then, a mutation operator is applied to the probability vector. After that, the cycle is repeated. The probability vector can be viewed 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. In contrast to PBIL which is estimating a distribution of promising solutions assuming that the decision variables are independent, various other approaches try to estimate distributions taking into account dependencies between decision variables.

2.4.2 Ant Colony Optimization

A recent and important class of nature inspired algorithms is that of ant algorithms [25]. These are algorithms inspired by the observation of social insect behavior, and in particular by the behavior of ant colonies. In these algorithms, the traditional emphasis on control, programming, and centralization is replaced by an emphasis on autonomy, emergence, and distributed functioning. A particularly successful research direction in ant algorithms, known as Ant Colony Optimization [46, 50, 48], is concerned with applications to discrete optimization problems.

Ant System (AS) is the earliest example of this kind of algorithms. AS was first applied to solve the Traveling Salesman Problem (TSP) and it achieved encouraging results, yet not competitive with the state of the art on large problem instances. AS

Algorithm 11 Ant Colony Optimization

```
while stopping criterion not satisfied do
  ScheduleActivities
      AntBasedSolutionConstruction()
      PheromoneUpdate()
      DaemonActions() {optional}
  end ScheduleActivities
end while
```

has been further modified and extended, and several variants have been designed. Ant Colony Optimization (ACO) metaheuristic is a general framework for ant algorithms (and their variants) applied to combinatorial optimization. In the following, we define the problem representation adopted in ACO and we outline the high level algorithm.

The main entities of ACO are artificial ants (hereafter called simply ants) that "walk" on a connected graph, called construction graph $G = (C, L)$, where arcs L (connections) fully connect nodes C (components). The combinatorial problem at hand is mapped onto G in such a way that feasible solutions to the original problem correspond to paths on G. Connections, components, or both, can have associated a pheromone trail and a heuristic value. Pheromone trails provide a kind of distributed long-term memory which encodes the history of the whole ants' search process. Heuristic values represent a priori information on the problem or dynamic heuristic information.

In ACO, ants are no longer reactive agents without memory. They are essentially stochastic solution construction procedures, equipped with memory to store the solution built (i.e., the path described in the graph) and heuristic information. Ants move on the basis of a construction policy that is a function of the problem constraints. They build paths by incrementally adding a node, among the feasible ones, to the current path by taking probabilistic decisions. A state transition rule returns the probability of adding a node to the current path. Once a solution is completed, the ant evaluates it and retraces the same path backward depositing on nodes (or arcs) an amount of pheromone proportional to the solution quality. This action is called *online delayed pheromone update*. It is also possible for the ants to add or remove pheromone during the path construction, executing the so called *step-by-step pheromone update*. The information provided by pheromone trails will guide the solution construction of future ants. Pheromone evaporation is achieved by executing a procedure called *pheromone trail evaporation*, which decreases pheromone on the whole graph (usually by decreasing each amount τ of a fraction $\rho\tau$, with $0 < \rho < 1$). ACO includes also optional activities called *daemon actions*, which are non local procedures, such as the application of local search to solutions, or the pheromone update on the path corresponding to the best solution found from the beginning of the run. The high level scheme of ACO is given in Algorihtm 11.

The ScheduleActivities construct does not specify how the three inner activities are scheduled and synchronized and the designer is free to specify how these procedures

interact.

As an example of a specific implementation of ACO, we briefly describe Ant System applied to the TSP.

The TSP can be represented in ACO as follows:

- nodes of G (the components) are the cities to be visited;
- a solution is an Hamiltonian tour in the graph;
- constraints are used to avoid cycles (an ant can not visit a city more than once).

The ant colony is composed of m ants that iterate the same sequence of actions until a termination condition is verified. At the beginning of the cycle ants are randomly put on the cities (nodes of the construction graph G). Starting from its start city, an ant moves from city to city to build an Hamiltonian tour. For the ant k, the probability of moving from city i to city j is given by the following state transition rule:

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} & \text{if} \quad j \in \mathcal{N}_i^k \\ 0 & \text{otherwise} \end{cases}$$

where τ_{ij} is the pheromone laid on arc (i,j), $\eta_{ij} = 1/distance(i,j)$ is the heuristic, the parameters α and β balance the relative influence of pheromone and heuristic, and $\mathcal{N}_i^k$ is the set of cities not yet visited by ant k.

The pheromone trail evaporation is ruled by the following formula:

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} \qquad \forall (i,j)$$

where ρ $(0 < \rho < 1)$ is the evaporation parameter.

Finally, the delayed pheromone update rule adjusts pheromone so as to make more preferable arcs belonging to short tours:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k \qquad \forall (i,j)$$

$$\Delta\tau_{ij}^k = \begin{cases} 1/L^k & \text{if arc } (i,j) \text{ is used by ant } k \\ 0 & \text{otherwise,} \end{cases}$$

where L_k is the length of the tour built by ant k. In Ant System no daemon rules are applied (e.g., every constructed solution can be used as initial solution for the application of local search, in a similar way to hybrid genetic algorithms).

Within the ACO metaheuristic framework, as shortly described above, the currently best performing versions in practice are Ant Colony System (ACS) [49] and $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [212].

Ant Colony System (ACS): The ACS algorithm has been introduced to improve the performance of AS. ACS is based on AS but presents some important differences. First, the daemon updates pheromone trails offline: at the end of an iteration of the algorithm (i.e., once all the ants have built a solution) pheromone is added to the arcs used by the ant that found the best solution from the start of the algorithm. Second, ants use a different decision rule to decide to which component to move next in the construction graph. The rule is called *pseudo-random-proportional* rule. With this rule, some moves are chosen deterministically (in a greedy manner), others are chosen probabilistically with the usual decision rule. Third, in ACS, ants perform only online step-by-step pheromone updates. These updates are performed to favor the emergence of other solutions than the best so far.

$\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS): $\mathcal{MM}$AS is also an extension of AS. First, the pheromone trails are only updated offline by the daemon (the arcs that were used by the iteration best ant or the best ant since the start of the algorithm receive additional pheromone). Second, the pheromone values are restricted to an interval $[\tau_{min}, \tau_{max}]$ and the pheromone trails are initialized to their maximum value τ_{max}. Explicit bounds on the pheromone trails avoid that the probability to construct a solution falls below a certain value greater than 0. This means that the chance of finding a global optimum never vanishes during the course of the algorithm.

Current research is focused on the theoretical foundations of ACO (convergence theorems have recently been proven [211]) and the investigation of relations between ACO and other search methods such as gradient descent and Monte Carlo algorithms [146]. Recently, the similarities between ACO algorithms and probabilitic learning algorithms such as EDAs have been recognized and studied. An important step into this direction was the development of the Hyper-Cube Framework for Ant Colony Optimization (HC-ACO) [24]. An extensive study on this subject has been presented in [244], where the authors present a unifying framework for so-called Model-Based Search (MBS) algorithms. Also, the close relation of algorithms like Population-Based Incremental Learning (PBIL) [10] to ACO algorithms in the Hyper-Cube Framework has been shown. We refer the interested reader to [244] for more information on this subject.

Successful applications of ACO include the application to routing in communication networks [43], the application to the Sequential Ordering Problem (SOP) [77], and the application to Resource Constraint Project Scheduling (RCPS) [143]. Further references to applications of ACO can be found in [51, 52].

2.5 Intensification and Diversification: The Key Concepts

In this section we consider metaheuristics from the standpoint of the use they make of intensification and diversification. Although the relevance of these two concepts is commonly agreed, so far there is no unifying description to be found in the literature. Descriptions are very generic and metaheuristic specific. Therefore most of them can be considered incomplete and sometimes they are even opposing. Depending on the paradigm behind a particular metaheuristic, intensification and diversification are achieved in different ways. In [23] a unifying view on intensification and diversification is proposed, based on an Intensification/Diversification frame which directly relates the two concepts. In the following, we discuss the main implementations of intensification and diversification, by observing that there are two levels: basic and strategic. The first level characterizes the basic algorithmic components, which are intrinsic to the algorithm. The second level, describes general strategies (usually of medium and long term) which are applied to improve the performance of the metaheuristic.

2.5.1 Intensification and Diversification

Every metaheuristic approach should be designed with the aim of effectively and efficiently exploring a search space. The search performed by a metaheuristic approach should be "clever" enough to both intensively explore areas of the search space with high quality solutions, and to move to unexplored areas of the search space when necessary. The concepts for reaching these goals are nowadays called intensification and diversification.

An implicit reference to the concept of "locality" is often introduced when intensification and diversification are involved. The notion of "area" (or "region") of the search space and of "locality" can only be expressed in a fuzzy way, as they always depend on the characteristics of the search space as well as on the definition of metrics on the search space (distances between solutions).

The literature provides several high level descriptions of intensification and diversification. In the following we cite some of them.

"Two highly important components of Tabu Search are intensification and diversification strategies. Intensification strategies are based on modifying choice rules to encourage move combinations and solution features historically found good. They may also initiate a return to attractive regions to search them more thoroughly. Since elite solutions must be recorded in order to examine their immediate neighborhoods, explicit memory is closely related to the implementation of intensification strategies. **The main difference between intensification and diversification is that during an intensification stage the search focuses on examining neighbors of elite solutions.** [...] **The diversification stage on the other hand encourages the search process to examine unvisited regions and to generate solutions that differ in various significant ways from those seen before.**" F. Glover

and M. Laguna in [90]
Later in the same book, Glover and Laguna write: "In some instances we may conceive of intensification as having the function of an intermediate term strategy, while diversification applies to considerations that emerge in the longer run."
Furthermore, they write: "Strategic oscillation is closely linked to the origins of tabu search, and provides a means **to achieve an effective interplay between intensification and diversification**."

"After a local minimizer is encountered, all points in its attraction basin lose any interest for optimization. The search should avoid wasting excessive computing time in a single basin and diversification should be activated. On the other hand, in the assumptions that neighbors have correlated cost function values, some effort should be spent in searching for better points located close to the most recently found local minimum point (intensification). **The two requirements are conflicting and finding a proper balance of diversification and intensification is a crucial issue in heuristics.**" R. Battiti in [12].

"**A metaheuristic will be successful on a given optimization problem if it can provide a balance between the exploitation of the accumulated search experience and the exploration of the search space to identify regions with high quality solutions in a problem specific, near optimal way.**" T. Stützle in [209].

"Intensification is to search carefully and intensively around good solutions found in the past search. Diversification, on the contrary, is to guide the search to unvisited regions. These terminologies are usually used to explain the basic elements of Tabu Search, but these are essential to all the metaheuristic algorithms. In other words, various metaheuristic ideas should be understood from the view point of these two concepts, and **metaheuristic algorithms should be designed so that intensification and diversification play balanced roles.**" M. Yagiura and T. Ibaraki in [242].

"Holland frames adaption as a tension between exploration (the search for new, useful adaptations) and exploitation (the use and propagation of these adaptations). The tension comes about since any move toward exploration – testing previously unseen schemas or schemas whose instances seen so far have low fitness – takes away from the exploitation of tried and true schemas. In any system (e.g., a population of organisms) required to face environments with some degree of unpredictability, **an optimal balance between exploration and exploitation must be found**. The system has to keep trying out new possibilities (or else it could "overadapt" and be inflexible in the face of novelty), but it also has to continually incorporate and use past experience as a guide for future behavior." M. Mitchel citing J.H. Holland in [154].

All these descriptions share a common view: that there are two forces for which an appropriate balance has to be found. Sometimes these two forces were described as opposing forces. However, lately some researchers raised the question, how opposing intensification and diversification really are.

Especially the TS literature advocates the view that intensification and diversification cannot be characterized as opposing forces. For example, in [90], the authors write: "Similarly, as we have noted, intensification and diversification are not opposed notions, for the best form of each contains aspects of the other, along a spectrum of alternatives."

Intensification and diversification can be considered as effects of algorithm components. We define an *I&D component* as any algorithmic or functional component that has an intensification and/or a diversification effect on the search process. Accordingly, examples of I&D components are genetic operators, perturbations of probability distributions, the use of tabu lists, or changes in the objective function. Thus, I&D components are operators, actions, or strategies of metaheuristic algorithms.

In contrast to the still widely spread view that there are components that have either an intensification or a diversification effect, there are many I&D components that have both. In I&D components that are commonly labeled as intensification, the intensification component is stronger than the diversification component, and vice versa. We can characterize two extremes: pure intensification and pure diversification. We will show that most (if not all) the components and strategy used in metaheuristics are located in between the two extremes. In other words, they all have both an intensification and a diversification effect, being one stronger than the other. We say that a component (or a strategy) has a pure intensification effect if it is solely guided by the objective function. Conversely, components only guided by functions other than the objective function (e.g., random, frequency-based rank, etc.) have a pure diversification effect[16].

2.5.2 Basic Level of Intensification and Diversification

The I&D components occurring in metaheuristics can be divided in basic (or intrinsic) ones and strategic ones. The basic I&D components are the ones that are defined by the basic ideas of a metaheuristic. On the other side, strategic I&D components are composed of techniques and strategies the algorithm designer adds to the basic metaheuristic in order to improve the performance by incorporating medium and long term strategies. Many of these strategies were originally developed in the context of a specific metaheuristic. However, many of these strategies can also be very useful when applied in other metaheuristics. In the following, we choose some basic I&D components that are inherent to a metaheuristic and show that most of the basic I&D components have an intensification character as well as a diversification character.

[16] In [23] this discussion is based on a frame which grafically show these relations.

For many components and strategies of metaheuristics it is obvious that they involve an intensification as well as a diversification component, because they make an explicit use of the objective function. For example, the basic idea of TS is a neighbor choice rule using one or more tabu lists. This I&D component has two effects on the search process. The restriction of the set of possible neighbors in every step has a diversifying effect on the search, whereas the choice of the best neighbor in the restricted set of neighbors (the best non-tabu move) has an intensifying effect on the search. The balance between these two effects can be varied by the length of the tabu list. Shorter tabu lists result in a lower influence of the diversifying effect, whereas longer tabu lists result in an overall higher influence of the diversifying effect. Another example for such an I&D component is the probabilistic acceptance criterion in conjunction with the cooling schedule in SA. The acceptance criterion is guided by the objective function and it also involves a changing amount of randomness. The decrease of the temperature parameter drives the system from diversification to intensification, eventually leading to a convergence of the system[17]. A third example is the following one. Ant Colony Optimization provides an I&D component that manages the update of the pheromone values. This component has the effect of changing the probability distribution that is used to sample the search space. It is guided by the objective function (solution components found in better solutions than others are updated with a higher amount of pheromone) and it is also influenced by a function applying the pheromone evaporation.

For other strategies and components of metaheuristics it is not immediately obvious that they have both, an intensification and a diversification effect. An example is the random selection of a neighbor from the neighborhood of a current solution, as it is done for instance in the kick-move mechanism of ILS. Initially one might think that there is no intensification involved and that this mechanism has a pure diversification effect caused by the use of randomness. However, for the following reason this is not the case. Many strategies involve the explicit or implicit use of a neighborhood. A neighborhood imposes a structure on the search space, since it defines the topology of the so-called *fitness landscape* [205, 206, 122, 128] which can be visualized as a labeled graph, as defined in Sec. 2.3.1. A fitness landscape can be analyzed by means of statistical measures. One of the common measures is the *auto-correlation*, that provides information about how much the fitness will change when a move is made from one point to a neighboring one. Different landscapes differ in their *ruggedness*. A landscape with small (average) fitness differences between neighboring points is called *smooth* and it will usually have just a few local optima. In contrast, a landscape with large (average) fitness differences is called *rugged* and it will be usually characterized by many local optima. Most of the neighborhoods used in metaheuristics provide some degree of smoothness that is higher than the one of a fitness landscape defined by a random neighborhood. This means that such a neighborhood is, in a sense, preselecting for every solution a set of neighbors for which the average fitness is not too different. Therefore, even when a solution is randomly selected from a set of

[17] Here we use the term convergence in the sense of getting stuck in the basin of attraction of a local minimum.

neighbors, the objective function guidance is implicitly present. The consequence is that even for a random kick-move there is some degree of intensification involved, as far as a non-random neighborhood is considered.

Dual considerations hold for a random mutation operator of an EC method. In the following we assume a bit-string representation and a mutation operator that is characterized by flipping each bit of the solution with a certain probability. The implicit neighborhood used by this operator is the completely connected neighborhood (i.e., every pair of solutions is directly connected). However, the neighbors have different probabilities to be selected. The ones that are closer (with respect to the Hamming distance) to the solution to which the operator is applied to, have a higher probability to be generated by the operator. Thus, with the same argument as above, we observe that the objective function is implicitly used as a guidance. The balance between intensification and diversification is determined by the probability to flip each bit. The higher this probability, the higher the diversification effect of the operator. In contrast, the lower this probability, the higher the intensification effect of this operator.

On the other side, there are some strategies that are often labeled as pure intensification. One example is the selection operator in EC algorithms. However, nearly all selection operators involve some degree of randomness (e.g., proportionate selection, tournament selection). This means that they also have a diversifying effect. The balance between intensification and diversification depends on the function that assigns the selection probabilities. If the differences between the selection probabilities are quite high, the intensification effect is higher, and similarly for the other extreme of having only small differences between the selection probabilities.

Even an operator like the neighbor choice rule of a steepest descent local search has a diversifying component. Indeed, the search is "moving" in the search space with respect to a neighborhood. A neighborhood can be regarded as a function other than the objective function, making implicit use of the objective function. Therefore, a steepest descent local search has both a strong intensification effect, but also a weak diversification character.

Based on these observations we conclude that probably most of the basic I&D components used in metaheuristics have both, an intensification and a diversification effect. However, the balance between intensification and diversification might be quite different for different I&D components.

2.5.3 Strategic control of intensification and diversification

The right balance between intensification and diversification is needed to obtain an effective metaheuristic. Moreover, this balance should not be fixed or only changing into one direction (e.g., continuously increasing intensification). This balance should rather be dynamical. This issue is often treated in the literature, both implicitly and explicitly, when strategies to guide search algorithms are discussed.

The distinction between intensification and diversification is often interpreted with respect to the temporal horizon of the search. Short term search strategies can be seen as the iterative application of tactics with a strong intensification character (for instance, the repeated application of greedy moves). When the horizon is enlarged, usually strategies referring to some sort of diversification come into play. Indeed, a general strategy usually proves its effectiveness especially in the long term.

The simplest strategy that coordinates the interplay of intensification and diversification and can achieve an oscillating balance between them is the restart mechanism: under certain circumstances (e.g., local optimum is reached, no improvements after a specific number of algorithm cycles, stagnation, no diversity) the algorithm is restarted. The goal is to achieve a sufficient coverage of the search space in the long run, thus the already visited regions should not be explored again. The computationally least expensive attempt to address this issue is a random restart.

Usually, the most effective restart approaches make use of the search history. Examples for such restart strategies are the ones based on concepts such as *global frequency* and *global desirability*. The concept of *global frequency* is well known from TS applications. In this concept, the number of occurrences of solution components is counted during the run of the algorithm. These numbers, called the global frequency numbers, are then used for changing the heuristic constructive method, for example to generate a new population for restarting an EC method or the initial solution for restarting a trajectory method. Similarly, the concept of *global desirability* (which keeps for every solution component the objective function value of the best solution it had been a member of) can be used to restart algorithms with a bias toward good quality solutions. Examples of the use of non-random restarts can be found also in in population-based methods. In EC algorithms the new population can be generated by applying constructive heuristics[18]. In ACO, this goal is addressed by smoothing or resetting pheromone values [212]. In the latter case, if the pheromone reset is also based on the search history, the action is located inside the I&D frame.

There are also strategies explicitly aimed at dynamically changing the balance between intensification and diversification during the search. A fairly simple strategy is used in SA, where an increase in diversification and simultaneous decrease in intensification can be achieved by "re-heating" the system and then cooling it down again (which corresponds to increasing parameter T and decreasing it again according to some scheme). Such a cooling scheme is called *non-monotonic* cooling scheme (e.g., see [138] or [166]). Another example can be found in Ant Colony System (ACS). This ACO algorithm uses an additional I&D component aimed at introducing diversification during the solution construction phase. While an ant is walking on the construction graph to construct a solution it reduces the pheromone values on the nodes/arcs of the construction graph that it visits. This has the effect to reduce for the other ants the probability of taking the same path. This additional pheromone update mechanism is called *step-by-step online pheromone update rule*. The interplay between this component and the other pheromone update rules (*online delayed phero-*

[18] See, for example, [75, 99].

mone update rules and *online pheromone update rule*) leads to an oscillating balance between intensification and diversification.

Some more advanced strategies can be found in the literature. Often, they are described with respect to the particular metaheuristic in which they are applied. However, many of them are very general and can be easily adapted and reused also in a different context. A very effective example is *Strategic Oscillation* [90][19]. This strategy can be applied both to constructive methods and improvement algorithms. Actions are invoked with respect to a critical level (*oscillation boundary*), which usually corresponds to a steady state of the algorithm. Examples for steady states of an algorithm are local minima, completion of solution constructions, or the situation were no components can be added to a partial solution such that it can be completed to a feasible solution. The oscillation strategy is defined by a pattern indicating the way to approach the critical level, to cross it and to cross it again from the other side. This pattern defines the distance of moves from the boundary and the duration of phases (of intensification and diversification). Different patterns generate different strategies; moreover, they can also be adaptive and change depending on the current state and history of the search process. Other representative examples of general strategies that dynamically coordinate intensification and diversification can be found in [13, 21, 22].

Furthermore, strategies can deal not only with single actions (e.g., variable assignments, moves), but can also guide the application of coordinated sequences of moves. Examples of such a strategy are given by so-called *ejection chain* procedures [90, 184, 185]. These procedures provide a mechanism to perform compound moves, i.e., compositions of different kinds of moves. For instance, in a problem defined over a graph (e.g., the VRP), it is possible to define two different moves: insertion and exchange of nodes; a compound move can thus be defined as the combination of an insertion and an exchange move. These procedures describe general strategies to combine the application of different neighborhood structures, thus they provide an example of a general diversification/intensification interplay. Further examples of strategies which can be interpreted as mechanisms to produce compositions of interlinked moves can also be found in the literature concerning the integration of metaheuristics and complete techniques [29, 201].

2.5.4 Hybridization of metaheuristics

We conclude this chapter by discussing a very promising research issue: the hybridization of metaheuristics. In fact, many of the successful applications that we have cited in previous sections are hybridizations[20].

One of the most popular ways of hybridization concerns the use of trajectory methods in population-based methods. Most of the successful applications of EC

[19] Indeed, in [90] and in the literature related to TS, many strategies are described and discussed.
[20] For a taxonomy of hybrid metaheuristics see [217].

and ACO make use of local search procedures. The reason for that becomes apparent when we analyze the respective strengths of trajectory methods and population-based methods.

The power of population-based methods is certainly the fact that they recombine solutions to obtain new ones. In EC algorithms and Scatter Search explicit recombinations are implemented by one or more recombination operators. In ACO and EDAs recombination is implicit, because new solutions are generated by using a distribution over the search space which is a function of earlier populations. This enables to make guided steps in the search space which are usually "larger" than the steps done in trajectory methods. In other words, a solution resulting from a recombination in population-based methods is usually more "different" from the parents than, say, a predecessor solution to a successor solution (obtained by applying a move) in Tabu Search. We also have "big" steps in trajectory methods such as ILS and VNS, but in these methods the steps are usually not guided (these steps are rather called "kick move" or "perturbation" indicating the lack of guidance). It is interesting to note, that in all population-based methods there are mechanisms in which good solutions found during the search influence the search process in the hope to find better solutions in-between those solutions and current solutions. In Path Relinking this idea is implemented in the most explicit way. The basic elements are guiding solutions (which are the good solutions found) and initiating solutions. New solutions are produced by applying moves to decrease the distance between the resulting solution and the guiding solution. In EC algorithms this is often obtained by keeping the best (or a number of best) solution(s) found since the beginning of the respective run of the algorithm in the population. In some ACO implementations (see for example [212, 22]) a particular pheromone updating schedule is applied. When the algorithm has nearly converged to a solution, only the best found solution since the start of the algorithm is used for updating the pheromone trails. This corresponds to "changing direction" and directing the search process toward a very good solution in the hope to find better ones on the way.

The strength of trajectory methods is rather in the way they explore a promising region in the search space. As in those methods local search is the driving component, a promising area in the search space is searched in a more structured way than in population-based methods. In this way the danger of being close to good solutions but "missing" them is not as high as in population-based methods.

In summary, population-based methods are better in identifying promising areas in the search space, whereas trajectory methods are better in exploring promising areas in the search space. Thus, metaheuristic hybrids that combine the advantage of population-based methods with the strength of trajectory methods are often very successful.

A second form of integration concerns systems that are usually called cooperative search. They consist of various algorithms exchanging information in some way. Cooperative search will be deeply discussed in Sec. 3.5.6. Finally, the integration of approximate and systematic (or complete) methods is nowadays an important and prolific research direction. We will overview and discuss this topic in chapter 7.

2.6 Conclusion

In this chapter, we have introduced metaheuristics and related concepts. First, we have discussed possible classifications, based on the main attributes of metaheuristic algorithms: population/single point, objective function, neighborhood structure and memory. Then, we have given a survey of the most important metaheuristics, by describing the basic algorithms along with variants and improvements. Finally, we have shown that metaheuristic algorithmic mechanisms and strategies are designed with the aim of exploiting the interplay between intensification and diversification. Most of metaheuristic components have both the characteristics at the same time, but one is usually stronger than the other. Intensification and diversification can be used as key concept to analyze, compare and design metaheuristics. In this chapter we have presented metaheuristics from an algorithmic standpoint. In the next chapter, we will describe an architecture that enables the implementation of metaheuristics by designing and connecting the algorithmic components.

Chapter 3

A Multi-level Architecture for Metaheuristics

In this chapter, we introduce an architecture that enables the design and implementation of metaheuristics in a component-based fashion. We move the focus from the algorithmic and conceptual viewpoint, to the software engineering approach.

We will first revisit metaheuristics in a multiagent perspective, by providing a formal definition of the agents and their coordination (Sec. 3.4). Then, in Sect. 3.5 we will show some examples that describe the implementation in this framework of some among the best known metaheuristic algorithms (already introduced in the previous chapter) and we will further extend the examples by considering cooperative search.

3.1 Motivations

Classifications of metaheuristics are useful when they help to outline structural similarities and differences among algorithms. Once some common key properties have been recognized, a general description can be formulated, and the algorithms of that class can be defined as specializations of it. A metaheuristics framework can be useful for:

- comparing existing algorithms;
- designing (new) hybrid algorithms;
- supporting software engineering.

In this framework, we first try to identify the common principles and basic components underlying metaheuristic algorithms. These components will be encapsulated in communicating software components with clear structure and interface. Components are organized in levels. Components belonging to lower levels are simple and,

in some cases, simply react to events, while components belonging to higher levels have deliberative capabilities, embed complex strategies and behave autonomously. For these reasons, in our system, components are called agents, even if the starting point of the *agent view* of our architecture is more oriented to software engineering rather than to Distributed Artificial Intelligence and Multi Agent Systems [236]. In fact, our architecture can be implemented as a centralized system where, instead of agents, we have software components such as objects.

We define a four levels architecture. In each level, one or more agents act: to the first level belong solution builders, to the second one solution improvers, then strategic agents live in the third level and coordinating agents in the fourth one. Agents belonging to the same level have the same structure. We will clearly define which is the structure of each agent, and its interface. Agents belonging to different levels communicate for exchanging results and information. We define the coordination scheme among agents as a labelled transition system.

The distinction and separation between different roles and levels is important for the comprehension of metaheuristics, their comparison, their design and implementation. In fact, in this way we are able to bring together flexibility in computing, possibly distribution in processing and heterogeneous forms of realizations. The framework proposed fulfills in fact the above mentioned features of a metaheuristic framework. First, it enables to compare existing algorithms from a structural perspective, as far as the number of agents, their structure and their communications are concerned. For example, algorithms providing a feedback from higher levels to lower ones are those enabling a deliberative capability like a form of *learning* or adaptation in the metaheuristic algorithm. Indeed, the feedback loop means that the search history is exploited to dynamically balance intensification and diversification, to adaptively tune parameters and to apply learning techniques, like the ones used in [47, 127, 152]. In this perspective, the framework can be useful for teaching metaheuristics, with the aim of describing the basic algorithmic components, their interfaces and the effect of communication.

Second, the architecture can be helpful in the design of (new) hybrid algorithms. For instance, we can start from an existing algorithm and we can add a further communication link between two levels. The new information exchange can be used to influence the behavior of one or more agents (e.g., exploitation of the search history). We may also add or substituting one or more agents in any level. Another possibility, strongly used in cooperative search, concerns the design of hybrid algorithms putting together existing metaheuristics. Cooperative search has been proved useful for solving large-scale optimization problems and multi-criteria optimization problems.

Finally, we can support software engineering since we isolate basic components which can be easily composed and re-used. In fact, we can change only one part of the architecture by maintaining the interface and changing the implementation without re-writing the whole application. The user can now compose his/her own strategy by putting together software components, without starting from scratch a new code each time an algorithm has to be implemented.

Algorithm 12 Adaptive Memory Programming

```
Initialize the memory
while stopping condition not met do
  Generate a new provisory solution s using data stored in the memory
  Improve s with a local search; let s' be the improved solution
  Update the memory using pieces of knowledge brought by s'
end while
```

3.2 Metaheuristic frameworks

There are different ways to describe metaheuristic algorithms. The most adopted criteria have been discussed in the previous chapter. Here we just mention a very interesting framework called *Adaptive Memory Programming* (AMP, [216]). AMP describes a class of metaheuristics as algorithms which use a memory to store information collected during the search process; this information is used to bias the solution construction and the future search (hence the term *adaptive*). AMP high level algorithm, as it appears in [216], is reported in Algorithm 12.

Within this general framework, several metaheuristics can be described, like Tabu Search, Ant Colony Optimization (ACO) and Scatter Search. Tabu Search explicitly handles a short term memory (the *tabu list*), which is used to avoid cycling among the same set of states. In population-based metaheuristics, memory is implicitly embedded in shared data structures (population itself in evolutionary algorithms and pheromone trails in ACO). AMP clearly points out the importance of the use of memory during the solution construction and the search. Nevertheless, the level of abstraction chosen enables only to informally introduce concepts of memory, stored information and solutions, and its usefulness is limited to a general description of algorithms which make use of search history. AMP cannot be used as a framework which helps to design hybrid algorithms, as its descriptive level is too general.

Other ways of describing metaheuristic algorithms have been proposed. An example mentioned in the previous chapter, is a framework for evolutionary computation algorithms [109]. In that framework, evolutionary algorithms are described depending on seven main features: individual representation, evolution process, neighborhood structure, information sources, infeasibility, intensification and diversification.

3.3 MAGMA: MultiAGent Metaheuristic Architecture

The weak notion of *agent* [60, 194, 236, 240] states that an agent is a (software) system that enjoys the following properties: *autonomy*, *social ability*, *reactivity*, and *pro-activeness*. In this context, we just adopt the metaphor of agent referring to a system able to build a solution, move over a landscape, communicate with other agents, be active (goal-oriented) and, possibly, be adaptive. Agents in our architecture

are reactive, since they must act as soon as their input is provided. Agents are autonomous, since they may need to have deliberative capabilities, as in the most general case they incorporate complex search strategies. The social ability is required only for communication; we do not take into account agent society formation and behavior. Our agents are pro-active since their goal is to apply the strategy to the model.

We devise different kinds of agents, with different functionalities, perspectives and goals. These agents act in a multi-level architecture (MAGMA, depicted in Figure 3.1), composed of some levels each of which corresponds to a different level of *abstraction*.

At each level there are one or more specialized agents, each implementing an algorithm. The LEVEL–0 provides a feasible solution (or a set of solutions) for the upper level; it can be considered the *solution* level. LEVEL–1 deals with solution improvement and agents perform a trajectory in the fitness landscape until a termination condition is verified; this can be defined as the level which deals with *neighborhood structure*. LEVEL–2 agents have a global view of the space, or, at least, their task is to guide the search toward promising regions and provide mechanisms for escaping from local optima. Therefore this can be defined as the *landscape* level.

Classical metaheuristic algorithms can be easily described via these three levels. In fact, we will show in Section 3.5 that using only three levels we are able to describe *simple* metaheuristics. With *simple* we mean non hybrid algorithms. When various search strategies (complete or incomplete) are combined, we have a cooperative search framework, where metaheuristics and other search method coexist and exchange results. In this case an additional level, LEVEL–3, coordinating different search strategies should be devised. Therefore this can be defined as the *coordination* level and deals with different landscapes and strategies.

Coordination among agents involves both communication and synchronization. A formal and general coordination model will be provided in Section 3.4. Communications between any two levels are possible, therefore an algorithm can be described as the result of the interaction of some agents (algorithmic components) each specialized for a specific task. Communication among agents can be implemented through any kind of mechanism and protocol. The designer is free of choosing the most suitable communication scheme. For example, agents can use a blackboard or a message passing mechanism.

Agents can be formally defined with tuples, whose elements describe computational capabilities, knowledge and goals. This definition encompasses all the current metaheuristics. Each tuple is composed of two main components: the model $\mathcal{M}$ and the strategy $\mathcal{S}$. In the following, we formalize the tuples for each level, while in section 3.5 we will specialize the tuples on specific algorithms.

3.3.1 Definition of the multi-level architecture

LEVEL–0 To this level belong *solution builder* agents, whose task is to produce starting solutions. LEVEL–0 agents, hereinafter referred to as L0-agents, deal with

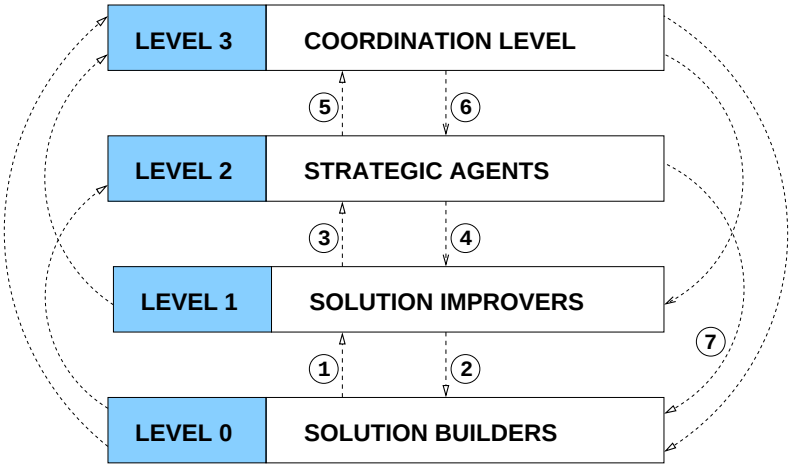

Figure 3.1: Multi-level architecture for metaheuristic algorithms. Note that only numbered arrows are indeed used by existing metaheuristics, while non numbered arrows can be used for extending existing algorithms.

problem entities such as variables, domain values and constraints or partial solutions, depending on the representation chosen. L0-agents' goal is to construct an initial solution for the upper level. L0-agents may exploit strategies like random initialization, greedy construction, probabilistic construction (e.g. pheromone based), etc. Their computational capabilities are usually quite limited, but can be very complex as we will show in Large Neighborhood Search. They are generally reactive agents, whose behavior is based upon a heuristic criterion (which may be dynamically changed by upper levels agents).

Agents of LEVEL–0 are defined by tuple $\mathcal{T}_0 = \langle \mathcal{M}_0, \mathcal{S}_0 \rangle$, where $\mathcal{M}_0 = \langle \mathcal{V}, \mathcal{D}, \mathcal{C}, \mathcal{OF} \rangle$. $\mathcal{V}$ are *components*, the domains $\mathcal{D}$ define the values that components can assume, $\mathcal{C}$ are the *constraints* and finally $\mathcal{OF}$ is the *objective function*. $\mathcal{S}_0$ is the *strategy* which will be specialized depending on the metaheuristic algorithm we are describing. We will provide some examples of strategies in Section 3.5. The *strategy* is the actual constructive algorithm, for instance it can be random, greedy, probabilistic, etc. *Components* can be variables or partial solutions, or any element which is used as a building block for solutions. *Constraints* are needed for the agent to construct solutions which are feasible with respect to the model (indeed, it is possible that they are infeasible for the original problem, as shown in Section 2.3.1 where an example of LS on SAT is discussed).

It is worth noting that L0-agents are constructive agents that are able to build a feasible solution for the model $\mathcal{M}_0$. This solution is computed depending on the strategy $\mathcal{S}_0$ which can be an approximate algorithm, like a greedy strategy, that simply finds the first solution. We will show that L0-agents can also implement a complete

algorithm, like *branch and bound* which computes the optimal solution with respect to the objective function. The fact that a single agent can compute the optimal solution has two advantages: the first is that our architecture can be used to describe not only metaheuristics, but also complete methods. Second, MAGMA can describe hybrid algorithms, like cooperative search and large neighborhood search, where complete solvers are integrated into metaheuristics [35, 65, 171] and L0-agents then optimally solve subproblems.

As an output LEVEL–0 provides one or more solutions to the upper levels. In addition, LEVEL–0 can be triggered each time a restart is performed.

LEVEL–1 This level contains *solution improver* agents. Each agent searches in a Fitness Landscape (FL) with a local search, trying to improve the solution it has received from another agent. LEVEL–1 agents (L1-agents) implement various search algorithms and they can constitute either single independent search agents, or cooperating agents. L1-agents deal with solutions and neighborhood structures. The usual concept of *short term memory* belongs to this level: agents can use their recent past to intensify the search or to escape from local optima. Therefore, L1-agents are usually not just reactive, but they may have deliberative capabilities even if rather limited.

L1-agents are defined by tuple $\mathcal{T}_1 = \langle \mathcal{M}_1, \mathcal{S}_1 \rangle$, where $\mathcal{M}_1 = \langle Sol\ , \mathcal{N}, \mathcal{H}, \mathcal{F} \rangle$. $\mathcal{S}_1$ is the strategy used to improve the solution and again it will be specialized on the specific metaheuristic. Let us consider now the model $\mathcal{M}_1$: *Sol* is the *initial solution*, $\mathcal{N}$ is the *neighborhood structure*, $\mathcal{H}$ is the search *history* and $\mathcal{F}$ is the *fitness function*.

The solution provided by LEVEL–0 (*arrow 1* in Figure 3.1) or any other agent (see for example *arrow 4*) is the starting point of the search, guided by the *strategy* which, in general, makes use of the search *history* and the *fitness function*. A very important element is the *neighborhood structure*, which defines the portion of search space visible from each state. The neighborhood structure can be also dynamic (as for Tabu Search) or the agent can dynamically change neighborhood during the search process, if suggested by upper level agents (*arrow 4*). For instance, the effectiveness of the combination of different neighborhoods has been proven very effective in Variable Neighborhood Search [105] and in a new metaheuristic called Multi-Neighborhood Search [80]. In general, every agent can receive not only simple pieces of information (such as solutions), but also more complex data such as neighborhoods, statistics or parts of the search history, etc.

The output of LEVEL–1 is the improved solution, which usually represents a local optimum or a state corresponding to search stagnation.

It is worth to stress that the more complex an agent is, the more difficult is the separation of basic algorithmic components. Therefore, instead of having LEVEL–1 agents with very sophisticated dynamic neighborhood structures, this architecture suggests to design simpler agents coordinated by an upper level agent. Thus, for example, metaheuristics with dynamic neighborhoods can be described as a system composed of LEVEL–1 agents with only one neighbor which alternate their search process under the coordination of a strategic agent of LEVEL–2.

LEVEL–2 LEVEL–2 agents (L2-agents) are *strategic* agents, since their main

role is to balance intensification and diversification. Todays most effective metaheuristics have usually a non trivial strategy to dynamically balance intensification and diversification, achieved by the use of search history (descriptions of such advanced strategies can be found in [90]). Moreover, L2-agents can sample or abstract the search space looking for promising regions to explore. With respect to the general architecture depicted in Figure 3.1 they can perform activities like the following:

- store the best solutions found among those provided by LEVEL–1 (*arrow 3*);
- suggest what building blocks (e.g. partial solutions) have the highest probability of being part of the optimal solution (*arrow 7*);
- suggest what regions of a (single) landscape are the most promising (*arrow 4*);
- dynamically bias the intensification/diversification balance (*arrows 4 and 7*);
- switch between two landscapes (*arrow 4*).

L2-agents deal with landscapes. *Long term memory* is implemented and exploited at this level, where data concerning the whole history of the search process of lower level agents are stored. One of the main uses of long term memory is the implementation of diversification strategies.

L2-agents are described by the tuple $\mathcal{T}_2 = \langle \mathcal{M}_2, \mathcal{S}_2 \rangle$, where $\mathcal{M}_2 = \langle \mathcal{P}, \mathcal{H}, \mathcal{F} \rangle$ and $\mathcal{S}_2$ is the strategy used for guiding lower level agents.

The model contains $\mathcal{P}$, i.e., the set of solutions of L1-agents, the *history* $\mathcal{H}$ and the *fitness function* $\mathcal{F}$. The set $\mathcal{P}$ of solutions provided by L1-agents can be used in population based methods, in which recombination and mutation operators are applied to the population. The *history* $\mathcal{H}$ is the structured ensemble of data collected during the overall search process. It includes information on every agent search process, for instance the best solution found. This information is exploited by the *strategy* which guides lower level agents.

Using levels 0, 1 and 2 we are able to describe almost all *simple* metaheuristic algorithms, i.e., algorithms that do not combine more than one strategy in each level. However, more complex metaheuristics can be devised, namely those integrating more than one (complete or incomplete) strategy. Suppose for example, that we have to describe a cooperative search mechanism that combines two metaheuristics and exchanges solutions. In this, case, while the two metaheuristics can be settled in the three level architecture, there is something missing, i.e., the coordination of the two metaheuristic. We therefore need an additional level that coordinates two algorithms, identifies which parts of the solutions should be exchanged, if any, and so on. In general, we need some deliberation capability that organizes lower level agents. For this purpose, we introduce an additional level which will be used to describe cooperative algorithms.

LEVEL–3 Agents belonging to this level (called L3-agents) coordinate lower level agents behavior. These agents know the model and strategy of all lower level

agents and coordinate their behavior and communications. For example, in a cooperative search framework, where more than one metaheuristic algorithm is involved, the information they communicate to each other is decided by a level-3 agent. If solutions are exchanged, the level-3 agent should for instance have a notion of distance between solution and exchange solution that are *enough different* for the other. This capability could also be applied to select, adapt and improve the lower level agents in population based metaheuristics (those actions involve communications expressed by *arrows* 5 and 6). As a final example, which will be explained in detail in section 3.5.6, we mention large neighborhood search (LNS). LNS is one of the most successful techniques for merging complete and incomplete methods. The idea is to start with a solution, fix some parts of it and optimally solve the rest. In this case, level-3 agents decide which parts should be fixed and which subproblems should be solved in order to eventually improve the solution.

Thus, L3-agents perform meta-reasoning on the behavior of agents at lower levels. The computational capabilities of L3-agents can be the highest among all the levels.

L3-agents are described by the tuple $\mathcal{T}_3 = \langle \mathcal{M}_3, \mathcal{S}_3 \rangle$, where $\mathcal{M}_3 = \langle \mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2 \rangle$ and $\mathcal{S}_3$ is the strategy used for guiding lower level agents. The model $\mathcal{M}_3$ contains the complete description of lower level agents and the strategy $\mathcal{S}_3$ defines the specific coordination scheme.

It is worth noting that each level has the same structure composed of a model and a strategy. Observe that the higher the level, the higher the abstraction. In fact, $\mathcal{M}_0$ is strongly problem dependent: components represent problem entities and constraints are relations among them. On the contrary, $\mathcal{M}_1$ manipulates solutions independently from their semantics and $\mathcal{M}_2$ further enlarges the gap between the problem and its representation since it focuses on intensification and diversification strategies. Finally, $\mathcal{M}_3$ can be considered as a meta-model containing the tuples of lower level agents.

An interesting observation is that levels 2 and 3 have a similar characteristic: they guide underlying levels. However, while level 2 coordinates trajectory methods, level 3 coordinates metaheuristic algorithms, but both have deliberative capabilities.

In MAGMA the definition of the environment is very simple. What agents perceive as "external" is just the set of agents to whom they communicate. Therefore, for any agent A_i, the environment is defined as the set of agents that send information to A_i and the set of agents receiving information from A_i.

3.4 Coordination

On the most general level of abstraction, we can describe agent coordination as the composition of basic states and actions. Agents wait for an input, then execute the algorithm specified by the strategy $\mathcal{S}$ and send the result to the destination (i.e., the specified receiver agent).

A particularly suitable formalism to describe this kind of coordination is the one of

Labelled Transitions Systems, LTS [220]. LTS are mainly used to formally define the coordination among processes, but they are also applied in the context of multiagent systems [223].

Formally, a LTS is a pair $(\mathbb{P}, \rightarrow)$, where $\mathbb{P}$ is a set of processes and $\rightarrow \subseteq \mathbb{P} \times \mathrm{Act} \times \mathbb{P}$ is an infix predicate. *Act* is a set of possible actions that can be performed by the processes. Actions are defined with respect to a chosen level of abstraction, therefore they can be general actions (e.g., execute a given algorithm) or specific actions (e.g., greedy solution construction). Usually, the following notation is used: $p \xrightarrow{a} q$, meaning that the process p can perform the action a and after its completion it reaches a state where q is the process remaining part. Actions of *Act* can be composed of computational activities (in a finite number for a finite time interval) and can be also synchronization of a process with the environment or the receipt of a signal sent by the environment.

LTS define the result of the execution of all the allowed actions for every process of the system. In case of alternatives (i.e., a process p is in a state such that it is allowed to perform more than one action), LTS do not specify the choice among the possible cases.

For the sake of readability, we will indicate state transitions and not process transitions, i.e., $p \xrightarrow{a} q$ means that the agent in state p can perform action a and after its completion it will be in state q.

We formalize the coordination among agents by means of LTS assuming the most general level of abstraction, which corresponds to the level of abstract classes in the object oriented terminology. On this level of abstraction, the actions agents can perform are three: *send*, *execute* and *receive*. The actions *send* and *receive* refer to the communication between agents (e.g., send solution to L1-agent, receive solution from L0-agent). *send* can also have as arguments a list of agents along with the respective messages to be sent. The action *execute* encapsulates the overall agent computation, i.e., we adopt a *blackbox* description of the computational part of the agents. The action *execute* is an abstraction of the execution of the particular algorithm specified by the strategy $\mathcal{S}$ of the agent tuple.

On the same level of abstraction, at each time stamp agents can assume one out of three states: `WAIT`, `READY` and `COMPLETED`. The state `WAIT` means that the agent is waiting for an input to start the execution of the algorithm. Agents are in state `READY` when they have received the needed input and they can start their execution. Finally, `COMPLETED` corresponds to the state reached after the termination of the execution (i.e., when the agent is ready to send its output to the receiver agent).

It is also possible that an agent needs more than one input before it can start the execution of the strategy. Therefore, we also add a fourth state that represents the condition in which an agent is waiting for the completion of the collection of inputs. We call this state `WAIT_FOR_ALL`.

The corresponding labelled transition system, which describes all the agents of the multi-level architecture is composed of four possible transitions:

- `WAIT` $\xrightarrow{\text{receive(msg,sender)}}$ `READY`

- WAIT_FOR_ALL $\xrightarrow{\text{receive}([msg_1, A_1], \ldots, [msg_n, A_n])}$ READY
- READY $\xrightarrow{execute(strategy)}$ COMPLETED
- COMPLETED $\xrightarrow{\text{send}(msg(s), receiver(s))}$ WAIT

Every specialization of metaheuristics in MAGMA can be defined also by instantiating this general LTS.

Finally, we note again that the notion of environment used in MAGMA is fairly simple, as it is defined, for every agent, as the set of senders and receivers.

3.5 Specializing MAGMA

MAGMA can be exploited either as a conceptual and implementation framework for metaheuristic algorithms, or as a metaphor for designing multiagent algorithms for optimization problems. In this section, we concentrate on the first topic and we show how some among the most effective metaheuristic algorithms can be described in terms of MAGMA. We will describe some specializations of the architecture showing that both trajectory and population based algorithms can be easily defined in our framework.

In the following sections, we will specialize the tuples describing the agents involved in each metaheuristic and the agent coordination through a LTS. Obviously, the models, i.e., $\mathcal{M}_i$, depend on the particular problem to be solved, thus here they are skipped. Thus, for each tuple, we will describe only the strategy.

There are metaheuristics which can just be described by L0-agents and L1-agents. For example, the basic Tabu Search can be realized with a single L0-agent which randomly constructs an initial solution and a single L1-agent which performs the search with one or more tabu lists. Of course, Tabu Search metaheuristics make often use of long term memory and therefore they need a further agent at LEVEL–2 which balances intensification and diversification, for instance by enlarging and restricting the tabu list. Typical examples of dynamic tabu length tuning can be found in [90] (for instance, a strategy called *strategic oscillation*) and in [13, 14], where an adaptive mechanism is proposed. The issue involving the dynamic control of intensification and diversification is one among the most relevant in metaheuristics research [13, 14, 23, 22, 32].

The detailed description and discussion of the metaheuristic algorithms has already be given in Chap. 2 and here they are presented just in their easiest and general form. It is worth noting that variants of these algorithms can be easily obtained by changing the algorithms implemented at any level. Moreover, it is always possible to introduce more than one agent per level. Communications between levels are very important as well, and in principle they involve all possible combinations of links between levels, even if the metaheuristic algorithms designed so far use just some of all possible communication links.

Algorithm 13 Greedy Randomized Adaptive Procedure

while stopping criterion not satisfied **do**
  Construct a solution s with a Greedy Randomized procedure
  Apply local search to s
  Memorize the best solution found
end while

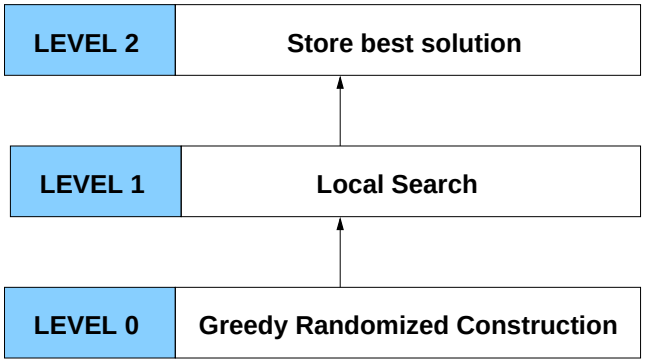

Figure 3.2: The Multi-level architecture version of GRASP.

3.5.1 Greedy Randomized Adaptive Procedure (GRASP)

We start our series of examples with a very simple algorithm: GRASP [59]. This algorithm is composed of a greedy construction procedure and a local search procedure (see the sketched Algorithm 13). The construction procedure builds a solution by applying a greedy function which, at each step, adds a randomly chosen component among the ones which have the highest greedy function value. Then, a local search phase starts; the choice of the local search algorithm depends on the problem to be solved. GRASP is easily described in MAGMA: the first level generates a solution by means of a greedy algorithm, a single L1-agent performs local search, and finally a L2-agent keeps track of the best solution found. Communications are unidirectional and the information flow from LEVEL–0 to LEVEL–1 and from LEVEL–1 to LEVEL–2. In Figure 3.2 the MAGMA version of GRASP is depicted.

The instantiation of the strategy in each tuple for GRASP is the following:

- $\mathcal{S}_0$ is a greedy random constructive strategy;
- $\mathcal{S}_1$ is a simple local search strategy implementing hill climbing;
- $\mathcal{S}_2$ stores the current best solution.

Coordination among agents composing GRASP can be defined by specializing the LTS given in Section 3.4. We always suppose that agents' models are instantiated by

an initialization process that we call INIT, except when explicitly stated that (a part of) the model is instantiated by a L3-agent. INIT sends a message *inst* containing all the information needed by the agents.

The coordination model for the L0-agent is the following:

- WAIT $\stackrel{\text{receive(inst,INIT)}}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_0)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send(sol,L1-agent)}}{\longrightarrow}$ WAIT

The coordination model for the L1-agent is the following:

- WAIT $\stackrel{\text{receive}(sol,L0-agent)}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_1)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(impr.sol,L2-agent)}{\longrightarrow}$ WAIT

Finally, for the L2-agent the coordination model is:

- WAIT $\stackrel{\text{receive}(sol,L1-agent)}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_2)}{\longrightarrow}$ COMPLETED
- COMPLETED $\longrightarrow$ WAIT

3.5.2 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is a well known population-based metaheuristic for designing algorithms for COPs [47, 48, 49]. The inspiring idea behind ACO is the foraging behavior of real ants and the way they communicate. Real ants deposit a pheromone trail while they are walking and select their direction in a stochastic way. The higher the pheromone value on a path, the higher the probability the ant chooses that path. Artificial ants are simple agents that construct a solution by choosing the next component in a probabilistic way (function of the pheromone). In ACO, pheromone is a mean to communicate good solution components. Several implementations of ACO have been proposed, often with different choices for the construction rules and the way to update pheromone trails [24, 49, 78, 192, 210, 212].

A generic ACO algorithm is sketched in Algorithm 14.

In Figure 3.3 the MAGMA specialization of ACO is depicted: we have N_a L0-agents (the ants) providing initial solutions by using a constructive procedure biased by pheromone trails. For brevity, we will indicate with $\mathbf{T}$ the matrix of pheromone values. N_a L1-agents perform local search and one L2-agent updates $\mathbf{T}$. The last piece

Algorithm 14 Ant Colony Optimization

while stopping criterion not satisfied **do**
  Construct a set of solutions $S = \{s_1, s_2, \ldots, s_m\}$ by using pheromone trails and heuristic information
  Apply local search to every solution in S
  Update pheromone trails
end while

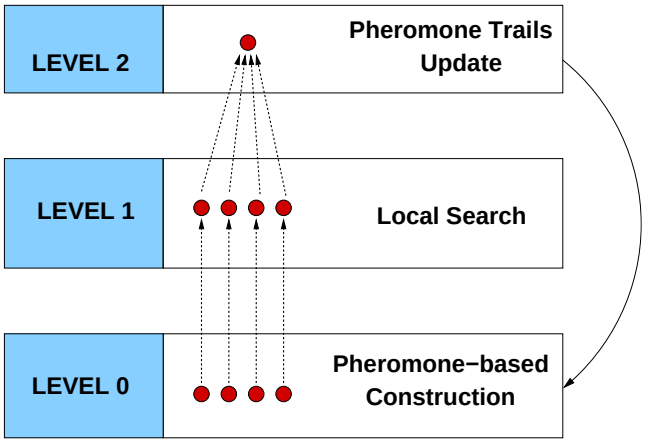

Figure 3.3: The Multi-level architecture version of ACO with local search.

of information (new pheromone values) is then used again by L0-agents to construct new solutions. We observe that in this case the communication is also from the third level to the first and this means that the process is self-adapting, as a feedback loop involves the initial and the final steps.

The instantiations of the strategy in each tuple for ACO are the following:

- $\mathcal{S}_0$ is a probabilistic constructive procedure;
- $\mathcal{S}_1$ can be any trajectory method;
- $\mathcal{S}_2$: L2-agent stores the current best solution and update pheromone trails.

The coordination model for the L0-agent i $(i = 1, \ldots, N_a)$ is the following:

- $\texttt{WAIT} \xrightarrow{\text{receive}(\mathbf{T}, INIT)} \texttt{READY}$
  the first iteration is performed on the basis of the initial values for pheromone
- $\texttt{WAIT} \xrightarrow{\text{receive}(\mathbf{T}, L2-agent)} \texttt{READY}$

after the first iteration the values are given by the L2-agent. Observe that it is not possible that both the transitions from state WAIT are active at the same time.

- READY $\stackrel{\text{Execute}(\mathcal{S}_0)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(sol, L1-agent_i)}{\longrightarrow}$ WAIT

The coordination model for the L1-agent i is the following:

- WAIT $\stackrel{\text{receive}(sol, L0-agent_i)}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_1)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(impr.sol, L2-agent)}{\longrightarrow}$ WAIT

Finally, for the L2-agent the coordination model is:

- WAIT_FOR_ALL $\stackrel{\text{receive}([s_1, L1-ag_1], \ldots, [s_n, L1-ag_n])}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_2)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(\text{new } \mathbf{T}, L0-ag_1, \ldots, L0-ag_{N_a})}{\longrightarrow}$ WAIT_FOR_ALL

3.5.3 Iterated Local Search (ILS)

Iterated Local Search (ILS) is a simple but powerful metaheuristic algorithm [208, 209]. It applies local search (or a more general trajectory method) to an initial solution until it finds a local optimum; then it perturbs the solution and restarts local search from the perturbed state. The importance of the perturbation (change in the solution) is obvious: a very small change in the solution could not make it possible to escape from the local optimum basin of attraction; on the other side, a too strong modification can be comparable to a simple random restart. In order to accomplish these requirements, several criteria have been adopted, most of which use the history of the search. A very simple version of ILS is described in Algorithm 15.

The design of ILS algorithms has several degrees of freedom in the choice of the initial solution, perturbation and acceptance criteria.

In the MAGMA framework, ILS can be described as follows (see Figure 3.4): one L0-agent provides an initial solution (either randomly, or heuristically generated), then it stops its activity. At LEVEL–1, one solution improver agent applies a local search algorithm to the solution and, when it meets a local optimum it stops, waiting for a new solution to improve. At LEVEL–2 the agent keeps track of the search process of the local search agent and, when it stops, it evaluates the new starting solution as a modification of the current one. Observe that, in this case, there is a continuous

Algorithm 15 Iterated Local Search

Generate initial solution s
Execute LS from an initial state s until a local optimum s^* is found
while stopping criterion not satisfied **do**
  Perturb s^* and obtain s'
  Execute LS from s' until it finds a local optimum $s^{*'}$
  On the basis of an *acceptance criterion* decide whether to set $s^* \leftarrow s^{*'}$
end while

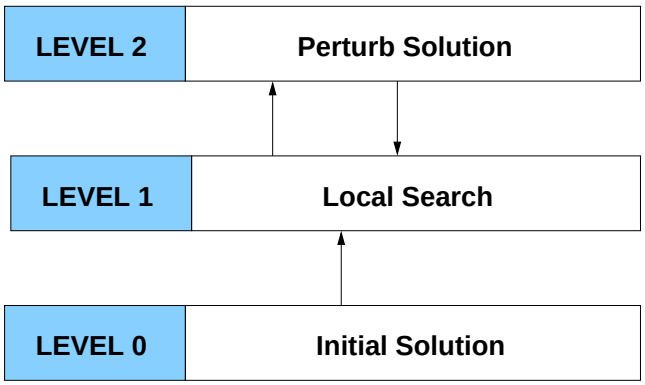

Figure 3.4: The Multi-level architecture version of ILS.

communication between LEVEL–1 and LEVEL–2, while LEVEL–0 participates just for the initialization, or for a random restart[1] (not considered in Figure 3.4). It can be observed that the structure of ILS is very similar to the structure of GRASP. However, the level-2 agent in ILS is much more *intelligent* than that of GRASP: in fact, it should guide the perturbation of the level-1 agent toward unexplored (and promising) areas.

The instantiations of the strategy in each tuple for ILS are the following:

- $\mathcal{S}_0$ is a randomly, or heuristic-based constructive procedure;
- $\mathcal{S}_1$ is any trajectory method;
- $\mathcal{S}_2$ is such that the LEVEL–2 agent stores the current best solution and perturbs the solution returned be the L1-agent.

The coordination model for the L0-agent is the following:

- `WAIT` $\xrightarrow{\text{receive(inst.,INIT)}}$ `READY`

[1] Random restart can, of course, be added to every approximate algorithm.

- READY $\stackrel{\text{Execute}(\mathcal{S}_0)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send(sol,L1-agent)}}{\longrightarrow}$ WAIT

The coordination model for the L1-agent is the following:

- WAIT $\stackrel{\text{receive}(sol, L0-agent)}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_1)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(impr.sol, L2-agent)}{\longrightarrow}$ WAIT

Finally, for the L2-agent the coordination model is:

- WAIT $\stackrel{\text{receive}(sol, L1-agent)}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_2)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(pert.sol, L1-agent)}{\longrightarrow}$ WAIT

3.5.4 Memetic Algorithms

In this section, we propose the application of the MAGMA framework to Memetic Algorithms (MAs) [158]. We include in this category genetic algorithms which apply also local search, in a way similar to ACO plus local search. In a MA (see Algorithm 16), an initial population of solutions is generated; then every solution is improved by applying local search. The resulting new population cooperates and/or competes to produce a new population. The cooperation is a way to exchange information among individuals and it can be implemented as genetic operators like mating and crossover; the competition can be implemented with the selection genetic operator or more elaborated mechanisms.

A MA in MAGMA is described as follows: many L0-agents generate the initial population of solutions. Each solution can be generated either randomly, or by means of a constructive procedure. L1-agents take the solutions from the lower level and improve them by using local search (either some steps of a local search algorithm, or until they reach a local optimum). At LEVEL–2 we have one agent whose task is to generate new solutions by applying recombination and mutation operators. Then the new population is produced by applying cooperation or competition operators. The new population is directly given as input to LEVEL–1. Figure 3.5 shows the multi-level architecture for a generic MA. Note that, since the L2-agent iteratively receives the solutions from all the L1-agents, applies cooperation and competition operators and sends the new solutions to the L1-agents (one for each agent), we represented this *one-to-many* communication by including the L1-agents in one single component.

The instantiation of the strategy in each tuple for MA has some similarities (at least for the first two levels) with ILS, with more than one agent per level (this appears also from the Figure):

Algorithm 16 Memetic Algorithm

Generate initial population S
while stopping criterion not satisfied **do**
  Apply local search to every individual $s \in S$
  Apply cooperation operators to S and obtain S'
  Apply competition operators to S' and obtain S''
  $S \leftarrow S''$
end while

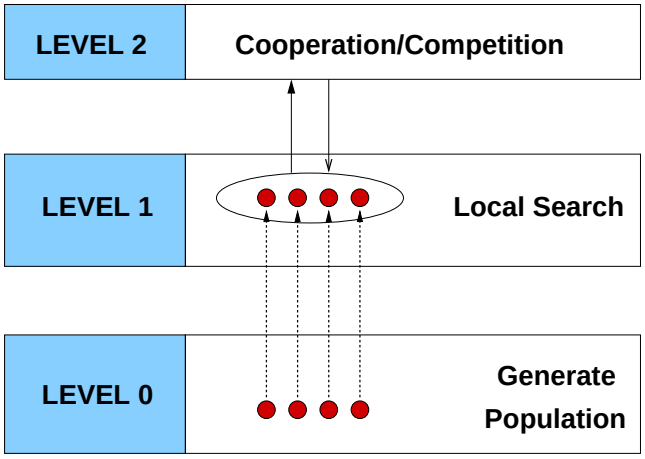

Figure 3.5: The Multi-level architecture version of Memetic Algorithms.

- $\mathcal{S}_0$: constructive procedure (e.g., random, heuristic, etc.), implemented by a population of L0-agents;
- $\mathcal{S}_1$: trajectory method, implemented by a population of L1-agents;
- $\mathcal{S}_2$: the L2-agent evaluates the population and applies recombination, mutation and selection/competition operators.

The coordination model for the L0-agent i ($i = 1, \ldots, P$, where P is the population cardinality) is the following:

- `WAIT` $\xrightarrow{\text{receive(inst.,INIT)}}$ `READY`
- `READY` $\xrightarrow{\text{Execute}(\mathcal{S}_0)}$ `COMPLETED`
- `COMPLETED` $\xrightarrow{\text{send}(I_i, L1-agent_i)}$ `WAIT`

The coordination model for the L1-agent i is the following:

- WAIT $\xrightarrow{\text{receive}(indiv, L0-agent_i)}$ READY
- READY $\xrightarrow{\text{Execute}(\mathcal{S}_1)}$ COMPLETED
- COMPLETED $\xrightarrow{\text{send(impr.indiv)}}$ WAIT

Finally, for the L2-agent the coordination model is:

- WAIT_FOR_ALL $\xrightarrow{\text{receive}([I_1, L1-ag_1], \ldots, [I_P, L1-ag_P])}$ READY
- READY $\xrightarrow{\text{Execute}(\mathcal{S}_1)}$ COMPLETED
- COMPLETED $\xrightarrow{\text{send}([I_1, L1-ag_1], \ldots, [I_P, L1-ag_P])}$ WAIT_FOR_ALL

3.5.5 A new algorithm: Guided Restart ILS

After the description of well known metaheuristics in MAGMA, we briefly show how MAGMA can be used to design a very simple new algorithm. In the following example, we do not claim to describe a brand new algorithm, but we rather aim at showing how to combine available algorithmic components and strategies.

If we consider ILS in the MAGMA framework, we observe three communication arrows: one from the L0-agent which provides the initial solution to the L1-agent. The remaining two arrows show the communication between L1-agent and L2-agent (local search and perturbation cycle). We note the absence of an arrow from LEVEL 2 to LEVEL 0, which characterizes ACO and other algorithms which exploit the search history to bias the initial solution construction. Therefore, we can add an arrow from the L2-agent to the L0-agent and enrich the strategy of the L2-agent by a diversification mechanism based on the search history. For instance, the L2-agent may accumulate statistics on the frequency of a list of recent solutions and bias the restarting process[2].

We implemented this diversification mechanism and compared its performance with respect to simple random restart ILS. On large MAXSAT instances the guided restart enables the algorithm to reach the best known solution with higher frequency than ILS with random restart.

Guided Restart ILS (GRILS) is based on basic ILS plus a restarting mechanism which makes use of the search history of previous restarts and the best solution found. The L2-agents keeps a list of solutions composed of the most recent k initial solutions and the best solution found since the beginning of the algorithm. The algorithm has been applied to tackle large MAXSAT problems, where solutions are represented by assignments to binary variables.

The restart list is used to compute a vector of probabilities which bias the construction of the initial solution for the restart. A simple random restart would use

[2]Interesting diversification strategies developed for ACO can be useful also for the other metaheuristics can be found in [22].

Table 3.1: Random Restart ILS vs. Guided Restart ILS. The best solution value found and the percentage of runs returning that value are reported.

Instance number	**Best known value** (unsat. clauses)	**RRILS**	**GRILS**
inst_1	309	34%	38%
inst_2	316	6%	8%
inst_3	315	20%	28%
inst_4	315	34%	56%
inst_5	315	14%	20%
inst_6	306	36%	44%
inst_7	315	20%	12%
inst_8	317	0%	12%

a probability vector with all the entries fixed to 0.5, i.e., every variable has probability 0.5 to be assigned to 1 in the new solution. Our aim is to bias the solution construction in favor of solutions which are likely not to belong to already explored regions (diversification). For each variable x_i, the frequency $freq_i$ of assignments to 1 is computed. This frequency is used as the probability to have the assignment $x_i = 0$ in the new initial solution. Therefore, the higher the frequency of ones in the last k restart solutions, the lower the probability of having the same assignment to x_i in the new initial solution. This diversification mechanism is counterbalanced by considering also the best solution found $\mathbf{x}^{\text{best}} = (x_1^{\text{best}}, x_2^{\text{best}}, \ldots, x_n^{\text{best}})$ to compute the probability vector. The frequency $freq_i$ is incremented if $x_i^{\text{best}} = 0$ and decremented for the opposite assignment. Therefore, the probability vector is slightly moved toward the current most promising region (in the spirit of path relinking [89]).

Preliminary results on large random MAXSAT instances show that GRILS finds the best known solution with higher frequency than ILS with simple random restart. The length of the list of the last recently visited solution has been set to 10. GRILS parameters (e.g., k) have not been tuned by means of an adaptive procedure, therefore the performance of GRILS can be further improved. In Table 3.1 results concerning MAXSAT instances with 1000 variables and 10000 clauses are reported[3]. For every instance, we report the best solution value found by the algorithms (which is also the best solution known, at our knowledge) in terms of number of unsatisfied clauses. For each algorithm we show the percentage of runs which returned the best solution value. As we can observe, GRILS returns the best known value with higher frequency than RRILS in all but one instances. For one instance GRILS was also able to find a better solution than RRILS. The algorithms run 50 times for each instance and they have been stopped after 60 seconds[4].

[3]Instances can be retrieved from
`http://rtm.science.unitn.it/intertools/sat/benchmark.html`

[4]They run on a Pentium III at 500 MHz, with 256 MB of RAM and 512 KB of cache memory.

3.5.6 Cooperative Search

In this section we show and discuss the definition of cooperative search algorithms in MAGMA. We will note that the level-3 agents are particularly important for the integration and cooperation of different problem solving strategies.

Cooperative search [6, 41, 111, 113, 203, 218, 219] consists of a search performed by agents which exchange information about states, models, entire sub-problems, solutions or other search space characteristics. Agents can be either homogeneous (implementing the same algorithm) or heterogeneous (different algorithms).

We refer to [111] for the definition of *cooperative search*: "cooperation involves a collection of agents that interact by communicating information [...]". Therefore, the main characteristic of cooperative search is the information exchange among agents during the search process. Cooperative search goes beyond the combination of metaheuristics, as it encompasses the integration of complete techniques (e.g., in multi-objective optimization), of mixed techniques (e.g., local search in constraint programming [35, 65, 171]). Moreover, in the field of Distributed Problem Solving, the subdivision of a problem in subproblems solved by cooperating agents is a typical issue. In the following we will limit our discussion on cooperative search achieved by the combination of different metaheuristics. We also give an example of the integration of metaheuristics with complete algorithms, to show that MAGMA can also describe hybrid algorithms.

We can distinguish some kinds of cooperative search depending on the model of the problem used by the agents and the type of information they exchange. Cooperating agents can either have the same model of the problem or they can have different models. In the latter case, some agents may also have models representing parts of the problem at hand (subproblems). Concerning information exchange, agents can exchange complete or partial solutions. Complete solutions represent a point in the solution space, whereas partial solutions represent structured areas of the search space.

Also, level-3 agents can identify promising solution building blocks used to feed other metaheuristic algorithms. In addition, a notion of distance among solution is essential in many cooperative search algorithms. The level-3 agents are devoted to compute this distance.

Moreover, information can be *positive* or *negative*. In the former case, (partial or complete) solutions are exchanged if they are estimated of good quality, or if they are considered hints for promising areas of the search space. In the latter case, information is used to avoid visiting states or areas of the search space which will not lead to optimal (or good) solutions (the most prominent example of negative information is given by nogoods [197]).

Typically, cooperative search algorithms are given by the parallel or interleaved execution of search algorithms. The algorithms can be different or they can be instances of the same algorithm working on different models or running with different parameters setting. What characterizes the cooperation is the type of information exchanged and how it is exploited by the agents.

The usefulness of the multiagent metaphor emerges when considering design issues like: synchronization, communication, information filtering and implementation. A meaningful example of cooperative search in a multiagent system is reported in [41]: in addition to search agents, there are *referees* agents whose work is to filter information. This architecture permits to develop separately search agents (focusing on search algorithms), agent which collect and filter (evaluate) information, communication mechanisms and policies.

It is important to mention that cooperative search algorithms (systems) show their effectiveness and efficiency even if they are sequentially implemented. This means that improvements arise from information exchange and not only from the parallel implementation [130, 139].

Cooperative search algorithms can be described in MAGMA by appropriately defining one or more than one LEVEL-3 agents which rule and supervise the information exchange. Since at each level the agents describing the algorithms have a well defined interface, solutions and other pieces of information can be easily exchanged under the coordination of a fourth level agent. Moreover, the definition of new cooperative search algorithms is quite straightforward. Indeed, as most metaheuristics algorithms can be defined in MAGMA with three levels, the cooperation of two or more metaheuristics is achieved by adding a LEVEL-3 agent.

Combining metaheuristics on the same model

When different metaheuristics tackle the same problem they usually search on different landscapes, or the explore the same landscape with a different strategy. Empirical results (see, for instance, [71]) show that some algorithms (agents) perform better than others on particular kinds of problems, while they are outperformed on other problems. It is conjectured that this depends upon the agent ability to exploit the fitness landscape characteristics. Among such properties are: ruggedness, number of local optima, distribution of local optima and topology of the basins of attraction [118, 144, 182, 206]. The choice of fitness function and operator defines such characteristics and it is reasonable that the combination of searches on different fitness landscapes (derived from the same original problem) can smooth the problem hardness (or, at least, effectively face it).

As an example of a cooperative search algorithm we describe the combination of a population based algorithm (memetic) with GRASP.

Suppose we have a system composed of a Memetic Algorithm (MA) collaborating with GRASP. Both metaheuristics have been already described in MAGMA, respectively in sections 3.5.4 and 3.5.1. Since the algorithms have a different strategy to explore the state space, the combination of them may hopefully result in a more effective algorithm. The aim of cooperative search is to let the two metaheuristics communicate and exchange results. In particular, we suppose that in every new generation of MA, or whenever MA needs diversification, we insert the k_G best solutions found by GRASP. Vice versa, suggestions for building the restricted candidate list of GRASP are derived by the best individuals of the MA. For instance, with a certain

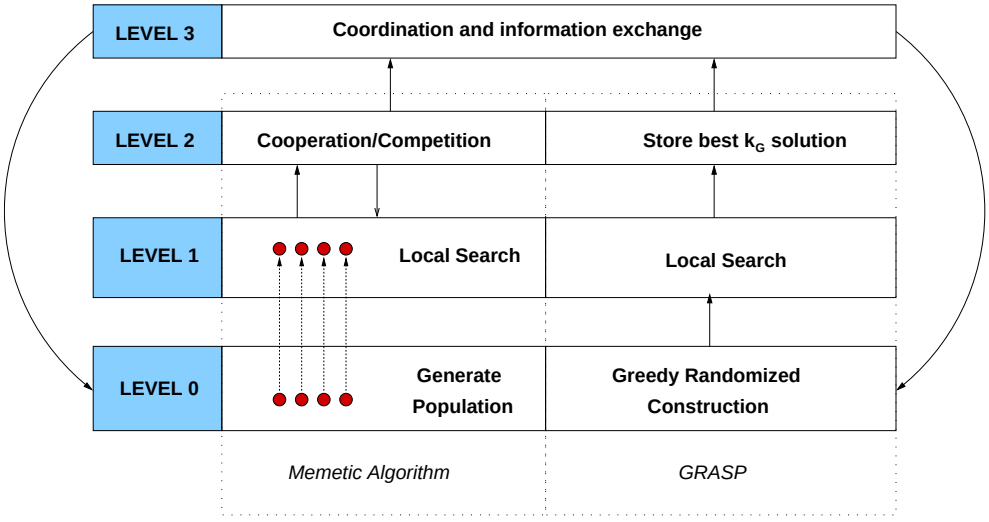

Figure 3.6: The Multi-level architecture version of Cooperative Search.

frequency (parameter of the algorithm) GRASP constructs the candidate list with an evaluation function which depends on the solution component frequency of the best k_{MA} individuals found by the MA and the heuristic.

L0, L1 and L2-agents, are those described in the previous sections for MA and GRASP. Observe that the strategy of the L2-agent of GRASP has been modified in order to enable it to maintain the k_G best found solutions, instead of only the best one. The tuples associated to these agents represent the model of an L3-agent who supervises and coordinates the information exchange. The strategy $\mathcal{S}_3$ defines the frequency of the solution exchange, builds the new population for MA and the restricted candidate list for GRASP (see figure 3.6). Its coordination model is defined as follows:

- WAIT $\xrightarrow{\text{receive}(s_1,\ldots,s_{k_G},L2-GRASP)}$ READY
- WAIT $\xrightarrow{\text{receive}(pop.,L2-MA)}$ READY
- READY $\xrightarrow{\text{Execute}(\mathcal{S}_3)}$ COMPLETED
- COMPLETED $\xrightarrow{\text{send}(cand.list,L0-GRASP)}$ WAIT
- COMPLETED $\xrightarrow{\text{send}(new\ pop.,L0-MA)}$ WAIT

The previous algorithm is an example of the cooperation of two different algorithms working on the same model, but searching with different strategies.

In the following example we show how MAGMA can also describe cooperative search algorithms composed of complete and approximate algorithms, working on different problem models.

Large Neighborhood Search

It is generally recognized that in local search algorithms the larger the neighborhood to explore, the higher the probability of finding better solutions, but the higher the computational effort needed to explore the neighborhood. Indeed, frequently large neighborhoods are just randomly sampled and not exhaustively explored. Complete techniques can be effectively integrated with local search in the exploration of large neighborhoods, resulting in a technique usually referred to as *Large Neighborhood Search* (LNS) [201].

The description of LNS is beyond the scope of this work and we just sketch the basic steps:

1. Construct initial solution through any incomplete algorithm.
2. Define a neighborhood $\mathcal{N}_k$ by selecting k variables among the n composing the solution.
3. Exhaustively explore $\mathcal{N}_k$ and find a solution s' locally optimal w.r.t. $\mathcal{N}_k$ (i.e., find the optimal solution of the subproblem $\mathcal{P}_{\mathcal{N}_k}$).
4. Go to step 2.

Suppose that we want to solve a Vehicle Routing problem where we have n trucks each starting and ending its tour in a depot. The trucks have to visit m customers within a given time window at a minimum cost. The cost is associated to each arc in the corresponding graph. Clearly, the problem can be complicated by many side constraints like truck capacity, union contract regulations, maximum length of each path etc. However, starting by this basic problem, we can intuitively describe a LNS framework. Suppose we have found a solution by using a local search or metaheuristic algorithm. We fix all routes but one (or two) and we optimally solve on the free variables a Travelling Salesman Problem with Time Windows (TSPTW). Clearly, the TSPTW is NP-hard, but if we consider small instances, there are very efficient methods that can be used to find the optimal solution [66] (by exploring a large neighborhood).

LNS can be defined in MAGMA as the cooperation of a complete solver (working on a subproblem) and a local search-based metaheuristic, for example ILS. In Figure 3.7 a very simple LNS is represented in terms of MAGMA levels. An L3-agent coordinates two algorithms: the one solving the subproblem (Figure 3.7:left) and the one providing the first solution (Figure 3.7:right). We suppose that ILS (implemented as described in Section 3.5.3 with three agents), provides the first solution to be improved by means of Large Neighborhood Search.

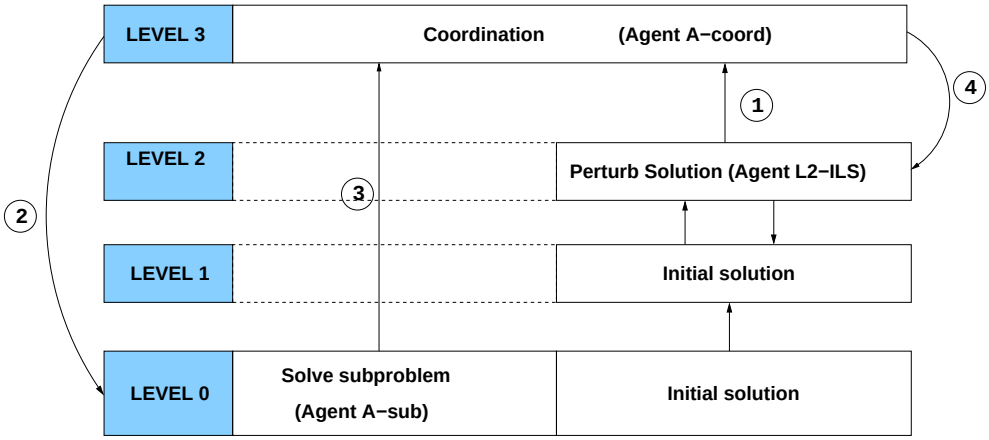

Figure 3.7: MAGMA version of an instance of Large Neighborhood Search.

The L3-agent receives the first solution from the agent of level-2 of ILS (*L2-ILS*) and defines $\mathcal{N}_k$ by fixing k out of n variables. The L3-agent communicates $\mathcal{N}_k$ to agent *A-sub* (the subproblem solver) at level 0 (arrow 2) by instantiating for it the subproblem model $\mathcal{M}_0$ in its tuple. Finally, the solution provided by *A-sub* is returned to the coordinating agent which again decides which variables to fix. This process can be iterated until a termination condition is reached. The *L2-ILS* agent can also be involved, from time to time, to perturb the solution provided by the agent *A-coord* (arrow 4) and to activate a further ILS phase.

We did not specify how *A-coord* defines $\mathcal{N}_k$ since many choices are possible. For instance, the neighborhood can be selected by choosing critical variables (e.g., bottleneck resources in scheduling problems).

Note that the ILS agents and the *A-sub* agent should not be aware of being part of a complex search strategy. They perform their usual tasks. Thus, the L3-agent is needed to enable the information exchange in a transparent way for lower level solvers.

The LTS corresponding to ILS is the following:
L0-agent of ILS:

- `WAIT` $\overset{\text{receive(inst.,INIT)}}{\longrightarrow}$ `READY`
- `READY` $\overset{\text{Execute}(\mathcal{S}_0)}{\longrightarrow}$ `COMPLETED`
- `COMPLETED` $\overset{\text{send(sol,L1-agent)}}{\longrightarrow}$ `WAIT`

L1-agent of ILS:

- `WAIT` $\overset{\text{receive}(sol,L0-agent)}{\longrightarrow}$ `READY`

- READY $\stackrel{\text{Execute}(\mathcal{S}_1)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(impr.sol, L2-agent)}{\longrightarrow}$ WAIT

L2-agent of ILS:

- WAIT $\stackrel{\text{receive}(sol, L1-agent)}{\longrightarrow}$ READY
- WAIT $\stackrel{\text{receive}(sol, A-coord)}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_2)}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send}(impr.sol, L1-agent)}{\longrightarrow}$ WAIT
- COMPLETED $\stackrel{\text{send}(impr.sol, A-coord)}{\longrightarrow}$ WAIT

A-sub:

- WAIT $\stackrel{\text{receive(inst,A-coord)}}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_{\text{A-sub}})}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send(sol,A-coord)}}{\longrightarrow}$ WAIT

A-coord:

- WAIT $\stackrel{\text{receive}(\text{sol}, L2\text{-}ILS)}{\longrightarrow}$ READY
- WAIT $\stackrel{\text{receive(sol,A-sub)}}{\longrightarrow}$ READY
- READY $\stackrel{\text{Execute}(\mathcal{S}_{\text{A-coord}})}{\longrightarrow}$ COMPLETED
- COMPLETED $\stackrel{\text{send(model,A-sub)}}{\longrightarrow}$ WAIT
- COMPLETED $\stackrel{\text{send(sol,L2-ILS)}}{\longrightarrow}$ WAIT

A plethora of variants of the proposed algorithm is possible, and they can be described in MAGMA by changing $\mathcal{S}_i$ at each level and by introducing other agents in the architecture.

3.6 Related work

A multiagent perspective of metaheuristics has relations both with object-oriented frameworks for the algorithms implementation and Distributed Artificial Intelligence (DAI). In this section we briefly discuss these issues.

Although MAGMA has been presented as a conceptual framework for metaheuristics, it can also be used as a framework for the algorithms implementation. In the literature, some implementation frameworks for local search-based metaheuristics have been introduced [5, 35, 44, 63, 133, 147]. They all share the same aims: on the one hand, fast prototyping and implementation, on the other hand, systematic comparison of algorithms. There are different choices for the implementation, but most of them provide a basic structure of two levels[5]: construction of the initial solution and further improvement. In [44] an interesting hierarchical structure of object-oriented classes goes into the direction of MAGMA (even though restricted to local search-based metaheuristics). The authors define four levels:

1. Basic components (I/O, state, moves)

2. Local Search problem elements (State manager, neighborhood explorer)

3. Local search-based metaheuristics (Iterative Improvement, Tabu Search, Simulated Annealing, etc.)

4. Local Search solving strategy (Simple solver, Token-ring solver, etc.)

When the fourth level of this architecture composes different algorithms, it can be interpreted as our LEVEL-3.

Note that the model component here are put altogether in the first level, while in MAGMA we allow to have different models/components in different levels.

Research on DAI and Multiagent Systems covers several theoretical and applicative topics. The relations with the multiagent approach for metaheuristics emerge mainly in *distributed search*, where agents cooperate to solve a search problem [134, 175, 236]. The basic approach consists of subdividing the search space, assigning to each agent a subspace and let each agent "locally" search in its own subspace. Protocols and algorithms have been developed to maintain consistency and to combine the solutions provided by the agents into a complete solution. Several cooperative techniques have also been proposed and studied, beside different synchronization methods. Even if distributed search deals with distributed implementation of classical complete search algorithms, concepts and researches about the formalization of communication and cooperation among (heterogeneous) agents may be effectively included in the specializations and implementations of MAGMA.

[5] As it can be expressed with our framework terminology.

3.7 Conclusion

In this chapter, we have introduced a multi-level architecture for describing and implementing metaheuristics. The architecture is organized in four levels: the first is devoted to the solution construction, the second to the improvement of the solution, the third to the implementation of long term strategies for intensification and diversification and the fourth is needed for coordinating lower levels. Algorithmic components are encapsulated in software agents, for which a clear interface is defined along with a coordination model.

This chapter concludes the general presentation and discussion on metaheuristics. We have first given an overview of the state of the art in Chap. 2, then we have introduced an infrastructure useful for implementation, in this chapter. In the following, we will concentrate on the application of metaheuristics to the Satisfiability Problem (SAT) and the Maximum Satisfiability Problem (MAXSAT). In the next chapter we will describe the design and implementation of two metaheuristics, for the first time applied to MAXSAT. Furthermore, in the subsequent chapters, we will deal with the important issue of the relation between problem structure and metaheuristics behavior.

Chapter 4

Metaheuristics Applications to MAXSAT

In the previous chapters, we have presented and discussed metaheuristic algorithms, without referring to a particular application. In the following chapters, we will discuss some aspects of the application of metaheuristics to the Satisfiability Problem (SAT) and the Maximum Satisfiability Problem (MAXSAT).

In this chapter we will first briefly present the metaheuristics applied so far to MAXSAT. Then we will describe two new applications, namely ACO and ILS. For each of them, we will discuss the scenarium of design issues, motivations for our choices, parameter setting procedures and experimental results. The results achieved with ACO are of good quality, though not competitive with ILS in terms of efficiency. On the other side, ILS performance is extremely good, especially for large unweighted MAXSAT instances.

This chapter aims at showing the design process of metaheuristics and it focuses on the algorithms themselves, rather than the analysis of the results and the study of the impact of problem structure on metaheuristic performance. This latter issue will be the subject of Chap. 5 and 6.

4.1 The MAXSAT Problem

MAXSAT is an optimization version of the Satisfiability Problem (SAT). SAT belongs to the class of NP-complete problems [79] and can be stated as follows: given a set of clauses, each of which is the logical disjunction of $k > 2$ *literals* (a literal is a variable negated or not), we ask whether an assignment to the variables exists that satisfies all the clauses. MAXSAT is an NP-hard problem and can be stated as follows: given

- n boolean variables $x_1, x_2, \ldots, x_n, \quad x_i \in \{0,1\}\ \forall i = 1, 2, \ldots, n$;
- $2n$ *literals*: l_i is a variable (x_i) or its negation ($\neg x_i$);

- m clauses c_j, $j = 1, \ldots, m$, where c_j is the disjunction of k literals, $l_{i_1} \vee l_{i_2} \vee \ldots \vee l_{i_k}$;
- optionally, a set of m weights $w_1, \ldots, w_m$ associated to the clauses;
- $F = \sum_{i=1}^{m} w_j \varphi(c_j)$, where $\varphi(c_j) = 1$ iff the clause c_j is satisfied by the assignment. If the weights are not defined (i.e., the instance is unweighted) we set $w_j = 1 \; \forall j = 1, \ldots, m$;

the objective is to find an assignment to the variables such that it maximizes F.

Among the best complete algorithms for MAXSAT we just mention resolution and backtracking [26] and branch & cut [125]. The drawback of complete techniques is that they can generally solve only relatively small instances. In order to face larger instances, metaheuristics are a good balance between efficiency and quality of solutions, as they generate suboptimal solutions in an amount of time compatible with practical applications.

4.2 Metaheuristics for MAXSAT

A first way to design local search algorithms to solve MAXSAT is to adapt well known local search techniques developed for SAT. The most effective are GSAT [200, 120], WalkSAT [199, 142] and their variants [115, 100, 83, 84, 190, 191]. These algorithms select from a *candidate* set (the *neighborhood*) a variable which, if flipped, would increase the objective function the most. Main differences among algorithms are in the definition of neighborhood and in the heuristics they use for selecting the variable to flip. More formally:

- $N(s) = \{z \mid Hamming\_distance(s, z) = 1\}$.
- $s^{(t+1)} = Heuristic\_Choice(N(s^t))$.

Some promising results have been obtained with different neighborhood structures in [243, 241, 190, 191].

Performance and behavior of such algorithms have been widely studied [198, 74, 178, 83, 84, 115]. It has been noticed that the search landscape, implicitly defined by the choice of the evaluation function and the neighborhood, is composed of large areas of solutions with a similar objective function value, called *plateaus*. It is generally recognized that, for SAT and MAXSAT, plateaus constitute a difficulty at least as hard as simple local optima. Local search generally reaches plateaus quickly, but once there, it becomes less effective, since the information gathered from the objective function value of neighboring solutions cannot guide it to a better region. Thus, more complex strategies are required; for example, by introducing a kind of memory in the search, or by dynamically changing the search landscape, or by changing the neighborhood structure.

4.2.1 History-based heuristics

Simple local search algorithms produce a *Markov chain* [58], as they follow a trajectory in the state space whereby the successor state is chosen depending only on the incumbent one. This implies the need of a restart after a local optimum, or the maximum allowed number of (non-improving) moves are reached. When good local optima have a small basin of attraction (i.e., they are reachable from a very small number of initial states), restart can be almost useless. But, if the local search memory is extended in such a way that it can recall a portion of past visited states, it can explore larger regions and avoid to get trapped in local optima.

A *soft* way of designing history-based searches is obtained by breaking ties (among variables which lead to the same objective function value increase) in favor of least recently flipped variables [115]. A stronger and more effective way is to forbid the last T flips; this leads to the well known metaheuristic called *Tabu Search* [90]. In [13, 14] the effectiveness of a tabu search approach for MAXSAT is shown. In this case, the dynamical system equations which define local search can be stated as follows:

- $N(s^t) = \{z \mid Hamming\_distance(s^t, z) = 1\}$.
- $Allowed(s^t) = N(s^t) - \{w \mid w$ can be obtained from s^t by applying a flip move performed less than T moves before.$\}$[1].
- $s^{(t+1)} = Heuristic\_Choice(Allowed(s^t))$.

Forbidden moves are kept in a list of length T, which is called *tabu list*. The longer T, the greater the number of not allowed moves. The tabu length tunes the balance between *intensification* and *diversification*: when T is small, once the search has reached a local optima (or a plateau) it continues to explore a small localized region; on the other hand, when T is large, the search is forced to explore a larger region, since many previous states are forbidden. Since the trade-off between intensification and diversification is crucial for local search, in [13, 14] a dynamic tabu length scheme is proposed. T is increased when the states visited by the search are repeated, and it is decreased when repetitions disappear for a sufficiently long period[2].

4.2.2 Dynamically changing landscapes

The landscape where local search moves is defined by the objective function and by the neighborhood structure. If local search is trapped in a local minimum, or it is wandering in a plateau, it can be redirected to other regions by changing the landscape shape. For example, a way to escape from a local maximum is to artificially decrease its value, until any neighbor becomes better. In MAXSAT this effect can be obtained in different ways. A very effective application is *Guided Local Search* [152], where the search is guided by varying weights associated to clauses. When the search does not

[1] The author set $1 \leq T \leq n-2$ to prevent $Allowed(s)$ from being empty.

[2] The algorithm proposed is more elaborated than the description we have given, but a detailed analysis is not in the scope of this discussion.

produce better solutions, weights associated to unsatisfied clauses are increased and thus the search is gradually forced to try to satisfy those clauses. This method was first introduced in [72, 73] as an improvement of GSAT applied to SAT problems.

A similar approach is the *Discrete Lagrangian-based search Method* [229], which is a discrete version of the usual Lagrangian method for continuous optimization. The basic idea is that an auxiliary coefficient is added to each clause weight and it is increased if the clause is not satisfied. This method differs from the previous one mainly in the coefficients update and in the local search used.

It is worth to mention the use of *non-oblivious functions* [13, 14, 15]. Since local search is directed by the objective function and a state can be a local optimum for an objective function, but not for another one, non-oblivious functions are introduced with the aim to escape from local optima. Non-oblivious functions weight in different ways satisfied clauses, depending on the number of literals which satisfy them, by favoring high redundancy. In [13, 14] it has been shown that a first search phase with non-oblivious function followed by a second phase with the usual (oblivious) objective function, leads to very good performances.

4.2.3 Variable Neighborhood Search

The search landscape can be modified also by varying the neighborhood structure. When the search reaches a local optimum, the neighborhood is enlarged or simply changed and the search can continue, hopefully toward a better local optimum. This process is the basic structure of a the *Variable Neighborhood Search* metaheuristic (VNS) [104, 102]. To apply VNS to MAXSAT it is first required the definition of a set of neighborhoods $N_k(s)$:

$z \in N_k(s)$ iff $Hamming\_distance(s, z) = k$

The high-level algorithm is sketched in the following:

1. Initialize: $k \leftarrow 1$, generate initial solution s_0, $s_{opt} \leftarrow s_0$, $z_{opt} \leftarrow F(s_0)$.
2. Repeat local search from a randomly chosen state $s_1 \in N_k(s_0)$ until a local optimum s^* is reached.
3. If $F(s^*) > z_{opt}$ then set $s_{opt} \leftarrow s^*$, $z_{opt} \leftarrow F(s^*)$, $k \leftarrow 1$; else set $k \leftarrow k + 1$ and goto step 2.

The algorithm requires also stopping conditions, the maximum value for k and it can be enhanced by more elaborate neighborhood managing schemata.

4.2.4 Constructive methods

Initial solutions are often randomly generated and so the success of search is due only to the search strategy. Nevertheless, more complex solution construction techniques can be adopted, with the aim to let the search start at a state in the basin of attraction of a good local optimum.

An interesting combination of constructive methods and local search is the *Greedy Randomized Adaptive Search Procedure* [186, 187]. An initial solution is constructed by using a combination of heuristics and then it is improved by local search. Constructive heuristics for MAXSAT incrementally assign values to variables in a *greedy* way, making at each step the assignment which decreases the total weight of not yet satisfied clauses the most. There are also other possibilities, based on different criteria (e.g., the number of positive and negative literals, the number of literals which satisfy a clause, etc.).

Ant Colony Optimization metaheuristic [47, 49, 43, 46] is another powerful example of constructive method combined with local search. The construction process is performed by a colony of agents (ants) which construct solutions by using the colony past experience and heuristic criteria. After an ant has constructed a solution, local search is applied to improve it. A more detailed description of ACO is given in Sec. 4.3.1.

4.2.5 Population heuristics

For the sake of completeness, it is worth mentioning also population heuristics such as *Genetic Algorithms* [93, 154, 28], *Memetic Algorithms* [158] and the yet cited Ant Colony Optimization metaheuristic. Population heuristics iteratively generate a set (population) of solutions, improve and combine them in order to obtain better solutions. To our knowledge, only Genetic Algorithms have been applied to MAXSAT [19, 179], with interesting results about the correlation between the behavior of genetic algorithms and problem structure. However, the effectiveness of the algorithms discussed seems not competitive with the cutting-edge metaheuristics.

As a conclusion for this overview on main metaheuristics for MAXSAT, we outline some current research trends. Most of the best performing cited algorithms combine different techniques to achieve an effective trade-off between intensification and diversification. Therefore, it is very important to know why some combinations are effective and on which kind of instances. To do this, it is necessary to develop formal models of the basic components of algorithms and analyze their interactions; furthermore, extensive simulations and deep statistical analyses are required to experimentally verify hypothesis and to generate new conjectures. Furthermore, problem structures have to be understood, via empirical analysis and formal models.

4.3 Two new applications

We developed two new applications to MAXSAT by specializing two metaheuristics, still not applied to this problem[3]. We first discuss the application of ACO, then the application of ILS. After the description of the algorithms and the design choices, we

[3]This work has been done mainly during the first phase of the European project called *Metaheuristics Network*.

will show experimental results. The results of ILS are very promising, as also proved by the experimental results obtained in the Metaheuristics Network project [145].

4.3.1 Ant Colony Optimization

In this section we will describe an ACO algorithm designed for solving MAXSAT problems.

Ant Colony Optimization has been already introduced in Chap. 2. In this section we deal with the Ant Colony System algorithm (ACS) [49], a particular instance of ACO (see Fig.17 for the basic algorithm). In ACS, each ant is initially positioned on a randomly chosen node of G and builds a solution by applying a probabilistic rule, called *state transition rule*. This probabilistic rule is biased by pheromone values in such a way that the higher the pheromone on a component/connection, the higher the probability to be selected. The general rule includes also heuristic information coded in the function η, which is combined with pheromone values, such that ants choose the component/connection depending on the joint value of past experience (pheromone) and heuristic (η). While building the solution ants "eat" some quantity of pheromone on the visited components/connections (this is called *step-by-step pheromone update*). After every ant has completed a solution, the *offline pheromone update* is applied to the components/connections of the best solution found so far, by adding a quantity of pheromone function of the quality of the solution.

Algorithm 17 High-level description of the ACS algorithm.

```
Initialize
while stopping criterion not satisfied do
  Position each ant in a starting node
  repeat
    for each ant do
      Choose next node by applying the state transition rule
      Apply step-by-step pheromone update
    end for
  until every ant has built a solution
  Update best solution
  Apply offline pheromone update
end while
```

Ants construct a solution by probabilistically choosing a node in their feasible neighborhood; in this work, we define the feasible neighborhood of an ant k as the set of pairs $(variable, value)$ such that the variable has not yet been assigned a value. The neighborhood chosen therefore implements the problem constraints that states that nodes associated to the same variable are not in a same solution. A key point for the algorithm designer is the choice of graph elements (components and connections) to which to associate pheromone. In this algorithm we decided to put pheromone on components. In this case, the amount of pheromone is proportional to the (learned)

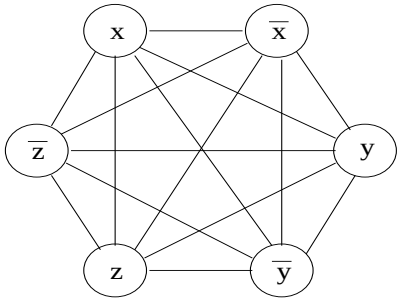

Figure 4.1: Construction graph for a MAXSAT instance with three variables. The correspondences between nodes and assignments are: $(x,1) \leftrightarrow x, (x,0) \leftrightarrow \overline{x}, (y,1) \leftrightarrow y, (y,0) \leftrightarrow \overline{y}, (z,1) \leftrightarrow z, (z,0) \leftrightarrow \overline{z}$.

desirability of having a particular assignment in the solution. An alternative possibility is to associate pheromone with connections. Thus, the quantity of pheromone on the connections between two nodes is proportional to the (learned) benefit of having the two corresponding assignments in the solution. Moreover, it is also possible to consider the sum of pheromone values on the edges which connect the candidate node to all the nodes already in the solution. These different approaches are discussed in [192].

4.3.2 ACO–MAXSAT Algorithm

The representation used is the following: each assignment (*variable,value*) is associated with a node in the construction graph, therefore, for a MAXSAT instance with n variables, we obtain a graph with $2n$ nodes (since *value* $\in \{0,1\}$). We can label the nodes with the pair (x_i, v_i), where x_i is a boolean variable ($i = 1, \ldots, n$) and $v_i \in \{0,1\}$ is the value assigned to the variable. The graph is fully connected. A solution of the problem is a path of length n, $S = \langle (x_{i_1}, v_{i_1}), \ldots, (x_{i_n}, v_{i_n}) \rangle$ such that for each pair $(x_i, v_j), (x_j, v_j) \in S$ is $x_i \neq x_j$. This constraint avoids cycles and inconsistencies in the solution.

Pheromone is deposited on nodes and also the heuristic information is evaluated for each possible assignment (x_i, v_i). An ant k constructs a solution in the following way: for n times, it adds a node to the current path (i.e., it assigns a value to a variable) by using the *pseudo-proportional transition rule*; if the ant is in node r, it chooses the next node s according to the following rule:

$$s = \begin{cases} arg\ max_{u \in J_k(r)} \{[\tau(u)]^\alpha \cdot [\eta(u)]^\beta\}, & \text{if } q \leq q_0 \text{ (exploitation)} \\ z, & \text{otherwise (biased exploration)} \end{cases}$$

where: q is a random number in [0,1], $J_k(r)$ is the set of allowed nodes (given

the path constructed by ant k till node r) and z is chosen with probability (*random-proportional rule*):

$$prob(z) = \begin{cases} \frac{[\tau(z)]^{\alpha} \cdot [\eta(z)]^{\beta}}{\sum_{u \in J_k(r)} [\tau(u)]^{\alpha} \cdot [\eta(u)]^{\beta}}, & \text{if } z \in J_k(r) \\ 0, & \text{otherwise} \end{cases}$$

α, β and q_0 are parameters of the algorithm.
The step-by-step pheromone update rule erases a quantity of pheromone from the nodes included in the solution constructed by each ant:

$$\forall \text{ ant } k, \forall s \in S_k : \tau(s) \leftarrow (1-\rho) \cdot \tau(s) + \tau_0,$$

where ρ and τ_0 are parameters in [0,1].
The offline pheromone update is evaluated according to the following rule:

$$\forall s \in S_{opt} : \tau(s) \leftarrow (1-\xi) \cdot \tau(s) + \xi \cdot g(S_{opt}),$$

where ξ is a parameter analogous to ρ and $g(S_{opt})$ is the normalized value of the solution S_{opt}.

The stopping criterion can be stated, for example, with a maximum execution time, a maximum number of cycles or the maximum error from the best solution known.

As heuristic information (quantified by η) for the solution construction we tried two different kinds of heuristic: *static* and *dynamic*. Static heuristics evaluate the desirability of an assignment on the basis of some instance properties (e.g., formula and weights). One among the simplest static heuristic consists of assigning to a variable the value that enables it to satisfy the greatest number of clauses (or, in a weighted MAXSAT, the highest sum of weights). Heuristic values $\eta((x,v))$ are computed by counting, for each variable x, the number of clauses in which it is a positive literal; then we assign to $\eta((x,1))$ the weighted sum of clauses satisfied by that assignment. We proceed in an analogous way to obtain $\eta((x,0))$. The results obtained with this heuristic were not good; moreover, often the use of η is misleading and it slows down the convergence toward a good solution. On the contrary, dynamic heuristic led to very good results. A dynamic heuristic changes the values of η during the solution construction, depending on some criteria. We adopted a heuristic function like the one defined in [187]: given a not yet assigned variable x, $\eta((x,v))$ is equal to the sum of weights of unsatisfied clauses which become satisfied if x is set to v. In formulas: $\eta((x_i,0)) = \sum_{j \in \Gamma_i^-} w_j$ and $\eta((x_i,1)) = \sum_{j \in \Gamma_i^+} w_j$, where Γ_i^- (resp. Γ_i^+) is the set of unsatisfied clauses which became satisfied if x_i is set to 0 (resp. 1). The advantage of this heuristic is obvious: depending on the past choices, η is re-calculated in order to evaluate the most promising remaining assignments. The drawback is in the execution time, since new values have to be computed at each construction step. Nevertheless, this heuristic information have been proved very useful and it led to good results, especially when combined with the pheromone values.

4.3.3 Parameters setting

Parameters setting in metaheuristics represents one of the most empirical procedures of the whole implementation process. In ACO algorithms, parameters are not independent and their value is also problem-dependent, but ACO procedures seem to be in general quite robust [49, 50]: once limited parameter ranges into reasonable intervals, the overall performance is not strongly affected by parameters variations. However, the optimization of algorithms requires tuning parameters. In [27] a genetic approach for parameters setting is presented and there is still room for developing automated procedures[4]. The tuning of parameters is an optimization process; it is nonlinear, because of the relations among parameters and it is multi-objective, since usually we want to optimize the effectiveness (quality of solution), the efficiency (computation time) and robustness (good average global performance over all the instances in terms of the last two objectives). We tackled this problem with a procedure similar to a gradient ascent algorithm in the space of parameters. First of all, we have to provide a benchmark problems set: in this case we chose a combination of unweighted and weighted instances. Then, we run different settings (derived, for example, from previous works on ACS) on the benchmark set and we select for each parameter p_h a set of values V_h $(h = 1, \ldots, N_{param})$. Thus, the problem is to find an assignment $\{(p_1, v_1), \ldots, (p_{N_{param}}, v_{N_{param}})\}$, $v_h \in V_h$, which leads to a good balance among effectiveness, efficiency and robustness. We start from a heuristically generated assignment $\{(p_1, v_1^0), \ldots, (p_{N_{param}}, v_{N_{param}}^0)\}$ and we iterate the following procedure: at step h, only parameters $p_1, \ldots, p_{h-1}$ have been assigned and the current assignment is $\{(p_1, v_1^*)$, ..., (p_{h-1}, v_{h-1}^*), ..., (p_h, v_h^0), ..., $(p_{N_{param}}, v_{N_{param}}^0)\}$. The algorithm is run on each instance for every possible value of parameter p_h, while the other parameters are kept constant. Then, the optimal value v_h^* is chosen. At the end, we obtain the assignment $\{(p_1, v_1^*), \ldots, (p_h, v_h^*), \ldots, (p_{N_{param}}, v_{N_{param}}^*)\}$ which is, at least, not worse than any other partial assignment $\{(p_1, v_1^*), \ldots, (p_{h-1}, v_{h-1}^*), \ldots, (p_h, v_h^0), \ldots, (p_{N_{param}}, v_{N_{param}}^0)\}$.

For the ACS algorithm applied to MAXSAT (thereafter referred to as ACS-MAXSAT), the parameters are:

- Number of ants.
- α and β, parameters which adjust the relative importance between pheromone and heuristic.
- ρ, parameter which controls the amount of pheromone "eaten" away by ants.
- ξ, evaporation parameter.
- τ_0, parameter involved in the step-by-step pheromone update.
- q_0, parameter involved in the transition rule.

[4] A recent very interesting and promising well-founded research on this topic can be found in [195].

4.3.4 Experimental Results

In the following we report experimental results, mainly focusing on the parameter optimization.

Instances from two different sets have been used:

- Random weighted instances (*jnh*).
  `http://www.research.att.com/~mgcr/data/maxsat.tar.gz`

- Random unweighted instances (*uuf*).
  `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB`

jnh have 100 variables and a number of clauses which ranges between 800 and 900. The *uuf* considered instances have 250 variables and 1065 clauses.

The initial setting is: 10 ants, $\alpha = \beta = 1$, $\rho = \xi = 0.01$, $\tau_0 = 0.01$ and $q_0 = 0.8$. For each instance we reported either the average error from the optimal solution (in the case of weighted instances, when the best solution is known), or the best average solution (in the case of unweighted instances, for which the optimal solution is not known). Moreover, the average time and number of iterations is reported. The averages are evaluated over 20 runs and each run is allowed a maximum of 100 cycles. The algorithm run on a Pentium II at 400MHz, with 512 MB of RAM and 512 KB of cache memory.

In Table 4.1 results for different values of the number of ants are reported. The lower the number of the ants, the lower the computational load. Nevertheless, with a small number of ants the learning process is not effective. The optimal number of ants depends also on the balance between exploration and exploitation (q_0), the amount of pheromone deposited (τ_0 and the amount depending on the solution value) and erased (ρ and ξ). From Table 4.1 we observe that the quality of solutions increases as the number of ants increases, whilst the time required increases approximately linearly. As optimal number of ants we choose 10, as the extra computational time required for 20 ants is not counterbalanced by the improvement in solution quality.

The effect of different values for ρ can be observed in Table 4.2. The optimal choice seems to be $\rho = 0.01$, because it gives the best average solution quality. Analogous considerations are valid for ξ, whose optimal value is 0.01 (see Table 4.3).

Table 4.4 reports the algorithm performance for different combinations of values of α and β. For $\beta/\alpha \in [0.5, 2]$ the performance is not highly perturbed; if we consider the quality of solutions, the best ratio is $\beta/\alpha = 1.5$. It is worth observing the first two columns: ($\alpha = 1$, $\beta = 0$) $\leftrightarrow$ only pheromone trails are used and ($\alpha = 0$, $\beta = 1$) $\leftrightarrow$ only heuristic is used. For weighted instances better solutions are obtained by using only pheromone trails, but with longer times. On unweighted instances, only heuristic information seems better. This phenomenon leads to conjecture that weighted instances are more structured than unweighted ones, thus ants can exploit this structure and find better solutions. However, the combination of the two sources of information enables to achieve the best average solution quality.

Table 4.1: Average error (for *uuf* instances the number of satisfied clauses is reported), time and number of iterations for different number of ants. The remaining parameters are: $\alpha = \beta = 1$, $\rho = \xi = 0.01$, $\tau_0 = 0.01$ and $q_0 = 0.8$

	Number of ants			
Instance	1	5	10	20
jnh1	1.025% 1.31 85.4	0.476% 6.64 83.1	0.386% 14.18 87.7	0.331% 24.17 72.3
jnh201	1.290% 1.21 79.5	0.505% 6.88 88.7	0.384% 12.74 82.4	0.344% 24.69 76.2
jnh301	1.526% 1.34 84.3	0.748% 7.20 86.5	0.483% 15.43 88.3	0.421% 28.67 83.9
uuf250-01	1025.6 4.03 79.6	1039.35 22.59 88.4	1042.7 42.37 85.7	1045.25 83.15 86.6
uuf250-02	1018.85 3.96 75.6	1031.5 23.78 88.0	1035.6 44.23 84.4	1038.5 85.53 87.0

Table 4.2: Average error (for *uuf* instances the number of satisfied clauses is reported), time and number of iterations for different values of ρ. The remaining parameters are: 10 ants, $\alpha = \beta = 1$, $\xi = 0.01$, $\tau_0 = 0.01$ and $q_0 = 0.8$

	ρ				
Instance	0.005	0.01	0.05	0.1	0.5
jnh1	0.389% 14.49 86.3	0.337% 13.78 84.7	0.405% 13.37 79.6	0.477% 12.57 74.95	0.704% 10.42 62.0
jnh201	0.422% 13.78 85.9	0.425% 12.90 83.1	0.423% 13.45 84.25	0.494% 11.82 73.2	0.710% 10.50 65.4
jnh301	0.630% 15.60 85.6	0.517% 14.43 82.1	0.524% 14.50 80.3	0.635% 15.98 88.0	0.962% 12.30 67.9
uuf250-01	1041.10 45.04 87.3	1042.3 41.06 82.7	1040.15 36.98 72.6	1037.75 32.59 64.15	1035.45 35.21 69.25
uuf250-2	1029.15 31.25 60.30	1041.10 45.04 87.3	1036.55 43.55 85.6	1033.35 35.33 69.6	1031.70 35.61 70.2

Table 4.3: Average error (for *uuf* instances the number of satisfied clauses is reported), time and number of iterations for different values of ξ. The remaining parameters are: 10 ants, $\alpha = \beta = 1$, $\rho = 0.01$, $\tau_0 = 0.01$ and $q_0 = 0.8$

	ξ				
Instance	0.005	0.01	0.05	0.1	0.5
jnh1	0.403% 13.39 80.8	0.373% 14.28 86.4	0.433% 14.49 80.6	0.450% 13.45 80.2	0.572% 14.23 85.9
jnh201	0.467% 12.50 79.0	0.440 % 13.69 86.9	0.443 % 12.51 77.3	0.537% 12.80 78.4	0.595% 12.68 79.8
jnh301	0.583% 15.25 85.1	0.575% 15.32 85.5	0.654% 14.25 78.8	0.727% 13.95 77.4	0.847% 15.21 85.4
uuf250-01	1042.40 40.23 79.4	1043.05 42.63 84.6	1041.45 47.04 86.4	1040.00 42.93 85.2	1037.90 44.21 88.0
uuf250-02	1035.60 41.41 82.1	1036.70 45.11 89.0	1035.50 46.40 89.2	1033.00 42.51 84.4	1031.80 43.31 86.1

Table 4.4: Average error (for *uuf* instances the number of satisfied clauses is reported), time and number of iterations for different combinations of α and β. The remaining parameters are: 10 ants, $\rho = \xi = 0.01$, $\tau_0 = 0.01$ and $q_0 = 0.8$

	α,β						
Instance	1 , 0	0 , 1	1 , 0.5	1 , 1	1 , 1.5	1 , 2	1 , 3
jnh1	0.629% 14.51 92.5	1.098% 9.89 63.0	0.419% 19.80 85.7	0.360% 13.88 84.3	0.394% 20.61 89.2	0.423% 14.13 84.1	0.444% 13.14 77.0
jnh201	0.432% 13.64 90.9	1.015% 6.41 42.8	0.432% 16.16 73.8	0.439% 13.43 84.9	0.376% 18.72 84.7	0.442% 13.44 82.3	0.448% 14.10 86.0
jnh301	0.665% 15.97 94.5	1.341% 7.80 45.7	0.582% 20.63 85.7	0.548% 15.59 85.9	0.519% 21.42 88.0	0.523% 16.55 89.8	0.599% 16.61 90.0
uuf250-01	1005.95 43.36 96.0	1030.45 19.21 42.8	1041.90 75.70 85.6	1043.05 43.26 85.8	1044.40 79.45 89.4	1043.50 42.47 80.0	1043.80 44.65 83.4
uuf250-02	1007.65 42.72 94.6	1023.45 14.03 31.3	1034.15 76.94 86.7	1035.85 40.69 80.2	1036.90 76.95 86.2	1037.20 42.82 81.3	1037.45 44.05 82.5

Parameter τ_0 determines the lowest value for the pheromone [211]. In Table 4.5 we can observe that the most promising values are between 0.01 and 0.05, with a slightly better performance for $\tau_0 = 0.01$.

Finally, from Table 4.6 we can assert that $q_0 = 0.8$ seems to give the best trade-off between exploration and exploitation.

Concluding, the best parameters setting is: 10 ants, $\alpha = 1$, $\beta = 1.5$, $\rho = \xi = 0.01$, $\tau_0 = 0.01$ and $q_0 = 0.8$.

Table 4.7 summarizes first results of ACS-MAXSAT with the optimal parameters setting.

4.3.5 Ant Colony plus Local Search

In the ACO framework it is included the possibility to apply local search after the construction of solutions. The basic algorithm is sketched in Alg. 18.

Algorithm 18 ACO plus Local Search high level algorithm.

while stopping criterion not satisfied **do**
  Ants construct a set of solutions $S = \{s_1, s_2, \ldots, s_{n_{ants}}\}$
  Apply local search to every solution in S and obtain a new set S'
  Update pheromone trails of S' by using the solution values of the elements of S'
end while

Local search is applied to each solution provided by the ants and the new solution substitutes the previous one. In this sense, we apply a kind of Lamarckian update [237], because the new set of improved solutions (population) replaces the previous one and pheromone update takes place over the new one. This is the usual way to insert local search into ACO algorithms, however there is also the possibility to apply a sort of Darwinian update, whereas the solutions improved by local search are used just to compute the amount of pheromone to add during the pheromone update phase.

As local search algorithm, we implemented a variant of GSAT, adapted for MAX-SAT. The basic structure of the algorithm, called MAX-GSAT, is reported in Alg. 19. The algorithm starts with an initial solution (in this case, provided by an ant) and, until the termination condition is reached, it "flips" the variable which produces the greatest $\delta = \omega^+ - \omega^-$, where ω^+ (ω^-) is the sum of weights of the unsatisfied (satisfied) clauses which become satisfied (unsatisfied) after the flip of the variable. As termination condition we adopted the following criterion: the search stops after $MAXMOVES$ flips without improvements. $MAXMOVES$ becomes therefore an algorithm parameter and, after some trials, we decided to set $MAXMOVES = number\ of\ variables$.

First results are reported in Table 4.8. As we can observe, local search strongly improves the algorithm performance. An improved version of MAX-GSAT which uses also Tabu Search[5] is under testing. The actual synergy between ACO and local search

[5] I thank Thomas Stützle for making available the source code of a very efficient implementation.

Table 4.5: Average error (for *uuf* instances the number of satisfied clauses is reported), time and number of iterations for different combinations of τ_0. The remaining parameters are: 10 ants, $\alpha = 1$, $\beta = 1.5$, $\rho = \xi = 0.01$ and $q_0 = 0.8$

	τ_0					
Instance	0.001	0.005	0.01	0.05	0.1	0.5
jnh1	0.510% 18.26 79.0	0.388% 18.68 80.7	0.395% 20.52 87.5	0.479% 18.27 79.2	0.596% 15.49 67.0	0.900% 10.75 46.4
jnh201	0.496 % 16.82 76.5	0.445% 17.91 81.3	0.429% 19.04 86.4	0.529% 17.62 79.8	0.628 % 15.57 69.7	0.889% 10.98 49.3
jnh301	0.782% 19.47 80.3	0.604% 20.62 85.3	0.572% 20.35 84.6	0.659% 19.98 82.7	0.828% 18.53 75.9	1.240% 15.12 62.2
uuf250-01	1039.95 78.51 87.8	1043.15 82.47 92.6	1042.90 73.36 82.6	1040.25 64.30 72.3	1039.00 57.71 65.0	1034.40 43.04 48.5
uuf250-01	1032.85 79.68 89.4	1037.20 80.92 90.6	1037.95 75.54 85.3	1034.05 64.78 73.1	1031.80 53.31 60.0	1026.05 55.78 63.0

Table 4.6: Average error (for *uuf* instances the number of satisfied clauses is reported), time and number of iterations for different combinations of q_0. The remaining parameters are: 10 ants, $\alpha = 1$, $\beta = 1.5$, $\rho = \xi = 0.01$ and $tau_0 = 0.01$

	q_0					
Instance	0	0.2	0.4	0.6	0.8	0.95
jnh1	0.847% 17.38 75.6	0.652% 19.21 81.8	0.493% 19.79 84.2	0.424% 19.25 82.5	0.350% 19.52 84.5	0.405% 19.24 83.5
jnh201	0.811% 17.41 80.7	0.559% 18.38 80.7	0.479% 19.40 84.9	0.405% 18.53 83.7	0.407% 18.38 83.4	0.465% 18.57 83.6
jnh301	1.131% 18.80 78.3	0.961% 19.08 77.5	0.760% 20.51 83.6	0.678% 20.72 85.3	0.577% 20.18 83.3	0.690% 18.66 77.5
uuf250-01	1024.75 68.43 75.7	1030.70 66.64 73.0	1035.15 69.80 78.0	1039.50 69.08 77.1	1042.85 75.38 84.7	1043.20 77.65 87.2
uuf250-02	1020.10 68.30 75.3	1024.80 72.95 80.4	1029.65 64.20 71.2	1034.70 73.87 82.4	1036.65 76.38 85.7	1033.85 82.78 93.2

Table 4.7: ACS-MAXSAT. No local search.

instance	av. error (av. best)	std.dev.	av. time	std.dev.	av. iter.	std.dev.
jnh1	0.350%	0.081	19.24	2.90	83.5	12.5
jnh201	0.407%	0.096	18.38	4.45	83.4	20.0
jnh301	0.577%	0.206	20.18	3.78	83.3	15.5
uuf250-01	1042.85	2.62	75.38	11.62	84.7	13.0
uuf250-02	1036.65	2.64	76.38	6.76	85.75	7.44

Table 4.8: ACS-MAXSAT plus local search.

instance	average error (average best)	std.dev.	average time	std.dev.	average iter.	std.dev.
jnh1	0.030%	0.017	8.18	9.57	21.9	26.9
jnh201	0.009%	0.014	5.44	5.86	15.25	17.69
jnh301	0.038%	0.037	10.24	13.39	25.4	34.41
uuf250-01	1060.5	1.57	7.08	5.17	6.05	4.79
uuf250-02	1059.95	1.66	9.50	8.41	8.20	7.81

is still an issue of current research.

4.3.6 An ILS Algorithm for MAXSAT

Iterated Local Search has been already introduced in Chap. 2. In this section, we describe our implementation for tackling MAXSAT.

The ILS algorithm we designed combines a Tabu Search (based upon GSAT local search) as basic local search, with WalkSAT moves as perturbation. Moreover, the tabu length is adaptively changed in order to gradually favor exploration when no improvements are not produced. The first phase of search is improved by using GSAT with parallel flips. The high level algorithm of ILS-MAXSAT is described in Figure 20. In the following we analyze each component in more detail.

Initial Solution

The importance of an initial solution depends upon the search strategy: a search with limited diversification requires a multi-start strategy and then a good initial solution; on the other hand, for a more elaborated search strategy a random initial solution could be the most efficient way to start. For ILS-MAXSAT we decided to use a random solution, because it is the fastest way of generating the initial solution. Consequently, we decided to improve the search strategy.

First phase of local search: Parallel MAX-GSAT

The goal of the first application of local search is to reach a local optimum, hopefully

Algorithm 19 MAX-GSAT

```
Input: a set of clauses α, MAXMOVES
Output: a truth assignment of α
T ← initial truth assignment
MOVES ← 0
besterror ← Eval(α, T) {Eval(α, T) returns the number of unsatisfied clauses}
bestsolution ← T
while MOVES < MAXMOVES do
  if T satisfies α then
    return T
  end if
  p ← a propositional variable such that a change in its truth assignment gives the
  maximum increase in the weighted sum of clauses of α that are satisfied by T
  T ← T with p reversed
  MOVES ← MOVES + 1
  error ← Eval(α, T)
  if error < besterror then
    MOVES ← 0
    bestsolution ← T
    besterror ← error
  end if
end while
return bestsolution
```

in a promising area of the search space. In [191, 190] a variant of GSAT is presented[6]: variables are divided in τ subsets of equal cardinality and for each subset the variable which fulfills the GSAT criterion is flipped. This results in the application of τ simultaneous variable flips. Depending on some instance properties, an optimal degree of parallelism τ_{opt} exists for which better solutions are found on average. τ_{opt} is, in general, not known in advance; nevertheless, from preliminary tests, we discovered that parallelism in any case boosts the search and produces local optima of better quality than single-flip local search. We adopted a decreasing τ, starting from a fraction of n (number of variables) and decreasing it at each step. The search stops when the maximum number of non-improving moves is reached. In Figure 4.2 the typical behavior of the first phase of MAX-GSAT for single and parallel flips is depicted. The parallel version reaches a good plateau faster than the single-flip version.

[6]This topic will be deeply discussed in Chap. 5.

Algorithm 20 Iterated Local Search (ILS)

$s_0 \leftarrow$ Generate_Initial_Solution_Randomly
$s^* \leftarrow$ Parallel_MAX–GSAT
while termination conditions not met **do**
  $s' \leftarrow$ WalkSAT_Perturbation($s^*, number\ of\ non\_improving\ moves$)
  Adapt_tabu_length($number\ of\ non\_improving\ moves$)
  $s^{*'} \leftarrow$ Tabu_Search(s')
  if $F(s^{*'}) > F(s^*)$ **then**
    $s^{*'} \leftarrow s^*$
  end if
end while

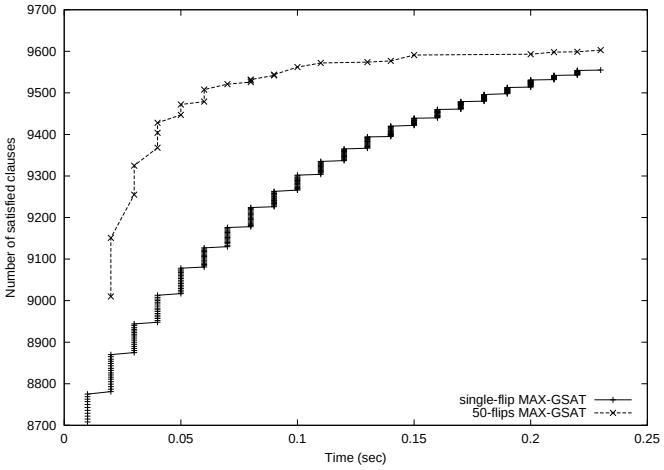

Figure 4.2: Typical behavior of single-flip and parallel-flips MAX-GSAT at the beginning of search. The plotted data are the result of a run of single-flip and 50-flips MAX-GSAT on a random generated instance of MAX-3SAT with 1000 variables and 10000 clauses.

Tabu Search

The local search used to reach the next local optimum after a perturbation is basically a MAX-GSAT with a variable length tabu list. The simple greedy hill climbing of GSAT is combined with the effective history-based tabu search with the intent to provide a simple yet powerful basic scheme to explore plateaus. The tabu length is incremented at each not accepted new local optimum and it is restarted when an acceptance occurs.

WalkSAT Perturbation

The choice of an elaborated perturbation instead of a simple random one is due to preliminary tests where it has been discovered that random perturbations are not effective for the MAXSAT instances attacked. The simplest WalkSAT version can be easily implemented on GSAT data structure, as it can be roughly considered a GSAT local search with a different neighborhood structure. WalkSAT randomly chooses a not satisfied clause and flips within it the variable which increases the sum of weights of satisfied clauses the most. WalkSAT and GSAT can be considered as dual approaches, since the first is a kind of repair algorithm, whilst the second is a typical greedy technique. The perturbation strength is the number of WalkSAT moves; it is incremented at each not accepted local optimum and restarted when the local optimum is accepted.

ILS-MAXSAT was first tested with a C++ implementation and further re-implemented in C using efficient data structures. In Table 4.9, results of the second implementation are reported. The averages are evaluated over 20 runs and each run is allowed a maximum time of 30 seconds. The algorithm run on a Pentium II at 400MHz, with 512 MB of RAM and 512 KB of cache memory.

4.4 Conclusion

In this chapter we have presented the application of metaheuristics to MAXSAT problems. In particular, we have discussed the implementation of ACO and ILS, for the first time applied to MAXSAT. We have seen that the concepts of intensification and diversification, discussed in Chap. 2, and modularity, described in Chap 3, have an important role in the design process of the two metaheuristics. Moreover, we have discussed a possible systematic procedure for tuning parameters. The experimental results show that ACO can achieve good results, but in conjunction with local search. ILS has a very good performance, both in terms of efficiency and effectiveness.

In this chapter, we focused on the algorithms, their structure and components, and on design issues. This is one of the two main topics of an engineering perspective of metaheuristics. A second topic concerns the empirical investigation of algorithm behavior and the study of the problem characteristics that have influence on the performance. In the following chapters we will investigate this second issue.

Table 4.9: Average results of ILS-MAXSAT on weighted and unweighted instances. *3sat500* and *3sat1000* instances can be retrieved from `http://rtm.science.unitn.it/intertools/sat/benchmark.html`.

instance	av. best	std.dev.	av. iter.	std.dev.	av. time (s)	std.dev.
jnh1	420925.00	0.00	15636.20	11598.84	0.21	0.15
jnh4	420827.00	9.71	113321.65	96102.29	1.55	1.35
jnh201	394238.00	0.00	1837.65	1282.87	0.02	0.01
jnh202	394148.25	39.97	87760.80	88245.98	1.12	1.13
jnh301	444848.60	6.12	83179.50	72040.54	1.16	1.01
jnh302	444459.00	0.00	37885.85	18524.51	0.53	0.26
uuf250-01	1064.00	0.00	53589.75	18646.87	1.07	0.34
uuf250-02	1063.00	0.00	129584.85	135922.35	2.33	2.29
uuf250-03	1064.00	0.00	53153.95	22476.21	1.07	0.40
3sat500_5000_1	4843.85	0.36	182337.70	94227.65	6.81	3.21
3sat500_5000_2	4840.75	0.44	234573.60	140915.76	8.66	4.87
3sat1000_10000_1	9689.65	1.03	297190.55	95478.42	19.74	5.50

Chapter 5

Criticality and Parallelism in SAT

The impact of problem structure on search is a relevant issue in AI research and related areas. Since the ultimate goal is to design and tune effective and efficient algorithms, the relations between structural problem features and algorithm performance have to be investigated. This topic has been recently received more attention, due to the following reasons: *(i)* real-world problems are often more difficult to solve than random generated problems of the same size and *(ii)* results obtained by applying statistical mechanics techniques (such as phase transition analysis [112]) have shown a strong correlation between search effectiveness and some critical parameters of the instances at hand.

In this chapter we investigate the relation between some SAT/MAXSAT instance features with the behavior of trajectory methods. We will define structural features on the basis of a constraint graph associated to the instances and in particular we will deeply investigate the impact of the node degree distribution on the application of parallel local moves. The results obtained generalize a phenomenon called *Criticality and Parallelism*, first discovered in combinatorial optimization problems such as the TSP and NK-spaces [139], since they apply to more complex search strategies and different kinds of problems and they also take into account structured instances.

Moreover, in Chap. 6, we also study the correlation between hardness of problems associated to constraint graphs characterized by the *small-world* topology, showing empirical evidence of higher hardness corresponding to small-world instances.

5.1 Structure of Satisfiability Problems

This section is devoted to the discussion of structure in SAT/MAXSAT problems. First, a general notion of structure is introduced, then we focus on a possible way of representing the relations among variables in SAT instances by defining a constraint

graph. Finally, we analyze the relevant parameters and discuss their impact on search.

5.1.1 What is *structure?*

The definition of structure emerging from the literature on Constraint Satisfaction Problems and Combinatorial Optimization Problems is usually based on the informal notion of a property enjoyed by non-random problems. Thus, *structured* is used to indicate that the instance is derived from a real-world problem or it is an instance generated with some similarity with a real-world problem. Commonly, we say structured for a problem which shows, under some abstraction, regularities such as well defined subproblems, patterns or correlations among problem variables.

There are also some quantitative measures of structure, such as entropy (see for example [110]), small-world proximity [230] and compression ratio [193].

The impact of problem structure on search performance has been studied from different perspectives. Studies on the impact of problem structure on heuristic search can be found, for example, in [17, 238, 232, 135]. Important results and observations on structure and problem hardness are reported in [98, 96, 97]. The effects of problem encoding are discussed in [116, 18]. Finally, the search algorithms behavior w.r.t. graph properties has been discussed in [231, 230].

5.1.2 Graph representations of SAT

Some problems suggest a natural structural description, since they have a representation suitable for structure analysis. A classical example are problems defined on graphs, such as the Graph Coloring Problem and the k-Cardinality Tree Problem. In general, for CSPs a constraint graph can be defined, where nodes correspond to variables and edges connect two variables if there exists a constraint involving them[1].

In general, we have to choose a level of abstraction and a suitable data structure to associate to the problem. Then we characterize the structure on the basis of relevant properties of the model we have obtained.

For SAT-like problems, the ones defined with binary variables and clauses, a graph with strong similarity with the CSP constraint graph can be defined. The graph associated with a SAT instance is an undirected graph $G = (V, A)$, where each node $v_i \in V$ corresponds to a variable and edge $(v_i, v_j) \in A$ $(i \neq j)$ if and only if variables v_i and v_j appear in a same clause. For instance, in Fig. 5.1 the graph corresponding to the formula $F_1 = (a \vee \neg b) \wedge (b \vee d) \wedge (c \vee \neg d \vee \neg e)$ is depicted.

Observe that the same graph corresponds to more than one formula, since nodes are connected by only one arc even if the corresponding variables belong to more than one clause. For example, the graph of Fig. 5.1 corresponds also to the following instances: $F_2 = (\neg a \vee \neg b) \wedge (\neg b \vee \neg d) \wedge (\neg c \vee \neg d \vee \neg e)$, $F_3 = (a \vee \neg b) \wedge (\neg a \vee b) \wedge (b \vee d) \wedge (c \vee \neg d \vee \neg e) \wedge (c \vee d \vee e)$. F_2 has the same number of clauses as F_1, but some clauses in F_2 are different. Also F_3 has the same associated graph, but it has a different number of clauses than the previous formulas.

[1] We limit our discussion to binary constraints.

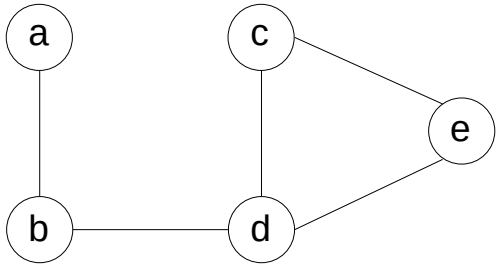

Figure 5.1: Constraint graph associated to a SAT instance.

Having a set of clauses associated to the same graph, makes this representation quite rough. Nevertheless, in the following, it will be shown that some properties of this SAT-associated graph strongly affect the behavior of metaheuristics on SAT and MAXSAT instances.

Graphs of different kind can be defined to study the structure of SAT problems. For instance, a graph can be defined with weights on edges to indicate the number of clauses involving the connected variables. Moreover, it is possible to construct a graph where nodes corresponds to literals instead of variables and adding one node for each clause.

For brevity, in the following we will refer to the simple graph defined at the beginning of this section as *SATgraph*.

5.1.3 Structural Properties of SAT Problems

In this section we consider the SATgraph associated to SAT/MAXSAT instances and its characteristics, in order to give the preliminary knowledge for the next sections.

The impact of graph properties on system behavior has been recently received attention, as witnessed by the wide spectrum of publications on the subject [235, 163, 4]. Graphs (or equivalently, networks) are the basic structure for a large number of phenomena and problems, from physics to economics, from computer science to social sciences. Many problems can be formulated in terms of networks which describe the relations among components and this abstraction enables us to characterize some general properties of the problem or the system.

Among the most famous examples we mention the *small-world* phenomenon [156, 61, 235, 234], that reached the popularity with the 1990's movie "Six degrees of separation". The small-world phenomenon was experimented first by the psychologist Stanley Milgram, who asked some persons to send a packet to persons they did not know, by just sending the packet to someone they knew and they considered closer to the target person. Thus, the delivery would have been achieved by a chain of acquaintances: the path between the source and the target is composed of persons who know their previous and next neighbor. Surprisingly, the average length of paths was fairly low: six. Beyond the popularity of the experiment, small-world is a very

important phenomenon emerging in very different and apparently distant fields. In chapter 6 we will discuss an example of the impact of the small-world property on the hardness of SAT instances.

In the following we define the properties which will be considered as relevant for the purposes of this thesis.

Given a non-oriented and simple graph, associated to a SAT instance, we consider the following parameters:

- Node degree
- Characteristic path length
- Clustering

In an instance with n variables, each node v_i, $i = 1, \ldots, n$, has a degree $q_i \in \{0, 1, \ldots, n-1\}$. For k-SAT problems, defined as conjunction of clauses with exactly k literal each, it holds $k - 1 \leq q_i \leq n - 1$. We define the *average connectivity* of the instance as the average node degree of the corresponding graph, i.e., $q = \frac{1}{n} \sum_{i=1}^{n} q_i$. Moreover, to make direct comparisons among instances with different number of variables, we also introduce the normalization of q: $\overline{q} = \frac{q}{n-1}$. In the following, we will use indifferently the expressions connectivity and node degree of an instance $\mathcal{I}$, being the second defined on the graph associated with $\mathcal{I}$. In order to compare the node degree distribution between instances, we will also consider the frequency of node degree $Freq(j)$ = 'frequency of a node connected to exactly j nodes' and the cumulative frequency $CumFreq(j)$ = 'frequency of a node connected to not more than j nodes'.

The connectivity gives a rough evaluation of the speed at which a modification occurring on a node affects the other nodes. The higher the connectivity, the stronger the "information spreading"[2]. The impact of connectivity will be discussed in detail in the following.

The characteristic path length $L(G)$ of a graph $G = (V, A)$ can be informally defined as the average path length between any pair of nodes. For simplicity, we will assume that the graph is connected, therefore L is always finite. Formally, the characteristic path length L of a graph G is defined as the median of the means of the shortest paths connecting each vertex $v \in V$ to all other vertices.

Finally, the clustering coefficient γ of a graph G quantifies the probability that, given node v_1 connected to v_2 and v_3, there is an edge between v_2 and v_3. For instance, friendship relations are characterized by a high value of γ. Formally, the clustering coefficient is defined on the basis of the notion of *neighborhood*. The *neighborhood* Γ_v of a node $v \in V$ is the subgraph consisting of the nodes adjacent to v (not including v itself). The clustering of a neighborhood is defined as:

$$\gamma_v = \frac{|E(\Gamma_v)|}{\binom{k_v}{2}},$$

[2] These considerations have been initially motivated by the research of Kauffman in [128, 129].

where $|E(\Gamma_v)|$ is the number of edges in Γ_v and k_v is the number of neighbors of v. Therefore, γ_v is the ratio between the number of edges of the neighborhood and the maximum number of edges it can have.

The clustering coefficient γ of a graph G is defined as the average of the clustering values γ_v for all $v \in V$.

Typically, random graphs are characterized by low characteristic path length and low clustering, whilst regular graphs (like lattices or rings) have high values for L and γ. Conversely, small-world graphs are characterized by low L and high γ.

In the following sections we will discuss the impact of the node degree on the behavior of local search for SAT in connection with a phenomenon called *criticality and parallelism* in combinatorial optimization.

5.2 Criticality and Parallelism in Combinatorial Optimization

In this section we report experimental results to support the empirical evidence for the presence in local search for SAT and MAXSAT of a phenomenon called *Criticality and Parallelism*, first discovered in combinatorial optimization problems such as the TSP and NK-spaces [139]. Furthermore, we show that the problem structure affects the behavior of local search, by observing that the connectivity of the graph associated with a SAT instance is a critical parameter.

The phenomenon called *criticality and parallelism* has been observed in the context of local search algorithms applied to combinatorial optimization problems [139, 130, 129], where local search is modified by applying more than one local move in parallel. It has been shown that the effectiveness of these algorithms depends on the parallelism degree τ (number of simultaneous moves): if τ increases, the solution quality[3] also increases up to a maximal point (corresponding to τ_{opt}) at which it starts to decrease. It has also been shown that τ_{opt} is negatively correlated with the *connectivity* among variables of the problem: the higher the connectivity, the lower τ_{opt}.

In [139, 45, 129, 130] some studies on the parallelization of local search algorithms are described. In [139], the authors apply a parallel version of Simulated Annealing to optimization on NK-landscapes [128, 129]. In brief, this approach can be described as follows. Suppose to have a minimization problem on N boolean variables; the search space can be represented as an *energy landscape*: the goal is to find a minimum in this landscape. Every variable x_i is associated to an energy value e_i, which is a function of x_i and other K variables. The objective function of the system (total energy) is $E = \frac{1}{N}\sum_{i=1}^{N} e_i$. A move from state s_1 to state s_2 results in an energy difference $\Delta E = E(s_2) - E(s_1)$. The application of the move operator is, in this case, simply a *flip* of a variable (i.e., $x_i \leftarrow \sim x_i$). The basic algorithm behaves as follows: it randomly

[3] In the following we will use the expression *solution quality* referring to the value of the objective function; in case of a minimization problem, the lower the objective function value, the higher the quality.

selects a variable and flips it; it accepts this move with probability 1 if $\Delta E \leq 0$ and with probability $\exp(-\Delta E/T)$ if $\Delta E > 0$. T is a *temperature* parameter, which controls the annealing schedule: the higher T, the higher the probability to choose a non-improving move. In the parallel version, every variable x_i $(i = 1, 2, \ldots, n)$ has probability p of being selected, that is, at each iteration pn parallel variable flips are tested on average. Hence, the degree of parallelism of search is $\tau = pn$. The authors discover that there is a p_{opt} for which the algorithm finds the solution with the lowest E: higher or lower values of p on average produce higher total energy values.

Since the effect of a variable flip on the objective function value is evaluated as if it was the only one to change, parallel (i.e., simultaneous) flips introduce a kind of noise in the energy evaluation. As observed in [142], the introduction of noise increases the effectiveness of local search, since it helps to escape from local optima. It is worth to note that [142] shows that the quality of solutions found increases as noise increases, up to a critical value above which the performance decreases again. However, differences and similarities between parallel LS and LS with noise have still to be completely discovered and explained.

Analogous results are reached in [129, 130], where yet a different approach is chosen. The COP is, in this case, the optimization of a NK-landscape with variables arranged in a bidimensional lattice; every variable corresponds to a cell in the lattice and K indicates the number of neighboring cells linked to it. The lattice is divided in P non-overlapping patches and a simple local search is applied in parallel to each patch. A variable is flipped if it decreases the energy of the patch to which it belongs. The authors find an optimal number of patches which allows the search to reach the lowest total energy value.

The underlying principle of the last approach is that, in order to optimize systems composed of conflicting elements, it is generally useful dividing the system in subsystems and optimize each of them independently. One of the effects of simultaneous changes is to help the search to avoid local optima, as they introduce a kind of *noise* due to the fact that each subset performs a local move supposing the other subsets do not change. Moreover, the authors claim that the optimal subdivision drives the system in a state such that subsystems coordinate themselves for a global optimization goal.

These works on parallelization of search propose very useful ideas for the improvement of local search for COPs and suggest new directions to understand local search behavior.

To summarize, the parallelization of local search can be achieved in different ways and the most important applied so far are:

- At each iteration, apply a local move on each variable (or a solution component) with probability p. This results in an average parallelism of pn, where n is the number of variables.
- Divide the problem in τ subsystems (which are, in general, not independent) and apply local search to optimize each of them independently.

Algorithm 21 GSAT

```
Input: a set of clauses α, MAX-FLIPS and MAX-TRIES
Output: a satisfying truth assignment of α , if found
for i := 1 to MAX-TRIES do
  T := a randomly generated truth assignment
  for j := 1 to MAX-FLIPS do
    if T satisfies α then
      return T
    end if
    p := a propositional variable such that a change in its truth assignment gives
    the largest increase in the total number of clauses of α that are satisfied by T
    with p reversed
  end for
end for
return "no satisfying assignment found"
```

- At each iteration, temporarily exclude some constraints between variables and apply a local move on the relaxed problem.

5.3 Parallel Local Search for SAT and MAXSAT

In this section we present one possible way of parallelizing local search to tackle SAT and MAXSAT. Then, we show and discuss experimental results.

5.3.1 Parallel GSAT

Since the previous results on criticality and parallelism in combinatorial optimization were obtained by applying a (quite simple) local search characterized by a strong hill climbing tendency, for our experiments on SAT we chose GSAT [200].

GSAT was first introduced in [200] as a greedy local search algorithm to solve SAT problems (see Alg. 21). In its basic version, it starts from a random assignment and looks for a satisfying assignment by moving from one state to another one in its neighborhood (defined as the set of states at Hamming distance equal to 1). Given a current state, the next state is chosen by *flipping* the variable that, if flipped, leads to the greater increment of satisfied clauses.

GSAT has a hill-climbing component because it tries to increase the number of satisfied clauses by moving toward the best neighboring state. Moreover, it is able to escape from some local optima and *plateaus* by using *sideways moves*, i.e., moves from a state to another with the same difference of satisfied clauses. Despite the effectiveness of sideways moves, GSAT can be stuck in small areas of the search space without escaping (i.e., it *stagnates*) and other more recent local search algorithms [115] perform better on SAT instances.

Algorithm 22 Parallel GSAT

```
Input: a set of clauses α, τ, MAXMOVES
Output: a truth assignment of α
T ← initial truth assignment
MOVES ← 0
besterror ← Eval(α, T) {Eval(α, T) returns the number of unsatisfied clauses}
bestsolution ← T
while MOVES < MAXMOVES do
  if T satisfies α then
    return T
  end if
  Divide the set of variables in τ subsets (randomly)
  for all subset X_k of variables (k = 1, 2, ..., τ) do
    p_k ← a propositional variable in X_k such that a change
    in its truth assignment gives the largest decrease in the number of clauses of
    α that are not satisfied by T
  end for
  T ← T with p_1, p_2, ..., p_τ reversed
  MOVES ← MOVES + 1
  error ← Eval(α, T)
  if error < besterror then
    MOVES ← 0
    bestsolution ← T
    besterror ← error
  end if
end while
return bestsolution
```

There is a very easy way to parallelize GSAT: the set of variables is divided in equal subsets; if n is the number of variables, the number of subsets corresponds to the parallelism degree τ and the cardinality of each subset is n/τ. The procedure we obtain (thereinafter referred to as PGSAT, see Alg. 22) behaves as follows: at each iteration, the subset are considered in parallel and the "best" variable for each of them is flipped. Therefore, after an iteration, τ variables have been flipped. This has an effect similar to the introduction of noise, because the possible flips are evaluated supposing that variables belonging to other subsets are not modified.

We would like to stress that the algorithms at hand are implemented sequentially. With "parallel moves" we mean "synchronous moves". Anyhow, these results could be beneficial also for implementations on parallel architectures.

Table 5.1: Results on 3-SAT random instances from SATLIB. Median time (s), median iterations and fraction of solved instances out of 100 are reported. τ_{opt} is in bold. Parameters used: MAX-FLIPS $= 5n$, as in [200]; the algorithm was stopped after 10 minutes.

Instances set	τ	Median Time (ms)	Median Iter.	Solved
uf20, 20-91	**1**	< 0.001	38	1
	2	< 0.001	123	1
	4	0.020	1002	1
	5	0.050	2045	1
uf50, 50-218	1	0.020	536	1
	2	0.020	516	1
	5	0.070	2241	1
	10	4.847	140547	0.8
uf100, 100-430	1	0.261	5114	1
	2	0.019	3553	1
	4	0.29	5434	1
	5	0.22	4160	1
	10	2.674	48094	0.88
uf150, 150-645	1	1.963	26635	1
	2	1.022	13438	1
	3	0.871	11405	1
	5	0.971	12291	1
	6	0.661	7809	1
	10	1.752	21878	0.92
uf200, 200-860	1	26.848	276885	0.97
	2	6.179	62594	1
	4	7.001	68512	0.97
	5	4.977	49289	0.99
	10	4.586	43877	0.91

5.3.2 Random Instances

In this section we present experimental results obtained by the application of PGSAT on random 3-SAT instances[4]; we first treat the case of Uniform Random (UF) 3-SAT instances (retrieved from the SATLIB benchmarks[5]). Then we report results on random *forced* instances[6]. The algorithm has been implemented in C and ran on a Pentium II 233MHz with 64 MB of RAM.

Each set of UF instances is composed of 100 satisfiable instances and is a problem in the threshold region [3, 153, 112] (i.e., $m/n \approx 4.3$, where n and m are respectively the number of variables and clauses). Results are summarized in Table 5.1.

We first can observe that (except for the 200-variables instances) there is an optimum value for τ which enables the algorithm to get the overall best performance

[4]In these experiments the termination condition is given by a maximum amount of computation time.

[5]http://aida.intellektik.informatik.tu-darmstadt.de/SATLIB/.

[6]Random generated instances with at least one satisfying assignment.

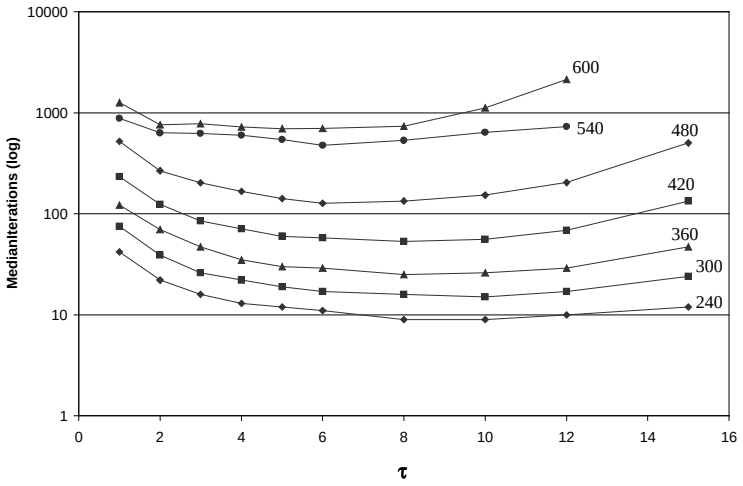

Figure 5.2: 120 variables. Median Iterations (in logarithmic scale) vs. τ (the number of clauses is reported at the end of the curves).

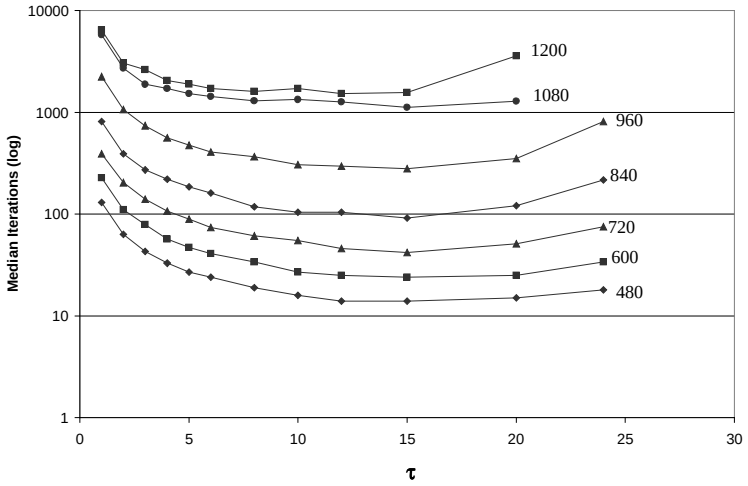

Figure 5.3: 240 variables. Median Iterations (in logarithmic scale) vs. τ (the number of clauses is reported at the end of the curves).

Table 5.2: Connectivity of uniform and forced formulas of 120 variables.

m	**Forced 3-SAT**	**Uniform 3-SAT**
240	0.104292	0.103939
300	0.126333	0.126110
360	0.147974	0.147897
420	0.169243	0.169293
480	0.190125	0.190238
540	0.210404	0.209885
600	0.229820	0.229594
660	0.249062	0.248930
720	0.267728	0.267714

(time, iterations and fraction of solved instances). The phenomenon is clear: in this case, as in [139], there exists an optimum degree of parallelism. There is also evidence of the fact that as the number of variables increases, also τ_{opt} increases.

We also generated two sets of 3-SAT random forced instances ($n = 120, 240$); each set is composed of 1000 instances for each value of m. Results on forced instances are reported in Fig. 5.2 and Fig. 5.3. In these graphics we plotted median iterations vs. parallelism for each set of instances. Median time has the same behavior and the fraction of solved instances is always 100%. First of all, we notice that for every set of instances there is a minimum in the search cost (corresponding to τ_{opt}); moreover, we observe that τ_{opt} slightly decreases as m increases and there is not evidence for a sharp transition of τ_{opt} in the transition region. Therefore, we can conclude that exists a different parameter linked to τ_{opt}, other than the ratio m/n. This topic will be the subject of the next section.

Graph connectivity

As stated in Sec. 5.1, the graph associated with a SAT instance is an undirected graph $G = (V, A)$, where each node $v_i \in V$ corresponds to a variable and edge $(v_i, v_j) \in A$ $(i \neq j)$ if and only if variables v_i and v_j appear in a same clause.

The *average connectivity* of an instance is $q = \frac{1}{n}\sum_{i=1}^{n} q_i$. and the normalization of q is $\overline{q} = \frac{q}{n-1}$.

In Table 5.2, values of $\overline{q}$ for uniform non forced and forced instances are reported: we observe that $\overline{q}$ decreases as n increases. Another observation derives from Fig. 5.4, where we plotted τ_{opt} against $\overline{q}$: the tendency depicted indicates that there is a strong negative correlation between τ_{opt} and $\overline{q}$. To reinforce this conjecture, we generated 3-SAT instances with the same $\overline{q}$ and we ran PGSAT over them. In Table 5.3 the results are reported: note that τ_{opt} is always comprised between 4 and 5.

Summary of first experiments set

The experiments performed on random 3-SAT instances suggest two observations:

Table 5.3: Results on Random 3-SAT formulas (100 instances for each set) with $\overline{q} = 0.2$. τ_{opt} is in bold.

	τ	**Median Time**	**Median Iter.**	**Solved**
$n = 50, m = 82, \overline{q} = 0.2$	1	< 0.001	9	1
	2	< 0.001	6	1
	5	< 0.001	5	1
	10	< 0.001	8	1
$n = 80, m = 220, \overline{q} = 0.2$	1	< 0.001	63	1
	2	< 0.001	39	1
	4	< 0.001	25	1
	5	< 0.001	23	1
	10	< 0.001	29	1
$n = 100, m = 352, \overline{q} = 0.2$	1	0.003	448	1
	2	0.002	287	1
	4	.001	138	1
	5	0.001	156	1
	10	0.002	245	1

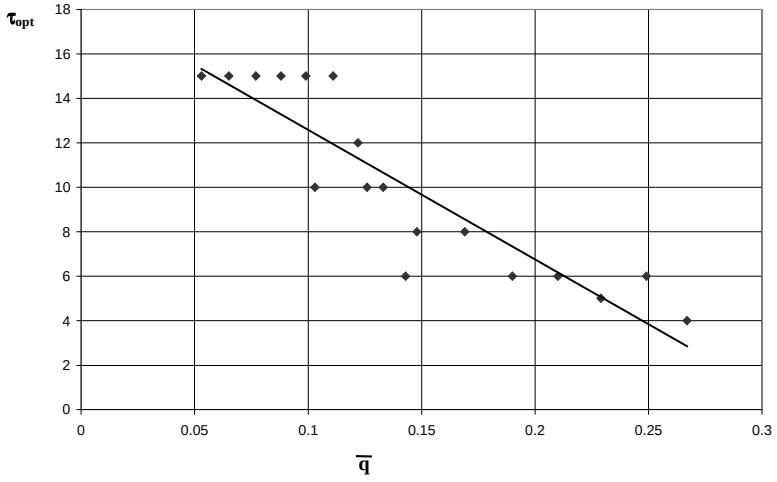

Figure 5.4: τ_{opt} vs. $\overline{q}$

1. There is a value of $\tau_{opt} \in \{1, \ldots, n\}$ that leads PGSAT to achieve the best performance. That is, the average time (and the average number of iterations) of PGSAT(τ_{opt}) is not higher than that of PGSAT(τ), for $\tau \in \{1, \ldots, n\}$.

2. τ_{opt} seems negatively correlated with the average connectivity of the SATgraph: the higher $\overline{q}$, the closer τ_{opt} to 1.

The first observation is not surprising. Indeed, since GSAT could stagnate in a small region of the search space, applying more than one flip in parallel might help the search to escape from that region. At the other extreme, if $\tau = n$ the search does not proceed, but it just oscillates. Therefore, it is likely that a value τ_{opt} such that $1 < \tau_{opt} < n$ enables the algorithm to perform better than with any other value of τ. Nevertheless, what makes the results very interesting is the second observation: the value of τ_{opt} seems to be linked with a structural property of the instance (the average connectivity $\overline{q}$), apparently independently of other instance parameters, such as the ratio m/n. Moreover, the supposed correlation between τ_{opt} and $\overline{q}$ perfectly fits in the previous results on criticality and parallelism in combinatorial optimization. In this case, the intuition behind the phenomenon is that when the relations between variables are loose (i.e., $\overline{q}$ is low), a variable flip affects a relatively small number of other variables, thus the system can be subdivided into several nearly independent subsets. On the contrary, tight relations produce a large network of dependencies among the variables and thus they are more sensitive to single flips.

To test whether the phenomenon is just a coincidence on satisfiable SAT instances and if it is general in the context of SAT-based problems, we also performed experiments on an optimization problem strongly related with SAT: MAXSAT.

5.3.3 Results on MAXSAT

We applied PGSAT also to MAXSAT instances, to test whether the connection between connectivity and τ_{opt} is influenced by the property of formulas to be satisfiable. We discovered that the phenomenon appears with the same characteristics also in unsatisfiable formulas. In this case, the evaluation of the algorithm is given in terms of the error returned after the termination condition is met. The error is defined as the number of unsatisfied clauses corresponding to the returned assignment. From now on, the termination condition is given by a maximum number of moves without improvement. Moreover, since we consider an average error, the algorithm is run several times on the same instance and, to make results independent of a particularly (un)favorable subdivision of variables, the composition of subsets it is reconfigured at each iteration.

We tested PGSAT on random generated unsatisfiable instances with three literals per clause (3-SAT), retrieved from SATLIB. We considered nine sets, from 50 to 250 variables and each set is composed of ten instances with a ratio between clauses (m) and variables (n) approximately equal to 4.3 (the *critical region* [153]). The number of subsets τ varies from 1 (only the best flip among all the variables is performed) to

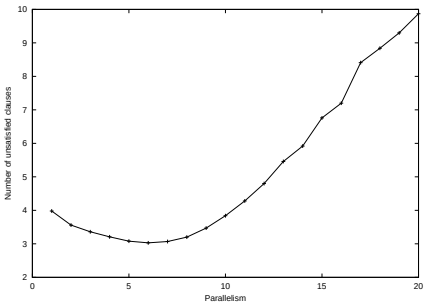

Figure 5.5: Average error vs. τ. Instance with 100 variables and 430 clauses. Results are averaged over 10000 runs.

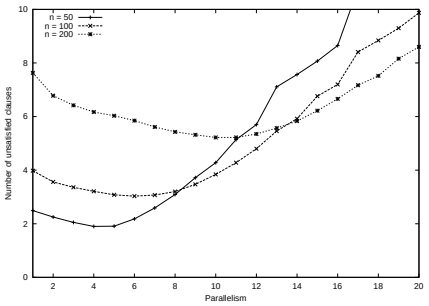

Figure 5.6: Average error vs. τ. Instances with 50,100,200 variables and 218,430,860 clauses respectively. Results are averaged over 10000 runs.

Table 5.4: Median and mean of optimal values of τ for each set of instances.

number of variables	Median τ_{opt}	Mean τ_{opt}
50	4	4.18
75	6	5.9
100	6	6.3
125	7	7.5
150	8	9.4
175	10	10.33
200	11	10.7
225	12	12.36
250	12	12.6

Table 5.5: Average connectivity and its normalized value for typical 3-SAT random generated instances.

number of variables	q	$\overline{q}$
50	19.88	0.4057
75	21.76	0.2941
100	22.88	0.2311
125	23.33	0.1881
150	23.99	0.1610
175	24.00	0.1379
200	24.11	0.1212
225	24.21	0.1881
250	24.34	0.0976

$n/2$; at the beginning of each iteration, a number of τ subsets is randomly generated[7].

The graph in Fig.5.5 reports the average error (number of unsatisfied clauses) as a function of τ for an instance with 100 variables and 430 clauses. This graph shows the typical behavior of the algorithm. Observe that the original algorithm ($\tau = 1$) reaches an average error of 4 and, as τ increases, the error decreases until a minimum at $\tau = \tau_{opt} = 6$; above that value the average error starts to increase.

For all sets of instances, a similar behavior has been noticed. Moreover, as can be observed in Fig.5.6, the higher the number of variables, the higher τ_{opt}. This result is summarized in Table 5.4, where for each set median and mean of τ_{opt} are reported[8].

Table 5.5 shows the typical values of $\overline{q}$ for the instances considered in this analysis.

[7] All subsets have equal cardinality, except for one which contains $\tau + n \; mod \; \tau$ variables.

[8] The average error has been evaluated over 100 runs and then median and mean of τ_{opt} over the 10 instances of each set have been considered.

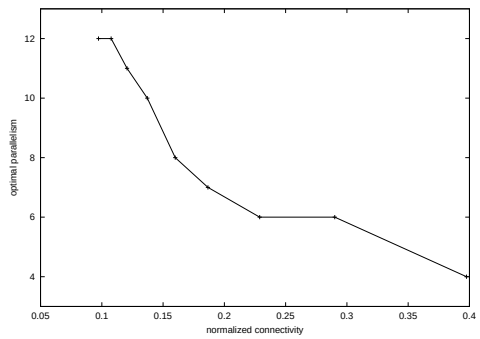

Figure 5.7: Optimal parallelism τ_{opt} vs. normalized average connectivity $\overline{q}$.

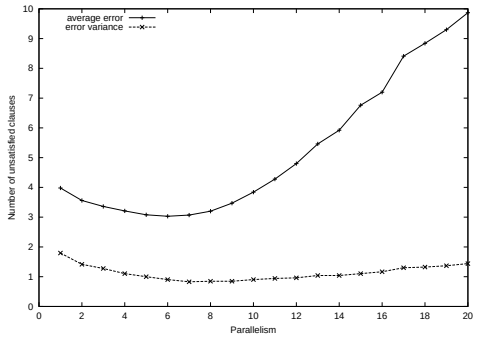

Figure 5.8: Average error and variance vs. τ. Instance with 100 variables. Results are averaged over 10000 runs.

Fig.5.7 shows the graphic combination of Table 5.4 and Table 5.5. We notice that the curve representing τ_{opt} vs. $\overline{q}$ is monotonically non increasing.

A further analysis of the statistics presented earlier in this section, permits to discover an interesting phenomenon. For example, the graph of Fig.5.8 shows the average error and the error variance as a function of τ for an instance with 100 variables. We observe that the minimum error is obtained for a value of τ slightly smaller than the value for which the variance has a minimum. We suppose that near the critical value τ_{opt} the algorithm achieves the highest effectiveness and thus it converges toward a smaller region around the best value it can find.

Concluding, we have shown that an optimal value of parallelism exists also for MAXSAT and that it is negatively correlated with the average connectivity of the instance. These results are in accord with the results discussed in the previous section and they can be seen also as a generalization of them, since they are related to an optimization version of SAT.

So far, we have presented and discussed experiments on PGSAT attacking random SAT/MAXSAT instances. From the observation of the results, we can conjecture that τ_{opt} is strongly correlated with the average node degree of the SATgraph. The testbed used, uniform randomly generated formulas, have however the disadvantage of imposing an average evaluation of the connectivity, supposing negligible the variance of the individual node degree. Nothing has been said about the distribution of the node degree and the hypothesis supporting the use of a global parameter like $\overline{q}$. Moreover, the SAT/MAXSAT instances considered in practice are not random but rather structured and often characterized by non random distributions of node degree. In order to overcome the drawbacks of experimenting on random instances, we investigated in two opposing directions. In the following sections we will first show the results obtained on SAT instance with SATgraph characterized by a fixed node degree (i.e., $q_i = q, i = 1, \ldots, n$). Then, in section 5.5, we will address the question of

whether and under what conditions the phenomenon discovered for random instances is still present in structured instances.

5.4 Constant-degree k-SAT instances

In the previous sections, we have characterized the connectivity of random 3-SAT instances with the average connectivity $\overline{q}$, which seems to be enough informative for uniform generated instances. In random 3-SAT instances, nodes have in general different degree, even though the values are mainly concentrated around the mean[9]. In order to perform experiments with the minimum amount of parameters which can vary, we devised a method to generate random k-SAT instances (for our purposes, $k = 3$) where variables have the same node degree. Therefore, the average connectivity equals the node degree of every node in the graph.

Instead of starting from a SAT formula, expressed as conjunction of clauses with fixed number of literals, we start from a graph with the desired connectivity properties and we use it as a skeleton for generating a SAT formula.

The starting graph is a *lattice graph*, in that each node is connected to a fixed number of lattice neighbors. Formally[10], a *d-lattice graph* is such that any node v is joined to its lattice neighbors u_i and w_i as specified by the following formulas:

$$u_i = [(v - i^\delta + n] \pmod n),$$
$$w_i = (v + i^\delta) \pmod n),$$

where n is the number of nodes, d is the graph dimension, γ is the number of neighbors per node, $i = 1, \ldots, \gamma/2$, $\delta = 1, \ldots, d$, and it is generally assumed that $\gamma \geq 2d$.

For instance, a 2-lattice graph with $gamma = 4$ is depicted in Fig. 5.9. Observe that the graph can be seen as a circular structure with adjunctive links connecting neighbors at distance 2. The graph has dimension 2, as it can be seen as a square grid with wraparound borders.

The lattice graph enjoys the desired property of having nodes with constant degree. Moreover, the node degree is a free parameter of the graph.

Once obtained the graph with the given connectivity, it is necessary to assign variables to nodes and to generate the clauses of the formula. The first step can be done very easily by assigning variables in order: variable x_i is assigned to node i, for $i = 1, \ldots, n$. The generation of clauses, i.e., of a formula that can be mapped into the given lattice graph, is a bit more complex. First of all, we remind that a SATgraph corresponds to a set of SAT instances. Therefore, it is important to define a given structure for the formula. There are quite a few choices at this point:

- Variable clause length
- Fixed clause length k. Which value for k?

[9] This topic will be further discussed in Sec. 5.5.

[10] The definition is taken from [234].

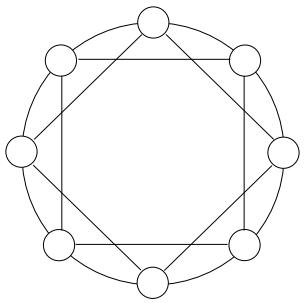

Figure 5.9: 2-lattice graph with $\gamma = 4$.

- Forced instances
- Controlled clauses/variables ratio (m/n).
- Which distribution for clauses and literals? Uniform?
- Satisfiable, unsatisfiable, both.
- ...

Our choice is to follow the usual experimental settings for random generated SAT instances: 3-SAT formulas with controlled ratio m/n. When needed, to obtain sat/unsat formulas we have filtered the generated instances with a complete algorithm, instead of generating forced formulas (which usually suffers of an anomaly, as they have usually a higher number of solutions than the corresponding non-forced ones).

In the following we describe the algorithm to generate 3-SAT instances with given ratio m/n on a lattice graph (thereafter called *lattice-3-SAT* instances). The generalization of the algorithm to k-SAT instances is straightforward. The high level algorithm is described in Alg. 23. The algorithm is structured in two phases. In the first phase a minimal set of clauses is generated to obtain a formula that can be represented by the given lattice graph. In the second phase, the required number of clauses is generated by adding clauses randomly chosen from the first set and by randomly changing the sign of literals.

In the first phase, clauses of three literals are constructed, by taking in turn each variable as a *pivot* and adding two subsequent variables (see Fig. 5.10,5.11). In order to avoid repetitions of clauses, for every variable x_i only subsequent variables $x_j, j > i$ (modulo n) are considered. Indeed, given the symmetry of the graph, the clauses involving the specular part of neighbors will be generated by using those neighbors as pivot (see Fig. 5.12,5.13).

A complete example of a lattice-3-SAT instance with $n = 6$, $m = 12$ and $\gamma = 4$ is the following:

Algorithm 23 Generation of a 3-SAT instance on a lattice graph

INPUT: n, m, γ {γ is the number of neighbors}
OUTPUT: 3-SAT formula $\Phi = \{C_1, \ldots, C_m\}$ with n variable and m clauses associated to a lattice graph with n nodes with γ neighbors each.

Build a lattice graph $G(n, \gamma)$ (on a circle) with n nodes with γ neighbors each;
Assign variables (clockwise) to nodes;
$\Phi \leftarrow \emptyset$
for $i = 1$ to $n - 1$ **do**
  The neighbors of x_i are $\mathcal{N}^+ = \{x_{i+1}, \ldots, x_{i+\gamma/2}\}$ (mod n) and $\mathcal{N}^- = \{x_{i-1}, \ldots, x_{i-\gamma/2}\}$ (mod n);
  for each pair x_j, x_{j+1} in $\mathcal{N}^+$ **do**
    Construct the clause $C = x_i \vee x_j \vee x_{j+1}$
    Negate each variable in C with probability 0.5;
    $\Phi \leftarrow \Phi \cup C$
  end for
end for
{Now the number of clauses is $|\Phi| = n(\gamma/2 - 1)$ }
while $|\Phi| < m$ **do**
  repeat
    Pick randomly a clause C' in Φ;
    Negate each variable in C' with probability 0.5;
  until a new clause C' is generated
  $\Phi \leftarrow \Phi \cup C'$
end while

$\Phi = \{(\neg x_1 \vee x_2 \vee x_3), (x_2 \vee \neg x_3 \vee x_4), (\neg x_3 \vee x_4 \vee x_5), (\neg x_4 \vee \neg x_5 \vee 6), (x_5 \vee x_6 \vee x_1), (x_6 \vee x_1 \vee x_2), (\neg x_5 \vee x_6 \vee \neg x_1), (x_3 \vee x_4 \vee \neg x_5), (\neg x_5 \vee x_6 \vee x_1), (x_2 \vee x_3 \vee \neg x_4), (x_6 \vee \neg x_1 \vee x_2), (\neg x_2 \vee x_3 \vee x_4)\}$

5.4.1 Experimental results

We have tested PGSAT on a benchmark composed of constant degree 3-SAT instances. The benchmark is composed of six sets of thirty instances. Each set contains instances with the same number of variables ($n = 20, 50, 100, 200, 400, 600$) with varying ratio between clauses and variables number ($m/n = 3, 4, 5$). For each instance type we randomly generated ten instances.

We run PGSAT 1000 times for every instance. When the ratio m/n is in the proximity of the critical value 4.3 (solvability threshold [3]) or above it, the instances are likely to be not satisfiable. Therefore, in order to keep a uniform evaluation, the comparison among different parallelism degrees is based on the number of unsatisfied clauses returned at the end of the execution.

Results in are plotted in Fig. 5.14, 5.15, 5.16 and 5.17. On the left we reported

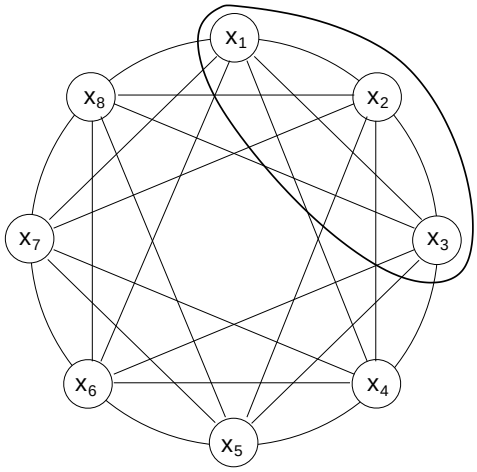

Figure 5.10: Construction of the first clause involving variable x_1.

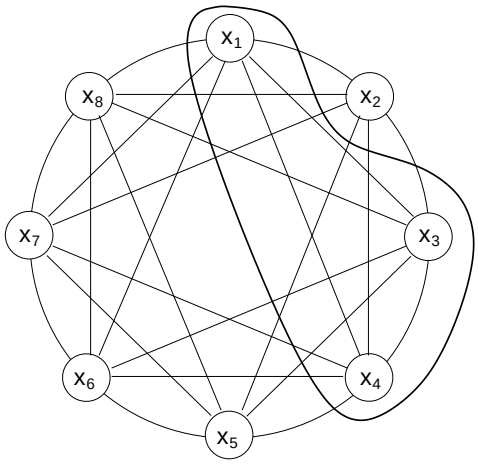

Figure 5.11: Construction of the second clause involving variable x_1.

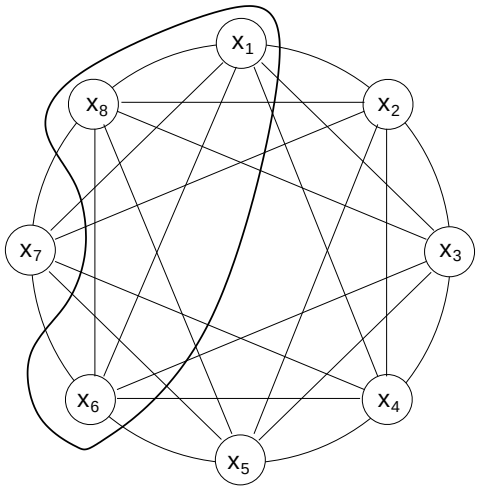

Figure 5.12: Construction of the third clause involving variable x_1. The pivot is variable x_6.

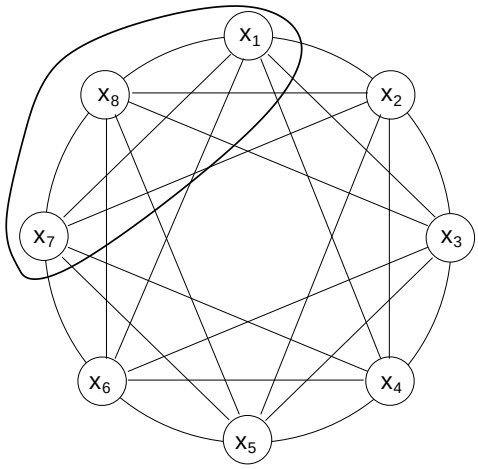

Figure 5.13: Construction of the fourth clause involving variable x_1. The pivot is variable x_7.

the average error against the parallelism, while on the right the rank of parallelism is plotted against the parallelism itself. The figures on the right part are obtained by ranking the values of τ depending on the average error. The lower the rank, the

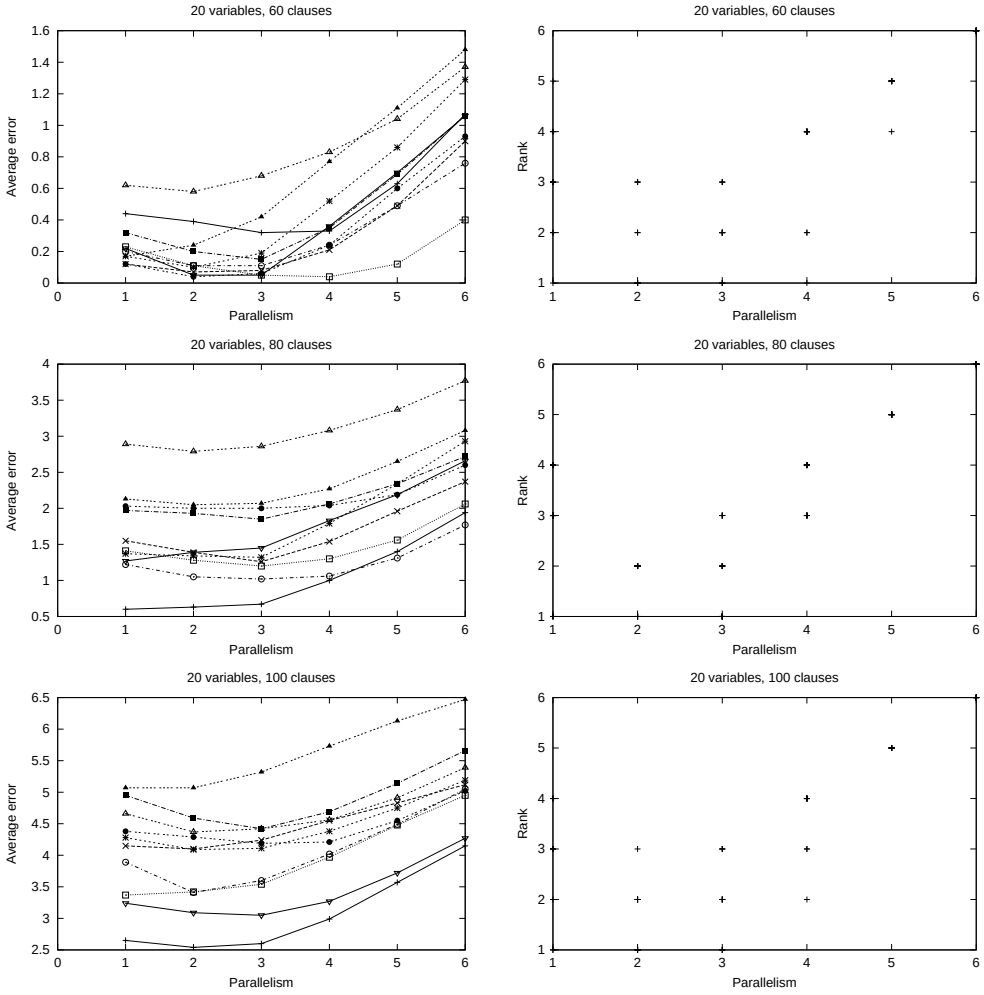

Figure 5.14: Left: Parallelism vs. Average error (number of unsatisfied clauses). Right: Parallelism vs. rank (the lower, the better).

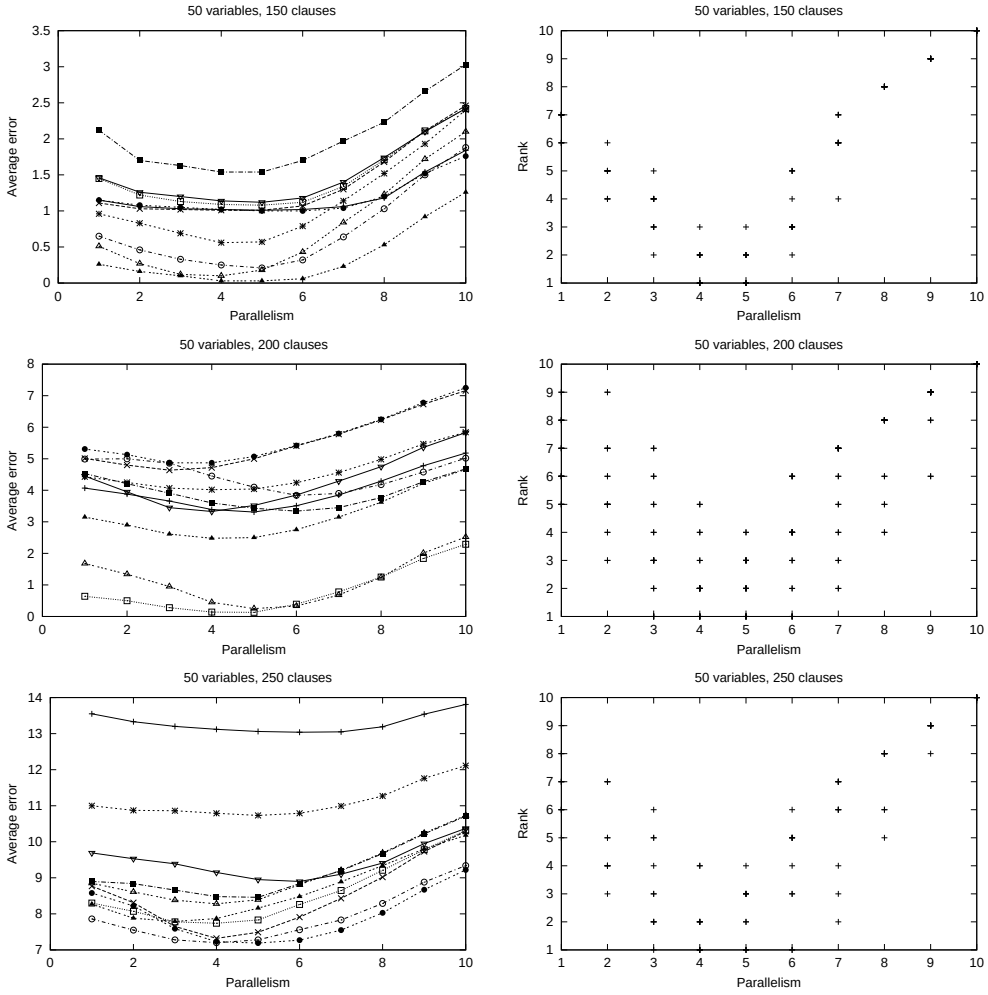

Figure 5.15: Left: Parallelism vs. Average error (number of unsatisfied clauses). Right: Parallelism vs. rank (the lower, the better).

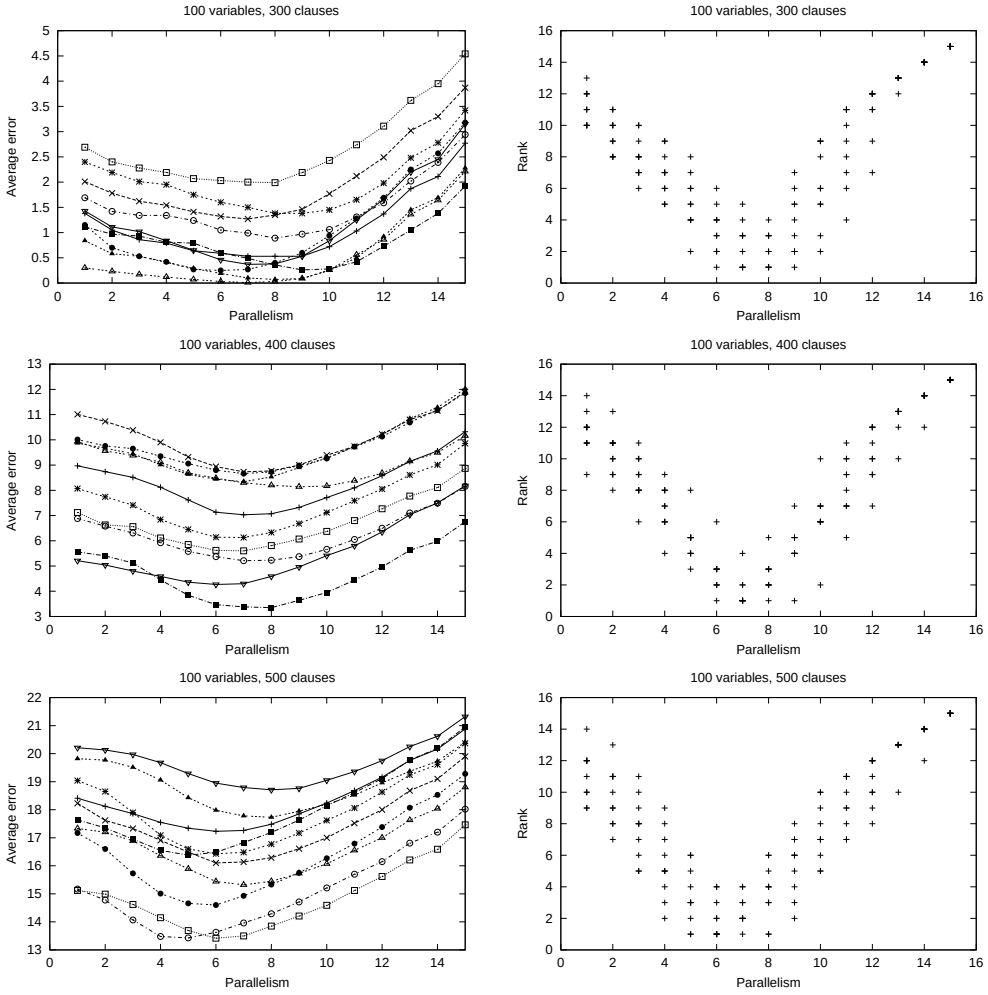

Figure 5.16: Left: Parallelism vs. Average error (number of unsatisfied clauses). Right: Parallelism vs. rank (the lower, the better).

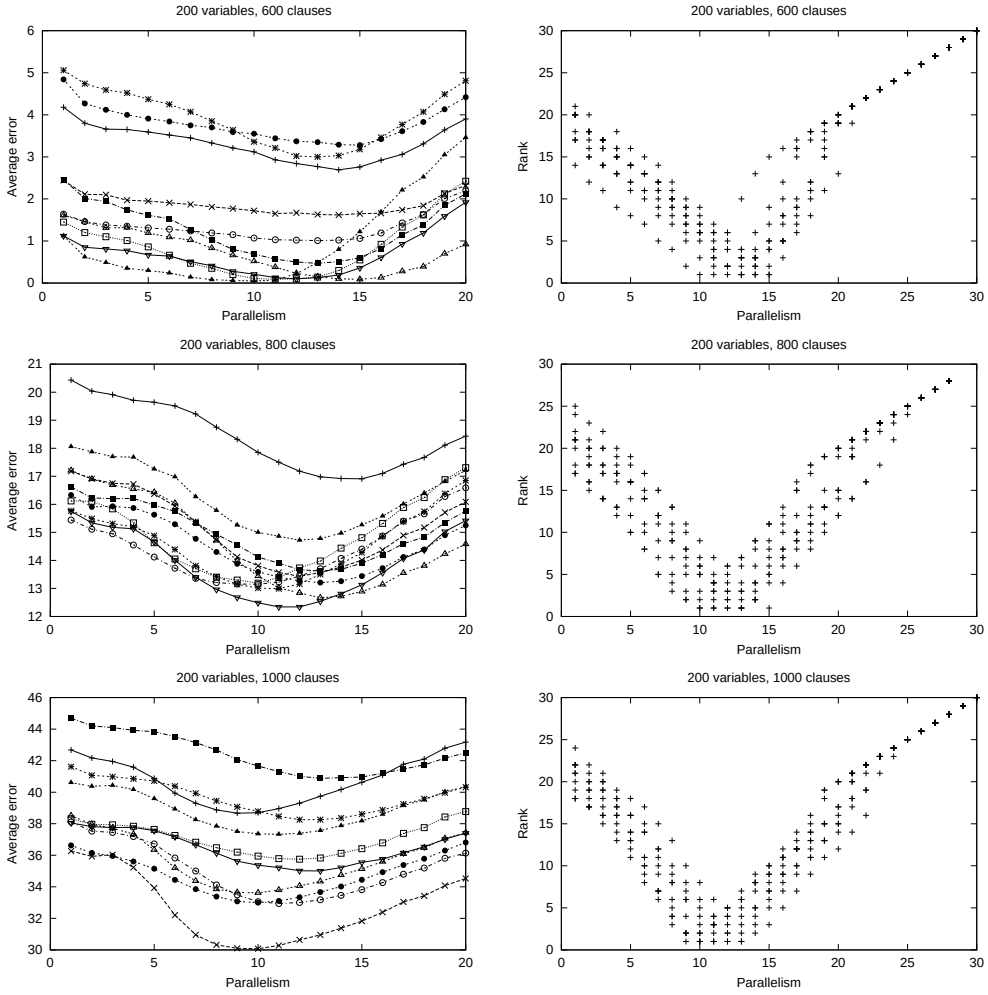

Figure 5.17: Left: Parallelism vs. Average error (number of unsatisfied clauses). Right: Parallelism vs. rank (the lower, the better).

better.

We observe two main points. First, even for instances with the same number of variables and clauses, the optimal parallelism is not exactly the same. Indeed, we can observe that τ_{opt} is confined in a range of values. This fact can be attributed to the variance among random generated instances. Furthermore, in some preliminary measurement, we observed that those differences may be explained by the correlation of the search landscape.

A second important observation is that the range of optimality is practically the same independently of the ratio m/n. This second observation supports the conjecture that the optimal parallelism mainly depends on the instance connectivity and it is independent of other parameters, such as the ratio m/n.

5.5 Structured Instances

In this section we address the question of whether and how parallel local search exhibits the *criticality and parallelism* phenomenon when performed on structured instances. First, we experimentally show that also for structured instances there exists an optimal value of parallelism which enables the algorithm to reach the optimal performance. Second, by analyzing the frequency of node degree of the graphs associated with the SAT instance, we observe that an asymmetric and not regular distribution strongly affects the algorithm performance with respect to τ.

5.5.1 Re-evaluating PGSAT on random instances

We analyzed the behavior of parallel PGSAT on random instances by using as termination condition a maximum number of moves without improvement[11]. As we will see, the results are qualitatively the same as those with obtained within a time limit (Sec. 5.3.2).

The variables are divided in τ subsets at random, before each iteration. All subsets have equal cardinality (n/τ), except for one which contains $n/\tau + n \ mod \ \tau$ variables. Results, averaged over 1000 trials, are shown in Fig.5.18. We can observe that for every value of n the average (resp. median) number of unsatisfied clauses returned by the algorithm has a minimum corresponding to a $\tau_{opt}(n)$. If we consider the average, the results are: $\tau_{opt}(20) = 1$, $\tau_{opt}(50) = 5$, $\tau_{opt}(100) = 7$, $\tau_{opt}(150) = 11$, $\tau_{opt}(200) = 11$, $\tau_{opt}(250) = 12$. For the success rate (i.e., the number of solved instances) results are analogous. We can note that the correspondence between the minimum error and the maximum success rate is fairly good, though it seems that the maximal success rate is reached just before reaching the minimum error. An important point to consider is that, at least in the context of random 3-SAT instances, both for maximal success rate and minimal error, the following relation holds: $n_1 \geq n_2 \Rightarrow \tau_{opt}(n_1) \geq \tau_{opt}(n_2)$. As previously noted, this relation derives from the fact that τ_{opt} is negatively correlated with q and it is not just an effect of size scaling. Indeed, at

[11] This cutoff value has been set to n.

fixed ratio between clauses and variables, q decreases as n increases. These results are in accordance with previous ones discussed in Section 5.3.2.

Connectivity distribution

In order to compare the node degree distribution between instances, we consider the frequency of node degree $Freq(j)$ = 'frequency of a node connected to exactly j nodes' and the cumulative frequency $CumFreq(j)$ = 'frequency of a node connected to not more than j nodes'. Fig.5.19 shows the cumulative frequency vs. the normalized node degree for random 3-SAT instances retrieved from SATLIB [117]. The instances belong to the threshold region (clauses/variables ≈ 4.3) and are satisfiable. Note that the curves are quite regular and, as the number of variables increases, they converge to a step function located at the average node degree. We can assume that the graph corresponding to a random 3-SAT instance is a random graph $G_{n,p}$ [164], where n is the number of nodes and p is the probability that any pair of nodes are connected. In fact, uniform random 3-SAT instances of SATLIB are generated by randomly selecting, for each clause, three literals among the complete set of $2n$ literals[12]. Thus, every pair of variables has the same probability to belong to a same clause. Note that the graph G associated with a SAT instance does not take into account the number of clauses involving the same pair of variables. Indeed, an edge just represents the fact that the variables it connects belong to at least one clause [13]. For random graphs like $G_{n,p}$ the distribution probability of connectivity follows a Poisson distribution, i.e.,

$$\text{prob}\{\text{a node is connected exactly to other } j \text{ nodes}\} = e^{-\lambda}\lambda^j/j!$$

where the parameter λ is the expected node degree, therefore, in our case, $\lambda = (n-1)\overline{q} = q$. For instance, in Fig.5.20 the frequency of a 3-SAT instance with 100 variables is plotted.

Experiments on Structured Instances

In this section we first analyze the node degree distribution of graphs associated with structured SAT instances. Then we show and discuss results of PGSAT, run with different values of τ, on two representative structured SAT instances.

Structured instances are characterized by the presence of some regularity in their components, for example graph problems based on ring or lattice topology, SAT problems obtained by encoding logic problems (circuit testing, inductive inference, planning, etc.). In SAT problems generated by an encoding procedure from other problems there are two sources of structure: the inherent structural properties of the original problem and the relations among variables introduced by the encoding procedure. As noted in [18], the inherent structure of the problem might be partially lost in the encoded formulation. However, independently of the origins of structure, the

[12]The description of the generation procedure is available at `www.satlib.org`.

[13]The relation between random graphs and k-SAT instances as a function of the number of variables, clauses and literals per clause is subject of current study.

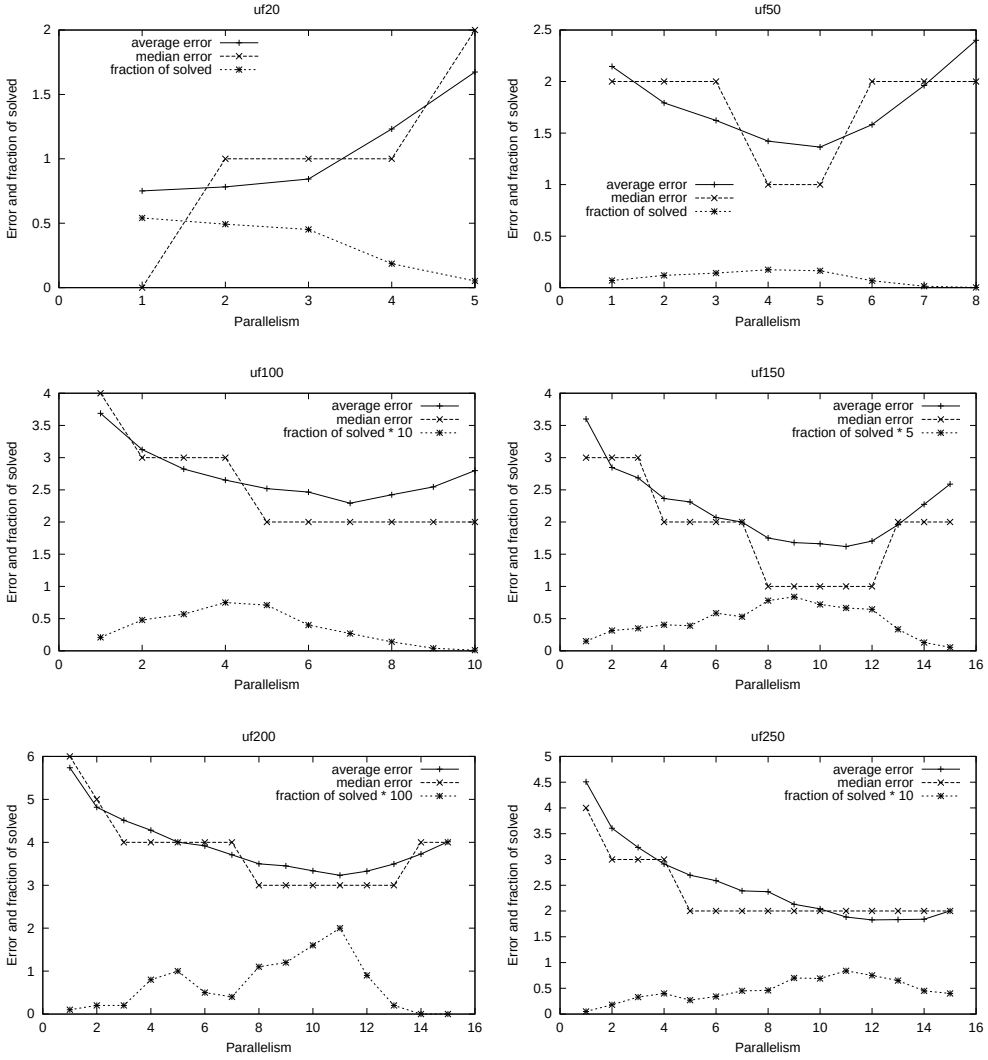

Figure 5.18: Average and median error (number of unsatisfied clauses) and fraction of solved instances (rescaled when necessary) vs. parallelism (τ) for random 3-SAT instances with $20, 50, 100, 150, 200$ and 250 variables. Results are averaged over 1000 trials. We can observe that for every value of n the average (resp. median) number of unsatisfied clauses returned by the algorithm has a minimum corresponding to a $\tau_{opt}(n)$. If we consider the average, the results are: $\tau_{opt}(20) = 1$, $\tau_{opt}(50) = 5$, $\tau_{opt}(100) = 7$, $\tau_{opt}(150) = 11$, $\tau_{opt}(200) = 11$, $\tau_{opt}(250) = 12$.

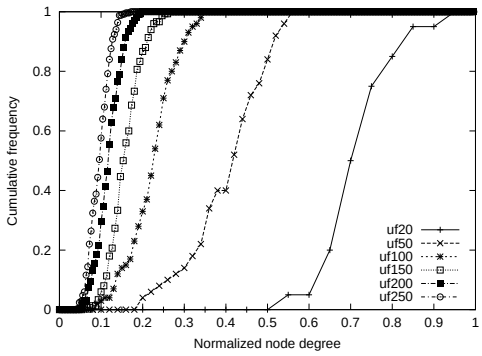

Figure 5.19: Cumulative frequency vs. normalized node degree for Uniform Random 3-SAT instances in the threshold region.

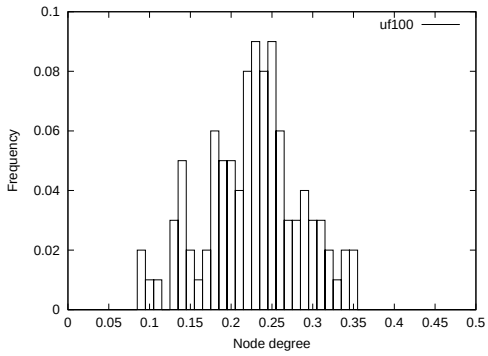

Figure 5.20: Frequency against normalized connectivity in random 3-SAT instance uf100-01. The instance has an average normalized connectivity of 0.229.

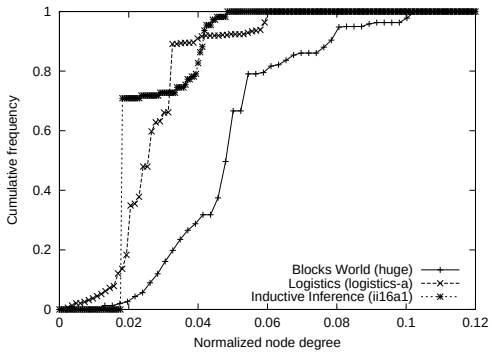

Figure 5.21: Cumulative frequency vs. normalized node degree in structured instances.

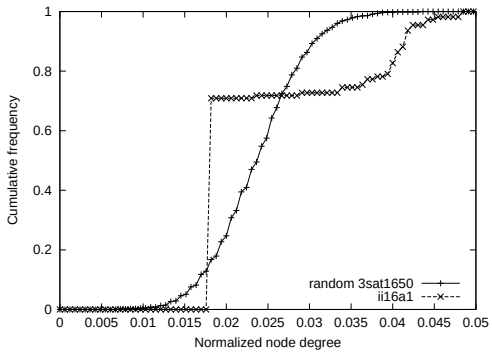

Figure 5.22: Cumulative frequency vs. normalized node degree in a structured instance (*ii16a1*) and in a random 3-SAT instance *3sat1650* of same size ($n = 1650$) and connectivity ($\overline{q} \approx 0.0239$).

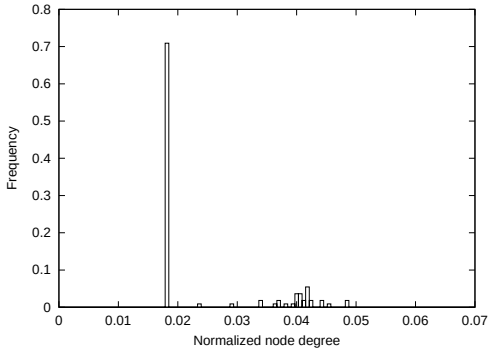

Figure 5.23: Frequency vs. normalized node degree in the instance *ii16a1*.

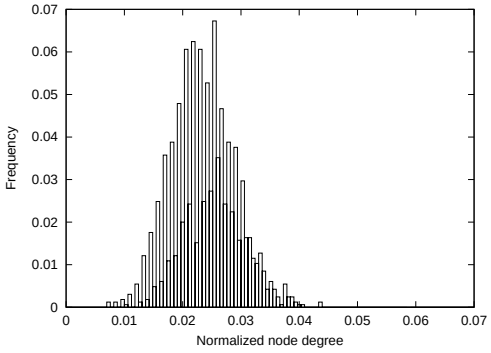

Figure 5.24: Frequency vs. normalized node degree in the instance *3sat1650*.

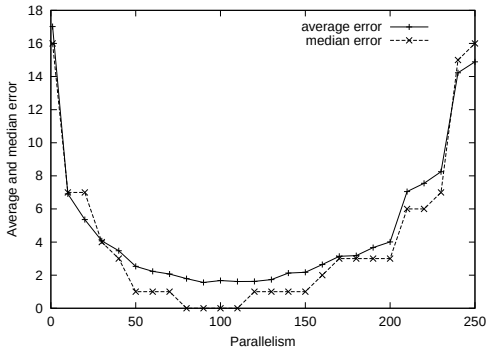

Figure 5.25: Average and median error against τ for PGSAT on the instance *ii16a1*.

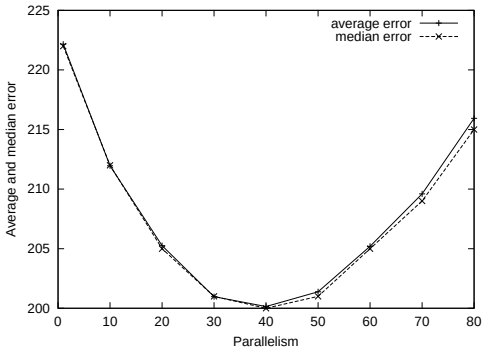

Figure 5.26: Average and median error against τ for PGSAT on the instance *3sat1650*.

SAT instances we consider clearly show node degree distributions very different with respect to random instances. Fig.5.21 shows the curve of cumulative frequency for structured SAT instances taken from SATLIB, produced by encoding a Blocks World Planning Problem (*huge*), a Logistics Planning Problem (*logistics-a*) and Inductive Inference (*ii16a1*). The plotted curves show apparent differences with those of random SAT problems. They are not as regular as random ones and they have gaps and plateaus, especially in the uppermost part of the curve. Structured instances thus have a more spread and non-uniform connectivity distribution.

The instance *ii16a1* is the most peculiar and differs the most from that of random instances. It has 1650 variables and a normalized average connectivity $\overline{q}_{ii16a1} = 0.0239$. Its cumulative frequency is shown in Fig.5.22, along with the cumulative frequency of a random 3-SAT instance of the same size and normalized connectivity (instance *3sat1650*). Fig.5.23 and Fig.5.24 plot the respective frequency of node degree. We can note that the node degree frequency of the structured instance is highly asymmetric and has a peak close to 0.018, corresponding to the large gap in the cumulative frequency. Therefore, *ii16a1* has a very large number of nodes with lower connectivity than the average. Conversely, the node degree frequency of the random instance is regular (it approximately fits the Poisson distribution with high mean) and the highest peak in frequency is very close to the mean.

Fig.5.25 and Fig.5.26 show the average and median error (number of unsatisfied clauses) respectively on *ii16a1* and *3sat1650* for PGSAT with different values of τ. Results are averaged over 500 trials. We first observe that also for the structured instance there exists an optimal value of τ. Nevertheless, despite the fact that the two instances have the same average connectivity, the optimal parallelism is higher for *ii16a1* than for *3sat1650*. We conjecture that the high number of nodes with low degree present in the instance *ii16a1* is the cause of higher optimal parallelism.

We performed the same kind of experiments on the *logistics-a* instance, by comparing it with a random 3-SAT instance with the same size (828 variables) and normalized average connectivity $\overline{q} = 0.0275$ (instance *3sat828*). Fig.5.27 and Fig.5.28 show the respective node degree frequency. Note that, while the distribution of the instance *3sat828* approximately follows a Poisson distribution, the distribution of *logistics-a* is not regular and has a high peak at normalized node degree 0.0326. The results of PGSAT performance with respect to τ are plotted in Fig.5.29 and Fig.5.30. The results are analogous, but dual, to the previous ones on *ii16a1*. We can observe that also for this structured instance there is a value of τ leading to a minimum average error, but in this case the optimal parallelism for *logistics-a* is lower than that of the random instance. This difference may be explained by observing that the highest peak in *logistics-a* node degree frequency corresponds to a node degree higher than the average value, which characterizes the random instance.

We can conclude this section by considering that the results obtained clearly show that an optimal parallelism value exists also for structured instances. Nevertheless, not surprisingly, this value is strongly affected by the asymmetric frequency of node degree. In particular, we observe that the highest peaks location seem the most relevant characteristic influencing the optimal parallelism.

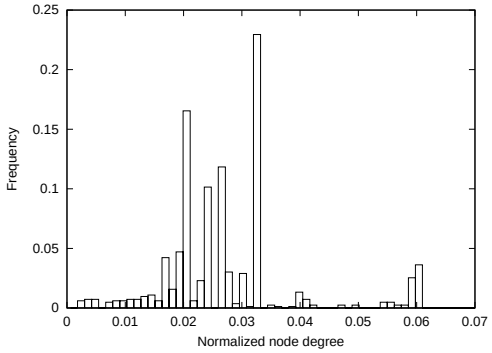

Figure 5.27: Frequency vs. normalized node degree in the instance *logistics-a*.

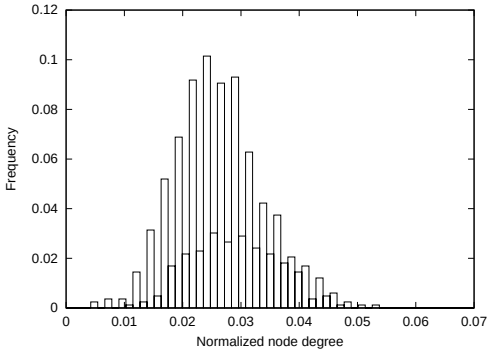

Figure 5.28: Frequency vs. normalized node degree in the instance *3sat828*.

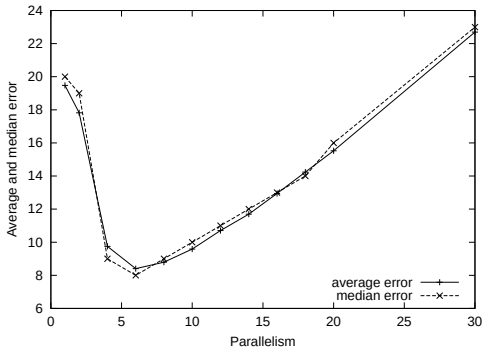

Figure 5.29: Average and median error against τ for PGSAT on the instance *logistics-a*.

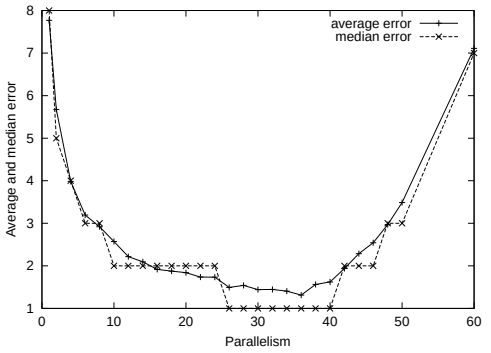

Figure 5.30: Average and median error against τ for PGSAT on the instance *3sat828*.

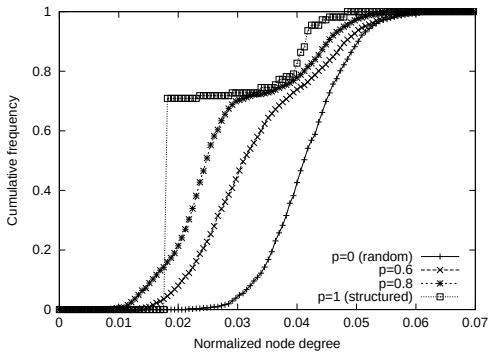

Figure 5.31: Cumulative frequency vs. normalized node degree for instances generated by morphing. $p \in [0,1]$ controls the randomness/structure ratio: the higher p, the higher the amount of structure.

5.5.2 Morphing from random to structure

In Sect.5.3.2 we have seen that the average connectivity affects the optimal parallelism for random instances. The results of the previous section show that for structured instances, characterized by asymmetric frequency distribution, the optimal parallelism is mostly affected by the highest peaks of node degree distribution. To investigate more deeply this difference, we generated instances with a controlled amount of *structure*, by means of a technique called *morphing* [82]. This methods enables to generate instances gradually morphing from source to destination instance by varying a parameter $p \in [0,1]$. The lower p used to generate an instance, the more similar to the source. To be applied on SAT problems, the method needs instances with the same number of variables (n) and clauses (m). A new SAT instance is generated by selecting each of the m clauses either from the source or the destination. The clause is chosen from the destination instance with probability p. We generated a satisfiable random 3-SAT instance with 1650 variables and 19368 clauses (*3sat1650_large*), the same number as *ii16a1*. With $p = 1$ we obtain *ii16a1* and with $p = 0$ *3sat1650_large*. Since p controls the number of clauses belonging to the structured instance *ii16a1*, it also measures the amount of structure in the generated instance. Fig.5.31 shows the node degree cumulative frequency of source (random), destination (structure) and instances generated by the morphing method. The node degree frequency of the considered instances is plotted in Fig.5.32. The results of 500 trials of PGSAT for different values of parallelism are shown in Fig.5.33. Observe that the optimal parallelism increases with p, therefore we can conjecture that the more similar the node degree frequency distributions of two instances, the closer should be their optimal parallelism degrees.

We can conclude by asserting two points. First, regardless of the instance type, the average connectivity is a rough, yet indicative, parameter for the optimal parallelism.

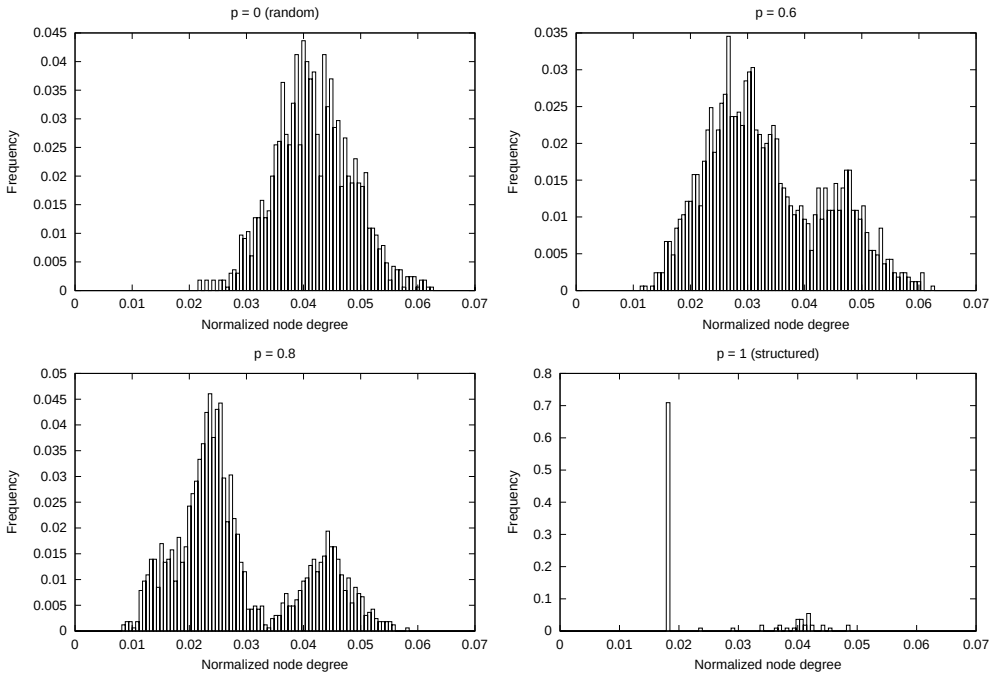

Figure 5.32: Frequency against normalized node degree for instances generated by morphing from a random instance (*3sat1650_large*) to a structured one (*ii16a1*).

Second, we have again found experimental evidence that in structured instances the highest peaks in the node degree distribution have a strong impact on the optimal parallelism value. Of course, we can not claim a statistical proof, that needs an exhaustive and deeper experimental analysis.

5.6 Tuning of τ

Beside the interest about the phenomenon itself, the parallelization can be used to improve local search, namely GSAT. The intuition behind the effectiveness of parallel local moves is that multi-flip moves help to escape from local minima and reach faster search space regions with low objective value. This behavior can be presumably explained by the fact that, at τ_{opt}, the distance between a state and its successor, reached after a multi-flip move, enables local search to achieve the optimal trade-off between intensification and diversification[14]. This conjecture is supported by experimental results in [130, 129] and by preliminary tests we performed on large MAXSAT

[14]We refer to the informal definition of *intensification* as the greedy exploitation of search history and *diversification* as the exploration of the search space.

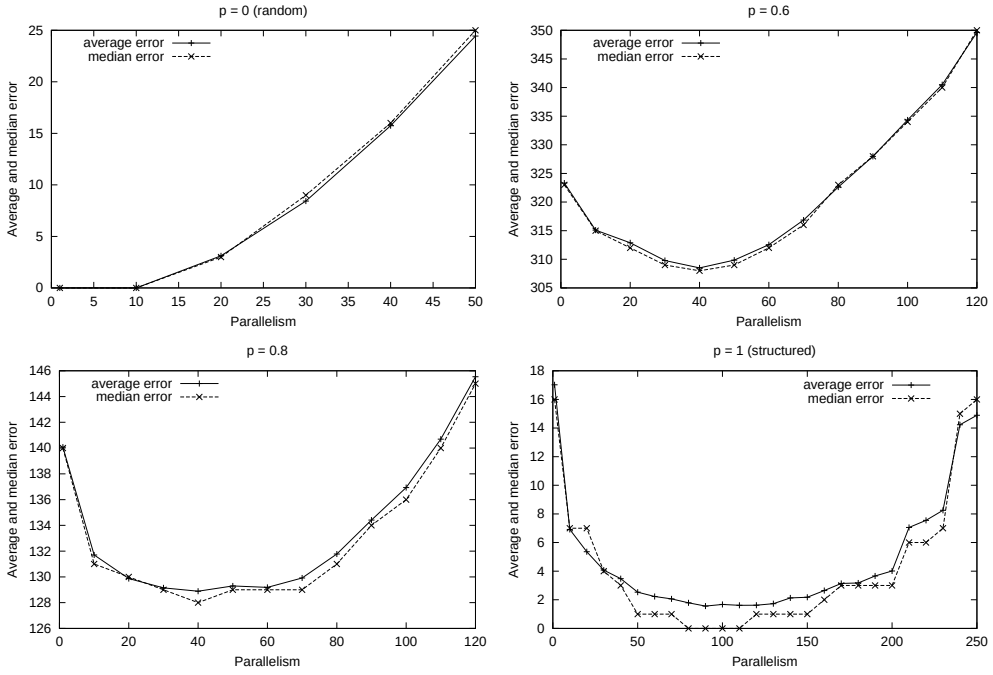

Figure 5.33: Average and median error of PGSAT run with different values of τ on instances generated by morphing from a random instance (*3sat1650_large*) to a structured one (*ii16a1*).

instances. For instance, consider Fig.5.34 where the initial iterations of PGSAT with single and multi-flips are compared. The algorithm is run on an unsatisfiable 3-SAT instance with 1000 variables and 10000 clauses, with $\tau = 1$ and $\tau = 50$. As we can note, the multi-flip version achieves larger solution improvements than the single flip one in the same amount of time.

Parallel flips may prevent the search to get trapped in local minima in similar way to random noise when added to local search [142, 199, 32]. The relations between parallel moves and noise are beyond the scope of this thesis. However, it is important to observe that the optimal number of parallel flips and the optimal noise level are related to properties of the search landscape, like the ruggedness, the distribution of local minima and the size of their basins of attraction [144, 122]. Since up to now theoretical results are still missing to compute the optimal parallelism value for a given instance, we need an empirical method to tune τ as close as possible to τ_{opt}. The easiest way is to run PGSAT for $\tau = 1, 2, \ldots$, using a small cutoff value and stop as soon as a minimum average error has been found. Nevertheless, this method is, in general, computationally expensive, as it might require several runs before finding τ_{opt}. Indeed, since the algorithm is stochastic, several runs have to be performed before achieving significant statistics.

We developed another method which requires just one short run for every candidate for τ_{opt}. This method is based on the observation of solution improvements PGSAT achieves during one execution. From a mathematical standpoint, our objective is to find the minimum of a curve $\epsilon(\tau)$ representing the average error ϵ with respect to τ. From the observation of experimental data, we can assume the curve convex and with one minimum. Therefore, τ_{opt} is given by the abscissa τ such that the derivative of the curve is zero, i.e., $\frac{d\epsilon(\tau)}{d\tau}|_{\tau=\tau_{opt}} = 0$. Since the values of τ are integer, an approximation of the derivative is $\Delta\epsilon(\tau) = \epsilon(\tau) - \epsilon(\tau - 1)$. Thus, the objective is to find $\overline{\tau}$ such that $\epsilon(\overline{\tau} - 1) - \epsilon(\overline{\tau}) \approx 0$. Fig.5.35 shows typical results obtained with short runs of PGSAT on the *logistics-a*[15], for which $\tau_{opt} \approx 6$. For every run length $(1, 10, 20, 50)$ we stored the objective function value (number of unsatisfied clauses) reached by PGSAT with different values of τ; we call this value $\epsilon(\tau)$. For each point (x, y) in the plot, the coordinate y is given by $\epsilon(x - 1) - \epsilon(x)$. For example, let us consider the *1st iteration* curve. After one iteration, the difference of the errors returned by PGSAT with $\tau = 1$ and $\tau = 2$ is 12, i.e., $\epsilon(1) - \epsilon(2) = 12$. All the runs have the same initial state and the algorithm uses the same random number generator seed. We can observe that at the *50th iteration* $\epsilon(\tau - 1) - \epsilon(\tau) \approx 0$ for $\tau = 9$ and the error difference approaches zero at $\tau = 7$. Moreover, we can observe that, except for the plot of the *1st iteration*, the curve decreases approximately linearly up to a point where the slope decreases quite abruptly. This point corresponds, with good approximation, to τ_{opt}. The fact that at the knee of the curve $\Delta\epsilon$ is not zero can be explained by observing that the solution improvements achieved with high parallelism (even $\tau > \tau_{opt}$) at the beginning of the search are higher than those of the remaining part of the search. The results obtained yield to the definition of a simple procedure

[15] We have chosen *logistics-a* just as a representative example of the experiments we made on the considered instances.

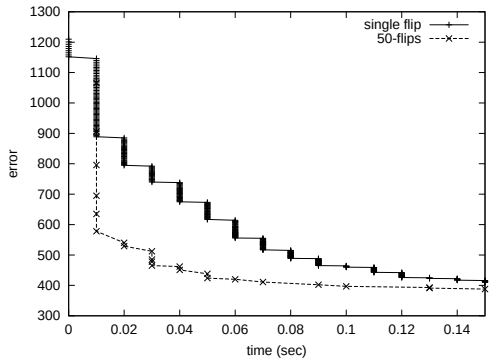

Figure 5.34: Single flip vs. multi-flip GSAT on an unsatisfiable 3-SAT instance with 1000 variables and 10000 clauses.

to tune τ: run PGSAT for a very small number of iterations for different values of τ and set it at the value corresponding to $\epsilon(\tau - 1) - \epsilon(\tau) \approx 0$. Alternatively, we can set τ to the value corresponding to the knee in the curve, thus further reducing the required number of iterations. Up to now, we have determined the near-optimal value of τ by directly observing the plotted data, but the development of an automatic procedure is subject of current work.

5.7 Discussion

The results discussed in the previous sections strongly support the conjecture that the criticality and parallelism phenomenon appears also in local search for SAT/MAXSAT problems.

It has been first shown that for different classes of problems (random and structured SAT, MAXSAT, lattice SAT) there is a value of τ that optimizes the algorithm performance. Furthermore, it has been discovered that, given a SATgraph, its connectivity strongly affect the optimal value of τ_{opt}. In case of SATgraph with constant node degree and random instances, the higher q (resp. $\overline{q}$), the lower τ_{opt}. In case of small differences among instances with the same connectivity, we have conjectured that the fitness landscape correlation correlation provides a justification for differences in τ_{opt}. These results are in accordance with the previous results found in the literature.

Beyond these experiments, the impact of SATgraph structure has been investigated, finding clues for the dependence of τ_{opt} on the frequency of node degree.

Nevertheless, we can not definitely conclude that the phenomenon is exactly the same found in [142], mainly for two reasons. The first reason is that, since experiments on a possible phase transition have not been performed, it might be possible that a phase transition in this case does not exist. It is important to observe that

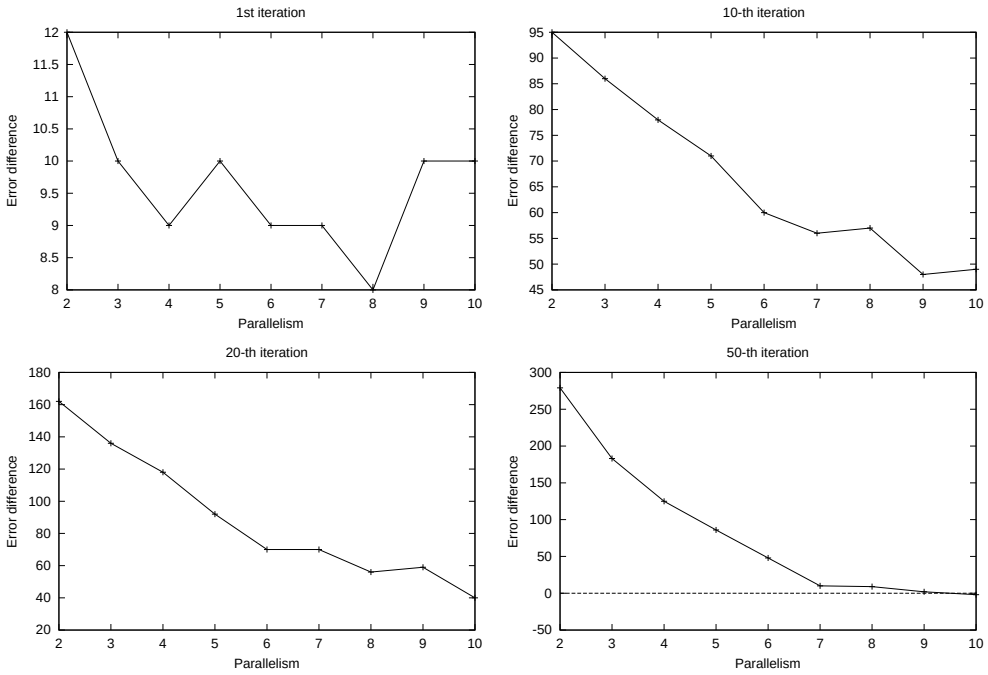

Figure 5.35: Difference of the error (number of unsatisfied clauses) reached after $1, 10, 20, 50$ iterations between successive values of parallelism. For each point (τ, y) in the plot, the coordinate y is given by $\epsilon(\tau - 1) - \epsilon(\tau)$, where $\epsilon(\tau)$ is the number of unsatisfied clauses returned by PGSAT with parallelism τ.

the algorithm applied is more effective than simple hill climbing, therefore the abrupt change in the behavior might be smoothed (as, indeed, is apparent from the plots in the previous sections). The second reason derives from the scientific method application. Even though the results presented strongly suggest (and even encourage) the conjecture, it is still early to assert statistical proof for the phenomenon.

However, the application of these results goes beyond the simple empirical investigation of hill climbing-like procedures. Algorithm other than GSAT might benefit from multi-flips application, for example GWSAT and WalkSAT, more effective on SAT than GSAT itself. There are several open issues to explore, like the exploitation of graph properties to define the subsets of strongly connected variables (instead of dividing variables at random), the investigation of the relation between criticality and parallelism and the introduction of noise in local search, the analytical study of connectivity and optimal parallelism.

Finally, we may suppose that there could be an analogous phenomenon also when complete algorithms are applied. The critical parameter can be, for instance, the randomization degree, or the number of assignments performed at each construction step.

5.8 Related Work

This work is inspired by work of Kauffman et al. on criticality and parallelism [139, 130, 129] and Monte Carlo algorithms for NK-models [45]. Even though not with the same objective, other works deal with multiple flips in local search for SAT and MAXSAT problems. Yagiura and Ibaraki [243] present a computational study of r-flips neighborhoods ($r = 2, 3$). Parkes [169] considers the parallelization of Walk-SAT [142, 199] for 3-SAT instances in the underconstrained region and observes that parallel flips does not degrade the local search, indirectly confirming our results. Indeed, in the underconstrained region the connectivity is low, therefore the optimal parallelism is high. Finally, Strohmaier [207] discusses the implementation of GSAT on a multi-flip neural network and Milano and Roli [150, 149] present a general method for tackling SAT with boolean networks, explicitly considering the possibility of multi-flips local search algorithms.

Chapter 6

Small-World Phenomenon and SAT

Small-world graphs [234, 235] are characterized by the simultaneous presence of two properties: the average number of links connecting any pair of nodes is low and the clustering is high. Social networks defined on the basis of friendship relationships are a typical example of graphs with a small-world topology.

The impact of small-world topology on search problems (e.g., Graph Coloring Problem) has been discussed in [230], where it is shown that many Constraint Satisfaction Problems and Combinatorial Optimization Problems have a small-world topology and the search cost can be characterized by a heavy-tail distribution.

In this chapter we report experimental results concerning the behavior of approximate and complete algorithms applied to SAT instances with a constraint graph characterized by a small-world topology. The primary reason why we consider also a complete algorithm is to show that some structural properties may have impact on very different search algorithms. Moreover, the comparison of complete and approximate algorithms with respect to problem structure enables the successful integration of both the techniques. We will show that the search cost of the complete solver is higher for small-world instances. A similar behavior can be conjectured also for local search algorithms.

6.1 Small-world SAT instances

In order to explore the behavior of search algorithms on small-world SAT instances, we generated a benchmark by morphing between instances constructed on lattice graphs and random instances. The core idea of the morphing procedure is derived from [82], where a method which enables to generate instances gradually morphing from a source to a destination instance. This procedure is also very similar to the one used in [235] to generate small-world graphs by interpolating between lattice graphs

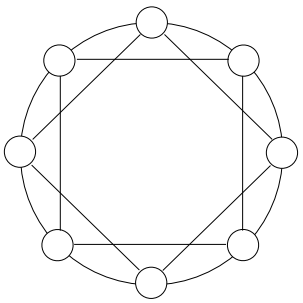

Figure 6.1: Example of lattice graph. Each node has 4 neighboring nodes.

and random graphs.

Lattice 3-SAT instances have been generated on the basis of a lattice graph (see Fig. 6.1), as described in Chap. 5.

The instances composing the benchmark are generated by morphing between a lattice SAT instance and a random one. Each instance is obtained by taking from the lattice SAT instance all the clauses except for a prefixed number which are randomly chosen from a random SAT instance with the same number of variables and clauses. This procedure is indeed very similar to the morphing procedure described in [82], but in this case we control the exact number of clauses taken from the destination instance. In this way it is possible to interpolate from lattice to random with the finest tuning and observe the arising of small-world properties in SAT instances.

In order to have a quantitative measure of the small-world characteristic, we introduce the *proximity ratio* μ [230], defined as the ratio between clustering and characteristic path length, normalized with the same ratio corresponding to a random graph, i.e., $\mu = (C/L)/(C_{rand}/L_{rand})$. In Fig. 6.2 the clustering and the characteristic path length of SAT instances gradually interpolating from lattice to random are plotted (in semi-log scale). We observe that L drops very rapidly with the introduction of clauses from the random instance. Conversely, C maintains a relatively high value for a higher amount of *perturbation*. The instances with low length and high clustering are characterized by the small-world property. This is also indicated by the proximity ratio curve, which approximately assumes its maximum in correspondence of that region.

We generated four sets of instances (respectively with 100, 200, 500 and 800 variables), each obtained by morphing between a lattice 3-SAT and a random 3-SAT with same number of variables and clauses. All the generated instances are satisfiable (unsatisfiable instances have been filtered by means of a complete solver). The ratio between the number of clauses and the number of variables is 3, lower than the so-called critical ratio (which is close to 4.3 for 3-SAT instances). This is due to the structure of lattice SAT instances which turned out to be almost all unsatisfiable at the critical ratio.

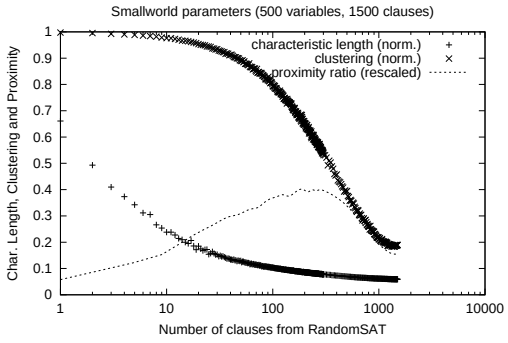

Figure 6.2: Characteristic path length L, clustering C and proximity ratio μ for instances generate by morphing from a lattice SAT instance and a random SAT instance of 500 variables and 1500 clauses. Observe that the maximum of μ approximately corresponds to the instances with maximal discrepancy between L and C.

In the next sections we present the results obtained by solving the generated instances both with a complete solver and local search algorithms.

6.2 Solving with Systematic Search

As a complete solving procedure we chose BerkMin [95], one of the most efficient complete SAT solvers available nowadays. The search cost has been evaluated as the number of variable assignments performed by the algorithm before solving the instance. In Fig. 6.3 the search cost for every set of instances is plotted along with the proximity ratio curve. We clearly observe that, after few perturbations the small-world property appears, as proved by the proximity ratio curve maximum. The scattered points representing the search cost follow approximately the same behavior as the proximity ratio. Therefore, we can attribute a higher search cost to small-world instances. This conjecture is confirmed by the correlation between search cost and proximity ratio, shown in Fig. 6.4: the higher the proximity ratio, the higher the search cost.

6.3 Solving with Local Search

We also solved the same benchmarks with two local search algorithms, namely Walk-SAT[1] [199] and GSAT [200]. Results are shown in Fig. 6.5 and Fig. 6.6 respectively. In each plot we reported the number of successes (out of 1000 runs) and the proximity

[1]The variable to flip has been chosen with the following heuristic: flip the variable with the highest GSAT-like score if it is not the most recently flipped, otherwise choose at random.

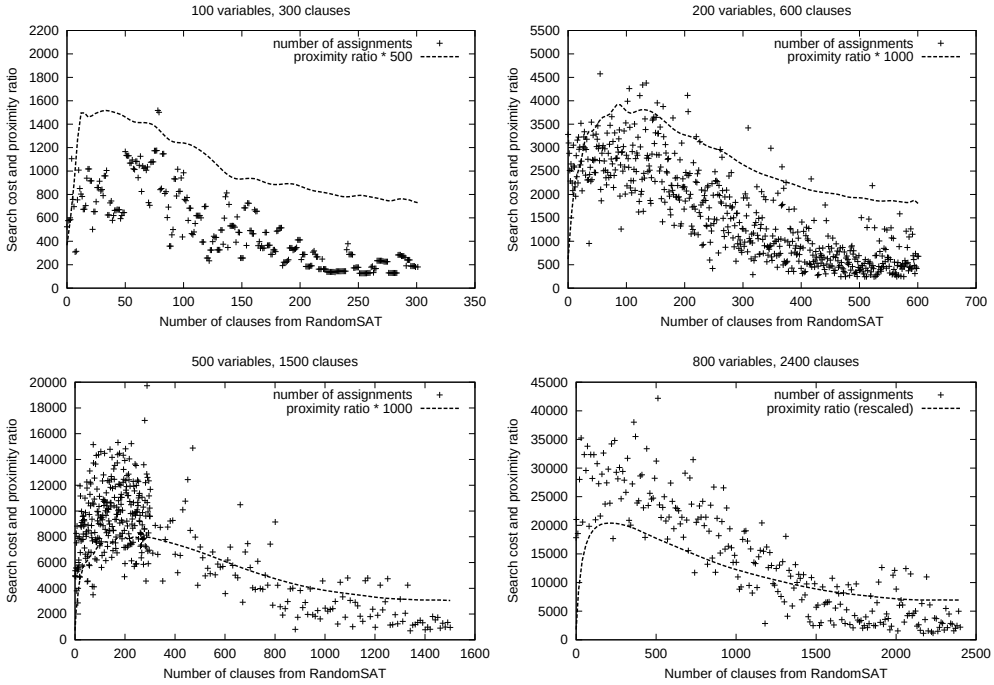

Figure 6.3: Search cost (number of assignments needed to solve the instance) for instance gradually interpolating between lattice SAT and random SAT. The proximity ratio (rescaled) is also plotted. Observe that the instances requiring the highest search cost correspond to those with the small-world property.

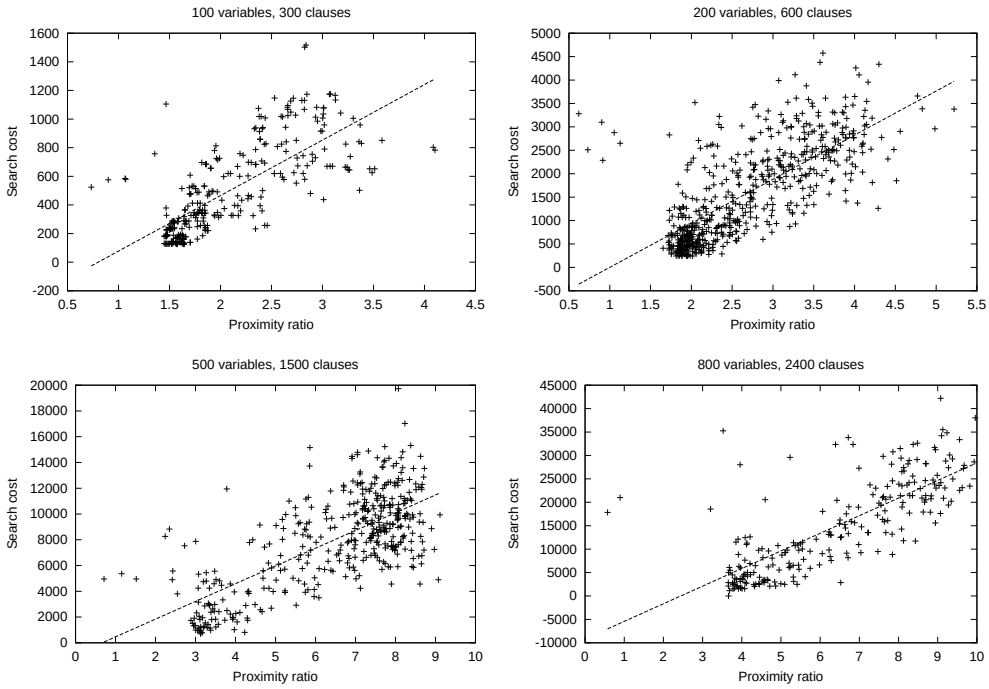

Figure 6.4: Correlation between search cost and proximity ratio.

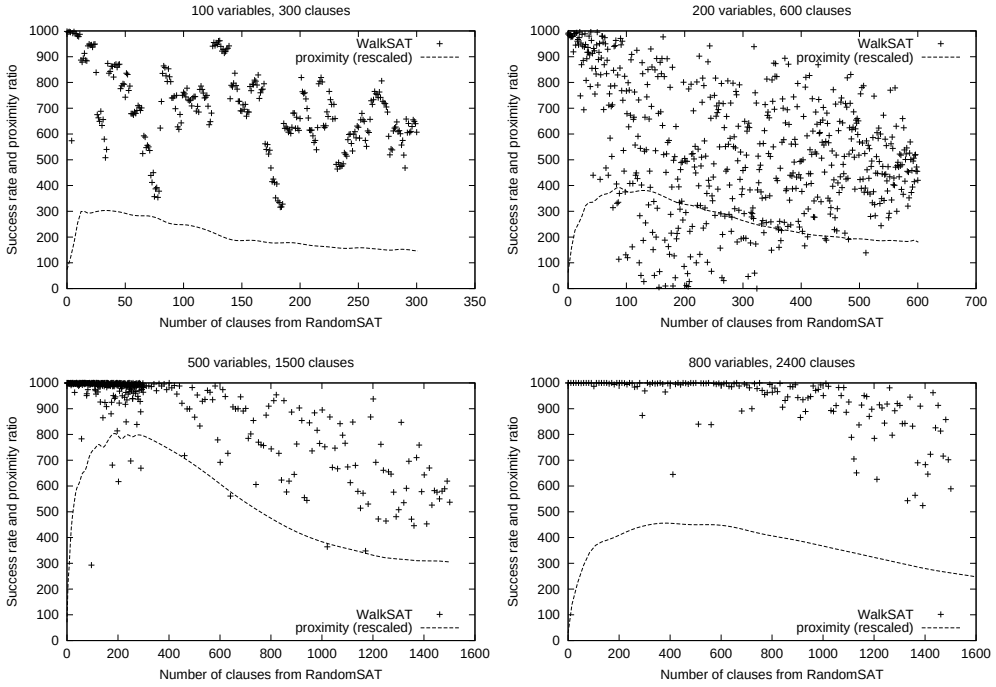

Figure 6.5: Success rate (out of 1000 runs) of WalkSAT on instances gradually interpolating between lattice SAT and random SAT. The proximity ratio (rescaled) is also plotted.

ratio. Results are not as clear as in the previous case, nevertheless some clues are present. Considering the results obtained with WalkSAT, we note that in the proximity of the small-world region, are located some among the hardest instances (indeed, the success rate is the lowest). This behavior is particularly apparent on the 200-600 instances, where the lowest success rate instances are located approximately around the maximum proximity. Nevertheless, the correlation between number of success out of 1000 runs and the proximity ratio does not confirm the hypothesis. Indeed, as can be observed from the plots in Fig. 6.7, the correlation is even in contradiction with the previous results on systematic search. The case of GSAT is different. GSAT performance is not as good as the WalkSAT one, since it always reach a lower success rate. Nevertheless, we can observe that the most difficult instances are located in the small-world region[2]. The correlation between success rate and proximity ratio is plotted in Fig. 6.8. In this case we can observe a negative correlation between proximity ratio and success ratio, i.e., the higher the proximity ratio, the harder the instance.

At the present, we can not claim any generality from these experiments and fur-

[2]The 800-2400 instances are indeed not solved in the range corresponding to small-world.

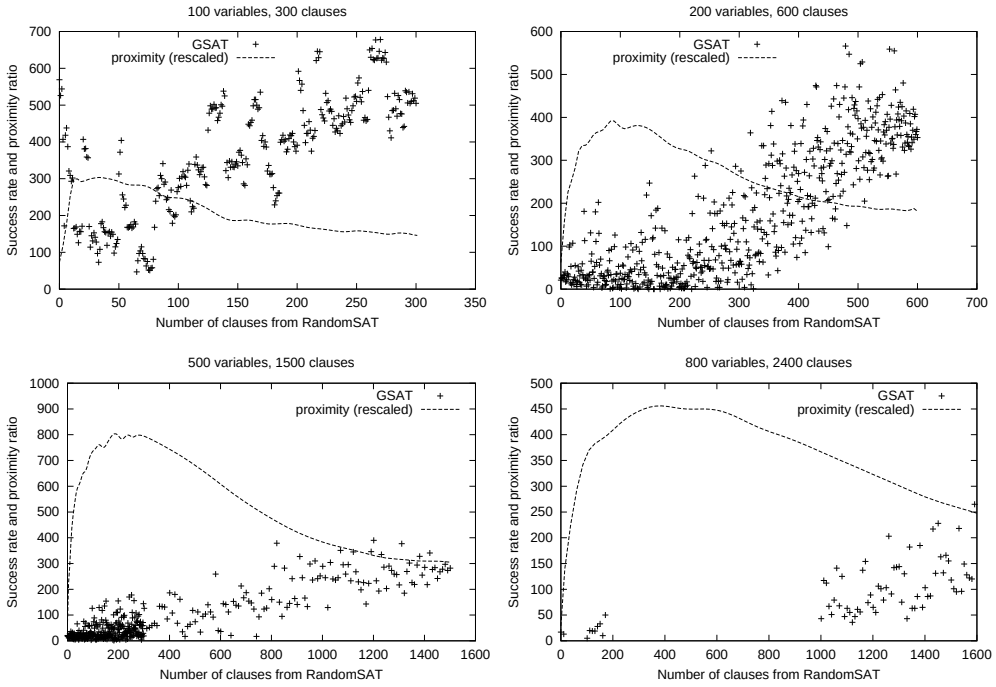

Figure 6.6: Success rate (out of 1000 runs) of GSAT on instances gradually interpolating between lattice SAT and random SAT. The proximity ratio (rescaled) is also plotted.

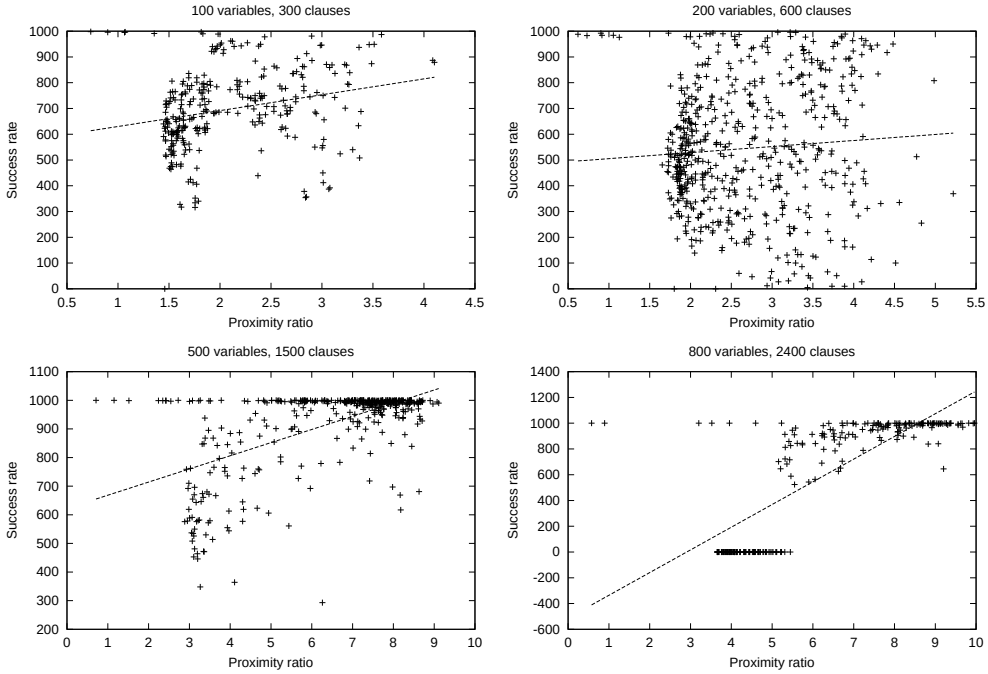

Figure 6.7: WalkSAT: Correlation between success rate and proximity ratio.

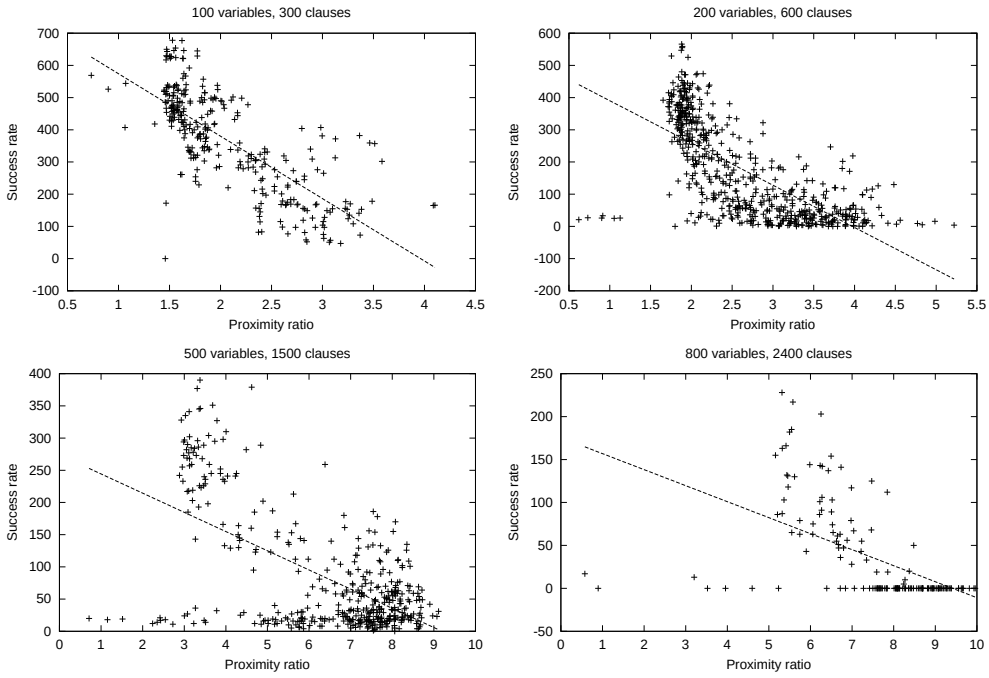

Figure 6.8: GSAT: Correlation between success rate and proximity ratio.

ther investigations are needed. If the conjecture on the positive correlation between instance hardness and proximity ratio is true, then the phenomenon may be explained considering the locality of decisions taken by the heuristics, as supposed in [230]: a locally good decision taken w.r.t. the clustering properties might be wrong with respect to the whole graph. This investigation open an interesting research direction concerning the analysis, comparison and design of effective heuristics.

Furthermore, a relevant research issue is the study of the relations between constraint graph properties (in this case, small-world) and characteristics of the search space.

Chapter 7

Work in Progress

In the following sections we will briefly illustrate a personal view of some connections between metaheuristics (mainly trajectory methods) and tree search along with some preliminary ideas and concepts that concern on-going work.

7.1 Integration of Metaheuristics and Complete Search

The integration of metaheuristics and systematic algorithms has recently produced very effective algorithms especially when applied to real-world problems. Discussions on similarities, differences and possible integration between metaheuristics and systematic search can be found in [76, 85, 107, 90]. A very successful example of such integration is the combination of metaheuristics and Constraint Programming (CP) [65, 171, 172, 36]. CP enables to model a COP by means of variables, domains[1] and constraints, which can be mathematical or symbolic (*global*). The latter ones involve a set of variables and describe subproblems, thus reducing the modeling complexity by encapsulating well defined parts of the problem into single constraints. Every constraint is associated to a *filtering* algorithm that deletes those values from a variable domain that do not contribute to feasible solutions. A CP system can be seen as the interaction of components (constraints) which communicate through shared variables. Constraints are activated as soon as a the domain of any variable involved has been changed. Then, they perform a propagation phase, i.e., they apply the filtering algorithm. This behavior stops as soon as there are no more values that can be removed from the domains or at least one domain is empty (i.e., no feasible solution exists). Since the complexity of the full constraint propagation is often exponential, the propagation may be not complete, hence, at the end of the propagation phase, some domains may contain unfeasible values. Hence, a search phase is started, such as Branch & Bound. A survey on the integration of metaheuristics and CP is provided in [65].

[1] We restrict the discussion to finite domains.

There are three main approaches for the integration of metaheuristics (especially trajectory methods) and systematic techniques (CP and tree search):

- Metaheuristics are applied before systematic methods, providing a valuable input, or vice versa.
- Metaheuristics use CP and/or tree search to efficiently explore the neighborhood.
- A "tree search"-based algorithm applies a metaheuristic in order to improve a solution (i.e., a leaf of the tree) or a partial solution (i.e., an inner node). Metaheuristic concepts can also be used to obtain incomplete but efficient tree exploration strategies.

The first approach can be seen as an instance of cooperative search and it represents a rather loose integration. The second approach combines the advantages of a fast search space exploration by means of a metaheuristic with the efficient neighborhood exploration performed by a systematic method. A prominent example of such a kind of integration is *Large Neighborhood Search* and related approaches [201, 29]. These approaches are effective mainly when the neighborhood to explore is very large. Moreover, many real-world problems have additional constraints (called *side constraints*) which might make them unsuitable for usual neighborhood exploration performed by metaheuristics. For instance, time windows constraints often reduce the number of feasible solutions in a neighborhood and could make local search not efficient, thus domain filtering techniques can effectively support neighborhood exploration. More examples can be found in [171, 172, 65]. The third approach preserves the search space exploration based on a systematic search (such as tree search), but sacrificing the exhaustive nature of the search [85, 107, 108, 151]. The hybridization is usually achieved by integrating concepts and machinery developed for metaheuristics (e.g., probabilistic choices, aspiration criteria, heuristic construction) into tree search methods. For example, instead of a chronological backtracking, a backjumping based on search history or information retrieved from local search samples can be performed. Another prominent example is the introduction of randomization in systematic techniques, as described in [96]. Many examples of this approach can be found in [65, 126, 196, 38, 176, 37].

7.1.1 Local Search vs. Nonsystematic Backtrack Search

Trajectory methods share some common characteristics with nonsystematic backtracking techniques. In fact, they both concentrate on promising areas of the search space and they are not complete. The latter ones are based on a search tree, whilst the former ones usually use memory in an unstructured way.

Typically, nonsystematic backtracking techniques [107, 108, 176] do not apply a chronological backtracking, but they backtrack to nodes suggested by a heuristic with the aim of exploring first the most promising areas of the search space and to avoid

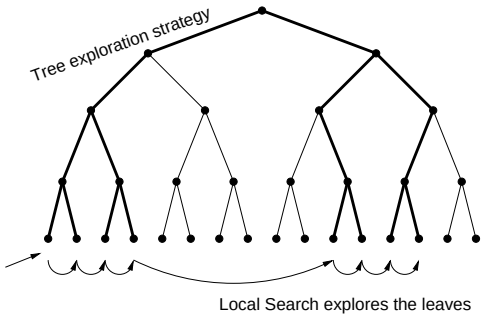

Figure 7.1: A pictorial representation of the search process performed by nonsystematic backtracking techniques and trajectory methods.

to be penalized by wrong earlier choices. Trajectory methods have almost the same goal, since they concentrate on the most promising areas of the search space and they are free to retract any past assignment. Figure 7.1 exemplifies the relation between the two approaches by enlightening that trajectory methods move along the fringe of the tree used by nonsystematic backtracking techniques.

Two are the main differences between the two approaches:

- Trajectory methods usually do not make use of a *structured* memory, but rather they exploit the search history by means of some of its features.
- Nonsystematic backtracking techniques usually do not exploit the auto-correlation of the search landscape[2], i.e., they do not explicitly take into account that two solutions that are close may differ only slightly in the objective function value.

Limited Discrepancy Search (LDS) [108] is a tree search particularly suitable for modifications in the direction of trajectory methods. LDS explores a binary search tree by means of a heuristic that suggests which branch to take at each decision point. We will call *reference solution* the leaf reached by following all the heuristic decisions. The algorithm explores the leaves (solutions) in increasing values of *discrepancy* (k) from the heuristic: it chooses all the branches suggested by the heuristic, except for k (see Fig. 7.2). The intuition behind this strategy is that, if the heuristic is good, it will suggest the wrong branch very few times. Therefore, it is more likely to find the optimal (or near-optimal) solutions very close to the reference solution.

LDS does not take into account the objective function and the correlation among solution values, i.e., it does not include intensification mechanisms such as hill-climbing. We introduced into LDS this mechanism, which is typical of local search, by allowing the dynamic changing of the reference solution[3]. As soon as a better

[2]See Chap. 2

[3]These ideas have been introduced in [151].

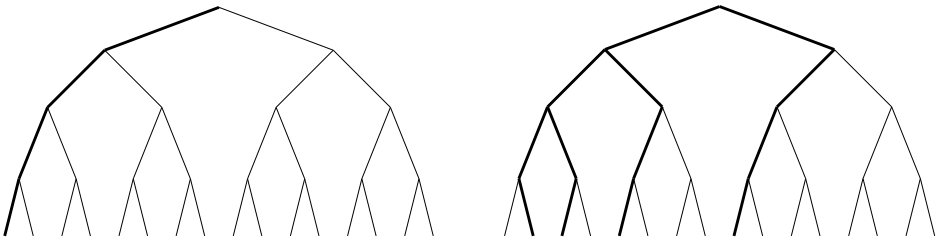

Figure 7.2: Tree exploration performed by LDS. On the left, the path suggested by the heuristic is explored. On the right, the leaves at discrepancy 1 from the reference solution are explored.

Algorithm 24 Climbing Discrepancy Search

```
k ← 1 {k is the discrepancy}
k_max ← n
s* ← InitialSolution() {s* is the reference solution proposed by the heuristics}
while k < k_max do
  Generate all leaves at discrepancy k w.r.t. s*: Leaves(s*, k)
  s' ← BestOf(Leaves(s*, k))
  if f(s') < f(s*) then
    s* ← s' {the new solution substitutes the reference one}
    k ← 1
  end if
  k ← k + 1
end while
```

solution is found, it is accepted as reference solution and (possibly) k is reset. This algorithm, that we called *Climbing Discrepancy Search*, is sketched in Alg. 24.

CDS indeed visits the solutions in the same order as Variable Neighborhood Descent (introduced in Sec. 2.3), provided that VND uses neighborhoods at increasing Hamming distances.

CDS has been proved more effective than LDS on preliminary experiments performed on random MAXSAT instances, reaching the optimal solution by exploring a considerably lower number of nodes.

CDS can be generalized by observing that the algorithm has two degrees of freedom: the choice of the solution among the set of leaves at discrepancy k from the reference solution (Choose()) and the acceptance criterion used to decide whether the chosen solution has to be accepted as the new reference one (Accept()). By specializing the functions Choose() and Accept(), we can obtain variants of CDS. The generalized version of CDS is reported in Alg. 25.

Observe that local search methods are characterized by the interplay of the functions Choose() and Accept(). For instance, Iterative Improvement chooses the best solution in the neighborhood and always accept it. Simulated Annealing chooses

Algorithm 25 High level generalized CDS

```
k ← 1 {k is the discrepancy}
s* ← InitialSolution() {s* is the reference solution proposed by the heuristics}
while Termination condition not met do
  Generate all leaves at discrepancy k w.r.t. s*: Leaves(s*, k)
  s' ← Choose(Leaves(s*, k), history)
  if Accept(s') then {if the new solution is accepted}
    s* ← s' {the new solution substitutes the reference one}
    k ← 1
  end if
  k ← k + 1
end while
```

at random a solution within the current neighborhood and accepts it with a probabilistic criterion. Another example is given by Tabu Search, which chooses the best solution in the neighborhood and accepts it depending on tabu and aspiration criteria. Therefore, variants of CDS can be designed by explicitly introducing ingredients from trajectory methods. Examples of such integration are given in Alg. 26 and Alg. 27, where a discrepancy-based Simulated Annealing and a discrepancy-based Tabu Search are respectively sketched.

The effectiveness of these algorithms have to be tested on several benchmarks and this experimental work is part of future research.

7.1.2 Ant Colony Optimization vs. Climbing Discrepancy Search

In this section we introduce some considerations about the relations between ACO and CDS. Subject of current work is the design, implementation and testing of hybrid metaheuristics based on these concepts.

The observations that follow are valid under these assumptions[4]:

- we restrict ACO to the pheromone-based solution construction mechanism;
- we suppose pheromone is on components and that the probability of adding a component to the current partial solution depends only upon the pheromone value of that component;
- we consider the basic version of CDS, where as soon as a better solution is found, it is accepted as the new reference one;
- finally, we consider binary trees.

To exemplify this case, we suppose to tackle MAXSAT. Solution components are assignments and pheromone is associated to components. If we represent the

[4] even though they can be also generalized

Algorithm 26 Discrepancy-based Simulated Annealing

$k \leftarrow 1$
Set k_{max}
$s^* \leftarrow$ InitialSolution()
while $k < k_{max}$ **do**
  Generate all not yet explored leaves at discrepancy k w.r.t. reference solution s^*: $Leaves(s^*, k)$
  while $Leaves(s^*, k) \neq \emptyset$ and No solution is accepted **do**
    Extract randomly a leaf s' from $Leaves(s^*, k)$
    $q = random(0, 1)$, $p = exp(-\frac{f(s') - f(s^*)}{T})$
    if $f(s^*) < f(s')$ or $q \leq p$ **then**
      $s^* \leftarrow s'$
      $k \leftarrow 1$
    end if
  end while
  $k \leftarrow k + 1$
end while

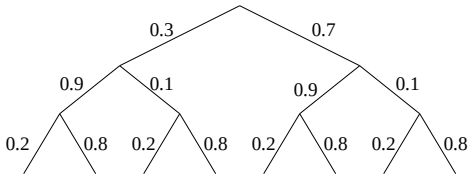

Figure 7.3: Search tree of the ACO solution construction. Labels on arcs reports the pheromone values of the corresponding assignments (for simplicity, we suppose pheromone values in the range $]0, 1[$).

search along a tree and we label arcs with the pheromone value of the corresponding assignment[5], we end up with a tree such as the one depicted in Fig. 7.3.

Ants use the pheromone values to construct a solution, i.e., they use pheromone as a guide to select a path along the search tree, from the root to a leaf. If the choice at the decision points was deterministic, all the ants would select the path with the highest pheromone values (as depicted in Fig. 7.4), in the same way as the construction of the reference solution in CDS (considering the pheromone values with the same role as the heuristic in CDS).

However, since the choice is probabilistic and the higher the pheromone value on a branch, the higher the probability to select that branch, the solutions constructed by the ants (i.e., the explored leaves of the tree) will share many of the assignments corresponding to the highest pheromone values, but some will be different. In other

[5]For simplicity, we suppose pheromone values in the range $]0, 1[$.

Algorithm 27 Discrepancy-based Tabu Search

```
k ← 1
Set k_max
s* ← InitialSolution()
while k < k_max do
  Generate all not yet explored leaves at discrepancy k w.r.t. reference solution s*:
  Leaves(s*, k)
  AllowedSet ← {z ∈ Leaves(s*, k)}
  if AllowedSet ≠ ∅ then
    s* ← BestOf(s*, AllowedSet) {the new solution substitutes the reference one}
    k ← 1
  else
    k ← k + 1
  end if
end while
```

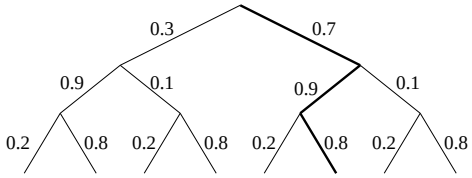

Figure 7.4: Search tree of the ACO solution construction. In bold, the path traced by following the highest pheromone values.

words, ants will construct solutions with different degrees of discrepancy from the reference solution. It is important to observe that the higher the discrepancy of a solution, the lower the probability of its construction. Therefore, we can assert that ants provide a kind of stochastic discrepancy search: they mainly concentrate around the reference solution, by preferring the exploration of solutions at low discrepancy from it, but they also have non zero probability to explore solutions at high discrepancy from the reference solution.

An important difference between the two algorithms concerns the strategy used for exploring the 'neighborhood' of the reference solution. This difference can be described in terms of intensification and diversification. In CDS, solutions at increasing discrepancy from the reference are explored: CDS deterministically increases the exploration radius, thus it proceeds from low to high diversification. At the other extreme, ACO first explores a large area (indeed, pheromone values are not strongly differing in the first phases of the search) and then it converges to one solution (ideally). Therefore, ACO follows the inverse pattern: first the highest diversification, then almost no diversification.

Moreover, note that, while in ACO the pheromone values are smoothly changed

toward a better solution found, in the basic version of CDS the reference solution is completely changed as soon as a better solution is found.

It should also be noted that ACO usually uses both pheromone and heuristic to guide the solution construction. In the first cycles of the algorithm, the heuristic is the strongest guidance, since pheromone values are still not much differentiated. Then, pheromone increases its influence and enables the search to exploit the search history. In CDS, the search history is exploited in the reference solution update mechanism, which, in its basic version, exploits only the most recent history.

The similarities and difference just discussed, suggest some possible integrations of mechanisms from ACO and CDS. In order to enable CDS to exploit the search history, the algorithm can be modified by introducing 'pheromone values' to implement a kind of distributed memory. Like in ACO, the rules that manage pheromone update define the strength of past decisions on the present. Furthermore, CDS can start with more than one reference solution and explore the neighborhoods of all of them and eventually converge toward the most promising one. CDS is typically deterministic, but randomization can also added. For instance, instead of deterministically increasing the discrepancy, probabilities can be assigned to the decision points, favoring the choice of lower discrepancy paths, but, at the same time, allowing the choice of leaves at higher discrepancy.

On the other side, ACO can be enriched by introducing some components from discrepancy search. First of all, a coordination mechanism can be used in order to prevent different ants from constructing the same solution, thus optimizing the computational resources with respect to the exploration of the search space[6]. This modification goes into the direction of reducing the randomness in ACO. Moreover, the intensification/diversification pattern in ACO can be inverted, by first favoring low diversification and increasing it during the computation. Of course, this choice needs a particular care, since it makes the metaheuristic unable to converge to a solution, which is often a required characteristic in approximate algorithms.

Finally, while tree search can make use of constraint propagation, this has not yet been tried for ACO. Indeed, ants can include constraint propagation in their construction procedure and thus reduce the search space.

[6]Observe that, depending on the pheromone updating rules, if a solution is constructed more than once, it may receive a greater amount of pheromone, thus it will be more favored in the future.

Chapter 8

Conclusion

The design and implementation of effective and efficient algorithms to tackle large real-world problems is nowadays a very active research field. Metaheuristics have been proven powerful and effective, though only in recent years researchers have started to systematically study metaheuristic algorithms and their performance.

In this thesis we have presented and described metaheuristics, focusing on their applications on SAT and MAXSAT problems. Moreover, we have discussed the impact of problem structure on algorithm behavior by studying how some constraint graph properties affect the search performance.

In the following, we briefly outline the main contributions of this work:

- We have first given a survey on metaheuristics, by introducing the most important algorithms, their variants and improvements. We have underlined the importance of intensification and a diversification and shown that metaheuristics can be conceptually analyzed on their basis.

- We have introduced a multi-level architecture that enables the design and implementation of metaheuristics in a component-based fashion, moving the focus from the algorithmic and conceptual viewpoint, to the software engineering standpoint. The architecture is organized in four levels: the first is devoted to the solution construction, the second to the improvement of the solution, the third to the implementation of long term strategies for intensification and diversification and the fourth is needed for coordinating lower levels. Algorithmic components are encapsulated in software agents.

- The issues of algorithm design and implementation are considered in the novel application of two metaheuristics to MAXSAT problems. We presented the development and implementation of Ant Colony Optimization (ACO) and of Iterated Local Search (ILS) metaheuristics. We also discussed design issues and choices, along with experimental results. We achieved particularly good results with Iterated Local Search.

- The observation of the behavior of metaheuristics on different sets of instances shows that the same algorithm may perform very differently on different kinds of instances. Moreover, the design of metaheuristics which effectively tackle real-world problems requires the study of the impact of problem structure on metaheuristic behavior. We dealt with this topic in the context of SAT and MAXSAT problems. We have first defined the structure of SAT/MAXSAT on the basis of a graph associated to the instances. This approach enables us to extract general properties of SAT and MAXSAT problem structure.

  We then studied the influence of some graph properties, in particular average connectivity and frequency of node degree, on parallel local search. We have found empirical evidence for the presence of an optimal number of parallel local moves that enables the algorithm to achieve the highest effectiveness. We found also that the optimal number of parallel moves is negatively correlated with the connectivity among variables. Furthermore, we have studied this phenomenon in detail, by analyzing random, structured and constant-connectivity instances. The results obtained can give insight into the Criticality and Parallelism phenomenon and into the behavior of trajectory methods on SAT/MAXSAT problems. Moreover, this study enabled us to improve the ILS metaheuristic applied to MAXSAT.

  Finally, we also investigated the hardness of small-world SAT instances (those with low characteristic path length and high clustering), finding interesting results which may support the conjecture that small-world instances are among the hardest to solve for both approximate and complete algorithms.

- We have concluded by outlining topics of our current research which tries to combine metaheuristics and tree search. The most relevant approaches relate to the integration of metaheuristics into discrepancy search, leading to the algorithm called Climbing Discrepancy Search and its variants. Moreover, the exploitation of pheromone based search in algorithms based on a search tree is a very promising subject concerning the integration of approximate and complete techniques.

The study of relations between structure and algorithm behavior is still a partially unexplored area. We briefly summarize some open questions:

- In this thesis we have just investigated one of the possible ways of characterizing structure. Other definitions for graphs are possible (such as weighted graphs), to capture different problem features.

- The relations between problem structure and search landscape should be investigated, in order to achieve an effective control of strategies and heuristics.

- Concerning Criticality and Parallelism, formal relations between connectivity and optimal parallelism have not yet be found. Moreover, the combination of parallel local moves and noise into local search has been proven very effective

in preliminary experiments we performed and this integration has still to be studied in detail.

- Finally, results and concepts discussed in this thesis may be effectively extended to tree search algorithms. For example, we may ask whether a kind of Criticality and Parallelism phenomenon can be discovered also in systematic search. Furthermore, the study of properties of graph associated to problem instances is not restricted to the application of metaheuristics, but it can be also extended toward systematic search and the integration of approximate and complete techniques.

Bibliography

[1] E. H. L. Aarts, J. H. M. Korst, and P. J. M. van Laarhoven. Simulated annealing. In Emile H. L. Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 91–120. Wiley-Interscience, Chichester, England, 1997.

[2] E. H. L. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, UK, 1997.

[3] D. Achlioptas and C. Moore. The asymptotic order of the random k -SAT threshold. In *Proceedings of FOCS 2002*, pages 779–788, 2002.

[4] L. A. Adamic and B. A. Huberman. Power-law distribution of the world wide web. *Science*, 287, 2000.

[5] A. A. Andreatta, S. E. R. Carvalho, and C. C. Ribeiro. An object-oriented framework for local search heuristics. In *Proceedings of the 26th Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '98)*, pages 33–45. IEEE, Piscataway, 1998.

[6] V. Bachelet and E.G. Talbi. Cosearch: a co-evolutionary metaheuritics. In *Proceedings of Congress on Evolutionary Computation – CEC'2000*, pages 1550–1557, 2000.

[7] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[8] T. Bäck, D. B. Fogel, and Z. Machalewicz, editors. *Handbook of Evolutionary Computation*. Institute of Physics Publishing Ltd, Bristol, UK, 1997.

[9] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report No. CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994.

[10] S. Baluja and R. Caruana. Removing the Genetics from the Standard Genetic Algorithm. In A. Prieditis and S. Russel, editors, *The International Conference on Machine Learning 1995*, pages 38–46, San Mateo, California, 1995. Morgan Kaufmann Publishers.

[11] Y. Bar–Yam. *Dynamics of Complex Systems*. Studies in nonlinearity. Addison–Wesley, 1997.

[12] R. Battiti. Reactive search: Toward self-tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, pages 61–83. John Wiley & Sons, Chichester, UK, 1996.

[13] R. Battiti and M. Protasi. Reactive Search, a history-base heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 2:Article 2, 1997.

[14] R. Battiti and M. Protasi. Solving MAX-SAT with non–oblivious functions and history–based heuristics. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, number 35 in DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, pages 649–667. American Mathematical Society, Association for Computing Machinery, 1997.

[15] R. Battiti and M. Protasi. Approximate algorithms and heuristics for MAX-SAT. In D. Z. Du and P. .M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 77–148. Kluwer Academic Publishers, 1998.

[16] R. Battiti and G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[17] J.C. Beck and M.S. Fox. Dynamic problem structure analysis as a basis for constrained -directed scheduling heuristics. *Artificial Intelligence*, 117:31–81, 2000.

[18] R. Béjar, A. Cabiscol, and C.P. Gomes C. Fernàndez, F. Manyà. Capturing structure with satisfiability. In T. Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001, 7th International Conference*, Lecture Notes in Computer Science, pages 137–152. Springer, 2001.

[19] A. Bertoni, P. Campadelli, M. Carpenteri, and G. Grossi. A genetic model: Analysis and application to MAXSAT. *Evolutionary Computation*, 3(8):291–309, 2000.

[20] S. Binato, W. J. Hery, D. Loewenstern, and M. G. C. Resende. A greedy randomized adaptive search procedure for job shop scheduling. In P. Hansen and C. C. Ribeiro, editors, *Essays and surveys on metaheuristics*. Kluwer Academic Publishers, 2001.

[21] C. Blum. ACO Applied to Group Shop Scheduling: A Case Study on Intensification and Diversification. In M. Dorigo, G. Di Caro, and M. Sampels, editors, *Proceedings of ANTS 2002 – Third International Workshop on Ant Algorithms*, volume 2463 of *Lecture Notes in Computer Science*, pages 14–27. Springer Verlag, Berlin, Germany, 2002.

[22] C. Blum. Ant Colony Optimization For The Edge-Weighted k-Cardinality Tree Problem. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 27–34, New York, 2002. Morgan Kaufmann Publishers.

[23] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. Technical Report IRIDIA/01-13, IRIDIA, Université Libre de Bruxelles, 2001.

[24] C. Blum, A. Roli, and M. Dorigo. HC–ACO: The hyper-cube framework for Ant Colony Optimization. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 2, pages 399–403, Porto, Portugal, 2001.

[25] E. Bonabeau, M. Dorigo, and T. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[26] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1999.

[27] H. M. Botee and E. Bonabeau. Evolving Ant Colony Optimization. *Adv. Complex Systems*, 1:149–159, 1998.

[28] P. Calégary, G. Coray, A. Hertz, D. Kobler, and P. Kuonen. A taxonomy of evolutionary algorithms in combinatorial optimization. *Journal of Heuristics*, 5:145–158, 1999.

[29] Y. Caseau and F. Laburthe. Effective Forget-and-Extend Heuristics for Scheduling Problems. In *Proceedings of CP-AI-OR'02 – Fourth Int. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, Ferrara (Italy), 1999. Also available at: www.deis.unibo.it/Events/Deis/Workshops/Proceedings.html.

[30] V. Cerny. A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[31] P. Chardaire, J. L. Lutton, and A. Sutter. Thermostatistical persistency: A powerful improving concept for simulated annealing algorithms. *European Journal of Operational Research*, 86:565–579, 1995.

[32] I. Charon and O. Hudry. The noising methods: A generalization of some metaheuristics. *European Journal of Operational Research*, pages 135:86–101, 2001.

[33] C. A. Coello Coello. An Updated Survey of GA-Based Multiobjective Optimization Techniques. *ACM Computing Surveys*, 32(2):109–143, 2000.

[34] D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46:93–100, 1990.

[35] B. De Backer, V. Furnon, and P. Shaw. An object model for meta-heuristic search in constraint programming. In *Proceedings of CP–AI–OR'99–Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, 1999.

[36] B. De Backer, V. Furnon, and P. Shaw. Solving Vehicle Routing Problems Using Constraint Programming and Metaheuristics. *Journal of Heuristics*, 6:501–523, 2000.

[37] F. Della Croce and V. T'kindt. A Recovering Beam Search algorithm for the one machine dynamic total completion time scheduling problem. *Journal of the Operational Research Society*, 2003. To appear.

[38] M. Dell'Amico and A. Lodi. On the Integration of Metaheuristic Strategies in Constraint Programming. In C. Rego and B. Alidaee, editors, *Adaptive Memory and Evolution: Tabu Search and Scatter Search*. Kluwer Academic Publishers, Boston, MA, 2002.

[39] M. Dell'Amico, A. Lodi, and F. Maffioli. Solution of the Cumulative Assignment Problem with a well–structured Tabu Search method. *Journal of Heuristics*, 5:123–143, 1999.

[40] M. L. den Besten, T. Stützle, and M. Dorigo. Design of iterated local search algorithms: An example application to the single machine total weighted tardiness problem. In *Proceedings of EvoStim'01*, Lecture Notes in Computer Science, pages 441–452. Springer, April 2001. Also available as technical report AIDA-00-07, Intellectics Group, Darmstadt University of Technology, Germany.

[41] J. Denzinger and T. Offerman. On cooperation between evolutionary algorithms and other search paradigms. In *Proceedings of Congress on Evolutionary Computation – CEC'1999*, pages 2317–2324, 1999.

[42] R. L. Devaney. *An introduction to chaotic dynamical systems*. Addison–Wesley, second edition, 1989.

[43] G. Di Caro and M. Dorigo. AntNet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research (JAIR)*, 9:317–365, 1998.

[44] L. Di Gaspero and A. Schaerf. EasyLocal++: An object-oriented framework for the design of local search algorithms and metaheuristics. In *Proceedings of MIC'2001 – Meta–heuristics International Conference* [192], pages 287–292.

[45] F. M. Dittes. Optimization on rugged landscapes: A new general purpose monte carlo approach. *Physical Review Letters*, 76(25):4651–4655, June 1996.

[46] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, DEI, Politecnico di Milano, Italy, 1992. pp. 140.

[47] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, 1999. Also available as Technical Report IRIDIA/99-1, Université Libre de Bruxelles, Belgium.

[48] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Art. Life*, 5(2):137–172, 1999.

[49] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the travelling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[50] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 26(1):29–41, 1996.

[51] M. Dorigo and T. Stützle. The ant colony optimization metaheuristic: Algorithms, applications and advances. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 251–285. Kluwer Academic Publishers, Norwell, MA, 2002.

[52] M. Dorigo and T. Stützle. *Ant Colony Optimization*. 2003. To appear.

[53] G. Dueck. New Optimization Heuristics. *Journal of Computational Physics*, 104:86–92, 1993.

[54] G. Dueck and T. Scheuer. Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing. *Journal of Computational Physics*, 90:161–175, 1990.

[55] A. E. Eiben, P.-E. Raué, and Z. Ruttkay. Genetic algorithms with multi-parent recombination. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, volume 866 of *Lecture Notes in Computer Science*, pages 78–87, Berlin, 1994. Springer.

[56] A. E. Eiben and Z. Ruttkay. Constraint satisfaction problems. In T. Bäck, D. Fogel, and M. Michalewicz, editors, *Handbook of Evolutionary Computation*. Institute of Physics Publishing Ltd, Bristol, UK, 1997.

[57] A. E. Eiben and C. A. Schippers. On evolutionary exploration and exploitation. *Fundamenta Informaticae*, 35:1–16, 1998.

[58] W. Feller. *An Introduction to Probability Theory and its Applications*. John Whiley, 1968.

[59] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[60] J. Ferber. *Multi-agent systems : an introduction to distributed artificial intelligence*. Addison-Wesley, 1999.

[61] R. Ferrer, C. Janssen, and R. V. Solé. Topology of technology graphs: Small world patterns in electronic circuits. *Physical Review E*, 63, 2001.

[62] P. Festa and M. G. C. Resende. GRASP: An annotated bibliography. In C. C. Ribeiro and P. Hansen, editors, *Essays and Surveys on Metaheuristics*, pages 325–367. Kluwer Academic Publishers, 2002.

[63] A. Fink and S. Voß. Reusable metaheuristic software components and their application via software generators. In *Proceedings of MIC'2001 – Meta–heuristics International Conference* [192], pages 637–641.

[64] M. Fleischer. Simulated Annealing: past, present and future. In C. Alexopoulos, K. Kang, W.R. Lilegdon, and G. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 155–161, 1995.

[65] F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Norwell, MA, 2002.

[66] F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the tsptw. *INFORMS Journal of Computing*, 14(4):403–417, 2002.

[67] D. B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, January 1994.

[68] G. B. Fogel, V. W. Porto, D. G. Weekes, D. B. Fogel, R. H. Griffey, J. A. McNeil, E. Lesnik, D. J. Ecker, and R. Sampath. Discovery of RNA structural elements using evolutionary computation. *Nucleic Acids Research*, 30(23):5310–5317, 2002.

[69] L. J. Fogel. Toward inductive inference automata. In *Proceedings of the International Federation for Information Processing Congress*, pages 395–399, Munich, 1962.

[70] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, 1966.

[71] C. Fonlupt, D. Robilliard, P. Preux, and E.G. Talbi. Fitness landscapes and performance of meta-heuristics. In S. Voß, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*. Kluwer Academic, 1999.

[72] J. Frank. Weighting for Godot: Learning heuristics for GSAT. In *Proceedings AAAI'96*, pages 338–343. AAAI Press, 1996.

[73] J. Frank. Learning short-term weights for GSAT. In *Proceedings IJCAI'97*, volume 1, pages 384–389, 1997.

[74] J. Frank, P. Cheeseman, and J. Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[75] B. Freisleben and P. Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In *International Conference on Evolutionary Computation*, pages 616–621, 1996.

[76] E. C. Freuder, R. Dechter, M. L. Ginsberg, B. Selman, and E. P. K. Tsang. Systematic Versus Stochastic Constraint Satisfaction. In *IJCAI 95*, volume 2, pages 2027–2032. Morgan Kaufmann, 1995.

[77] L. M. Gambardella and M. Dorigo. Ant Colony System hybridized with a new local search for the sequential ordering problem. *INFORMS Journal on Computing*, 12(3):237–255, 2000.

[78] L. M. Gambardella, É. D. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50:167–176, 1999.

[79] M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[80] L. Di Gaspero and A. Schaerf. Multi-neighborhood local search for the course timetabling problem. In *Proceedings of the 4th International Conference on Practice and Theory of Automated Timetabling (PATAT-2002)*, 2002.

[81] M. Gendreau, G. Laporte, and J.-Y Potvin. Metaheuristics for the Vehicle Routing Problem. In P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*, volume 9 of *SIAM Series on Discrete Mathematics and Applications*, pages 129–154. 2001.

[82] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of AAAI99*, pages 654–660, 1999.

[83] I. P. Gent and T. Walsh. The enigma of SAT hill-climbing procedures. Research Paper 605, Dept. of Artificial Intelligence, University of Edinburgh, 1992.

[84] I. P. Gent and T. Walsh. An empirical analysis of search in GSAT. *JAIR*, 1:47–59, Sep 1993.

[85] M. L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[86] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[87] F. Glover. Future paths for integer programming and links to artificial intelligence. *Comp. Oper. Res.*, 13:533–549, 1986.

[88] F. Glover. Tabu Search Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[89] F. Glover. Scatter search and path relinking. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, Advanced topics in computer science series. McGraw-Hill, 1999.

[90] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[91] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.

[92] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, MA, 1989.

[93] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison–Wesley, 1989.

[94] D. E. Goldberg, K. Deb, and B. Korb. Don't worry, be messy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, La Jolla, CA, 1991. Morgan Kaufmann.

[95] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT Solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[96] C.P. Gomes. Structure, duality, and randomization – common themes in AI and OR. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, 2000.

[97] C.P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[98] C.P. Gomes and D. Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Extensions*, 2002.

[99] J. J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 42–60. Morgan Kaufmann Publishers, 1987.

[100] J. Gu. Local search for the satisfiability (SAT) problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(4):1108–1129, July-August 1993.

[101] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In *Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy, 1986.

[102] P. Hansen, B. Jaumard, N. Mladenović, and A.D. Parreira. Variable neighbourhood search for maximum weight satisfiability problem. Le Cahiers du GERAD G–2000–62, Group for Research in Decision Analysis, 2000.

[103] P. Hansen and N. Mladenović. Variable Neighborhood Search for the p-Median. *Location Science*, 5:207–226, 1997.

[104] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In S. Voß, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*, chapter 30, pages 433–458. Kluwer Academic Publishers, 1999.

[105] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467, 2001.

[106] G. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical Report No. 99010, IlliGAL, University of Illinois, 1999.

[107] W. D. Harvey. *Nonsystematic Backtracking Search.* Phd thesis, CIRL, University of Oregon, 1269 University of Oregon; Eugene, OR USA 97403-1269, 1995.

[108] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 20-25 1995. Morgan Kaufmann, 1995.

[109] A. Hertz and D. Kobler. A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, 126:1–12, 2000.

[110] T. Hogg. Which search problems are random? In *Proc. of AAAI98*, pages 438–443, 1998.

[111] T. Hogg and A. Huberman. Better than the best: The power of cooperation. In *SFI 1992 Lectures in Complex Systems*, pages 163–184. Addison-Wesley, 1993.

[112] T. Hogg, B. A. Huberman, and C. P. Williams. Phase transitions and the search problems. *Artificial Intelligence*, 81(1–2), 1996. Special issue on Phase Transitions and Search Problems.

[113] T. Hogg and C.P. Williams. Solving the really hard problems with cooperative search. In *Proceedings of AAAI93*, pages 213–235. AAAI Press, 1993.

[114] J. H. Holland. *Adaption in natural and artificial systems.* The University of Michigan Press, Ann Harbor, MI, 1975.

[115] H. H. Hoos and T. Stützle. Towards a characterisation of the behaviour of stochastic local search algorithms for sat. *Artificial Intelligence*, 112:213–232, 1999.

[116] H.H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of IJCAI-99*, pages 988–993, 1999.

[117] H.H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, pages 283– 292. IOS Press, 2000.

[118] W. Hordijk. A measure of landscapes. *Evolutionary Computation*, 4(4):335–360, 1996.

[119] L. Ingber. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics – Special Issue on Simulated Annealing Applied to Combinatorial Optimization*, 25(1):33–54, 1996.

[120] Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. In *1st International Joint Workshop on Artificial Intelligence and Operations Research.*, 1995.

[121] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: a case study. In E.H. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997.

[122] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, Univ. of New Mexico, Albuquerque, NM, 1995.

[123] T. Jones. One operator, one landscape. Santa Fe Institute Technical Report 95-02-025, Santa Fe Institute, 1995.

[124] D. E. Joslin and D. P. Clements. "Squeaky Wheel" Optimization. *Journal of Artificial Intelligence Research*, 10:353–373, 1999.

[125] S. Joy, J. Mitchell, and B. Borchers. A branch and cut algorithm for MAX-SAT and weighted MAX-SAT. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *AMS/DIMACS Series in Discrete Mathematics and Applications.* American Mathematical Society, April 1997.

[126] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139:21–45, 2002.

[127] L. P. Kaelbing, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[128] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford University Press, 1993.

[129] S. A. Kauffman. *At home in the universe.* Oxford University Press, New York, 1995.

[130] S. A. Kauffman and W. Macready. Technological evolution and adaptive organizations. *Complexity*, 26(2):26–43, March 1995.

[131] P. Kilby, P. Prosser, and P. Shaw. Guided Local Search for the Vehicle Routing Problem with time windows. In S. Voß, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*, pages 473–486. Kluwer Academic, 1999.

[132] S. Kirkpartick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, 13 May 1983*, 220(4598):671–680, 1983.

[133] F. Laburthe and Y. Caseau. SALSA: A language for search algorithms. In M. Maher and J. F. Puget, editors, *Principle and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 1998.

[134] V. R. Lesser. An overview of DAI: Viewing distributed AI as distributed search. *Journal of Japanese Society for Artificial Intelligence-Special Issue on Distributed Artificial Intelligence*, 5(4):392–400, 1990.

[135] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems. In P. Van Henteryck, editor, *Proceedings of CP02 - Eighth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*. Springer, 2002.

[136] H. R. Lourenço, O. Martin, and T. Stützle. A beginner's introduction to Iterated Local Search. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 1, pages 1–6, Porto – Portugal, 2001.

[137] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, 2002.

[138] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34(1):111–124, 1986.

[139] W. G. Macready and S. A. Kauffman A. G. Siapas. Criticality and parallelism in combinatorial optimization. *Science*, 271:56–59, January 1996.

[140] O. Martin and S. W. Otto. Combining Simulated Annealing with Local Search Heuristics. *Annals of Operations Research*, 63:57–75, 1996.

[141] O. Martin, S. W. Otto, and E. W. Felten. Large-step markov chains for the traveling salesman problem. *Complex Systems*, 5(3):299–326, 1991.

[142] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 321–326, Menlo Park, July 27–31 1997. AAAI Press.

[143] D. Merkle, M. Middendorf, and H. Schmeck. Ant Colony Optimization for Resource-Constrained Project Scheduling. *IEEE Transactions on Evolutionary Computation*, 6(4):333–346, 2002.

[144] P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, Advanced topics in computer science series. McGraw-Hill, 1999.

[145] http://www.metaheuristics.net/, 2000. Visited in January 2003.

[146] N. Meuleau and M. Dorigo. Ant Colony Optimization and Stochastic Gradient Descent. *Artificial Life*, 8(2):103–121, 2002.

[147] L. Michael and P. van Hentenryck. Localizer: A modeling language for local search. *INFORMS Journal of Computing*, 11:1–14, 1999.

[148] Z. Michalewicz and M. Michalewicz. Evolutionary computation techniques and their applications. In *Proceedings of the IEEE International Conference on Intelligent Processing Systems*, pages 14–24, Beijing, China, 1997. Institute of Electrical & Electronics Engineers, Incorporated.

[149] M. Milano and A. Roli. Boolean networks-based algorithms for the satisfiability problem. In *Electronic Proceedings of the Workshop on Experimental Analysis of Algorithms for Artificial Intelligence*, 1999. AI*IA working group on Knowledge Representation and Reasoning.

[150] M. Milano and A. Roli. Solving the satisfiability problem through boolean networks. In Evelina Lamma and Paola Mello, editors, *Lecture Notes in Artificial Intelligence - AI*IA99: Advances in Artificial Intelligence*, volume 1792, pages 72–83. Springer, 2000.

[151] M. Milano and A. Roli. On the relation between complete and incomplete search: an informal discussion. In *Proceedings of CP-AI-OR'02 – Fourth Int. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 237–250, Le Croisic (France), 2002.

[152] P. Mills and E. Tsang. Guided Local Search for solving SAT and weighted MAX-SAT Problems. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000*, pages 89–106. IOS Press, 2000.

[153] D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions of sat problems. In *Proceedings, Tenth National Conference on Artificial Intelligence*, pages 459–465. AAAI Press/MIT Press, July 1992.

[154] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT press, Cambridge, MA, 1998.

[155] N. Mladenović and D. Urošević. Variable Neighborhood Search for the k-Cardinality Tree. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 2, pages 743–747, Porto – Portugal, 2001.

[156] J. M. Montoya and R. V. Solé. Small world patterns in food webs. *Journal of Theoretical Biology*, 214:405–412, 2002.

[157] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Toward memetic algorithms. Tech. Rep. Caltech Concurrent Computation Program 826, California Institute of Technology,Pasadena, California, USA, 1989.

[158] P. Moscato. Memetic algorithms: A short introduction. In F. Glover D. Corne and M. Dorigo, editors, *New Ideas in Optimization.* McGraw-Hill, 1999.

[159] H. Mühlenbein. Evolution in time and space – the parallel genetic algorithm. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms.* Morgan Kaufmann, San Mateo, USA, 1991.

[160] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV*, volume 1411 of *Lecture Notes in Computer Science*, pages 178–187, Berlin, 1996. Springer.

[161] H. Mühlenbein and H.-M. Voigt. Gene Pool Recombination in Genetic Algorithms. In I. H. Osman and J. P. Kelly, editors, *Proc. of the Metaheuristics Conference*, Norwell, USA, 1995. Kluwer Academic Publishers.

[162] G. L. Nemhauser and A. L. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[163] M.E.J. Newman. Random graphs as models of networks. SFI Working Paper 02-005, Santafe Institute, 2002.

[164] M.E.J. Newman, S.H. Strogatz, and D.J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E*, 64, 2001.

[165] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job-shop problem. *Management Science*, 42(2):797–813, 1996.

[166] I. H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[167] I. H. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:513–623, 1996.

[168] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity.* Dover Publications, Inc., New York, 1982.

[169] A.J. Parkes. Distributed local search, phase transitions, and polylog time. In *Workshop "Stochastic Search Algorithms" at IJCAI-01*, 2001.

[170] M. Pelikan, D. E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. Technical Report No. 99018, IlliGAL, University of Illinois, 1999.

[171] G. Pesant and M. Gendreau. A view of local search in Constraint Programming. In *Principles and Practice of Constraint Programming - CP'96*, volume 1118 of *Lecture Notes in Computer Science*, pages 353–366. Springer-Verlag, 1996.

[172] G. Pesant and M. Gendreau. A Constraint Programming Framework for Local Search Methods. *Journal of Heuristics*, 5:255–279, 1999.

[173] L. S. Pitsoulis and M. G. C. Resende. Greedy Randomized Adaptive Search procedure. In P.M. Pardalos and M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 168–183. Oxford University Press, 2002.

[174] M. Prais and C. C. Ribeiro. Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal on Computing*, 12:164–176, 2000.

[175] N. V. N. Prasad, S. E. Lander, and V. R. Lesser. Cooperative learning over composite search spaces: Experiences with a multi-agent design system. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, volume 1, pages 68–73. AAAI Presss/MIT Press, 1996.

[176] S. Prestwich. Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. *Annals of Operations Research*, 115:51–72, 2002.

[177] N. J. Radcliffe. Forma Analysis and Random Respectful Recombination. In *Proceedings of the Fourth International Conference on Genetic Algorithms, ICGA 1991*, pages 222–229. Morgan Kaufmann Publishers, San Mateo, California, 1991.

[178] S. Rana. *Examining the role of local optima and schema processing in genetic search.* PhD thesis, Colorado State University, Fort Collins, Colorado, 1999.

[179] S. Rana and D. Whitley. Genetic algorithm behavior in the MAXSAT domain. In A. E. Eiben, T. Bck, M. Schoenauer, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN V, 5th International Conference*, Lecture Notes in Computer Science 1498, pages 785–794. Springer Verlag, 1998.

[180] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, 1973.

[181] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publishing, Oxford, England, 1993.

[182] C. R. Reeves. Landscapes, operators and heuristic search. *Annals of Operations Research*, 86:473–490, 1999.

[183] C. R. Reeves and J. E. Rowe. *Genetic Algorithms: Principles and Perspectives. A Guide to GA Theory.* Kluwer Academic Publishers, Boston (USA), 2002.

[184] C. Rego. Relaxed Tours and Path Ejections for the Traveling Salesman Problem. *European Journal of Operational Research*, 106:522–538, 1998.

[185] C. Rego. Node-ejection chains for the vehicle routing problem: Sequential and parallel algorithms. *Parallel Computing*, 27(3):201–222, 2001.

[186] M. G. C. Resende and T. A. Feo. A GRASP for satisfiability. In M.A. Trick, editor, *The Second DIMACS Implementation Chellenge*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science. 1995.

[187] M. G. C. Resende, L. S. Pitsoulis, and P. M. Pardalos. Approximate solution of weighted MAX-SAT problems using GRASP. In J. Gu and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 393–405. American Mathematical Society, 1997.

[188] M. G. C. Resende and C. C. Ribeiro. A GRASP for graph planarization. *Networks*, 29:173–189, 1997.

[189] C. C. Ribeiro and M. C. Souza. Variable neighborhood search for the degree constrained minimum spanning tree problem. *Discrete Applied Mathematics*, 118:43–54, 2002.

[190] A. Roli. Criticality and parallelism in GSAT. *Electronic Notes in Discrete Mathematics*, 9, 2001.

[191] A. Roli and C. Blum. Critical parallelization of local search for MAX-SAT. In *Proceedings of AI*IA 2001 - 7th Congress of Italian Association of Artificial Intelligence.*, 2001.

[192] A. Roli, C. Blum, and M. Dorigo. ACO for maximal constraint satisfaction problems. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 1, pages 187–191, Porto – Portugal, 2001.

[193] A. Roli and F. Zambonelli. Emergence of macro spatial structures in dissipative cellular automata. In *Proc. of ACRI2002: Fifth International Conference on Cellular Automata for Research and Industry*, volume 2493 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2002.

[194] S. J. Russell and P. Norvig. *Artificial Intelligence. A modern approach.* Simon & Schuster Company, 1995.

[195] M. Sampels, C. Blum, M. Mastrolilli, and O. Rossi-Doria. Metaheuristics for Group Shop Scheduling. In J.J. Merelo Guervós et al., editor, *Proceedings of PPSN-VII, Seventh International Conference on Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, pages 631–640. Springer Verlag, Berlin, Germany, 2002. Also available as technical report TR/IRIDIA/2002-07, IRIDIA, Université Libre de Bruxelles.

[196] A. Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proc.IJCAI-97*, pages 1254–1259, San Mateo CA (USA), 1997. Morgan Kauffman.

[197] T. Schiex and G. Verfaillie. Nogood recording fot static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[198] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *AAAI-92: Proceedings 10th National Conference on AI*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1993.

[199] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for local search. In *Proc. 12th National Conference on Artificial Intelligence, AAAI'94, Seattle/WA, USA*, pages 337–343, 1994.

[200] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press.

[201] P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In M. Maher and J.-F. Puget, editors, *Principle and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*. Springer, 1998.

[202] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer. A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97, 1997.

[203] L. Sondergeld and S. Voß. Cooperative intelligent search using adaptive memory techniques. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*, chapter 21, pages 297–312. Kluwer Academic Publishers, 1999.

[204] W. M. Spears, K. A. De Jong, T. Bäck, D. B. Fogel, and H. de Garis. An overview of evolutionary computation. In Pavel B. Brazdil, editor, *Proceedings of the European Conference on Machine Learning (ECML-93)*, volume 667, pages 442–459, Vienna, Austria, 1993. Springer Verlag.

[205] P. F. Stadler. Towards a theory of landscapes. In R. Lopéz-Peña, R. Capovilla, R. García-Pelayo, H. Waelbroeck, and F. Zertuche, editors, *Complex Systems and Binary Networks*, volume 461 of *Lecture Notes in Physics*, pages 77–163. Springer Verlag, Berlin, New York, 1995. Also available as SFI preprint 95-03-030.

[206] P. F. Stadler. Landscapes and their correlation functions. *Journal of Mathematical Chemistry*, 20:1–45, 1996. Also available as SFI preprint 95-07-067.

[207] A. Strohmaier. Multi-flip networks: Extending symmetric networks to real parallelism. Technical report aida-96-08, FG Intellektik, TU Darmstadt, 1996.

[208] T. Stützle. Iterated local search for the quadratic assignment problem. Technical report aida-99-03, FG Intellektik, TU Darmstadt, 1999.

[209] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications.* DISKI - Dissertationen zur Künstliken Intelligenz. infix, Sankt Augustin, Germany, 1999.

[210] T. Stützle and M. Dorigo. ACO algorithms for the quadratic assignment problem. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 33–50. McGraw-Hill, 1999.

[211] T. Stützle and M. Dorigo. A short convergence proof for a class of ACO algorithms. *IEEE Transactions on Evolutionary Computation*, 6(4):358–365, 2002.

[212] T. Stützle and H. H. Hoos. $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[213] G. Syswerda. Simulated Crossover in Genetic Algorithms. In L.D. Whitley, editor, *Proceedings of the second workshop on Foundations of Genetic Algorithms*, pages 239–255, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[214] http://www.tabusearch.net. Visited in January 2003.

[215] E. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[216] E. D. Taillard, L. M. Gambardella, M. Gendreau, and J. Potvi. Adaptive memory programming: A unified view of meta-heuristic. *European Journal of Operational Research*, 135:1–16, 2001.

[217] E-G. Talbi. A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics*, 8(5):541–564, 2002.

[218] M. Toulouse, T.G. Crainic, and B. Sansò. An experimental study of the systemic behavior of cooperative search algorithms. In S. Voß, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*, chapter 26, pages 373–392. Kluwer Academic Publishers, 1999.

[219] M. Toulouse, K. Thulasiraman, and F. Glover. Multi-level cooperative search: A new paradigm for combinatorial optimization and application to graph partitioning. In *Proceedings of the fifth International Euro-Par Conference on Parallel Processing*, Lecture Notes in Computer Science, pages 533–542, 1999.

[220] R. J. van Glabbeek. The linear time – branching time spectrum. the semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*. North-Holland, 2001.

[221] C. H. M. van Kemenade. Explicit filtering of building blocks for genetic algorithms. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 494–503, Berlin, 1996. Springer.

[222] P. J. M. Van Laarhoven, E. H. L. Aarts, and J. K. Lenstra. Job Shop Scheduling by Simulated Annealing. *Operations Research*, 40:113–125, 1992.

[223] M. Viroli and A. Omicini. Modeling agents as observable sources. *Journal of Universal Computer Science*, 8(4):423–451, 2002.

[224] M. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. Complex Adaptive Systems. MIT Press, 1999.

[225] S. Voß, S. Martello, I. H. Osman, and C. Roucairol, editors. *Meta-Heuristics - Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[226] C. Voudouris. *Guided Local Search for Combinatorial Optimization Problems*. PhD thesis, Department of Computer Science, University of Essex, 1997. pp. 166.

[227] C. Voudouris and E. Tsang. Guided Local Search. *European Journal of Operational Research*, 113(2):469–499, 1999.

[228] A. S. Wade and V. J. Rayward-Smith. Effective local search for the Steiner tree problem. *Studies in Locational Analysis*, 11:219–241, 1997. Also in *Advances in Steiner Trees*, ed. by Ding-Zhu Du, J. M.Smith and J.H. Rubinstein, Kluwer , 2000.

[229] B. W. Wah and Y. Shang. Discrete lagrangian-based search for solving MAX-SAT problems. In *Proc. 15th Int. Joint Conf. on Artificial Intelligence, IJCAI*, pages 378–383, Aug. 1997.

[230] T. Walsh. Search in a small world. In *proceedings of IJCAI99*, pages 1172–1177, 1999.

[231] T. Walsh. Search on high degree graphs. In *Proceedings of IJCAI-2001*, 2001.

[232] J.P. Watson, L. Barbulescu, A.E. Howe, and L.D. Whitley. Algorithm Performance and Problem Structure for Flow-Shop Scheduling. In *ixteenth National Conference on Artificial Intelligence (AAAI-99)*, 1999.

[233] R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 1998. Stanford University Bookstore.

[234] D.J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.

[235] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[236] G. Weiss, editor. *Multiagent Systems. A modern approach to distributed artificial intelligence*. The MIT Press, 2000.

[237] D. Whitley, S. Gordon, and K. Mathias. Lamarckian evolution, the Baldwin effect and function optimization. In *Proceeding of PPSN–III, Third International Conference on Parallel Problem Solving from Nature*, pages 6–15, Berlin, Germany, 1994. Springer Verlag.

[238] C. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:72–117, 1994.

[239] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[240] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

[241] M. Yagiura and T. Ibaraki. Efficient 2 and 3-flip neighborhood search algorithms for the MAX-SAT: Experimental evaluation. *Journal of Heuristics*, 7:423–442, 2001.

[242] M. Yagiura and T. Ibaraki. On metaheuristic algorithms for combinatorial optimization problems. *Systems and Computers in Japan*, 32(3):33–55, 2001.

[243] M. Yagiura and T.Ibaraki. Analyses on the 2 and 3-flip neighborhoods for the MAXSAT. *Journal of Combinatorial Optimization*, 3:95–114, 1999.

[244] M. Zlochin, M. Birattari, N. Meuleau, and M. Dorigo. Model-based search for combinatorial optimization. Technical Report TR/IRIDIA/2001-15, IRIDIA, Université Libre de Bruxelles, Belgium, 2001.