MAGMA:
A Multiagent Architecture for Metaheuristics

Michela Milano
DEIS - Università di Bologna
Viale Risorgimento, 2
40136 Bologna - Italia
email: mmilano@deis.unibo.it

Andrea Roli
DEIS - Università di Bologna
Viale Risorgimento, 2
40136 Bologna - Italia
email: aroli@deis.unibo.it

(Preprint)

Abstract

In this work we introduce a multiagent architecture (MAGMA, MultiAGent Metaheuristics Architecture) conceived as a conceptual and practical framework for metaheuristic algorithms. Metaheuristics can be seen as the result of the interaction among different kinds of agents: the basic architecture contains three levels each hosting one or more agents. Level-0 agents build solutions, level-1 agents improve solutions and level-2 agents provide the high level strategy. In this framework, classical metaheuristic algorithms can be smoothly accommodated and extended. The basic three level architecture can be enhanced with the introduction of a fourth level of agents, level-3 agents, coordinating lower level agents. With this additional level, MAGMA can also describe, in a uniform way, cooperative search and, in general, any combination of metaheuristics.

We describe the whole architecture, the structure of agents in each level in terms of tuples, and the structure of their coordination as a Labelled Transition System. We propose this perspective with the aim to achieve a better and clearer understanding of metaheuristics, obtain hybrid algorithms, suggest guidelines for a software engineering-oriented implementation and for didactic purposes. Some specializations of the general architecture will be provided in order to show that existing metaheuristics (e.g., GRASP, Ant Colony Optimization, Iterated Local Search, Memetic Algorithms) can be easily described in our framework. We describe cooperative search and large neighborhood search in the proposed framework exploiting level-3 agents. We show also that a simple hybrid algorithm, called Guided Restart Iterated Local Search, can be easily conceived as a combination of existing components in our framework.

Metaheuristics, agents, combinatorial optimization, constraint satisfaction.

1 Introduction

Metaheuristic techniques are used to solve hard problems (generally NP-hard problems) instead of, or in conjunction with, exact algorithms. When the problem size becomes very large, exact algorithms might be too computationally expensive. In these cases, approximate algorithms (which do not guarantee to find an optimal solution) are a very effective alternative to complete methods. Furthermore, the usefulness of metaheuristics has been recently proved in successful combination with complete methods (like complete search strategies in Constraint Programming [23, 52]). Metaheuristics are approximate algorithms which encompass and combine constructive methods, local search strategies, local optima escaping strategies and population-based search. They include, but are not restricted to: Tabu Search [30], Simulated Annealing [43], Evolutionary Computation [9], Memetic Algorithms [50], Scatter Search [29], Ant Colony Optimization [11, 18], GRASP [20], Iterated Local Search [61, 62] and Variable Neighborhood Search [32, 33]. Surveys and current research on metaheuristics can be found in [1, 6, 11, 68].

In this paper, we revisit metaheuristics in a multiagent perspective and provide a framework, MAGMA, where existing and hybrid metaheuristic algorithms can be smoothly accommodated. We first provide motivations of this paper in Section 2. In Section 3 we survey the most commonly used classifications of metaheuristics. Section 4 defines basic concepts for the definition of metaheuristics. Then we describe MAGMA in Section 5 and provide a formal definition of the agents and their coordination in Section 6. In Section 7 we show some specializations of MAGMA that describe some among the best known metaheuristic algorithms. We describe the MAGMA architecture of GRASP, Ant Colony Optimization, Iterated Local Search and Memetic Algorithms. In addition, cooperative search and Large Neighborhood Search are considered. We also describe a hybrid algorithm, called Guided Restart Iterated Local Search, which can be easily implemented in our framework. Finally, we present related works and some conclusions.

2 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 work, 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 [71]. 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 [16, 41, 49]. 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. We believe this is the first step toward the integration of metaheuristic components in open systems where the ever growing need of extra functionalities and integration with other types of software is strongly required.

3 Metaheuristic algorithms classifications

There are different ways to classify and describe metaheuristic algorithms. Perhaps, the most familiar and easiest is the description based on the algorithm origins. There are nature-inspired algorithms, like Genetic Algorithms and Ant Algorithms, and non nature-inspired ones like Tabu Search and Iterated Local Search. This classification is quite rough for two main reasons: *(i)* recent hybrid algorithms, e.g., memetic algorithms [50], do not completely fit neither class or, in a sense, they fit both at the same time, *(ii)* this classification is not helpful to compare algorithms.

Another way to classify metaheuristics is differentiating them between single and population heuristics. The first are commonly 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 (continuous or not) in the search space during the search process. Population metaheuristics, conversely, generate search processes which dynamically describe the evolution of a set of points in the search space. The most important population based methods are Evolutionary Computation, Memetic Algorithms, Scatter Search and Ant Colony Optimization.

Algorithm 1 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
```

A very interesting framework is called *Adaptive Memory Programming* (AMP, [65]). 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 [65], is reported in Algorithm 1.

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. For example, in [34] a framework for evolutionary computation algorithms is discussed, where evolutionary algorithms are described depending on seven main features: individual representation, evolution process, neighborhood structure, information sources, infeasibility, intensification and diversification.

4 Problem, Models and Fitness Landscape

In this section we introduce the problem classes metaheuristics usually attack and we briefly discuss some design issues of problem modelization. Furthermore, we introduce the fitness landscape, which is an important component of our definition of search process.

4.1 Search Problem Formalization

The widest class of problems metaheuristics deal with are Constraint Satisfaction Problems (CSPs) and Combinatorial Optimization Problems (COPs). They can be defined by:

- a set of variables, $X = \{x_1, \ldots, x_n\}$ representing problem entities;
- variable domains, D_i $\forall i = 1, \ldots, n$, i.e., values that variables can assume;
- constraints among variables, $c_j(Y_j)$, $Y_j \subseteq X, \forall j = 1, \ldots, m$ limiting the combinations of feasible assignments;
- an *objective function* to be optimized (minimized or maximized).

In CSPs, the objective function is constant, as they are solved by any feasible solution, whilst for COPs it is usually a non trivial function. There are important variants of these problems: MAX-CSPs [26], in which the objective is to maximize the number of satisfied constraints (or a weighted sum of them) and Multi-criteria COPs [31, 69] in which more than one criterion has to be optimized.

A *model* is a non-ambiguous and formal definition of the problem: which are the main entities, the constraints, the costs (or values) associated with configurations of entities and the goal.

Often the problem entities are directly modeled and used by the algorithm. The choice of a specific model is, in general, "algorithm-oriented"(see Figure 1), because it has to be tailored to the algorithm that will be used to solve the problem. This holds both for systematic constructive methods and metaheuristics. For example, a problem modeled via binary variables and linear inequalities will be effectively solved through Integer Programming techniques, while the same model is not in general the best one for a Constraint Programming finite-domain solver, even if it can sometimes be applied. In the metaheuristics context, if, for instance, simple genetic algorithms are used, solutions may be coded as binary strings. The model may also define an alternative objective function, for example by introducing penalties for infeasible solutions, or by using smoothing functions to shape the search landscape. Observe that the search process may also dynamically change models (e.g., by swapping between two different objective functions [3]).

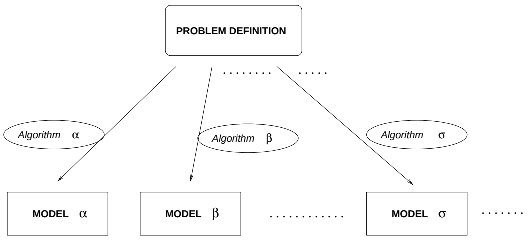

Figure 1: Conceptual process from the original problem to the representation used in the algorithm. The model definition is algorithm dependent: solution space, neighborhood structure and objective function have to be tailored to the chosen metaheuristics.

4.2 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 $\mathcal{N} : S \rightarrow 2^S$ 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*, $F: S \rightarrow \mathbb{R}^+$.

The Fitness Landscape (hereafter referred to as FL) can be interpreted as a graph (see Figure 2) in which nodes are solutions (with their

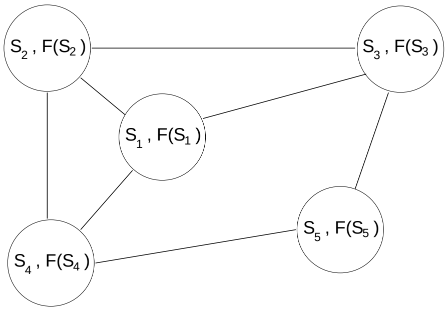

Figure 2: Example of undirected graph representing a fitness landscape. Each node is associated with a solution s_i and its corresponding fitness value $F(s_i)$. Arcs represent transition between states by means of φ. Undirected arcs correspond to symmetric neighborhood structure.

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) 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 is weakly optimally connected). Nevertheless, there are some exceptions of effective neighborhood structures which do not enjoy this property [51].

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 [39], 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 [19].

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 [39, 40]. 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 is theoretically supported by the *No Free Lunch Theorem* [72].

4.3 Examples

In this section we present some examples of the previously introduced concepts.

4.3.1 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

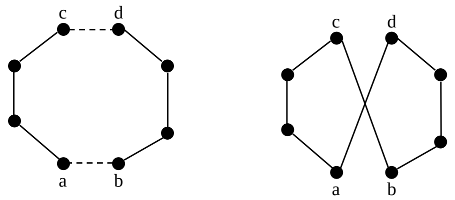

Figure 3: Example of 2-exchange move. The tour can be reconstructed in only one way.

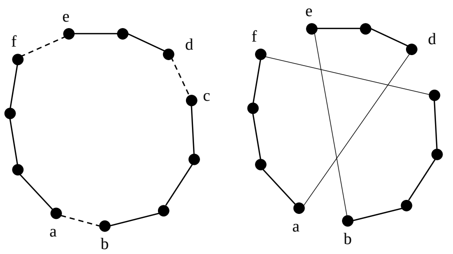

Figure 4: Example of 3-exchange move. After the removal of three arcs there are six possible ways to reconstruct the tour. In this figure just one of them is shown.

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 [38]. 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 3). In Figure 4 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.

Model 2

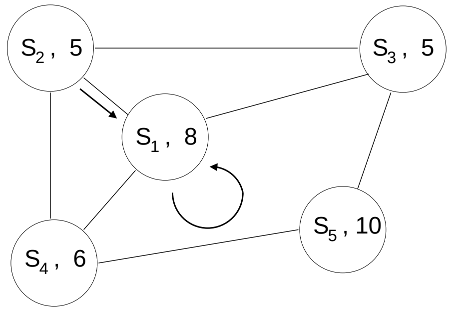

Figure 5: An example of hill-climbing local search on a FL. If the agent starts at node s_2 it moves toward the best neighbor s_1 whose fitness value is 8. From node s_1 there are no other possibilities to move, because its neighbors have lower fitness. The agent has reached a *local optimum* and cannot reach the *global optimum* s_5.

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 $3 \to 5 \to 4 \to 2 \to 1$. 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.

4.3.2 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[1]. Thus, feasible solution for the SAT problem are mapped in optimal solutions in the model.

5 MAGMA: MultiAGent Metaheuristic Architecture

The weak notion of *agent* [21, 56, 71, 73] 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 6), 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 7 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 6. 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 7 we will specialize the tuples on specific algorithms.

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

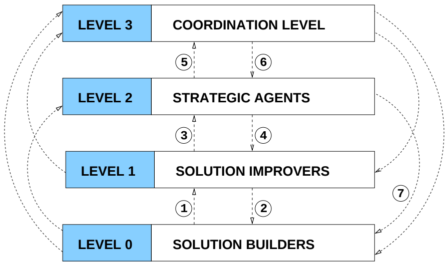

Figure 6: Multi-level architecture for metaheuristic algorithms. Note that only numbered arrows are indeed used by existing metaheuristics, while non numbered arrows can be used for extending existing algorithms.

5.1 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 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 7. 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 4.3.2 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 [12, 23, 52] 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 6) 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 [33] and in a new metaheuristic called Multi-Neighborhood Search [15]. 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 [30]). 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 6 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 7.6, 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.

6 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 [28]. LTS are mainly used to formally define the coordination among processes, but they are also applied in the context of multiagent systems [70].

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:

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

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.

7 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 (some examples have been described in Section 4.3). 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 [30] (for instance, a strategy called *strategic oscillation*) and in [3, 4], 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 [3, 4, 6, 8, 10].

The detailed description and discussion of the metaheuristic algorithms is beyond the scope of this paper and 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 2 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

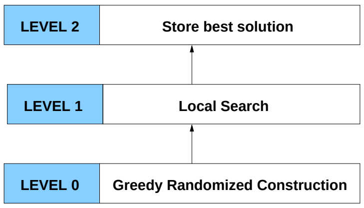

Figure 7: The Multi-level architecture version of GRASP.

7.1 Greedy Randomized Adaptive Procedure (GRASP)

We start our series of examples with a very simple algorithm: GRASP [20]. This algorithm is composed of a greedy construction procedure and a local search procedure (see the sketched Algorithm 2). 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 7 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 6. 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:

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

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

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

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

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

7.2 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is a well known population-based metaheuristic for designing algorithms for COPs [16, 17, 18]. 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

Algorithm 3 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

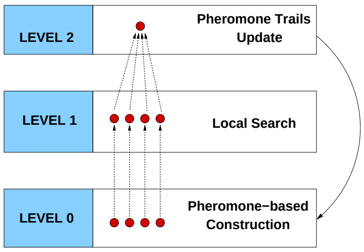

Figure 8: The Multi-level architecture version of ACO with local search.

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 [7, 18, 27, 55, 63, 64].

A generic ACO algorithm is sketched in Algorithm 3.

In Figure 8 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 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:

- `WAIT` $\xrightarrow{\text{receive}(\mathbf{T}, INIT)}$ `READY`

  the first iteration is performed on the basis of the initial values for pheromone
- `WAIT` $\xrightarrow{\text{receive}(\mathbf{T}, L2-agent)}$ `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` $\xrightarrow{\text{Execute}(\mathcal{S}_0)}$ `COMPLETED`
- `COMPLETED` $\xrightarrow{\text{send}(sol, L1-agent_i)}$ `WAIT`

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

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

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

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

7.3 Iterated Local Search (ILS)

Iterated Local Search (ILS) is a simple but powerful metaheuristic algorithm [61, 62]. 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

Algorithm 4 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

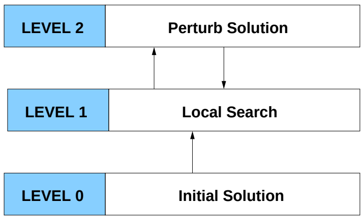

Figure 9: The Multi-level architecture version of ILS.

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 4.

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 9): 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 communication between LEVEL–1 and LEVEL–2, while LEVEL–0 participates just for the initialization, or for a random restart[2] (not considered in Figure 9). 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`
- `READY` $\xrightarrow{\text{Execute}(\mathcal{S}_0)}$ `COMPLETED`
- `COMPLETED` $\xrightarrow{\text{send(sol,L1-agent)}}$ `WAIT`

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

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

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

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

7.4 Memetic Algorithms

In this section, we propose the application of the MAGMA framework to Memetic Algorithms (MAs) [50]. 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 5), an initial population of solutions is generated; then every solution is improved by applying local search. The resulting

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

Algorithm 5 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

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 10 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):

- $\mathcal{S}_0$: 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.

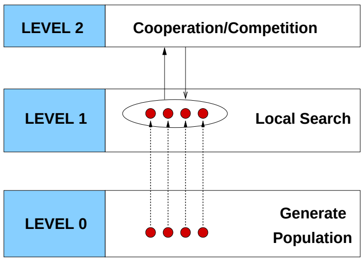

Figure 10: The Multi-level architecture version of Memetic Algorithms.

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`

7.5 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[3].

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. The description of the algorithm and experimental results are reported in the Appendix.

We believe that the use of MAGMA can clarify the interaction of algorithmic components, enabling the designer to build new combinations, starting from basic blocks and communication schemes.

7.6 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 [5, 13, 35, 36, 59, 66, 67] 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).

[3]Interesting diversification strategies developed for ACO can be useful also for the other metaheuristics can be found in [8].

We refer to [35] 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 [12, 23, 52]). 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 [57]).

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 [13]: 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 [42, 46].

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.

7.6.1 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, [25]) 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 [37, 47, 54, 60]. 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 7.4 and 7.1. 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 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 11). 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`

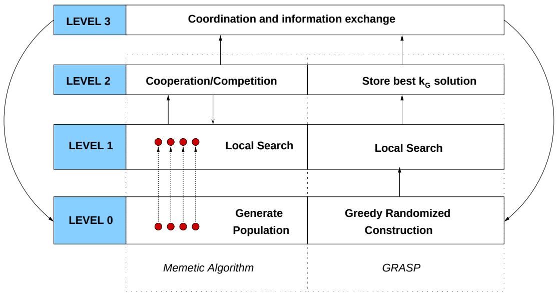

Figure 11: The Multi-level architecture version of Cooperative Search.

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.

7.6.2 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) [58].

The description of LNS is beyond the scope of the paper 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 [24] (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 12 a very simple LNS is represented in terms of MAGMA levels. An L3-agent coordinates two algorithms: the one solving the subproblem (Figure 12:left) and the one providing the first solution (Figure 12:right). We suppose that ILS (implemented as described in Section 7.3 with three agents), provides the first solution to be improved by means 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

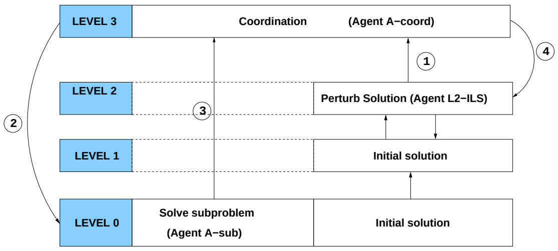

Figure 12: MAGMA version of an instance of Large Neighborhood Search.

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:

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

L1-agent of ILS:

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

L2-agent of ILS:

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

A-sub:

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

A-coord:

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

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.

8 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 [2, 12, 14, 22, 44, 48]. 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[4]: construction of the initial solution and further improvement. In [14] 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)

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

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 [45, 53, 71]. 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.

9 Discussion and Conclusion

In this paper, we have discussed a multiagent approach to metaheuristics. We have described the multi-level architecture MAGMA, where agents are characterized by different levels of abstraction and objectives. Agents are described through a uniform view of tuples containing a model and a strategy. Furthermore, we have provided a coordination model to formally define synchronization and communication among agents.

We have shown that MAGMA can describe existing metaheuristics by specializing the architecture general scheme. In addition, cooperative search, large neighborhood search and in general any combination of metaheuristics and complete algorithms can be described in MAGMA.

The basic characteristics of MAGMA are the multilevel structure and the organization as multiagent system. We believe that these two properties could enable the decomposition of algorithms into basic components and be useful in bridging the gap between *ad hoc* algorithm design and general metaheuristics software libraries/packages.

There are some relevant issues that should be mentioned. In MAGMA, all direct communications between levels are allowed. In the previous examples, only some among the possible communications have been exploited and many possibilities to combine and design new algorithms have to be investigated. The explicit presence of levels permits to clearly identify information flows. For example, algorithms which have a loop LEVEL–0 $\rightarrow$ LEVEL–1 $\rightarrow$ LEVEL–2 $\rightarrow$ LEVEL–0 exhibit a feedback loop. Feedback loops characterize self-adapting systems and emerging phenomena, e.g., ACO dynamics.

The communication patterns in MAGMA involve agents belonging to different levels. However, the communication between agents at the same level is possible too, by means of L3-agents. In fact, L3-agents can also provide simple communication links between agents at any level.

An interesting possibility is to use LEVEL–3 to perform resource management for the lower levels. For instance, a L3-agent can decide the priorities among L2-agents and allot more or less CPU time to agents depending on the requirements. This strategy could be useful in case of anytime algorithms, where as soon as a solution is required, it has to be provided. L3-agents can also select lower level agents as a function of the quality of the solutions provided.

Finally, we think MAGMA can be useful as a teaching tool, because it is structured in interacting software components and levels of abstraction. Indeed, recent successful approaches to teaching Artificial Intelligence are based upon the multiagent description of algorithms (see, for instance [56]).

Acknowledgments

We would like to thank Paolo Torroni, Andrea Omicini, Christian Blum and the anonymous reviewers for useful comments and suggestions on this work. This work was supported by the SOCS project, funded by the CEC, contract IST-2001-32530. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

10 Appendix

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 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 [29]).

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 1 results concerning MAXSAT instances with 1000 variables and 10000 clauses are reported[5]. 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[6].

Table 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%

References

[1] E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization.* Wiley-Interscience, 1997.

[2] 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*

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

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

USA '98), pages 33–45. IEEE, Piscataway, 1998.

[3] R. Battiti and M. Protasi. Reactive Search, a history-base heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 1997.

[4] 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.

[5] V. Bachelet and E.G. Talbi. Cosearch: a co-evolutionary metaheuristics. In *Proceedings of Congress on Evolutionary Computation – CEC'2000*, pages 1550–1557, 2000.

[6] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. Technical Report IRIDIA/01-13 Université Libre de Bruxelles, Belgium, 2001.

[7] 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.

[8] 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, *Ant Algorithms Third International Workshop, ANTS 2002*, volume 2463 of *Lecture Notes in Computer Science*, pages 14–27. Springer, 2002.

[9] P. Calegari, G. Coray, A. Hertz, D. Kobler, and P. Kuonen. A taxonomy of evolutionary algorithms in combinatorial optimization. *Journal of Heuristics*, 5:145–158, 1999.

[10] I. Charon and O. Hudry. The noising methods: A generalization of some metaheuristics. *European Journal of Operational Research*, 135:86–101, 2001.

[11] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimization*. Advanced topics in computer science series. McGraw-Hill, 1999.

[12] 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.

[13] 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.

[14] 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*, volume 1, pages 287–292, Porto – Portugal, 2001.

[15] 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). Gent (Belgium), 21-23 August 2002.

[16] 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.

[17] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.

[18] 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.

[19] A.E. Eiben and Z. Ruttkay. Constraint satisfaction problems. In T. Bäck, D. Fogel, and M. Michalewicz, editors, *Handbook of*

Evolutionary Computation. IOP Publishing Ltd. and Oxford University Press, 1997.

[20] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[21] J. Ferber. Multi-agent systems : an introduction to distributed artificial intelligence. Addison-Wesley, 1999.

[22] A. Fink and S. Voss. Reusable metaheuristic software components and their application via software generators. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 2, pages 637–641, Porto – Portugal, 2001.

[23] F. Focacci and F. Laburthe and A. Lodi. Local Search and Constraint Programming. In *Handbook of Metaheuristics*, F. Glover and G. Kochenberger editors, Kluwer Academic Publishers, Norwell, MA, 2002.

[24] F. Focacci, A. Lodi and M. Milano A Hybrid Exact Algorithm for the TSPTW. INFORMS *Journal of Computing*, 14(4):403–417,2002.

[25] C. Fonlupt, D. Robilliard, P. Preux, and E.G. Talbi. Fitness landscapes and performance of meta-heuristics. In Stefan Voss, Silvano Martello, Ibrahim Osman, and Catherine Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization.* Kluwer Academic, 1999.

[26] E. Freuder and R. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[27] 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.

[28] R.J. van Glabbeek. The linear time – branching time spectrum. The semantics of concrete, sequential processes. In *Handbook of Process Algebra.* J.A. Bergstra, A. Ponse and S.A. Smolka editors. North-Holland, 2001.

[29] 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.

[30] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

[31] M. Hansen. Tabu search for multiobjective optimization: MOTS. 13th International Conference on Multiple Criteria Decision Making, 1997.

[32] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In Stefan Voss, Silvano Martello, Ibrahim Osman, and Catherine Roucairol, editors, *Meta-heuristics: advances and trends in local search paradigms for optimization*, chapter 30, pages 433–458. Kluwer Academic Publishers, 1999.

[33] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of operational Research*, 130:449–467, 2001.

[34] A. Hertz and D. Kobler. A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, 126:1–12, 2000.

[35] T. Hogg and A. Huberman. Better Than the Best: The Power of Cooperation. *SFI 1992 Lectures in Complex Systems*, 163–184, Addison-Wesley, 1993.

[36] T. Hogg and C.P. Williams. Solving the really hard problems with cooperative search. In *Proceedings of AAAI93*, pages 213–235. AAAI Press, 1993.

[37] W. Hordijk. A measure of landscapes. *Evolutionary Computation*, 4(4), 1996.

[38] 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.

[39] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, Univ. of New Mexico, Albuquerque, NM, 1995.

[40] T. Jones. One operator, one landscape. Santa Fe Institute Technical Report 95-02-025, Santa Fe Institute, 1995.

[41] L.P. Kaelbing, M.L. Littman and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial intelligence Research*, 4:237–285, 1996.

[42] S.A. Kauffman and W. Macready. Technological evolution and adaptive organizations. *Complexity*, 26(2):26–43, March 1995.

[43] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science, 13 May 1983*, 220(4598):671–680, 1983.

[44] 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.

[45] 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.

[46] W.G. Macready and S.A. Kauffman A.G. Siapas. Criticality and parallelism in combinatorial optimization. *Science*, 271:56–59, January 1996.

[47] 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.

[48] L. Michael and P. van Hentenryck. Localizer: A modeling language for local search. *INFORMS Journal of Computing*, 11:1–14, 1999.

[49] P. Mills and E. Tsang. Guided Local Search for solving SAT and weighted MAX-SAT Problems. In I. Gent, H. van Maaren, and T. Walsh Eds., SAT2000, pp.89–106. IOS Press. 2000.

[50] P. Moscato. Memetic algorithms: A short introduction. In F. Glover D. Corne and M. Dorigo, editors, *New Ideas in Optimization*. McGraw-Hill, 1999.

[51] Nowicki, E. and Smutnicki, C. A fast taboo search algorithm for the job-shop problem. *Management Science*, volume 42, number 2, pages 797–813, 1996.

[52] G. Pesant and M. Gendreau. A View of Local Search in Constraint Programming. In E.C. Freuder, editor, *Principles and Practice of Constraint Programming - CP'96*. LNCS 1118, pages 353–366. Springer-Verlag, Berlin Heidelberg, 1996.

[53] N.V.N. Prasad, S.E. Lander, and V.R. Lesser. Cooperative learning over composite search spaces: Experiences with a multiagent design system. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, volume 1, pages 68–73. AAAI Press/MIT Press, 1996.

[54] Reeves, C. Landscapes, operators and heuristic search. *Annals of Operations Research*, volume 86, pages 473–490, 1999.

[55] A. Roli, C. Blum, and M. Dorigo. ACO for maximal constraint satisfaction problems. In *Proceedings of MIC'2001 – Metaheuristics International Conference*, volume 1, pages 187–191, Porto – Portugal, 2001.

[56] S.J. Russell and P. Norvig. *Artificial Intelligence. A modern approach.* Simon & Schuster Company, 1995.

[57] T. Schiex and G. Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, volume 3, number 2, pages 187–207, 1994.

[58] 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.

[59] L. Sondergeld and S. Voss. Cooperative intelligent search using adaptive memory techniques. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Metaheuristics: advances and trends in local*

search paradigms for optimization, chapter 21, pages 297–312. Kluwer Academic Publishers, 1999.

[60] P.F. Stadler. Landscapes and their correlation functions. *J. Math. Chem.*, 20:1–45, 1996. SFI preprint 95-07-067.

[61] T. Stützle. Iterated local search for the quadratic assignment problem. Technical report aida-99-03, FG Intellektik, TU Darmstadt, 1999.

[62] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications.* DISKI - Dissertationen zur Künstliken Intelligenz. infix, 1999.

[63] 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.

[64] T. Stützle and H. H. Hoos. $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[65] E.D. Taillard, L.M. Gambardella, M. Gendreau, and J. Potvin. Adaptive memory programming: A unified view of metaheuristics. *European Journal of Operational Research*, 135:1–16, 2001.

[66] 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.

[67] M. Toulouse, K. Thulasiraman and F. Glover. Multi-Level Cooperative Search: A New Paradigm for Combinatorial Optimization and an Application to Graph Partitioning. Proceedings of Euro-Par'99, 1999.

[68] 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, 1999.

[69] Luc N. Van Wassenhove and Ludo F. Gelders. Solving a bicriterion scheduling problem. *European Journal of Operational Research*, 4(1):42–48, 1980.

[70] M. Viroli and A. Omicini. Modeling Agents as Observable Sources. *Journal of UniversalComputer Science*, 8(4):423–451, 2002.

[71] G. Weiss, editor. *Multiagent Systems. A modern approach to distributed artificial intelligence.* The MIT Press, 2000.

[72] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[73] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.