UNIVERSITATEA „POLITEHNICA“ din BUCUREȘTI

Facultatea de Automatică şi Calculatoare

Catedra de Calculatoare

TEZĂ DE DOCTORAT

Fenomene Emergente în Sistemele Bazate pe Agenți

Emergent Phenomena in Agent-Based Systems

Autor: Ing. Șerban Iordache

Conducător de doctorat: Prof. dr. ing. Florica Moldoveanu

BUCUREȘTI

Contents

1. INTRODUCTION 4

1.1. Motivation 4

1.2. Original contributions 5

1.3. Scientific publications in connection with this thesis 6

1.4. Thesis outline 7

2. BACKGROUND 9

2.1. Agent-based systems 9

2.2. Emergence 11

2.3. Cellular automata 12
2.3.1. One-dimensional cellular automata 13
2.3.2. Two-dimensional cellular automata 17

2.4. Ant colony optimization 19
2.4.1. The behavior of foraging ants 19
2.4.2. Ant colony optimization for the traveling salesman problem 21
2.4.3. The ant colony optimization metaheuristic 22

3. A DIFFERENT VIEW ON EMERGENCE 24

4. METFREM – A META-FRAMEWORK FOR THE STUDY OF EMERGENCE 32

4.1. Goals and objectives of MetFrEm 34

4.2. Related work 35

4.3. Concepts and structure of MetFrEm 36

4.4. A formal description of MetFrEm 37

4.5. The algorithmic structure 40

4.6. Analysis of MetFrEm 41

4.7. Modeling various systems in MetFrEm 42
4.7.1. Case study: Cellular automata 42
4.7.2. Case study: Ant Colony Optimization 46
4.7.3. Case study: Predator-Prey models 50

5. THE CONSULTANT-GUIDED SEARCH METAHEURISTIC 53

5.1. Introduction 53

5.2. The CGS Metaheuristic 54
5.2.1. Method Description 54
5.2.2. The Metaheuristic 55
5.2.3. Method Analysis 57

5.3. Modeling CGS in MetFrEm 58

5.4. Positioning of CGS 60

5.5. CGS applied to the Traveling Salesman Problem 61
5.5.1. The CGS-TSP algorithm 62
5.5.2. Implementation 65
5.5.3. Parameter tuning 65
5.5.3.1. CGS-TSP tuning 65
5.5.3.2. ACS tuning 66
5.5.3.3. MMAS tuning 66
5.5.4. Experimental results 67

5.6. CGS combined with local search for the Traveling Salesman Problem 70
5.6.1. Local search 70
5.6.2. Applying local search to CGS-TSP 71
5.6.3. CGS-TSP with confidence 71
5.6.4. Experimental setup 71
5.6.5. Experimental results 72

5.7. CGS applied to the Quadratic Assignment Problem 75
5.7.1. The CGS-QAP algorithm 75
5.7.2. Experimental setup 76
5.7.3. Experimental results 77
5.7.4. Variants of the CGS-QAP algorithm 79
5.7.4.1. The CGS_V-QAP algorithm 79
5.7.4.2. The CGS_2-QAP algorithm 79
5.7.4.3. The CGS_{2V}-QAP algorithm 80

5.8. CGS applied to the Generalized Traveling Salesman Problem 80
5.8.1. The local-global approach to the Generalized Traveling Salesman Problem 80
5.8.2. The Hybrid Algorithm for the GTSP 82
5.8.3. Strategy and solution construction 85
5.8.4. An algorithm variant using confidence 86
5.8.5. Local Search 87
5.8.6. Experimental setup 88
5.8.7. Computational results 89

5.9. Conclusions and future work 92

6. CONCLUSIONS 94

APPENDIX A.
AGSYSLIB - A SOFTWARE TOOL FOR AGENT-BASED PROBLEM SOLVING 96

A.1. The AgSysLib framework and library 96

A.2. Agent-based problem solving with AgSysLib 97
A.2.1. The “Strange Collisions” problem 97
A.2.2. Configuration 98
A.2.3. Listeners 100
A.2.4. Experimentation and tuning 101
A.2.5. Debugging 106

A.3. Conclusions 107

REFERENCES 108

1. INTRODUCTION

1.1. Motivation

Emergence is one of the most intriguing phenomena exhibited by complex systems. It consists in the appearance of system-level features that do not characterize the elements composing the considered system. Therefore, these new features are sometimes described as unexpected or surprising. An emergent phenomenon occurs when the system components are governed by simple rules, but the macroscopic behavior resulting from their interaction is complex. In many cases it is very difficult or even impossible to predict this behavior by analyzing the system components. Often, the concept of emergence is summarized by the phrase "the whole is greater than the sum of its parts". Emergence is exhibited by decentralized systems having no global control structure, where the observed behavior is a consequence of the local interactions between independent entities, generically called agents.

A classic example of emergence is offered by a colony of social insects, such as ants, termites or bees. Such a colony can be seen as a highly adaptive macro-organism, although each individual is an unintelligent insect, which uses only simple rules to respond to stimuli in the environment. Some species of termites are able to build mounds reaching a height of several meters. Due to a complicated system of tunnels and chambers that provides passive cooling, the temperature inside these mounds remains almost constant, regardless of the outside temperature. Architects and engineers have taken inspiration from this model in order to design buildings that regulate the temperature and humidity using only natural means [4]. However, a termite mound is not designed by architects and the building process is not coordinated by engineers. Each termite acts according to a simple algorithm, but at the colony level the result is simply amazing.

Emergent phenomena can be observed in a large variety of systems. Examples include the formation of ripple patterns in a sand dune, the occurrence of traffic jams, the price setting in a decentralized market, the growth of a snowflake or the formation of a hurricane. However, the most striking instances of emergence can be found in biological systems: a living cell emerges from the interaction of its constituent molecules; the immune system, which is able to protect the organism against diseases, emerges from the combined action of several types of lymphocytes; brain cells self-organize into a complex neural network, which produces intelligence and even consciousness.

All matter in our universe is composed of elementary particles. Therefore, even the most complex phenomena are ultimately the result of the interaction of a few types of elementary particles. How is this possible? How can these elementary particles self-organize into increasingly higher structures? How can life emerge from inanimate matter? How can goal-directed behavior emerge from particles that have no goals? And how can intelligence and consciousness emerge from particles that possess no intelligence?

While these questions are significant enough to justify the study of emergence, there is another driving force behind the research presented in this thesis: the problem of engineering emergent behavior. Due to the growing number of decentralized, agent-based applications,

this represents an important issue, for which no generally accepted methodology is currently available. Traditional software engineering does not take advantage of the emergent behavior exhibited by agent-based applications. Most agent-oriented methodologies regard emergent phenomena as undesired and try to suppress the "unexpected" behavior by constraining the actions of individual agents. Of course, it would be preferable to design applications that make use of the emergent behavior instead of avoiding it, but this is a difficult task, taking into consideration the apparent unpredictability of emergent phenomena.

In the last years, researchers have taken inspiration from nature in order to design algorithms that produce certain desired emergent behavior. Many natural systems are able to adapt to dynamical environments and can perform efficiently certain tasks for which no feasible conventional algorithms are known. It is therefore tempting to mimic such natural systems in order to obtain a similar behavior. An example is offered by Ant Colony Optimization [45], which is a heuristic method inspired by ant foraging. In their way back from a food source, ants deposit small amounts of chemicals called pheromones. These pheromones can be sensed by other foragers, which are more likely to follow the trails having a stronger concentration of pheromones. This simple foraging strategy leads eventually to the discovery of the shortest path between the nest and the food source. In other words, the shortest path emerges from the interaction between ants and their environment. Ant Colony Optimization is inspired by this emergent phenomenon and it has been initially used to solve the traveling salesman problem. Since then, it has been successfully applied to many other combinatorial optimization problems and it has been also adapted for continuous optimization problems. One notable application of this method is to dynamic network routing problems [37]. These are difficult real-world problems, because the traffic load, the network topology and other characteristics of the network vary in time.

Although nature is a powerful source of inspiration, it is not always possible to find a natural system that exhibits a particular emergent behavior. Moreover, it is not sufficient to identify an appropriate natural system, but it is also necessary to understand its working, in order to mimic it. Therefore, an important question is how to design an agent-based system that exhibits a desired emergent behavior, when no source of inspiration can be found in nature. In this thesis, we explore both theoretical and practical approaches to tackle this problem.

We are mainly interested in engineering complex, adaptive behavior, similar to that exhibited by living organisms, because it is very difficult to obtain such behavior using traditional software engineering techniques. A main hypothesis that guides our research is that in order to engineer such behavior, it is preferable to focus on heterogeneous systems with simple agents than to consider homogenous systems with complex agents.

1.2. Original contributions

There are three main contributions of this thesis:

- **A mathematical formalism of emergence in agent-based systems.** Emergent phenomena are often described as unexpected, surprising or hard to predict. Therefore, many definitions of emergence involve a certain degree of subjectivity. Other definitions give a rigorous description of the properties characterizing emergent phenomena, but they may not capture all aspects of emergence, or they may require very complex computations in order to decide whether or not a certain behavior is emergent. We take a different view on emergence, which allows us to provide a definition that is both objective and suitable for practical purposes. Our formalism is based on the idea that a definition of emergent phenomena should only be concerned

with how these phenomena arise and it should not address the properties of the emergent phenomena.

- **MetFrEm – a meta-framework for the study of emergence.** In order to study emergent phenomena in a rigorous manner, we introduce a meta-framework called MetFrEm, which allows the modeling of various algorithmic frameworks comprising a population of interacting agents. MetFrEm favors the modeling of highly heterogeneous decentralized systems with agents that follow simple rules.
- **The Consultant-Guided Search (CGS) metaheuristic.** We propose a swarm intelligence metaheuristic that makes use of the emergent behavior exhibited by a population of interacting virtual persons. We apply this metaheuristic to several classes of combinatorial optimization problems and report the experimental results, which show that CGS is able to achieve state-of-the-art performance:
  - *the Traveling Salesman Problem (TSP)* - Our experiments with and without local search show that CGS clearly outperforms the two best performing Ant Colony Optimization algorithms for the TSP: Ant Colony System [43] and MAX-MIN Ant System [106].
  - *the Quadratic Assignment Problem (QAP)* - Our CGS algorithm for the QAP is significantly better than MAX-MIN Ant System [104], which is currently the best Ant Colony Optmization algorithm for this class of problems.
  - *the Generalized Traveling Salesman Problem (GTSP)* - Computational results show that there is no statistical significant difference between our algorithm and the memetic algorithm of Gutin and Karapetyan [55], which is currently the best published heuristic for the GTSP.

In addition, the work presented in this thesis has led to the creation of three open source software packages, which are of practical importance for the research community:

- **AgSysLib** (http://agsyslib.sourceforge.net/). AgSysLib is a software tool for agent-based problem solving. It assists users in all aspects related to the design, implementation, debugging and tuning of agent-based algorithms. AgSysLib is both a framework and a library and it features a component-based architecture, which permits to build algorithms in a modular way and facilitates the experimentation and analysis of different variants of an algorithm.
- **SwarmTSP** (http://swarmtsp.sourceforge.net/). SwarmTSP is a Java library of swarm intelligence algorithms for the Traveling Salesman Problem (TSP) and for the Generalized Traveling Salesman Problem (GTSP). It implements all Consultant-Guided Search algorithms for the TSP and GTSP proposed in this thesis, as well as several Ant Colony Optimization algorithms: Ant System, Ant Colony System, MAX-MIN Ant System, Elitist Ant System, Rank-Based Ant System and Best-Worst Ant System.
- **SwarmQAP** (http://swarmqap.sourceforge.net/). SwarmQAP is a Java library of swarm intelligence algorithms for the Quadratic Assignment Problem (QAP). It implements all Consultant-Guided Search algorithms for the QAP proposed in this thesis, as well as the MAX-MIN Ant System algorithm.

1.3. Scientific publications in connection with this thesis

Iordache, S. *Consultant-Guided Search - A New Metaheuristic for Combinatorial Optimization Problems*. In: GECCO 2010: Proceedings of the 12th Genetic and Evolutionary Computation Conference, Portland, Oregon, USA, ACM Press, 2010, pp. 225-232 [65] (**nominated for best paper award**).

Iordache, S. *Consultant-Guided Search Algorithms with Local Search for the Traveling Salesman Problem*. In: 11th International Conference Parallel Problem Solving from Nature - PPSN XI. LNCS 6239, Krakow, Poland, Springer, 2010, pp. 81-90 [63].

Iordache, S. *Consultant-Guided Search Algorithms for the Quadratic Assignment Problem*. In: Hybrid Metaheuristics - 7th International Workshop, HM 2010. LNCS 6373, Vienna, Austria. Springer, 2010, pp. 148-159 [61].

Iordache, S., Moldoveanu, F. *AgSysLib - A Software Tool for Agent-Based Problem Solving*. In: Scientific Bulletin of "Politehnica" University of Bucharest, C Series (Electrical Engineering), vol. 73, issue 2, ISSN 1454-234x, 2011 [66].

Iordache, S. *A Framework for the Study of the Emergence of Rules in Multiagent Systems*, In: Katalinic, B. (Ed.), Proceedings of the 20th International DAAAM Symposium, Vienna, Austria, ISSN 1726-9679, 2009, pp. 1285-1286 [60].

Iordache, S., Pop, P.C. *An Efficient Algorithm for the Generalized Traveling Salesman Problem*. In: A. Quesada-Arencibia et al. (Eds.), Proceedings of the 13-th International Conference on Computer Aided Systems Theory (EUROCAST 2011), Las Palmas de Gran Canaria, Spain, ISBN: 978-84-693-9560-8, 2011, pp. 264-266 [67].

Iordache, S. *Consultant-Guided Search combined with local search for the traveling salesman problem*. In: GECCO 2010 companion: Proceedings of the 12th annual conference companion on Genetic and evolutionary computation. ACM Press, 2010, pp. 2087-2088 [64].

Iordache, S. *Consultant-Guided Search algorithms for the quadratic assignment problem*. In: GECCO 2010 companion: Proceedings of the 12th annual conference companion on Genetic and evolutionary computation. ACM Press, 2010, pp. 2089-2090 [62].

Pop, P.C., **Iordache, S**. *A Hybrid Heuristic Approach for Solving the Generalized Traveling Salesman Problem*. In: GECCO 2011: Proceedings of the Genetic and Evolutionary Computation Conference, Dublin, Ireland, ACM Press, 2011 (accepted) [91].

1.4. Thesis outline

Chapter 2 outlines the main concepts and techniques relevant to the content of this thesis. After a short introduction of agent-based systems, we present different aspects of the notion of emergence and we discuss several emergent phenomena exhibited by cellular automata. Then, we introduce Ant Colony Optimization, as an example of a problem solving technique that makes use of emergent behavior.

In Chapter 3, we develop a mathematical formalism for the study of emergence, which puts emergence in an agent-oriented context, consistent with the frame imposed by the problem of engineering emergent behavior.

In Chapter 4, we propose a meta-framework for the study of emergence, called MetFrEm, which can be used to describe various algorithmic frameworks comprising a population of interacting agents. The design goals of our meta-framework reflect the main objectives and hypotheses of this thesis. After an intuitive description of the concepts and structure of MetFrEm, we provide a formal description of this meta-framework. Then, we illustrate by means of a few case studies how various systems can be modeled in MetFrEm.

In Chapter 5, we introduce Consultant-Guided Search, a new metaheuristic for combinatorial optimization problems, inspired by the possibility to view the interactions in MetFrEm from

the perspective of clients that receive advice from consultants. We apply this metaheuristic to a few classes of problems (the traveling salesman problem, the quadratic assignment problem and the generalized traveling salesman problem) and we compare the results with those obtained by state-of-the-art algorithms.

Appendix A describes AgSysLib, a software tool that we have developed in order to assist in agent-based problem solving. We identify the difficulties encountered during the design, implementation, debugging and tuning of a new agent-based algorithm and we show how this tool helps in overcoming them. AgSysLib is both a library and a framework and it has played a major role in the development of the algorithms presented in Chapter 5.

2. BACKGROUND

In this chapter, we introduce the basic concepts of agent-based systems and emergent phenomena, and we present a few examples of agent-based systems exhibiting emergent behavior. We limit our discussion to those aspects that are relevant to the content of this thesis and we encourage the reader to consult the references for further details.

2.1. Agent-based systems

Put simply, an agent-based system is a system of interacting agents. Therefore, in order to define an agent-based system, it is necessary to clarify what is meant by the notion of *agent*. There is, however, no generally accepted definition for this term. Agent-based systems play a central role in various fields and, usually, the meaning ascribed to the term *agent* differs from field to field and even within the same field. In the next paragraphs, we present the most important research areas dealing with agent-based systems and we discuss the interpretations they give to the notion of agent. We should mention that the boundaries between these research areas are not always clear. They often overlap and, sometimes, positioning a research in a specific area is rather a matter of perspective, reflecting the point of view adopted by a researcher.

Most research concerned with agent-based systems can be subsumed under the field of *artificial intelligence*. This is a vast research field and it is difficult to define the concept of *agent* in such a general context. However, Russel and Norvig [96] offer the following definition: "An *agent* is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors".

Agent-Based Modeling and Simulation (ABMS) is concerned with creating representations of existing complex systems in terms of interacting agents. These agent-based models can be used to simulate the evolution of the corresponding complex systems, to predict their behavior, to test hypotheses about these systems or to gain insight into their underlying mechanisms. Macal and North [80] consider that for the purposes of ABMS, the defining characteristics of an agent are:

- an agent is autonomous and self-directed.
- an agent is modular or self-contained.
- an agent is social, interacting with other agents.

The two authors of [80] also identify a few additional properties that are frequently associated with agents, but which are not necessarily a defining characteristic of them:

- an agent may live in an environment.
- an agent may have explicit goals that drive its behavior.
- an agent may have the ability to learn and adapt its behaviors based on its experiences.
- an agent may have resource attributes that indicate its current stock of one or more resources.

ABMS has applications in a variety of areas, such as biology [89], supply chains [73], agent-based social simulation [31] or agent-based computational economics [112]. Most systems of interest are *complex adaptive systems* (CAS) [83]. These are systems that "change and reorganize their component parts to adapt themselves to the problems posed by their surroundings" [56]. In the context of CAS, "*agents* are semi-autonomous units that seek to maximize their fitness by evolving over time" [40]. The ability to adapt and to learn from previous experiences is considered a major characteristic of an agent in CAS. However, we should note that some complex systems are adaptive, although their agents are governed by hard-coded rules. One example is an ant colony, which is able to adapt to changes in the environment, although ants are not intelligent and they act based on fixed rules. These rules do not change during the life of the ants, but the ant colony exhibits a complex, adaptive behavior. An important research direction in CAS is the study of the complex emergent phenomena produced by the interaction of agents following relatively simple rules.

A large number of definitions for the concept of *agent* have been offered in the field of *multi-agent systems* (MAS). They differ, sometimes significantly, in the characteristics ascribed to an agent. In general, these definitions impose more capabilities than in the case of ABMS and, frequently, the main abstraction used is that of an *intelligent agent*. Wooldridge [127] proposes the following definition:

> An *agent* is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.

Frequently, agents are only indirectly defined, by enumerating their required characteristics. Wooldridge and Jennings [128] identify and examine two major usages of the term *agent*. The first one, referred as the "weak notion of agency" is characterized by the following properties:

- autonomy: agents can control their own actions.
- social ability: agents interact by means of a communication language.
- reactivity: agents can sense and react to changes in the environment.
- pro-activeness: agents can take the initiative to interact.

A more restrictive meaning of the term *agent* is given by the "strong notion of agency", which implies the existence of mental traits such as knowledge, beliefs, desires, intentions, obligations [98] or even emotions [7].

One characteristic that appears in one form or another in almost all definitions of an agent is that of *autonomy*. This is another concept for which no generally accepted definition exists. A classification of different forms of autonomy proposed in the literature is given in [20]. Autonomy represents the central concept of a recent bottom-up paradigm called *autonomy oriented computing* [75], which refers to agents as *autonomous entities* and defines autonomy as follows:

> *Autonomy* of an entity refers to its condition or quality of being self-governed, self-determined, and self-directed. It guarantees that the primitive behavior of an entity is free from the explicit control of other entities.

For some researchers, autonomy has a more specific meaning than that conveyed by the above definition. For example, d'Inverno and Luck [38] relate autonomy with the existence of *motivations*, for which they give the following definition:

> A *motivation* is any desire or preference that can lead to the generation and adoption of goals and that affects the outcome of the reasoning or behavioural task intended to satisfy those goals.

Motivations are higher-level components that can generate *goals*. The two authors exemplify the difference between motivations and goals by discussing how the *goal* of robbing a bank may have greed as *motivation*. They introduce a hierarchy of concepts starting with the general notion of *entity* and ending with that of *autonomous agent* [38]:

- An *entity* is something that comprises a non-empty set of attributes, a set of actions, a set of goals and a set of motivations.
- An *object* is an entity with a non-empty set of actions.
- An *agent* is an object with goals.
- An *autonomous agent* is an agent with motivations.

In the absence of a generally accepted definition, it is necessary to specify what we mean by *agent* in the context of this thesis. Since we are mainly interested in the complex behavior that emerges from the interaction of agents governed by only simple rules, we need a definition that imposes very few restrictions on the agents. In our research, we regard as agents even very simple entities, such as the cells of a cellular automaton. The only requirement we impose to an agent is to act autonomously, that is, to decide by itself what actions to take, without receiving commands from an external entity. In chapter 3, we develop a formalism for the study of emergence, which allows us to give rigorous definitions for the major concepts related to agent-based systems.

2.2. Emergence

Emergence consists in the appearance of system-level features that do not characterize the elements composing the considered system. Examples include the emergence of life from inanimate matter or the emergence of consciousness from the interaction of a large number of neurons. Another example is an ant colony, where ants returning from a food source lay pheromones on the ground on their way back to the nest. Other ant foragers are able to sense these pheromones and are attracted by the trails with higher concentrations of pheromone. In time, pheromone trails corresponding to the shortest paths between nest and food sources emerge from the collective behavior of individual ants.

While it is easy to grasp intuitively the meaning of emergence, it is nonetheless difficult to give a rigorous definition of it. In this respect, emergence is similar to concepts such as intelligence, consciousness or life.

Different people ascribe different meanings to this term, but, in general, they fall into two distinct classes: strong emergence and weak emergence. For Chalmers [22], a system exhibits *strong emergence* if a "high-level phenomenon arises from the low-level domain, but truths concerning that phenomenon are not *deducible* even in principle from truths in the low-level domain". Consciousness is often presented as a potential instance of strong emergence. However, the possibility of strong emergence is a subject of debate in philosophy, because its existence would require rethinking our conception of nature. Most scientists involved in natural sciences reject this kind of emergence and they refer instead to weak emergence in their discussions about emergence. Chalmers [22] considers that a system exhibits *weak emergence* if a "high-level phenomenon arises from the low-level domain, but truths concerning that phenomenon are *unexpected* given the principles governing the low-level domain".

Beside strong and weak emergence, Bedau [9] identifies a third kind of emergence, which he calls *nominal emergence*. This is characterized by the existence of "a macro property that is the kind of property that cannot be a micro property". For example, the fluidity and transparency of water are macro properties that cannot exist at the level of its constituent molecules.

In this thesis, we are mainly concerned with weak emergence. Therefore, if not otherwise stated, we use the term *emergence* in the following to refer to this kind of emergence. Chalmer's above definition of weak emergence is subjective, because it is based on the existence of *unexpected* features, thus putting emergence "in the eye of the observer". An objective definition is offered by Bedau [10], who considers that the macrostate of a system is weakly emergent if it can be derived from the system's microdynamic only by simulation. However, this definition may be too restrictive. For example, the appearance of pheromone trails indicating the shortest paths between the nest of an ant colony and some food sources is not an emergent phenomenon according to Bedau's definition. Moreover, using this definition, the question of emergence is in general undecidable.

In our research, we are interested in engineering emergent behavior, that is, in designing agent-based systems that produce certain desired behavior. In this context, we attach a broader meaning to the term *emergence*, because, for our purposes, it is not relevant whether the exhibited behavior is perceived as unexpected or not. We are only concerned with the difficult task of finding sets of simple rules for agents, in order to obtain a desired system behavior. Our own definition of emergence is given in chapter 3, where we develop a mathematical formalism for the study of emergence in agent-based systems.

2.3. Cellular automata

Emergent phenomena can be observed even in simple computational systems such as cellular automata. These are deterministic, discrete-time systems, characterized by only local interactions. The cells of a cellular automaton are placed on a regular grid. At each time step, the cells update simultaneously their state, based on a given rule. The number of possible states is finite and the update rule is the same for all cells. For a given cell, the update rule takes into consideration the state of a number of nearby cells.

The history of cellular automata begins in the late 1940s and is associated with the mathematicians Stanisław Ulam and John von Neumann [12]. At Ulam's suggestion, von Neumann has used cellular automata as abstract model for a self-replicating machine. He was able to design a *universal constructor* using a two-dimensional cellular automaton with 29 states and he showed that for a particular start configuration, this constructor makes an infinite number of copies of itself [117]. Later, Codd has designed a universal constructor with only 8 states [24], and Banks has further reduced this number to only 4 states per cell [5].

In 1970, the mathematician John Horton Conway has devised a two-dimensional cellular automaton called the *Game of Life* [51], which has become widely known. In the 1980s, Stephen Wolfram has published numerous articles on cellular automata [124], being a major contributor to the progress of this research area. In the last decades, a large amount of work has been dedicated to the study of cellular automata and they have found applications in a variety of fields. Wolfram's book *A New Kind of Science*, published in 2002 [119], covers a broad range of experimental research related to cellular automata. In spite of their simplicity, cellular automata can exhibit complex emergent behavior. This has led Zuse [130] and Fredkin [49] to hypothesize that our universe is actually a huge cellular automaton.

An infinite number of types of cellular automata can be obtained by combining different values for the factors that characterize a cellular automaton. Some of these factors are:

- **the number of dimensions**: most frequently studied are one- and two-dimensional cellular automata, which can be visualized graphically in a natural way.
- **the number of states**: the most simple cellular automata have only two states (usually denoted by 0 and 1), which can be represented graphically using two different colors (usually black and white).
- **the neighborhood**: the cells taken into consideration by the update rule in order to determine the new state of a given cell constitute its neighborhood. If the update rule also takes into account the current state of the given cell, the neighborhood includes the cell itself.
- **the update rule**: this is the function that computes the new state of a cell based on the current states of the cells in its neighborhood. The update rule is frequently represented in a tabular form.
- **the cellular universe type**: in the case of a finite grid, it is necessary to specify how to apply the update rule at the edges. One possibility is to consider that outside the grid exist virtual cells, which remain always in a constant given state. Another possibility is to consider a circular or toroidal arrangement of the cells.

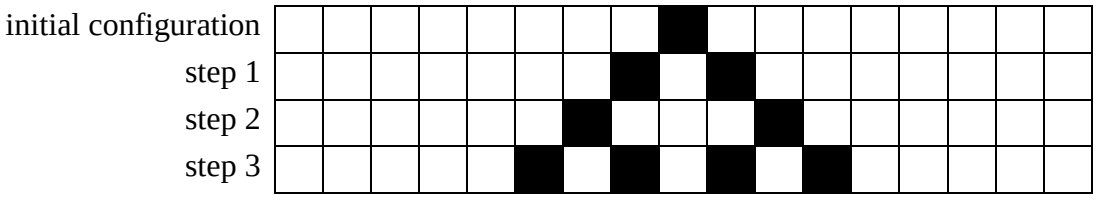
2.3.1. One-dimensional cellular automata

We start our discussion by presenting the simplest class of one-dimensional cellular automata, called *elementary cellular automata*. These have only two states (labeled 0 and 1) and the cellular universe is infinite, that is, the cells are arranged on a line that extends infinitely in both directions. The neighborhood of a cell is given by its two adjacent cells and by the cell itself. Since every cell can be in one of the two possible states, there are $2^3 = 8$ possible state combinations for the cells in a neighborhood. For each of these 8 combinations, there are two possible ways to update the cell state: setting it to 0 or setting it to 1. Therefore, there are $2^8 = 256$ possible update rules, that is, there are 256 types of elementary cellular automata. The update rule for one of these elementary cellular automata is given in the table below:

Current configuration	111	110	101	100	011	010	001	000
New cell state for the middle cell	0	1	0	1	1	0	1	0

Wolfram [122] has introduced an identification system of cellular automata, based on the sequence of binary digits representing the new state of a cell in the update rule table. The corresponding encoding for the above cellular automaton is 01011010, which is the binary representation of the number 90. Therefore, this cellular automaton can be referred to as "the *rule 90* elementary cellular automaton".

The time evolution of a one-dimensional cellular automaton can be visualized by representing the configuration at each time step as a row of black and white cells. Considering that initially a single cell was in state 1, the figure below presents the first steps in the evolution of the *rule 90* cellular automaton:



It can be observed how the cells states change based on the simple rules defined in the automaton's table. The time evolution corresponds to the appearance of a graphical pattern. The pattern obtained after 300 steps is presented in the figure below:

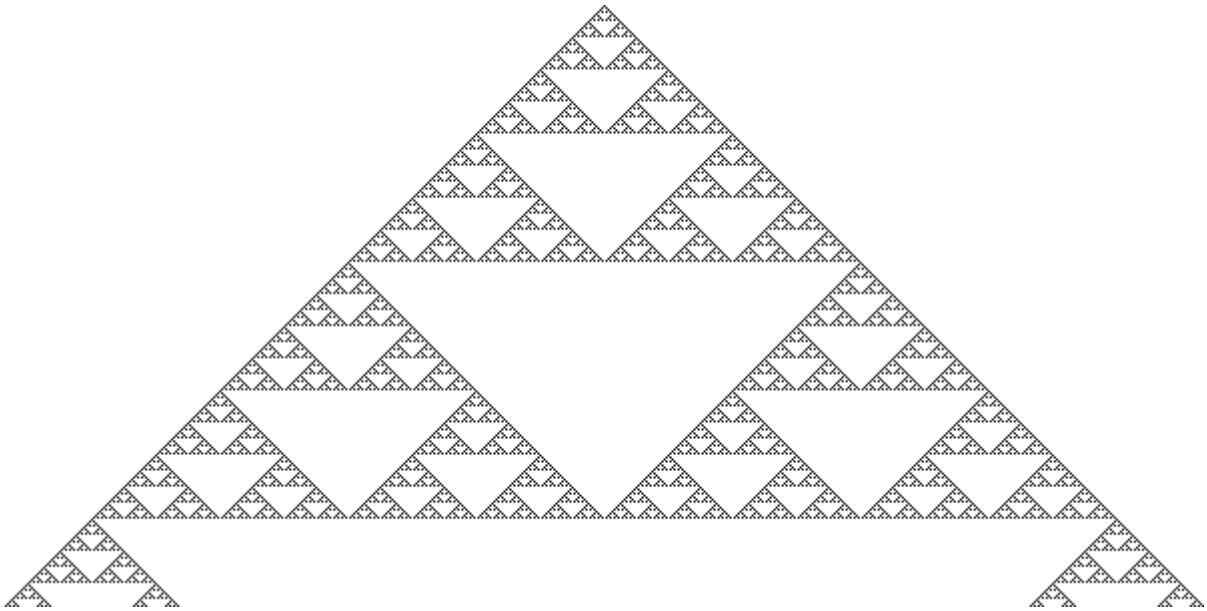


Fig. 2.1. Three hundred steps in the evolution of the *rule 90* elementary cellular automaton. (source: Wikimedia Commons)

The above figure is named "the Sierpiński triangle", after the Polish mathematician who described it in 1915. The resulting patterns differ from one cellular automaton to another: they can be symmetrical, asymmetrical or apparently chaotic. Wolfram [125] has identified a series of particularities of these patterns and has categorized the cellular automata in four classes:

Class 1 cellular automata

From almost any initial configuration, these cellular automata evolve to a homogeneous state. Using the dynamic systems terminology, this final state can be described as a *limit point attractor*.

Class 2 cellular automata

Depending on the initial configuration, these cellular automata evolve to stable or periodical configurations. In terms of dynamical systems, they are analogous to *limit cycles* (also known as *periodic attractors*). Changes in the initial configuration may affect the final state, but they remain localized in a small region. Therefore, class 2 cellular automata act as *filters* on the initial configuration.

Class 3 cellular automata

These automata exhibit chaotic, aperiodic behavior. For almost any initial configuration, the statistical properties of the resulting pattern, such as the proportion of non-zero cells, become very similar after a sufficient number of steps. Cellular 3 automata are very sensitive to the initial configuration, small changes in the initial states leading to increasingly large changes in subsequent states. They are analogous to the *strange attractors* typically found in chaotic dynamic systems.

Class 4 cellular automata

Propagating structures appear during the evolution of these cellular automata. In terms of complexity, they are between the class 2 and class 3 cellular automata.

Among the above classes, the most interesting are class 3 and class 4 cellular automata, because they show that very simple rules can lead to a very complex behavior.

The best known class 3 cellular automaton is *rule 30*, whose update rule table is given below:

Current configuration	111	110	101	100	011	010	001	000
New cell state for the middle cell	0	0	0	1	1	1	1	0

The figure below presents the first 200 steps in the evolution of this cellular automaton:

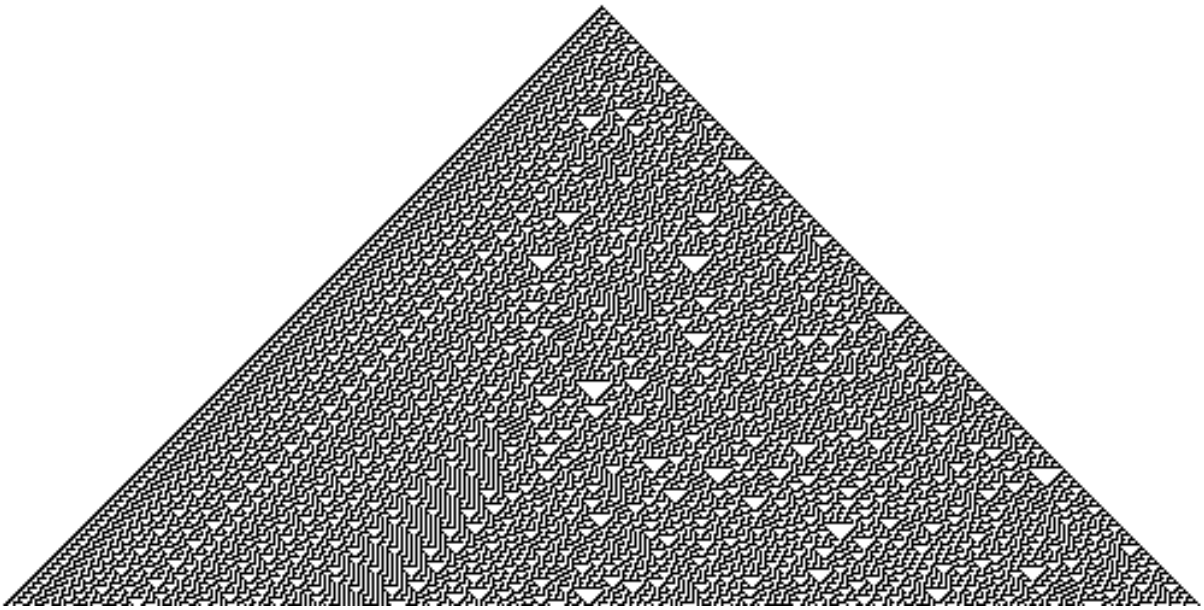


Fig. 2.2. Two hundred steps in the evolution of the *rule 30* elementary cellular automaton. (source: Wikimedia Commons)

The resulting pattern is chaotic. Although triangles and other small structures can be identified, there is no periodicity in their occurrence. Rule 30 is non-linear and computationally irreducible: there is no simpler method to predict the cell states after a given number of steps than applying at each step the automaton's update rule. Wolfram [120] has observed that the digit sequence corresponding to the cell states evolution in this cellular automaton passes the standard statistical tests for randomness and has proposed a random number generator [121] based on rule 30, which has been implemented in the *Mathematica* software package [123].

The update rule table of *rule 110*, which is a class 4 elementary cellular automaton, is given below:

Current configuration	111	110	101	100	011	010	001	000
New cell state for the middle cell	0	1	1	0	1	1	1	0

The figure below presents the first 250 steps in the evolution of this cellular automaton:

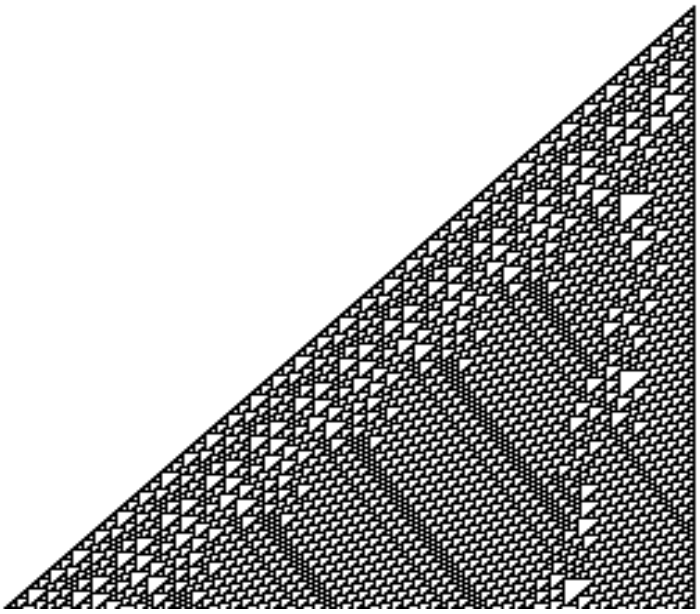


Fig. 2.3. Two hundred fifty steps in the evolution of the *rule 110* elementary cellular automaton. (source: Wikimedia Commons)

This automaton exhibits a complex behavior, which is neither chaotic nor repetitive and is characterized by the occurrence of localized propagating structures interacting in complex ways. Matthew Cook [27] has proved that *rule 110* is computation universal, by showing that it can emulate a cyclic tag system. This is an important theoretical result, because it shows that even the simple rules of an elementary cellular automaton are able to produce highly complex behavior.

Although extremely simple, *rule 110* is not the simplest computation universal system. In his book "A New Kind of Science" [119], Wolfram has conjectured that a particular Turing machine with only two states and three colors is also computation universal and, in 2007, he has announced a 25000$ award to the first person being able to prove or disprove this conjecture. A few months later, Alex Smith, a then 20-year-old student, has proved the conjecture [100] and has won the prize.

In order to use the *rule 110* automaton as a computational system, it would be necessary to encode in its initial configuration both the problem to be solved and the Turing machine capable to solve it. This would lead to a huge initial configuration, which makes *rule 110* unusable for practical purposes. In practice, however, one is interested to solve a particular problem, rather than having a computation universal system capable to solve any problem. Therefore, the question arises, whether it is possible to find a cellular automaton able to solve a particular problem. Until now, we have discussed only elementary cellular automata, which have two states and a neighborhood composed of three cells: the two adjacent cells and the cell itself. There are, however, an infinite number of one-dimensional cellular automata, which can be obtained by varying the number of states and the number of cells in the neighborhood. Hence, there is a good chance that a one-dimensional cellular automaton capable of solving the given problem exists.

Mitchell *et al*. [84] describe a series of experiments in which they use genetic algorithms that evolve a population of cellular automata in order to solve a particular problem. Each automaton can be seen as a computing system whose program is stored in the update rule table and whose input is given by the initial configuration. The result of the computation is given by the configuration obtained after a given number of steps or when a termination condition is met. One of their experiments concerns the majority problem, which consists in

deciding whether or not the initial configuration contains a majority of cells in state 1. For this purpose, the authors have considered one-dimensional cellular automata with two states and a neighborhood composed of 7 cells: the cell itself and its 3 neighbors on either side. For this type of automata, there are $2^7 = 128$ possible state combinations for the cells in a neighborhood. Therefore, the number of cellular automata of this type is: $2^{128} \approx 3 \cdot 10^{38}$. Since an exhaustive search is not feasible in this case, the authors have been able to find good approximate solutions by using genetic algorithms. This is an example of engineering emergent behavior in a multi-agent system, where cells represents the agents and the desired behavior is to obtain a final configuration that contains only 1s or only 0s, depending on which state occurs more frequently in the initial configuration.

2.3.2. Two-dimensional cellular automata

The most studied two-dimensional cellular automata use a rectangular grid, but other topologies are also possible. The figure below presents cellular automata based on hexagonal and triangular lattices:

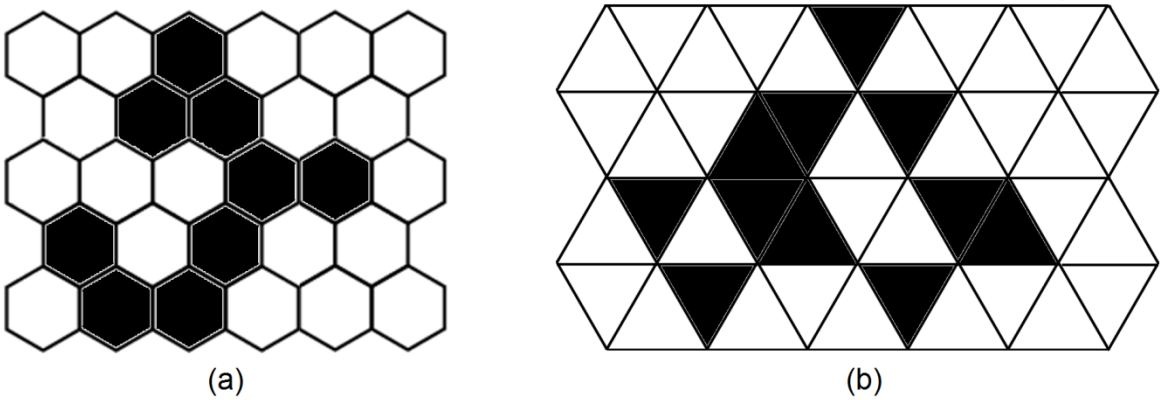


Fig. 2.4. Non-rectangular topologies: (a) hexagonal lattice; (b) triangular lattice.

For a cellular automaton with a rectangular grid, the most frequently used neighborhoods are: the *Moore neighborhood*, which includes the eight surrounding cells, and the *von Neumann neighborhood*, which includes only the four orthogonally surrounding cells (Fig. 2.5). Usually, the cell itself is also part of the neighborhood.

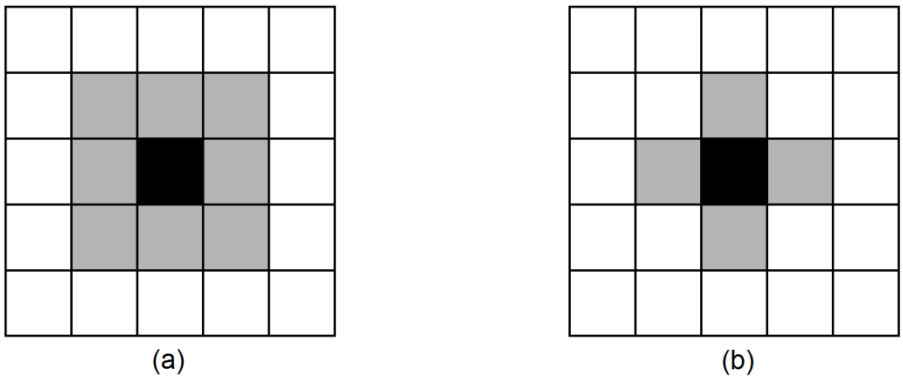


Fig. 2.5. (a) Moore neighborhood; (b) von Neumann neighborhood.

For a two-dimensional cellular automaton with two states and using a *Moore neighborhood*, there are $2^9 = 512$ possible combinations for the states of the cells in the neighborhood, which leads to a number of $2^{512} \approx 10^{154}$ possible cellular automata. Using a von *Neumann neighborhood*, there are $2^5 = 32$ possible state combinations, leading to $2^{32} \approx 4 \cdot 10^9$ possible cellular automata.

A frequently studied category of cellular automata is that of *totalistic automata*. For these automata, the update rules depend only on the number of state 1 cells in the neighborhood. For a Moore neighborhood, there are 10 possible combinations (because the number of cells in state 1 is between 0 and 9), therefore there are $2^{10} = 1024$ possible totalistic automata. Analogous, there are $2^6 = 64$ possible totalistic automata with von Neumann neighborhood.

Another category is that of *outer totalistic automata*, whose update rules depend on the state of the given cell and on the number of remaining cells with state 1 in the neighborhood. For a Moore neighborhood, the number of state 1 cells in the neighborhood not including the cell itself is between 0 and 8, which gives 9 possible values. Since the update rules also take into consideration the state of the given cell, there are 2 x 9 = 18 possible combinations, leading to a number of $2^{18} = 262144$ possible outer totalistic automata. Analogous, there are $2^{10} = 1024$ outer totalistic automata with von Neumann neighborhood.

In order to visualize the time evolution of a two-dimensional cellular automaton, a method similar to that presented for one-dimensional automata can be used, thus obtaining a three-dimensional representation. Another possibility, usually used in computer simulations, is to visualize dynamically the evolution: the representation remains two-dimensional, but it changes at each step, in order to reflect the current state of the cellular automaton. This is a more attractive alternative, because, in many cases, the dynamic of the structures arising during the evolution creates the impression of watching a computer game, as in the case of Conway's *Game of Life*.

Conway's Game of Life

The *Game of Life* has been devised in 1970 by the mathematician John Horton Conway and it has become wildly known after being presented by Martin Gardner in the "Mathematical Games" column of "Scientific American" [51]. The game is nothing more than an outer totalistic two-dimensional cellular automaton with two states and using a Moore neighborhood. A cell in state 1 is considered a live cell, while a cell in state 0 is considered a dead cell. Because the automaton is outer totalistic, we use the term "neighbor" in this context to refer to any surrounding cell, but not to the cell itself. Conway's Game of Life uses the following update rule:

- A live cell with less than 2 or more than 3 live neighbors dies.
- A live cell with 2 or 3 live neighbors stays alive.
- A dead cell with exactly 3 live neighbors becomes a live cell.

The only task of a player is to initialize the start configuration, after which the automaton evolves without player's intervention. The game has attracted great interest due to the unexpected patterns that arise and their surprising evolution. The philosopher Daniel C. Dennett has repeatedly referred to this game [34][35][36] in order to illustrate the emergence of complex philosophical concepts such as consciousness or free will from a set of relatively simple deterministic physical rules governing our universe.

There are many types of patterns that can emerge in Game of Life, such as: *still lives* (static patterns), *oscillators* (repeating patterns), *starships* (patterns translating across the grid), *puffers* (starships leaving debris behind), *guns* (oscillators that emit starships), *rakes* (puffers that emit starships) or *breeders* (patterns exhibiting quadratic growth, such as: guns that emit rakes, puffers that emit guns, rakes that emit puffers or rakes that emit other rakes). The most known pattern, presented in the figure below, is called *glider* and it is the smallest possible starship:

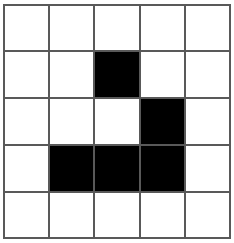


Fig. 2.6. Glider

Conway [11] has shown that Game of Life is Turing complete. His proof is based on emulating the basic building blocks of a computer by using Game of Life's patterns. For example, glider guns can be used as clocks that generate pulses in the form of gliders at regular intervals. By studying the interactions between gliders or between gliders and other patterns, Conway has noticed interesting properties that allow emulating logical gates. For example, if two gliders collide in a particular way, they can annihilate themselves completely. This behavior can be exploited in order to create a NOT gate, by using a glider gun whose stream of gliders collide with the input stream. The presence of a glider in the input stream is interpreted as a bit of value 1, while its absence (i.e., a pattern of blank cells) is interpreted as a bit of value 0. After interacting with the stream produced by the glider gun, gliders in the input stream are annihilated, which can be interpreted as the conversion of a bit of value 1 into a bit of value 0. Conversely, since a glider produced by the glider gun continues to exist after the interaction with a pattern of blank cells, it can be interpreted as a bit of value 1 obtained from the conversion of a byte of value 0 present in the input stream. Using more complex interactions, it is possible to emulate other building blocks of a computer, such as AND gates, OR gates or counters. Finally, it is possible to combine these blocks in order to obtain a finite state machine connected to two counters, which is capable of universal computation. Based on Conway's ideas, Paul Rendell has implemented a Turing machine in Game of Life [93].

2.4. Ant colony optimization

As mentioned before, emergent phenomena can be observed in many natural systems, and scientists often take inspiration from nature in order to design algorithms that make use of the emergent behavior exhibited by these systems. Frequently, such emergent phenomena can be seen in biological systems composed of many individuals that coordinate their actions in a decentralized way. Examples of such systems include: colonies of social insects (such as ants, termites, bees or wasps), flocks of birds, schools of fish, herds of land animals or crowds of people. Swarm intelligence [16] is a research field that tries to identify and understand the mechanisms that lead to emergent behavior in these systems. By mimicking the working of such systems, it is possible to devise algorithms capable of performing complex tasks for which no feasible conventional algorithms are known. Some of the best known swarm intelligence techniques are: ant colony optimization [45], particle swarm optimization [88] or bee inspired algorithms [69].

Ant colony optimization (ACO) is currently the best known and the most used swarm intelligence technique. Since we refer repeatedly to it in the next chapters, we give in this section a description of this technique.

2.4.1. The behavior of foraging ants

In their way to a food source and back to the nest, ants deposit on the ground small amounts of chemicals called pheromones. These pheromones can be sensed by other foragers, which are more likely to follow the trails having a stronger concentration of pheromone. Although

very simple, this foraging strategy proves to be very effective. The interactions between ants are mediated by pheromones. This indirect form of communication, where traces left in the environment by an individual influence the subsequent actions of other individuals or even of the individual itself, is called *stigmergy* [53] [114]. An important characteristic of stigmergy is the locality of information, illustrated in our case by the fact that only nearby ants are able to sense the pheromones.

Deneubourg *et al.* [33] have performed a series of experiments in order to study the behavior of a colony of Argentine ants. In one experiment, they have used a double bridge with branches of equal length to connect the nest with a food source, as seen in Fig. 2.7. Initially, ants choose randomly between the two branches and lay pheromones on their way. Due to random fluctuations, after some time the concentration of pheromone on one branch becomes higher than on the other branch. Therefore, this branch attracts more ants, which in turn deposit more pheromones, thus further increasing the gap between the pheromone concentrations on the two branches. Due to this positive feedback, all ants converge eventually to the same branch.

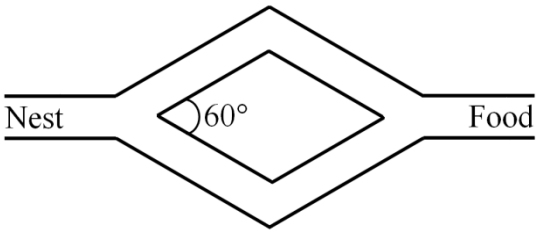


Fig. 2.7. Double bridge experiment with branches of equal length. (modified from [33])

In a variant of the above experiment, Goss et al. [52] have used branches of different length (Fig. 2.8). In this case, random fluctuations are no longer important, because this time another aspect plays a more significant role: ants that have chosen the short branch are the first to return to the nest. Therefore, the concentration of pheromone on the short branch becomes higher than that on the long branch and the positive feedback mechanism described before leads to the convergence of all ants to the short branch.

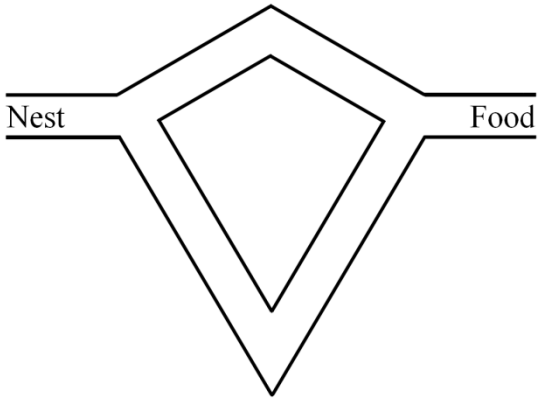


Fig. 2.8. Double bridge experiment with branches of different length. (modified from [52])

Ant colony optimization (ACO) algorithms are based on the observation that a pheromone trail corresponding to the shortest path between the nest and the food source emerges from the stigmergic interactions of ants. The traveling salesman problem has been the first problem solved using ACO algorithms. In the next subsection, we use this problem to illustrate how ACO algorithms work. Then, in subsection 2.4.3, we present the formalization of ACO as a metaheuristic.

2.4.2. Ant colony optimization for the traveling salesman problem

The traveling salesman problem (TSP) consists in finding the shortest round-trip tour through a given set of n cities. It can be represented as a weighted graph, whose vertices correspond to the cities and whose arcs correspond to the paths between cities. Each arc has an associated weight indicating the distance between the cities connected by this arc. In general, it is assumed that there is a path between any two cities, that is, the problem graph is complete. Otherwise, it can be transformed into a complete graph, by associating infinite weights to the missing arcs. This way, the missing arcs do not affect the result, because they cannot be part of the solution. If the distance between two cities does not depend on the direction in which the corresponding arc is traversed, the problem is symmetric and its graph is undirected. Otherwise, the problem is asymmetric and it can be represented by a directed graph having two arcs between each pair of nodes i and j: one arc from i to j and one arc from j to i. In general, these arcs have different weights. For simplicity, we discuss only the case of symmetric problems. If we denote by d_{ij} the distance between nodes i and j, then $d_{ij} = d_{ji}$.

Since ants deposit pheromones during their walk, the concentration of pheromone becomes higher on the paths traversed more frequently. ACO algorithms consider that each arc has an attached value corresponding to the pheromone concentration on the given arc. The pheromone concentration on the arc connecting the cities i and j is denoted τ_{ij}. At start, the pheromone concentration on all arcs is initialized with a value τ_0.

ACO algorithms use a population of artificial ants, which mimic to some degree the behavior of real ants. Each artificial ant constructs at each iteration a solution to the problem. In order to avoid visiting a city more than once, each ant keeps a list of the nodes already visited in the current iteration. Usually, at the start of a new iteration each ant is placed on a randomly chosen city. At each construction step, an artificial ant has to decide which city to visit next. For this purpose, it uses a stochastic rule that takes into account the pheromone concentration on the arcs to the potential next cities, as well as some heuristic information. Most algorithms use as heuristic information the inverse of the distance to the potential next node. If the current node is i and the candidate node is j, the heuristic information is $\eta_{ij} = 1/d_{ij}$.

The probability of an ant located at city i to choose j as the next city is given by:

$$p_{i,j} = \frac{{\tau_{i,j}}^{\alpha} \cdot {\eta_{i,j}}^{\beta}}{\sum_{l \in \mathcal{N}_i} {\tau_{i,l}}^{\alpha} \cdot {\eta_{i,l}}^{\beta}} \tag{2.1}$$

where $\mathcal{N}_i$ represents the set of unvisited cities in the neighborhood of city i, and the parameters α and β determine the influence of the pheromone concentration and that of the heuristic information.

At the end of each iteration, when each ant has constructed a solution to the problem, the pheromone concentration on each arc is updated. The update rule is based on two factors that affect the pheromone concentration in the case of real ants: the pheromone evaporation and the pheromone accumulation on the traversed paths. The pheromone concentration on the arc connecting nodes i and j is updated according to the following formula:

$$\tau_{i,j} \leftarrow (1 - \rho) \cdot \tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k} \tag{2.2}$$

The first term corresponds to pheromone evaporation. The parameter $\rho \in (0,1]$ gives the evaporation rate. The second term represents the amount of pheromone deposited by all ants that have traversed the arc (i, j) during the current iteration. Considering that the artificial ant

k has constructed the tour T_k having the total length C_k, the amount of pheromone deposited by this artificial ant on the arc (i, j) is given by:

$$\Delta\tau_{i,j}^{k} = \begin{cases} \frac{1}{C_k}, if\ (i,j) \in\ T_k \\ 0, otherwise \end{cases} \tag{2.3}$$

The above equations correspond to the first ACO algorithm, called Ant System, which has been introduced by Marco Dorigo [42]. Many extensions of this algorithm have been proposed, such as: Elitist Ant System [41], Ant-Q [50], Rank-based Ant System [18], Ant Colony System [43] or MAX-MIN Ant System [106]. The last two variants, which are among the best-performing ACO algorithms, are briefly described in the next paragraphs.

Ant Colony System (ACS)

Unlike the original Ant System algorithm, ACS decides which city to be visited next by using an action choice rule that exploits more aggressively the experience gained by ants. The probability that an ant chooses a given potential node as the next to be visited depends on a random variable q uniformly distributed in the range [0,1] and on a constant parameter $q_0 \in [0,1]$. If the current value of q is such that $q \leq q_0$, the artificial ant chooses the node l that maximizes the expression $\tau_{i,l} \cdot {\eta_{i,l}}^{\beta}$. Otherwise, the node to be visited next is chosen based on the probabilities given by formula (2.1), where the parameter α is set to 1. This action choice rule is called *pseudorandom proportional*.

Another difference between ACS and the original algorithm is that pheromone evaporation and pheromone deposit occur only on the arcs corresponding to the best-so-far tour. However, this strategy may lead to stagnation, because all ants converge to the best-so-far tour. Therefore, in addition to the global pheromone trail update, which takes place at the end of each iteration, ACS performs a local update, at each construction step: each time an artificial ant traverses an arc, it removes a certain amount of pheromone from the arc, in order to favor the exploration of alternative paths in the future.

MAX-MIN Ant System (MMAS)

Pheromone evaporation and pheromone deposit is restricted to the arcs corresponding either to the best-so-far tour (as in ACS) or on the arcs corresponding to the iteration-best tour. While ACS avoids stagnation by performing a local pheromone trail update, MMAS achieves this goal by limiting the value range of the pheromone concentration. This way, MMAS prevents the excessive growth of pheromone concentrations on some paths and the excessive decrease on some other paths. Pheromone concentrations are constrained to the interval $[\tau_{min}, \tau_{max}]$, where τ_{max} is recomputed at each iteration, based on the length C_{bs} of the best-so-far tour: $\tau_{max} = 1/C_{bs}$. The minimum value allowed for pheromone concentrations is computed as: $\tau_{min} = \tau_{max}/a$, where a is a constant parameter.

Additionally, pheromone concentrations are reinitialized in MMAS each time the system approaches stagnation or when no improved tour has been found after a given number of iterations.

2.4.3. The ant colony optimization metaheuristic

After the initial application to the TSP, ACO techniques have been adapted to many other classes of problems. The set of concepts characterizing all ACO algorithms have been formalized as a metaheuristic by Dorigo et al. [44]. Metaheuristics are algorithmic templates used to specify problem-independent optimization strategies, which can be instantiated in

order to define problem-specific heuristics. The pseudocode description of the ACO metaheuristic [45] is given below:

```
procedure ACOMetaheuristic
    ScheduleActivities
        ConstructAntsSolutions
        UpdatePheromones
        DaemonActions       % optional
    end-ScheduleActivities
end-procedure
```

Fig. 2.9. The ACO metaheuristic in pseudocode. (from [45])

ACO techniques can be applied to any combinatorial optimization problem for which a solution consists of a sequence of components and for which it is possible to map the problem space to a construction graph having solution components as nodes. This way, a solution to the problem can be represented as a path in the construction graph. As shown in the pseudocode above, ACO is based on three activities: ConstructAntsSolutions, UpdatePheromones and DaemonActions. The metaheuristic does not specify whether these activities should be executed in parallel or sequentially, or whether they should be synchronized in some way. These aspects are specific to each algorithm and they are determined in each case by the particular instantiation of the ScheduleActivities construct.

During the ConstructAntsSolutions phase, artificial ants incrementally build solutions, by walking through the construction graph. At each step, a new node of the construction graph is visited and the corresponding component is added to the partial solution. The walk is stochastic and biased by the pheromone trails and by some heuristic information.

Pheromone concentrations are changed during the UpdatePheromones activity. Concentrations increase when ants deposit pheromones on the traversed arcs and they decrease as a result of evaporation. Since more pheromones are deposited on arcs that are traversed more often or on arcs belonging to high quality tours, these arcs become more attractive for the artificial ants. Evaporation allows forgetting past decisions, thus allowing the exploration of new areas and preventing an algorithm to converge too rapidly to a suboptimal region in the search space.

A series of optional centralized actions are grouped under the DaemonActions activity. A typical example is a local optimization procedure that can be used to improve the solutions constructed by the ants.

3. A DIFFERENT VIEW ON EMERGENCE

The research presented in this thesis is driven by the issue of engineering emergent behavior. This is a difficult problem, because the mechanisms behind emergence are not completely understood and emergent phenomena are characterized by their apparent unpredictability.

In this chapter, we develop a mathematical formalism for the study of emergence, in order to gain insight into the principles of emergent behavior. Our formalism puts emergence in an agent-oriented context, consistent with the frame imposed by the problem of engineering emergent behavior.

As discussed in section 2.2, there are many definitions for emergence in the literature. They range from intuitive to formal and they differ in the aspects of emergence they are able to capture. Many formal definitions start from a preliminary intuitive definition and try to express it in a rigorous manner. In doing this, they are usually concerned with the aspects that distinguish emergent properties or behaviors from non-emergent ones. In some cases, this distinction can be easily formalized, but there is no general procedure that allows to decide whether a given phenomenon is emergent or not. One example is Darley's definition [30], which states that a "true emergent phenomenon is one for which the optimal means of prediction is simulation". In other cases, it is more difficult to express formally the defining characteristics of an emergent phenomenon, and the definitions actually formalize a procedure for detecting emergence. This is, for example, the approach taken by Bonabeau and Dessalles [15]. In many definitions, emergence is detected by means of an observer. In some cases, this leads to a subjective description, putting emergence "in the eye of the observer". Examples include the Turing test for emergence [17] or the use of an observer that relies on the surprise element [95]. There are, nevertheless, several attempts to provide an objective definition of emergence. One example is the approach taken by Crutchfield [29], which introduces the notion of *intrinsic emergence*, characterized by an increase in intrinsic computational capability. Besides their theoretical importance, objective definitions are appealing because they offer in principle a way to automatically detect emergent phenomena. However, they usually involve very complex computations, making them unsuitable for practical purposes.

In this thesis, we take a different view on emergence, which allows us to provide a definition that is both objective and suitable for practical purposes. We argue that a definition of emergent phenomena should only be concerned with how these phenomena arise and it should not address the properties of the emergent phenomena. In our view, these properties should be the subject of an entire research field and not part of the definition of emergence. From this perspective, a definition should be given only in terms of the processes that produce emergent phenomena. We propose an informal definition that represents the starting point in developing our formalism:

Definition 3.1 (informal) *An emergent behavior is the behavior exhibited by a decentralized agent-based system.*

Obviously, the above definition is objective, because it does not depend on the way an observer perceives the behavior. It is also suitable for practical purposes, because it eliminates the need to detect emergent phenomena. However, one may argue that this definition is too broad, making almost any behavior emergent. Let us consider for example a decentralized agent-based system where all agents die or settle down to a stable state after one time step. Should we really consider this an emergent behavior? Our answer is "yes". While this is an uninteresting behavior, it is nonetheless an instance of the possible emergent behaviors exhibited by a decentralized agent-based system. We consider that the class of emergent behaviors includes all behaviors produced by a decentralized agent-based system and we regard the study of the properties of emergent behaviors as a scientific discipline, which we will refer to as *emergent behavior theory* (EBT).

Of course, EBT is focused on "interesting" behaviors, but what aspects of an emergent behavior make it interesting? There is no absolute answer to this question, because it depends on the observer watching the emergent behavior. One may therefore argue that we have eliminated the subjectivity from the definition of emergence only to move it to the EBT. However, this kind of subjectivity can be found in many research fields. Let us take an example from computational complexity theory, by considering the Traveling Salesman Problem (TSP), which is known to be NP-hard. An instance of TSP with a cost matrix containing only zeros is trivial to solve and it is of no interest for a researcher studying this class of problems, but it nonetheless represents an instance of this class of NP-hard problems. One researcher may be interested in non-trivial TSP instances for which exact algorithms are able to find solutions within reasonable time. Another researcher may find interesting those TSP instances that are intractable for exact algorithms, but for which heuristic algorithms obtain very good results. Yet, another researcher may consider that a TSP instance is interesting only if no exact algorithm or heuristic is able to find a good solution within reasonable time. Furthermore, a researcher may be surprised to find out that an algorithm can easily solve a given TSP instance, but it is not able to solve another TSP instance, although the researcher cannot detect any significant differences in the structure of the cost matrices corresponding to these two instances.

The example above shows that subjective notions like "interesting" or "surprising" are not uncommon in a research field, although definitions of concepts related to the given research field do not usually involve subjective notions. Defining emergence in terms of the characteristics of the exhibited behaviors is like arguing that only "useful", "interesting" or "surprising" instances of the TSP should be considered as members of this class of problems. For this reason, we consider that our approach to exclude the behavior properties from the definition of emergent behavior is a necessary step toward a theory of emergence.

In order to give a more formal definition of emergence, it is necessary to clarify the meaning of the concepts involved by definition 3.1. We start with the notion of decentralized agent-based system. When can we say that a system is decentralized? Intuitively, a system is decentralized if it lacks a central control structure. However, for any conceivable agent-based system, it is possible to build a model that involves a central control structure. This can be done for example by viewing the agents as merely passive elements, unable to take actions on their own. In this case, the set of rules associated with a given agent is not considered a characteristic of this (degenerated) agent, but it is instead viewed as part of a complex set of rules used by a central control structure. This central control structure acts on each agent in accordance with the corresponding set of rules. To exemplify, let us consider an artificial life system based on agents that act without the need of a central control structure. This system can be implemented as a computer simulation where the agents are nothing more than data structures manipulated by the program. In this case, the program clearly represents a central control structure, although the model used is a decentralized one.

The above discussion shows that it is more accurate to talk about the model used to describe a given system than about the system itself. Therefore, it would be preferable to give a definition of emergence in terms of agent-based models rather than agent-based systems. Still, it is not clear whether decentralization is an objective or rather a subjective characteristic of a model.

We try to capture the essence of decentralization by examining how a centralized system could be modeled as a decentralized one. In order to do this, it is necessary to remove the central control structure while preserving the behavior of the system. A control structure is actually nothing more than an agent that coordinates the activities of other agents. In doing this, it needs information about the state of each agent. Based on this information and on a set of rules, the control structure issues a series of commands in order to control its subordinate agents. A possible approach to eliminate the control structure is to implement its logic in each agent, thus transforming the control structure into a passive agent. Instead of receiving commands from the control structure, each agent can decide itself what action to do next. However, this implies that each agent has information about the state of the other agents. This problem can be solved by implementing a kind of stigmergic communication between agents. For this purpose, the passive agent corresponding to the former control structure can be used as a blackboard on which each agent writes information about its state and from which each agent can retrieve the needed information. The resulting model exhibits the same behavior as the original system, but the central control structure has been replaced by a passive agent playing the role of an environmental element that mediates the communication between agents.

Using the above method, it is possible to transform a centralized agent-based system into a system lacking a central control structure. However, it is questionable whether the resulting model is decentralized. In our view, the answer to this question is "no". Although the model lacks a control structure, it still contains a central structure, represented by the passive agent used to mediate the communication. We argue that that the defining characteristic of decentralization is the existence of only local interactions between agents and that the presence of a control structure has no relevance in deciding whether a model is decentralized or not. From this perspective, we propose the following informal definition:

Definition 3.2 (informal) *An agent-based model is decentralized if no agent is able to interact with all other agents.*

The above definition is ambiguous, because it does not explicitly disallow the interaction of an agent with all other agents by means of stigmergic communication. Stigmergy implies the existence of environment elements that mediate the communication, but making references to environment elements would complicate the definition of a decentralized agent-based model. We choose instead to treat the environment as a collection of agents, thus integrating the environment into the agent-based model. This may seem a radical decision, but we consider that there is no fundamental difference between agents and environment elements. Agents are able to perceive their environment and act upon it, but they are also able to perceive the state of other agents and to interact with them. One can argue that environmental elements are typically passive entities, which do not actively interact with other agents, while agents are typically active, autonomous entities. However, this is not necessarily true. For example, the air is an environmental element, but it could take the form of a tornado that interacts in a devastating way with an autonomous agent like a human been. Moreover, depending on the level of observation, the same entity may play the role of an agent or an environment element. Finally, our universe is probably the best example that an agent-based system does not necessarily have an environment.

In order to clarify what we mean by interaction in the context of an agent-based model, we propose the following definition:

Definition 3.3 (informal) *An interaction of a group of agents is an exchange of information between the agents in the considered group, which leads to a change in the state of at least one of the agents.*

There are a few more things we have to take into account before proceeding with our mathematical formalism. One of them is that the set of agents with which an agent interacts may change over time. Another one is that, in general, the number of agents does not remain constant during the evolution of the system. Some agents may die, while other agents may be created. Who should be responsible for creating and destroying agents? Should we place the logic of creating and destroying agents at the system level or at the level of individual agents? In our view, having the logic at the system level would represent an inherent loss of decentralization. Moreover, let us notice that placing the logic at system level is equivalent to having a central control structure that decides when a subordinate agent should die and when a new subordinate agent should be created. Since a central control structure can be always seen as an agent, it means that there is no loss of generality in placing the logic at the level of individual agents.

One of the most interesting aspects of emergence is that even very simple rules are able to produce complex behavior. Therefore, we consider that in order to understand the mechanisms of emergence, one should start by studying the behavior exhibited by systems containing agents that are governed by only simple rules. The question is how to express the notion of simple rules in a formal way. One idea is to use algorithmic complexity concepts such as Kolmogorov complexity [76]. However, this approach is not practical, because Kolmogorov complexity is not computable. Therefore, we take a pragmatic approach and we propose to use abstract syntax trees (AST) in order to assess the complexity of agent rules. Assuming that a method to represent agent rules as ASTs has been agreed upon, we measure the complexity of the rules of an agent as the number of leaves in the corresponding AST. This is, of course, not an ideal measure, because it depends on the language used to express the rules and because the same rules can be expressed in many ways. Nevertheless, it is a useful measure for practical purposes.

We are ready now to give a formal definition of agent-based models. The notations that appear in the next paragraphs use upper indices to denote agents and lower indices to denote time steps.

Definition 3.4 *An agent-based model is a discrete-time dynamical system described by a tuple* $(\mathrm{S}, \mathrm{L}, \mathrm{A}_0)$*, where:*

- S is the state space.
- L is the language used to describe agent rules, together with a definite method of representation as AST.
- A_0 is the initial finite set of agents. An agent is a tuple $(s_0, \sigma, f, \varphi)$, where:
  - $s_0 \in S$ is the initial state.
  - $\sigma : 2^S \to 2^{\mathcal{A}}$ is the selection function, which returns the set of agents with which this agent interacts. ($\mathcal{A}$ denotes the set of all possible agents, that is, the set of all possible tuples (s, σ, f, φ). The notation $2^{\mathcal{M}}$ denotes the power set of $\mathcal{M}$.)
  - $f : 2^S \to S$ is the interaction rule, which computes the new state of the agent based on the states of the agents with which it interacts. f is described in the language L.
  - $\varphi : S \to 2^{\mathcal{A}}$ is the transformation rule that decides if the agent should die and/or other agents should be created. The return value of φ is the agent itself, if the

agent continues to exist and no new agents are created; it is the empty set, if the agent dies and no new agents are created; it is a finite set of agents (possibly containing the agent itself), if new agents are created. φ is described in the language L.

- The interaction between agents is reflexive in the sense that if an agent a interacts with an agent b at a given moment t, then b also interacts with a at the moment t:

  $$\forall t \in \mathbb{N},\ \forall a, b \in A_t,\ \ b \in \sigma^a(s_t^a) \Leftrightarrow\ a \in \sigma^b(s_t^b)$$

- The state of an agent evolves over time according to the following rule:
  $s'_{t+1} = f(\{s_t\} \cup \{s_t^a \mid a \in \sigma(s_t)\})$
  In the above equation, s'_{t+1} denotes the preliminary state of the agent at the moment $t+1$, that is, the state before applying the transformation rule. The definitive configuration of the agent-based model at the moment $t+1$ is:
  $$A_{t+1} = \bigcup_{a \in A_t} \varphi^a(s'^a_{t+1})$$

Next, we provide a definition for a decentralized agent-based model, which is a formalization of the definition 3.2:

Definition 3.5 *An agent-based model is decentralized iff:*

$$\sigma^a(s_t^a) \cup \{a\} \subset A_t, \forall t \in \mathbb{N}, \forall a \in A_t.$$

In studying emergence, it is interesting to analyze and compare the behaviors obtained in a variety of agent-based models, ranging from centralized to strongly decentralized. Therefore, it is useful to have a measure of how decentralized an agent-based model is. Let us first notice that the condition imposed by the definition 3.5 can be alternatively expressed as:

$$|\sigma^a(s_t^a)| + 1 < |A_t|, \forall t \in \mathbb{N}, \forall a \in A_t.$$

Based on this observation we provide the following definition:

Definition 3.6 *The centralization level of an agent-based model is a quantity γ given by:*

$$\gamma = \max_{t \in \mathbb{N}} \max_{a \in A_t} \frac{|\sigma^a(s_t^a)| + 1}{|A_t|}$$

Using the above measure, a strongly decentralized agent-based model can be defined as a model with a very low centralization level:

Definition 3.7 *An agent-based model is strongly decentralized iff* $\gamma \ll 1$.

According to definition 3.1, emergent phenomena require a decentralized model. However, some agent-based models are only partially decentralized. Should we consider in this case the resulting behavior as emergent? In order to better understand this question and what we mean by partially decentralized models, let us examine the agent-based models used by the Ant Colony Optimization (ACO) algorithms for the traveling salesman problem, which have been described in section 2.4. In almost all ACO algorithms, ants update the pheromone trails at the end of each construction phase by depositing pheromones on the arcs contained in the newly constructed tour. In Ant System, the amount of pheromone is inversely proportional to the length of the constructed tour. Other variants of ACO use update strategies that require global knowledge about the solution quality. In algorithms such as Elitist Ant, MAX-MIN Ant System or Ant Colony System the pheromone update process is strongly biased in favor of the best-so-far or iteration-best solution. In Rank-Based Ant System a ranking of the solutions

based on their quality is performed before updating the pheromone trails. Ant System is the algorithm that mimics most closely the behavior of real ants, but it is clearly outperformed by all other ACO algorithms. Nevertheless, Ant System is the only algorithm that uses a decentralized model. The other algorithms use global information that implies the existence of a central structure. However, the global information is not needed at each time step, but only at the end of the construction phase. During the construction phase, ants' actions are decentralized. Therefore, we denote as *partially decentralized* the model used by these algorithms. Should we consider that the behavior exhibited by Ant System is emergent, while the behavior exhibited by the other algorithms is not, because they use only partial decentralized models? In our opinion, the answer is "no", although the behavior exhibited by Ant System could be denoted as *pure emergent*, in order to differentiate it from the *partially emergent* behavior exhibited by the other algorithms.

In accordance with the discussion above, we offer the following definition for a partially decentralized agent-based model:

Definition 3.8 *An agent-based model is partially decentralized iff:*

$$\exists\, T \subseteq \mathbb{N}, such\ that\ |T| = \infty\ and\ \ \sigma^a(s_t^a) \cup \{a\} \subset A_t, \forall t \in T, \forall a \in A_t$$

Notice that in the above definition we require that there is an infinite set of time steps at which the model behaves like a decentralized one. For an agent-based model where the condition imposed by the definition 3.8 is satisfied only for a finite set T of time steps, there exists a moment τ from which on the agents' actions are centralized at each time step. Therefore, we do not consider such a model as partially decentralized. Conversely, if there is a finite set T of time steps such that the decentralization condition is satisfied for each $t \in \mathbb{N}\backslash T$, then it exists a moment τ from which on the agents' actions are decentralized at each time step. Hence, such a model could be considered decentralized.

As mentioned before, we are mainly interested in emergent phenomena exhibited by agents that follow only simple rules. Therefore, we need a measure for the complexity of agent rules.

Definition 3.9 *Given a function* g *described in a language* L *that specifies a definite method of representation as AST, the rule complexity of* g *relative to the language* L*, noted* $\rho_L(g)$*, is given by the number of leaves in the AST representation of g associated with the language* L*.*

Using this measure, we introduce a definition for the rule complexity of an agent-based model:

Definition 3.10 *The rule complexity of an agent-based model is a quantity* r *given by:*

$$r = \max_{t \in \mathbb{N}} \max_{a \in A_t} (\rho_L(f^a) + \rho_L(\varphi^a))$$

There is a large class of agent-based models for which the set of agents does not change over time: agents do not die and no new agents are created. In such models, the transformation rule of each agent is the identity function, which has a rule complexity of 1. Since the set of agents does not change over time, the rule complexity of the model is determined by the initial set of agents. In this case, we can write:

$$r = \max_{a \in A_0} (\rho_L(f^a) + 1)$$

Definition 3.11 *The dynamic of a decentralized agent-based model, expressed in terms of system-level properties, is called pure emergent behavior.*

Definition 3.12 *The dynamic of a partially decentralized agent-based model, expressed in terms of system-level properties, is called partially emergent behavior*.

The two definitions above refer to system-level properties, but this is a concept that is not involved in our definition of an agent-based model. What characteristics of a system are regarded as system-level properties depends on the perspective of an observer. In general, there is a loss of information when a system is described in terms of system-level properties instead of describing it in terms of its agents' states, although in most cases the system-level description is more useful. For example, describing a gas in terms of its temperature and pressure is more useful than specifying the positions and velocities of each of its molecules. Nevertheless, knowing only the temperature and pressure of a gas, one cannot compute the position and velocity of a particular molecule. This loss of information makes the behavior of an agent-based model – as defined by the two definitions above – a subjective notion, because it depends on the set of system-level properties used to describe the dynamic of the system. However, this does not represent a problem from the practical perspective of engineering emergent behavior. Since in this case the desired behavior is specified a priori, the set of system-level properties is implicitly imposed by the need to detect this given behavior.

The formalism introduced in this chapter is a first step towards an *Emergent Behavior Theory* (EBT). It is consistent with our view that emergent phenomena should be defined only in terms of the computational models able to produce them, while the characteristics of these phenomena should represent an entire research topic of EBT. From this perspective, EBT should identify and analyze different classes of emergent behavior, such as:

- emergent behavior characterized by an increase in complexity;
- emergent behavior characterized by pattern formation;
- chaotic emergent behavior;
- emergent behavior characterized by attractors;
- emergent behavior for which simulation is the shortest way to predict it.

There are many questions to which EBT should find an answer. The list below contains only a few of them:

- How is pure emergent behavior different from partially emergent behavior? For practical purposes, is it the mixed approach offered by partially emergent behavior preferable to the pure approach?
- In what respects is the emergent behavior exhibited by homogenous agent-based systems different from the emergent behavior exhibited by heterogeneous systems? Which type of system should be preferred when engineering emergent behavior? Homogenous or heterogeneous?
- How affects the centralization level the emergent behavior? Is there a threshold that must be exceeded in order to be able to obtain a certain behavior? Is there an optimum value of the centralization level?
- Is there a minimum number of agents needed to obtain a certain behavior? Is there an optimal number?
- How does relate the complexity of agents' rules and the number of agent types needed to achieve a desired behavior?
- What is the best methodology to engineer emergent behavior?

How should one proceed in order to answer these questions? Let us first note that the above questions are too broad to admit a simple answer. Moreover, some of the problems raised by these questions may be undecidable in the general case. Therefore, one should initially address more specific versions of these questions, by restricting for example the class of desired behavior taken into consideration, by considering a particular methodology of

engineering the emergent behavior or by putting additional constraints on the agent-based models used. However, we hypothesize that even for such specific questions, only a fraction of them are decidable. Since there are virtually endless specific questions that could be posed and since one cannot know a priori whether an answer can be found for a particular question, it is difficult to decide which questions are the most suitable to be tackled.

We propose to identify promising specific questions by performing computational experiments and choosing those questions for which empirical answers can be found. Then, using the formalism introduced in this chapter, one can try to give rigorous proofs of these empirical answers. This approach implies the existence of an experimental framework for the study of emergence, which must be compatible with our mathematical formalism. A meta-framework designed by us to meet these requirements makes the subject of the next chapter.

4. METFREM – A META-FRAMEWORK FOR THE STUDY OF EMERGENCE

In order to study emergent phenomena in a rigorous manner, we need a framework that allows the modeling of virtually any agent-based system in a unified way. For this purpose, we develop a **Met**a-**Fr**amework for **Em**ergence, called *MetFrEm*, which can be used to describe various algorithmic frameworks comprising a population of interacting agents. MetFrEm is a meta-framework, because it is typically used to model abstract, high-level algorithms such as metaheuristics, which are themselves frameworks allowing to describe specific algorithms.

By modeling different high-level algorithmic specifications in the same meta-framework, we can make a rigorous comparison of the methods considered. MetFrEm provides a set of concepts and rules that must be used to model the desired systems and it imposes a general algorithmic structure for these systems. At a first glance, constraining all models to the same algorithmic structure may seem to limit the range of systems that can be expressed in MetFrEm. This is not true. The algorithmic structure prescribed by MetFrEm requires only describing the working of each modeled system using a fixed template. It does not prevent any agent-based system from being able to be modeled in the meta-framework. Instead, the imposed algorithmic structure actually improves the usefulness of MetFrEm. Due to this constraint, the methods expressed in the meta-framework are somewhat devoid of their underlying metaphor. Therefore, we can better identify which aspects are similar and which are really different when comparing various algorithmic frameworks.

Since our meta-framework is used to model meta-algorithms, its structure must be defined at the meta-meta-algorithm level. To better understand what this means, we make an analogy with the four-layered architecture introduced by OMG (Object Management Group) [113] for modeling languages (Figure 4-1).

The topmost layer in the OMG architecture defines the language used by MOF (Meta-Object Facility) to specify metamodels. Possible such metamodels are: the UML language, UML profiles (e.g., SysML - *Systems Modeling Language* or SPEM - *Software Process Engineering Modeling*) or other metamodels (e.g., CWM – *Common Warehouse Metamodel* or ODM – *Ontology Definition Model*). Specific models, positioned at the M1 Layer, can be specified using these metamodels. Finally, real world object instances are created based on specific models.

Similarly, we can describe a four-layer architecture containing MetFrEm at the topmost layer (Figure 4-2). Meta-algorithms are positioned at Level 2 in this architecture. An instantiation of a meta-algorithm represents an algorithm for a specific class of problems. For example, the MAX-MIN Ant System algorithm [106] for the Traveling Salesman Problem is an instantiation of the Ant Colony Optimization metaheuristic. The last level in this architecture is represented by the application of an algorithm to a particular problem instance.

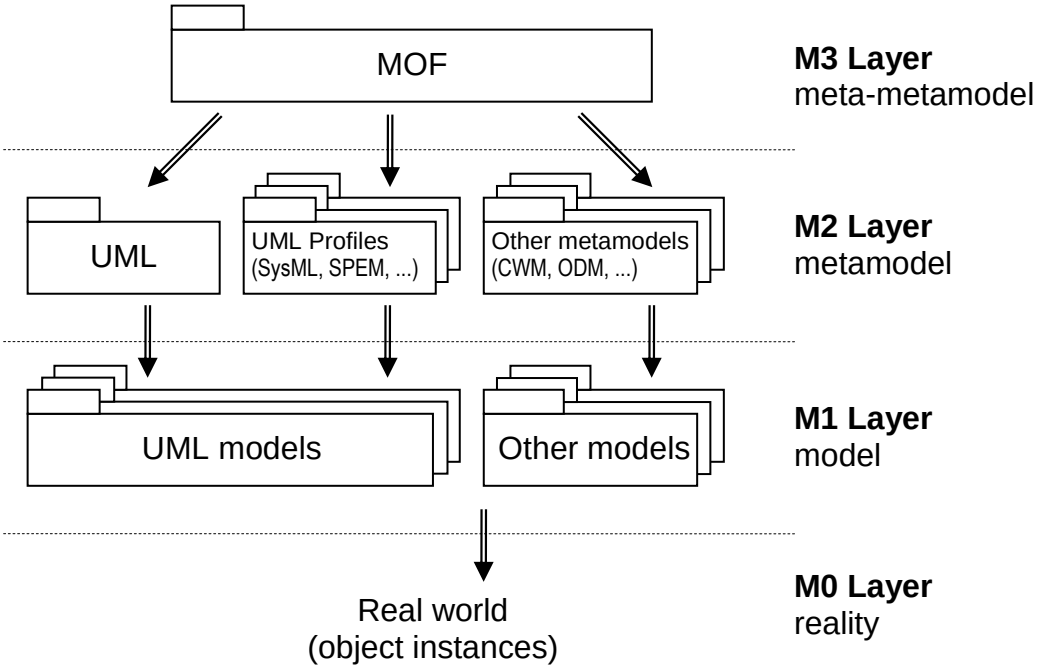


Figure 4-1. The OMG four-layer modeling architecture.

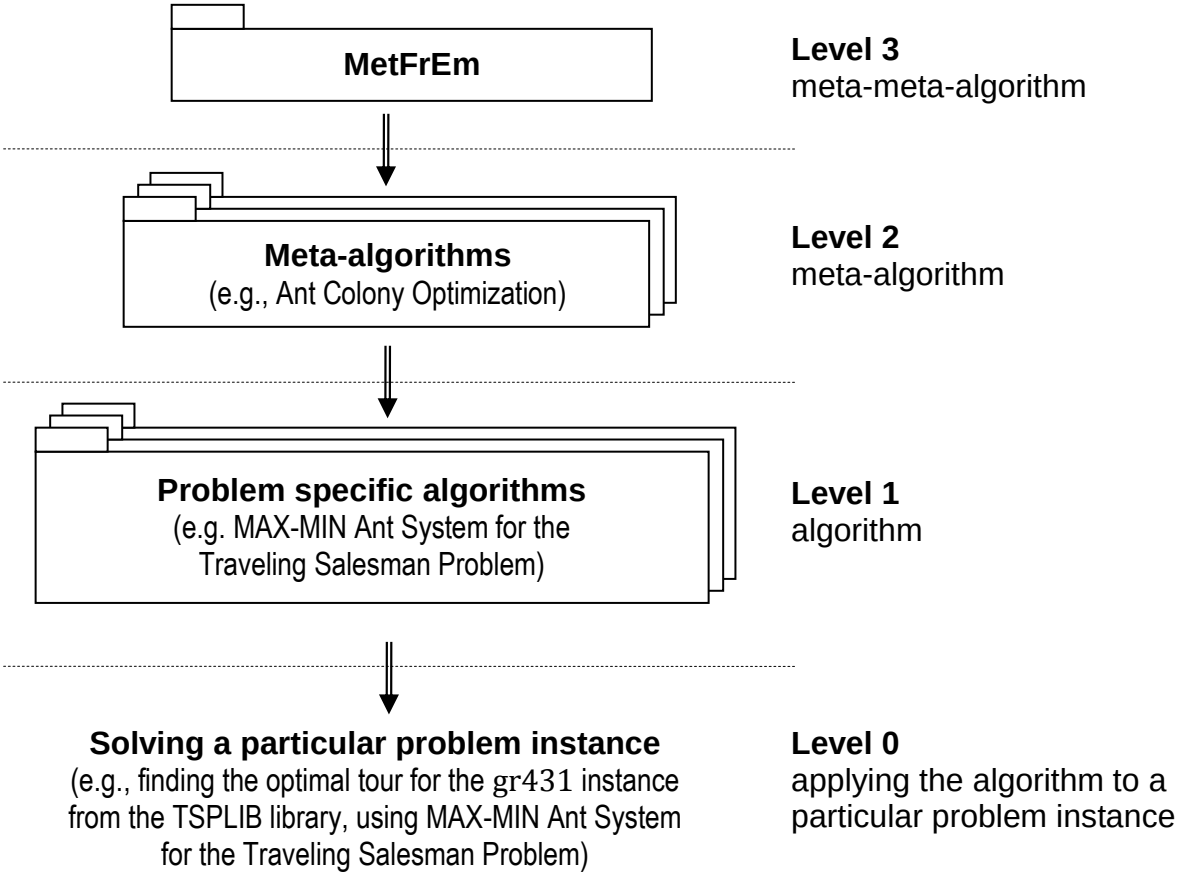


Figure 4-2. The MetFrEm four-layer architecture.

4.1. Goals and objectives of MetFrEm

In the introductory chapter, we have mentioned a series of questions and hypotheses that motivate our research. We take them into consideration in order to establish the goals of our meta-framework. The main design objectives of MetFrEm are:

- **to allow the modeling of virtually any agent-based algorithmic framework.** Since emergent phenomena may occur in virtually any agent-based system, our meta-framework should not restrict the type of systems that can be modeled.
- **to favor the modeling of strongly decentralized systems.** Although we do not try to mimic natural systems, our research is focused on building models that could explain some of the properties and phenomena observed in these systems. It is highly improbable that a centralized control structure can arise by pure chance in the systems of the type considered in our research. Therefore, we constrain many of the models proposed in this thesis to be strongly decentralized. We should note, though, that there is no reason to forbid centralized control for optimization algorithms, and, in practice, most optimization heuristics present some form of centralized control. For example, swarm intelligence techniques like Ant Colony Optimization (ACO) or Particle Swarm Optimization (PSO) keep track of the best-so-far solution and use this information in various ways (e.g., for updating the pheromone trail in ACO or for adjusting the velocity in PSO). MetFrEm should allow us to clearly identify which models use centralized control and which not.
- **to favor the modeling of systems with agents that follow only simple rules.** In order to understand the underlying principle of emergence, it is necessary to study the mechanisms that allow simple rules to produce complex behavior. Moreover, we focus on agents with simple rules, because in nature only such agents could appear by pure chance. These agents have no intelligence, beliefs, desires or intentions. They pursue no goals and there is no meaning associated with the simple rules that govern their behavior. Therefore, there is no need to introduce such high-level concepts in our meta-framework. We are instead interested in how these concepts may emerge from the interaction of our simple agents.
- **to facilitate the modeling of algorithmic frameworks with highly heterogeneous agents.** One characteristic of many agent-based computational models is that individuals are relatively homogeneous. For example, there is only one species of ant in most Ant Colony Optimization algorithms and only one type of particle in Particle Swarm Optimization. On the other hand, one of the hypotheses mentioned in the introductory chapter is that it is easier to engineer a desired emergent behavior in a system with heterogeneous agents. Our meta-framework should be able to model systems with homogeneous agents, like swarm intelligence techniques, but one important goal of MetFrEm is to facilitate the modeling of algorithmic frameworks with highly heterogeneous agents. For this purpose, a system comprising a great number of agent types should be described in a generic manner in the meta-framework, without the need to explicitly state the interaction rules between each pair of agent types. An agent of a given type has to know how to interact with agents of other types without even needing to be aware of the existence of these other types. Another related goal of MetFrEm is to facilitate the modeling of systems where new types of agents are continually created during the evolution of the systems. This implies the possibility to automatically generate new types of agents.
- **to offer a unitary approach to the modeling of direct communication and stigmergy.** Two major forms of communication between agents have been observed in natural systems like colonies of social insects. The first one is direct communication, used for example by honey bees. They are able to perform a so called “waggle dance”,

through which they inform other bees in the hive about the location and quality of a food source. The second form of communication, called stigmergy, is an indirect form of communication, based on traces left in the environment by an agent. These traces influence the subsequent actions of other agents or even of the agent itself. Stigmergy can be observed for example in some species of ants, which leave a pheromone trail on their way to the food source and back to the nest. In our opinion, there is no fundamental difference between these two types of communication, because the distinction between agents and environment elements is a subjective one, depending on the perspective taken. Therefore, our meta-framework should treat all types of communication in a unitary fashion.

Emergence is often associated with the occurrence of hierarchical structures. However, we do not require that MetFrEm facilitates the modeling of such structures. While they represent an important aspect of self-organization in complex systems, we are rather interested in whether and how hierarchical structures can emerge in decentralized systems with simple agents. Therefore, a typical framework considered in our research can be used in experiments aiming to observe the appearance of hierarchical structures, but it does not try to model them explicitly.

4.2. Related work

Existing frameworks for the study of emergence differ in their goals and in the aspects of emergence they try to capture. Artificial chemistries investigate the emergence of life and of its underlying evolutionary mechanisms by introducing a framework that takes inspiration from real chemical systems. A comprehensive review is given in [39], where the following definition is offered: "an artificial chemistry can be defined by a triple (S, R, A), where S is the set of all possible molecules, R is a set of collision rules representing the interaction among the molecules, and A is an algorithm describing the reaction vessel or domain and how the rules are applied to the molecules inside the vessel". One of the first and best known models of artificial chemistry is the *AlChemy* [48] system, which uses λ-calculus to describe the interactions between molecules.

John H. Holland [57] argues that in complex systems, emergence can be associated with the existence of procedures able to generate dynamic behaviors, together with constraints that restrict the set of such possible behaviors. Consequently, he introduces a framework called *constrained generating procedures* (CGP), which permits the modeling of virtually any complex system. The basic elements in CGP are *mechanisms*, which are characterized by their *state* and their *transition function*. Mechanisms have inputs, but no outputs. However, it can be considered that the state of a mechanism represents its output. Therefore, when one mechanism is connected to another, the state of the first mechanism is transformed by an *interface function* into a sequence of values, which are placed on one of the second mechanism's inputs. In the standard framework, the connections between mechanisms are fixed, but Holland also proposes an extension of his framework, called *constrained generating procedures with variable geometry* (CGP-v), which allows mechanisms to make and break their connections.

Another framework for the study of emergence, introduced by Kubík [72], is based on grammar systems. In this framework, agents are represented by grammars that operate on a number of shared symbolic tapes, which form the environment. In order to define emergence, Kubík provides a formal interpretation of the phrase "the whole is greater than the sum of its parts" using concepts available in his framework. Thus, a system exhibit emergent behavior,

if the language generated by the interacting grammars cannot be generated by the superimposition of the individual languages of these grammars.

4.3. Concepts and structure of MetFrEm

Universe
We denote as *universe* an agent-based model represented in MetFrEm. The universe representation is consistent with the definition of an agent-based model introduced in the previous chapter. This means that a universe is a discrete time model that has no environment. Environment elements can be represented as ordinary agents, which are called *mechanisms* in MetFrEm's terminology. A metamodel in MetFrEm involves only one universe, which in turn contains all metamodel's mechanisms.

Mechanism
In order to denote an agent in MetFrEm, we adopt the term *mechanism*, which was introduced by Holland to describe his CGPs. However, the structure of a mechanism in MetFrEm differs from the structure of a mechanism in CGP. An important characteristic of mechanisms in MetFrEm is that they know how to interact with any other mechanisms, without needing to know what types of mechanisms exist in the universe.

Property
The *internal state* of a mechanism is characterized by the values of its *internal properties*. It is not mandatory that each type of mechanism has the same set of internal properties. However, the set of properties that characterizes a mechanism is a subset of a finite *global set of properties* defined in the given universe. When there is no possible confusion we will use the term *properties* to denote the internal properties and the term *state* to denote the internal state of a mechanism. Each property has a numeric value that can change over time. In general, a mechanism does not fully expose its state to other mechanisms. Moreover, in different contexts, a mechanism can expose different sets of properties, by using the appropriate *view* for the given context.

View
A *view* represents a set of properties exposed by a mechanism to other mechanisms. This set of properties can be a subset of the internal properties or it can involve properties whose values aggregate the values of some internal properties. The properties exposed by a view must also be a subset of the global set of properties of the universe. Each mechanism provides three views, which will be discussed in the following paragraphs: the *observable view*, the *active view* and the *reactive view*. Each of the three views has a corresponding *view function*, which is used to compute the values of the exposed properties. Accordingly, these view functions are: the *observable view function* v_O, the *active view function* v_A and the *reactive view function* v_R. We call *observable state* of a mechanism the set of values returned by the observable view function v_O.

Neighborhood
At each time step, a mechanism can initiate an interaction with a subset of mechanisms. This subset is selected from a family of potential subsets, which represents the *neighborhood* of the mechanism at the given time step. Each mechanism has an associated *neighborhood function* η, which returns the neighborhood of the mechanism at a given time step, based on the *observable states* of the other mechanisms in universe.

Evaluation function
In order to choose the subset of mechanisms with which it initiate an interaction at a given time step, a mechanism has to evaluate each subset in its neighborhood. This operation is performed by using an *evaluation function* ψ, which returns a numerical value indicating how suitable a given subset is. The suitability of a subset is computed based on the values of the properties exposed by the *observable view* of each of the mechanisms in the subset.

Selection function
Based on the suitability values returned by the evaluation function, a mechanism selects the subset of mechanisms with which it initiates an interaction, by using a *selection function* σ.

Interaction functions
In MetFrEm, one of the mechanisms involved in an interaction plays an active role and it is called the *initiator* of the interaction. The other mechanisms, which are determined by the selection function σ, represent the *target* of the interaction and play a reactive role. During the interaction, the initiator exposes its state to the target mechanisms by means of its *active view*, while a target mechanism exposes its state to the initiator and to other target mechanisms by means of its *reactive view*. In general, the state of a mechanism changes as a result of the interaction. This change is reflected by changes in the values of the internal properties of the mechanism. The new internal state is computed by applying an *interaction function*. Each mechanism has two associated interaction functions: an *active interaction function* f_A and a *reactive interaction function* f_R. Which function is used depends on the role played by the mechanism in interaction: the initiator mechanism computes its new state by using the *active interaction function*, while the target mechanisms compute their new states by using the *reactive interaction function*.

Transformation function
As a result of an interaction, a mechanism may continue to exist, it may die or it may create new mechanisms. These operations are performed by a *transformation function* φ, which replaces the mechanism with a set of other mechanisms. If the mechanism continues to exist and no new mechanisms are created, this set is represented by the mechanism itself. If the mechanism dies and no new mechanism are created, the transformation function returns an empty set. If new mechanisms are created, the transformation function returns a finite set of mechanisms, which also contains the mechanism itself, if it continues to exist.

4.4. A formal description of MetFrEm

In this section, we give a formal description of the concepts presented in the above paragraphs. We first introduce the *set of global properties* of a universe:

$$\mathcal{P} = \{p_1, p_2, p_3, \dots\ p_z\}$$

where $z \in \mathbb{N}$ is the number of global properties.

Each element p_i of the set $\mathcal{P}$ identifies a measurable property. For example, a universe that models microscopic particles could involve properties such as: mass, position and velocity. In MetFrEm, the value of a property is a dimensionless quantity expressed as a real number.

We denote by $\mathcal{M}$ the infinite set of all possible mechanisms. A *universe* in MetFrEm is a tuple ($\mathcal{P}$, M_0), where $\mathcal{P}$ is the global set of properties and $M_0 \subset \mathcal{M}$ is the initial finite set of mechanisms.

A *mechanism* is defined as a tuple (P_I, P_O, P_A, P_R, s_0, υ_o, υ_A, υ_R, η, ψ, σ, φ, f_A, f_R), where:

- $P_I \subseteq \mathcal{P}$ is the set of internal properties;
- $P_O \subseteq \mathcal{P}$ is the set of observable view properties;
- $P_A \subseteq \mathcal{P}$ is the set of active view properties;
- $P_R \subseteq \mathcal{P}$ is the set of reactive view properties;
- $s_0 \in \mathbb{R}^{|P_I|}$ is the initial internal state, that is, the initial set of values of the internal properties;
- $v_o : \mathbb{R}^{|P_I|} \to \mathbb{R}^{|P_O|}$ is the observable view function, which computes the set of values representing the observable state of the mechanism;
- $v_A : \mathbb{R}^{|P_I|} \to \mathbb{R}^{|P_A|}$ is the active view function, which computes the set of values exposed during an interaction initiated by this mechanism;
- $v_R : \mathbb{R}^{|P_I|} \to \mathbb{R}^{|P_R|}$ is the reactive view function, which computes the set of values exposed during an interaction in which this mechanism is a target;
- $\eta : 2^{\mathbb{R}} \to 2^{2^{\mathcal{M}}}$ is the neighborhood function, which returns a family of subsets of mechanisms with which this mechanism could initiate an interaction. The return value is computed based on the observable states of all mechanisms in the universe.
- $\psi : 2^{\mathbb{R}} \to \mathbb{R}$ is the evaluation function, which computes the suitability of a subset of mechanisms from the family of potential subsets, based on their observable states.
- $\sigma : 2^{\mathbb{R}} \to 2^{\mathcal{M}}$ is the selection function, which chooses the subset of mechanisms with which this mechanism initiates an interaction, based on the suitability of each potential subset.
- $\varphi : \mathcal{M} \to 2^{\mathcal{M}}$ is the transformation function, which returns a set of mechanisms reflecting the transformation undergone by this mechanism at the end of an interaction: it may continue to exist, it may die and/or it may create new mechanisms.
- $f_A : 2^{\mathbb{R}} \to 2^{\mathbb{R}}$ is the active interaction function used by the initiator mechanism in order to compute its new internal state, based on its current state and on the values of the reactive properties of the target mechanisms.
- $f_R : 2^{\mathbb{R}} \to 2^{\mathbb{R}}$ is the reactive interaction function used by a target mechanism in order to compute its new internal state, based on its current state, on the values of the active properties of the initiator mechanism, and on the values of the reactive properties of the other target mechanisms.

In the next paragraphs, we show how the dynamic of the universe results from the application of the functions associated with the mechanisms. At the same time, we analyze whether a MetFrEm universe represents an agent-based model in the sense given by the definition 3.4 introduced in the previous chapter. A first apparent problem is posed by the selection function σ. In the definition 3.4, the domain of the selection function is the set of all agents' states. In contrast, the input of a selection function in MetFrEm is given by the set of suitability values of potential subsets. Let us see how the set of mechanisms with which a given mechanism initiates the interaction is computed in MetFrEm. The neighborhood of a mechanism μ, that is, the set of potential subsets of mechanisms with which it can initiate an interaction at a time step t, is computed as:

$$\mathcal{N}_t^{\mu} = \eta^{\mu}\left(\bigcup_{m \in M_t} v_O(s_t^m)\right) \tag{4.4}$$

where M_t is the set of all mechanisms that exist at time step t and s_t^m is the internal state of the mechanism m at time step t.

The set of suitability values corresponding to each potential subset is given by:

$$B_t^\mu = \bigcup_{N \in \mathcal{N}_t} \psi^\mu(\bigcup_{m \in N} \upsilon_O^\mu(s_t^m)) \tag{4.5}$$

The actual subset of mechanisms with which a given mechanism μ initiates an interaction is:

$$V_t^\mu = \sigma^\mu(B_t^\mu) \tag{4.6}$$

In MetFrEm, a mechanism can interact with other mechanisms in an active or in a reactive way. The set of all mechanisms with which a mechanism interacts must therefore include the mechanisms involved in either of these types of interactions. For a given mechanism μ, this set is:

$$W_t^\mu = V_t^\mu \cup \{m \in M_t \mid \mu \in V_t^m\} \tag{4.7}$$

It can be noticed that although the set of mechanisms with which a given mechanism interacts is determined in a rather complex way, the result depends ultimately only on the internal states s_t^m of the mechanisms in the universe. Therefore, it is possible to construct a function σ^* representing a composition of σ, ψ, η and υ_O, having as domain the set of all mechanisms' states and returning the set of interacting mechanisms. The σ^* function has the same form and the same meaning as the evolution function from the definition 3.4.

A second issue concerning the compatibility of a MetFrEm universe with the formal definition of an agent-based model is given by the interaction functions. In MetFrEm, it is possible that at a given time step t, a mechanism is the subject of several interactions: it may be the initiator in one of these interactions and the target in the others. The new state after an interaction in which the mechanism acts as initiator is computed using the active interaction function f_A. The new state after an interaction in which the mechanism acts as target is computed using the reactive interaction function f_R. The final state of the mechanism is obtained by successively applying these functions for each interaction of the given mechanism. If we denote by μ the initiator mechanism, its preliminary state after the interaction, that is, the state before applying the transformation rule, is given by:

$$s'^\mu_{t+1} = f_A^\mu(\{s_t^\mu\} \cup \{\upsilon_R^m(s_t^m) \mid m \in V_t^\mu\}) \tag{4.8}$$

The preliminary state of a target mechanism λ after the interaction is:

$$s'^\lambda_{t+1} = f_R^\lambda(\{s_t^\lambda\} \cup \{\upsilon_A^\mu(s_t^\mu)\} \cup \{\upsilon_R^m(s_t^m) \mid m \in V_t^\mu, m \neq \lambda\}) \tag{4.9}$$

It can be observed that, ultimately, the preliminary states are determined only by the internal states s_t^m of the mechanisms in the universe. It is therefore possible to construct an interaction function f^* having as domain the set of all mechanisms' states and returning the state of a mechanism after all interactions undergone at a given time step t. The f^* function has the same form and the same meaning as the interaction function from the definition 3.4.

In addition, let us notice that MetFrEm defines the interaction between agents in a way that is inherently reflexive in the sense required by the definition 3.4: if a mechanism $m1$ interacts with a mechanism $m2$ at a given moment t, then $m2$ also interacts with $m1$ at the moment t.

Finally, note that the definition of the transformation function φ in MetFrEm is identical with that introduced in the definition 3.4. The definitive configuration of the universe at time step $t + 1$ is:

$$M_{t+1} = \bigcup_{m \in M_t} \varphi^m(s'^m_{t+1}) \tag{4.10}$$

Based on the observations above, we conclude that a universe in MetFrEm is compatible with the formal definition of an agent-based model.

4.5. The algorithmic structure

The evolution of a universe in MetFrEm is presented in pseudocode in the figure below:

```
1  procedure MetFrEm ()
2      M ← initializeUniverse ()
3      while (termination condition not met) do
4          executePreliminaryActions (M)      // optional
5          foreach initiator ∈ M do
6              neighborhood ← getNeighborhood (initiator)
7              suitabilities ← evaluateNeighborhood(neighborhood)
8              targets ← selectTargets (neighborhood, suitabilities)
9              initiator.state ← activeInteraction (initiator, targets)
10             replacement ← transform(initiator)
11             M ← (M \ {initiator)} ∪ replacement
12             foreach target ∈ targets do
13                 target.state ← reactiveInteraction (target, initiator, targets)
14                 replacement ← transform(target)
15                 M ← (M \ {target)} ∪ replacement
16             end foreach
17         end foreach
18         executeFinalActions (M)          // optional
19     end while
20 end procedure
```

Figure 4-3. Pseudocode of the evolution of a universe in MetFrEm.

During the initialization phase (line 2), the initial set of mechanisms is created and the state of each mechanism is configured. The algorithm enters then the main loop (lines 3-19). In order to allow modeling of systems that do not perfectly fit the formal structure specified by MetFrEm, the algorithm offers two optional procedures: *executePreliminaryActions* (line 4) and *executeFinalActions* (line 18). Typically, these procedures perform global operations that cannot be modeled as decentralized actions, but the meta-framework does not impose any restriction on what operations they can execute. For example, in some Ant Colony Optimization algorithms, only the best-so-far ant is allowed to update the pheromone trails. This operation requires global knowledge about the solutions constructed by all ants and could be therefore implemented by one of these optional procedures.

While MetFrEm does not specify the actions performed by the optional procedures, the working of the other procedures referred in the pseudocode above is completely determined by the underlying functions and views introduced by the formal description. The *getNeighborhood* procedure (line 6) uses the neighborhood function η in order to determine the set of potential subsets of mechanisms with which the initiator could interact, in accordance with formula 4.4. The *evaluateNeighborhood* procedure (line 7) uses the evaluation function ψ in order to compute the suitability value of each set of mechanisms in the neighborhood, in accordance with formula 4.5. The *selectTargets* procedure (line 8) uses

the selection function σ in order to choose the set of target mechanisms, in accordance with formula 4.6. The *activeInteraction* procedure (line 9) uses the active interaction function f_A in order to compute the state of the initiator after the interaction, in accordance with formula 4.8. The *reactiveInteraction* procedure (line 13) uses the reactive interaction function f_R in order to compute the state of a target mechanism after the interaction, in accordance with formula 4.9. The *transform* procedure (lines 10 and 14) uses the transformation function φ in order to determine the set of mechanisms that replaces a given mechanism, in accordance with formula 4.10.

4.6. Analysis of MetFrEm

As mentioned before, one of the main goals in designing MetFrEm was the possibility to model highly heterogeneous agent-based systems, where a mechanism knows how to interact with any other mechanism, without needing to know what types of mechanisms exist in the universe. For this purpose, the interaction depends only on the application of mechanism specific functions and it is defined only in terms of property values exposed by views. The concept of type is not even used in MetFrEm. However, this does not mean that MetFrEm prevents the modeling of systems that rely on the concept of type. This would be a major constraint, because many agent-based algorithms require that the agents are aware of the types of other agents and implement different interaction rules based on the types of agents that interact. It is easy to model such an algorithm in MetFrEm, by treating the type of a mechanism as one of its observable properties. This way, all mechanism specific functions are able to different implement rules for different types of mechanisms. Although for the considered model the *type* property has a special meaning, from MetFrEm's perspective, it is nothing more than a regular observable property.

In the formal description of MetFrEm, properties can have only numerical values. However, since a numerical encoding is possible for any type of information, we can consider without loss of generality that any type of values is permitted for properties.

While modeling a meta-algorithm, it is not required to use all concepts offered by MetFrEm. For example, in many cases the set of mechanisms does not change over time. Mechanisms do not die and no new mechanisms are created. This means that the transformation function φ is the identity function, which implies that lines 10, 11, 14 and 15 from Figure 4-3 have no effect in this case. There are also many meta-algorithms where agents fully expose their internal state. In such cases, the view functions v_O, v_A and v_R are all identity functions. In other cases, these functions are projections that simply return a subset of the values corresponding to the internal properties, while in most complex scenarios the return values of these functions aggregate some of the values of the internal properties. Similarly, there are meta-algorithms where an agent interacts at each time step with the same fixed set of agents. In this case, the neighborhood function η returns a single constant set of mechanisms, the evaluation function ψ become unnecessary and the selection function σ returns the single set of mechanisms in the neighborhood. Finally, it is possible that the set of targets returned by the selection function is empty. Even in this case, the state of the initiator may change as a result of the interaction with the empty set. Such a change is equivalent with an evolution in time of the initiator's state. In many cases, this time evolution is a component independent of the "real" interactions of a mechanism. Therefore the active interaction function f_A can be implemented as a superposition of two other functions: the time evolution function and the "real" interaction function.

4.7. Modeling various systems in MetFrEm

MetFrEm's first purpose is to help the study of emergence and the implementation of new meta-algorithms and agent-based systems that make use of emergent behavior. However, modeling existing algorithms and systems in MetFrEm is also useful, because it permits a comparative analysis and a better understanding of their characteristics. In this section, we use several such existing algorithms and systems as a means to illustrate the use of MetFrEm.

In order to model a system in MetFrEm, one must provide a description of the properties, views and functions specified in the formal representation of this meta-framework. This description refers in turn to operations expressed in terms of the modeled system. When MetFrEm is used for meta-modeling, that is, for modeling a meta-algorithm or a high-level representation of a system, some of these operations are not completely specified. They become fully specified only when the template provided by the meta-algorithm is applied to a specific scenario.

If necessary, the MetFrEm model may also specify a series of operations to be performed by the optional procedures *executePreliminaryActions* and *executeFinalActions* referred in the pseudocode from Figure 4-3.

In order to distinguish the properties available in the global set from their corresponding values, we adopt the convention to start the name of a property with a capital letter, while values of this property should be denoted by using a lowercase as the first letter. For example, the global set of properties could contain a property called *Velocity*. The corresponding value of this property for a given mechanism at a given time step should be denoted as *velocity*.

Additionally, in the case studies presented below, we use the following notations:

- id – the identity function: $id(x) = x, \forall x$
- $idsel$ – the function that returns the unique element of the set representing its argument: $idsel(\{x\}) = x, \forall x$. The return value is unspecified if the argument of this function is a set containing more than one element.
- $zero$ – the function that always returns 0: $zero(x) = 0, \forall x$
- $empty$ – the function that always returns an empty set: $empty(x) = \emptyset, \forall x$
- π – the projection on a set of elements. It is typically used to describe view functions that simply return a subset of the values corresponding to the internal properties. For example, consider a universe having the set of internal properties P_I = {*Mass*, *Velocity*} and the set of observable properties P_O= {*Velocity*}. The observable view function is simply a projection of the internal state on the elements of P_O, that is, on *Velocity*: $\upsilon_O = \pi_{P_O} = \pi_{Velocity}$.

4.7.1. Case study: Cellular automata

Cellular automata (CA), presented in section 2.3, are deterministic systems characterized by only local interactions. The type of a CA depends on several factors, such as: the number of dimensions, the number of states, the type of neighborhood, the update rule or the type of cellular universe. We first show how a meta-model of CA can be described in MetFrEm, then we instantiate this meta-model for two specific types of CA: a one-dimensional CA implementing rule 110 and the two-dimensional CA representing Conway's Game of Life.

In order to distinguish concepts that have identical names but different meanings in MetFrEm and CA, we use the terms *cell state*, *cell neighborhood* and *cellular universe* to denote the concepts associated with CA.

In a CA, the cell states are updated simultaneously, but the algorithmic structure imposed by MetFrEm allows only sequential update of the mechanisms' states. In order to model a CA in MetFrEm, we divide the cell interactions in two phases. In the first phase, which we call the *computation phase*, the new value for each cell state is computed, but the cell states are not updated. In the second phase, which we call the *change phase*, the cell states are updated with the values computed in the previous phase.

Each cell of a CA can be implemented as a mechanism in MetFrEm. A cell is characterized by its position and its state. Since we model the evolution of a CA as a two-phase process, a mechanism needs two additional properties: a property indicating the current phase in the CA evolution and a property corresponding to the new cell state. Therefore, the global set of properties is:

$\mathcal{P}$ = {*Position*, *CellState*, *NewCellState*, *Phase*}

The *Position* property indicates the location of the mechanism in the cellular universe. The corresponding values are points in a d-dimensional space, where d represents the number of dimensions of the cellular universe: *position* $\in \mathbb{N}^d$. We denote the values corresponding to individual directions in this multidimensional space by adding a suffix that indicates the particular direction considered. For example, in a three-dimensional cellular universe, the position of a cell is described by the values: *position.x*, *position.y* and *position.z*.

If we denote by q the number of possible cell states of the considered CA, then the values of the properties *CellState* and *NewCellState* are natural numbers less than q:

cellState $\in \{0, 1, \dots, q-1\}$
newCellState $\in \{0, 1, \dots, q-1\}$

The *Phase* property admits two possible values: *phase* $\in \{0, 1\}$. The value 0 corresponds to the *computation phase*, while the value 1 corresponds to the *change phase*.

The meta-model is a generic description, which is not bound to a particular type of CA. Therefore, the operation of a CA cannot be fully specified at this level. In particular, it is not specified what cells compose the neighborhood of a given cell and how the state of a cell is updated based on the states of the cells in its neighborhood. For this reason, we introduce two generic functions, which must be instantiated in order to obtain the model of a concrete type of CA. The first function, called *cellNeighborhood*, returns a family of sets containing only one set[1]: the set of cells that constitute the neighbors of a given cell. The second function, called *updateRule*, computes the new state of a cell based on the states of its neighboring cells. In our MetFrEm model, this function is called only during the *computation phase*.

The cellular universe can be described in MetFrEm as follows:

$P_I = \mathcal{P}$ = {*Position*, *CellState*, *NewCellState*, *Phase*}
P_O = {*Position*}
$P_A = \emptyset$
P_R = {*CellState*}
$\upsilon_O = \pi_{P_O}$
$\upsilon_A = empty$
$\upsilon_R = \pi_{P_R}$
η = *cellNeighborhood*
$\psi = zero$

[1] It is necessary that the return value is a family of sets instead of just a set, in order to be compatible with the formal definition of the neighborhood function η.

$\sigma = idsel$
$\varphi = id$
$f_R = id$
f_A – pseudocode description using the variables *initiator* and *targets* referred in Figure 4-3:

```
if initiator.phase = 0 then
    targetCellStates ← {cell.cellState | cell ∈ targets }
    initiator.newCellState ← updateRule (initiator.cellState, targetCellStates)
else
    initiator.cellState ← initiator.newCellState
end if
initiator.phase ← 1 - initiator.phase
```

Our meta-model describes the interaction between a cell and its neighbors in a unidirectional way: only the initiator changes its state as a result of the interaction. The targets, that is, the neighbor cells, expose their cell states, but they are not affected by the interaction. Therefore, the initiator does not expose any properties and the reactive interaction function is the identity function. Since the cell neighborhood is fixed, there is no need for an evaluation function. Cells are not destroyed and no new cells are created during the evolution of the cellular universe, therefore the transformation function is the identity function.

The active interaction function computes the new cell states and stores them as *NewCellState* properties during the *computation phase* and updates the *CellState* properties during the *change phase*. The value of the *Phase* property is switched at each MetFrEm time step.

The CA meta-model perfectly fits the structure imposed by MetFrEm, therefore it does not need the optional procedures *executePreliminaryActions* and *executeFinalActions*.

MetFrEm model of the rule 110 one-dimensional CA
In order to apply our meta-model to a concrete type of CA, we must instantiate the generic functions *cellNeighborhood* and *updateRule*. The neighborhood of a one-dimensional CA is composed of its two adjacent cells. The location of a cell is described by a single coordinate: *position.x*. Therefore, the *cellNeighborhood* function can be described in pseudocode as follows:

```
function cellNeighborhood (initiator)
    leftCell ← idsel ({cell | cell.position.x = initiator.position.x – 1})
    rightCell ← idsel ({cell | cell.position.x = initiator.position.x + 1})
    return [leftCell, rightCell]
end function
```

The update rule for the rule 110 CA is given in the table below:

Current configuration	111	110	101	100	011	010	001	000
New cell state for the middle cell	0	1	1	0	1	1	1	0

The *updateRule* procedure can be described in pseudocode as follows:

```
function updateRule (initiatorCellState, targetCellStates)
    pattern ← targetCellStates[0] || initiatorCellState || targetCellStates[1]
    if pattern ∈ {111, 100, 000} then
        return 0
    else
        return 1
    end if
end function
```

In the above pseudocode, the symbol || denotes the concatenation operation.

MetFrEm model of Conway's Game of Life

Conway's Game of Life (described in subsection 2.3.2) is a two-dimensional CA, therefore the location of a cell is described by two coordinates: *position.x* and *position.y*. It uses a Moore neighborhood, which means that the neighborhood of a cell comprises its eight surrounding cells. The *cellNeighborhood* function can be described in pseudocode as follows:

```
function cellNeighborhood (initiator)
    return {cell | (cell.position.x - initiator.position.x) ∈ {-1, 1},
                   (cell.position.y - initiator.position.y) ∈ {-1, 1}}
end function
```

A cell in state 1 is considered a live cell, while a cell in state 0 is considered a dead cell. Conway's Game of Life uses the following totalistic update rule:

- A live cell with less than 2 or more than 3 live neighbors dies.
- A live cell with 2 or 3 live neighbors stays alive.
- A dead cell with exactly 3 live neighbors becomes a live cell.

The corresponding *updateRule* procedure can be described in pseudocode as follows:

```
function updateRule (initiatorCellState, targetCellStates)
    liveNeighbors ← 0
    foreach cell ∈ targetCells do
        if cell.state = 1 then
            liveNeighbors ← liveNeighbors + 1
        end if
    end foreach
    if initiatorCellState = 1 then
        if  liveNeighbors < 2 or liveNeighbors > 3 then
            return 0
        else
            return 1
        end if
    else
        if  liveNeighbors = 3 then
            return 1
        else
            return 0
        end if
    end if
end function
```

4.7.2. Case study: Ant Colony Optimization

Ant Colony Optimization (ACO), presented in section 2.4, is a metaheuristic inspired by the behavior of real ant colonies. It is typically used to solve combinatorial optimization problems where a solution can be incrementally constructed by adding at each step a new component from a given set of available components. We denote by $C = \{c_1, c_2, \ldots, c_{N_C}\}$ the finite set of components, where N_C is the number of components. A partial solution is a finite length sequence of components: $x = (c_{i_1}, c_{i_2}, \ldots, c_{i_k})$, where the number of components cannot exceed a given maximum value n: $k \leq n$. A complete solution is also considered a partial solution.

The components C can be seen as the nodes of a *construction graph* $G_C = (C, L)$, where L is the set of arcs that fully connects the components of C. In ACO, the solution construction is a randomized walk on the construction graph. At each construction step, an ant has to decide which component to visit next. This decision is biased by the concentration of pheromone on the arcs connecting the current component with the potential next components. Additionally, ants' decisions may be influenced by some heuristic information. The pheromone trail is updated by the ants themselves, by depositing a certain amount of pheromone.

Instead of interacting directly, ants coordinate their activities via stigmergy. We can consider that an ant interacts with an arc connecting two components, by perceiving the existing concentration of pheromone and by modifying it through the deposition of a new amount of pheromone. Therefore, our meta-model should contain two types of mechanisms: ants and arcs between components. As mentioned before, since MetFrEm does not use the concept of type, the type of a mechanism must be represented by a property.

An ant can perform a local pheromone trail update, by depositing pheromone on the current arc at each construction step, or it can perform a global update at the end of the construction phase, by depositing pheromone on the path representing a complete solution. In some algorithms, both methods are combined. However, only the local pheromone trail update can be in general regarded as a MetFrEm interaction. While a global update could be also modeled as a MetFrEm interaction, by considering that an ant interacts with all arcs contained in its newly constructed solution, most algorithms require global knowledge in order to decide which ants are allowed to perform the global pheromone trail update. Therefore, we treat the global update as an operation not fitting the algorithmic structure imposed by MetFrEm and we implement it as one of the operations performed by the optional procedure *executeFinalActions*. Most ACO algorithms use global update, because it improves the performance. Some early variants of the Ant System algorithm, called *ant-quantity* and *ant-density* [25] are among the algorithms that use exclusively local update. Ant Colony System (ACS) [43], which is one of the best-performing ACO algorithms, combines local and global pheromone trail update.

In order to create a meta-model of ACO in MetFrEm, we introduce a number of generic functions that, at this level, are only described in terms of their purpose. A concrete representation of these functions must be provided when instantiating the meta-model in order to describe a specific ACO algorithm. The generic functions are:

- *chooseStartComponent* – this function is called at the beginning of a solution construction phase, in order to choose the first component in the partial solution.
- *getNeighborComponents* – given a mechanism of type *ant*, this function returns a set of components representing the potential next destinations of the ant[2]. The main factor

[2] In order to be compatible with the formal definition of the neighborhood function η, this function must actually return a family of sets, each set containing a single element that represents a component.

that influences the result returned by this function is the current position of the ant, that is, the last element in the partial solution constructed by the given ant during its randomized walk on the construction graph. Since the construction graph is completely connected, this function may return, in principle, the entire set of components C. However, in most cases, there are a series of constraints that drastically reduce the number of components in the return set. For example, in many algorithms, an ant is not allowed to walk to an already visited component. In addition, many algorithms use *candidate lists* containing only a relatively small number of components, which are in some sense "close" to the current component. The argument of this function may be an empty partial solution at the beginning of a solution construction phase. In this case, the return value is the empty set $\emptyset$.

- *getCost* – given a partial solution, this function computes its cost. In many cases, this cost is the result of applying the objective function of the considered optimization problem to the partial solution. If the partial solution given as argument is an array containing only one component, the function returns 0. If the array contains two components, the result corresponds to the cost associated with the arc identified by these components.
- *getComponentSuitability* – given an arc that connects the current location of an ant with a potential next location, this function returns a value proportional to the probability to choose this arc.
- *chooseNextComponent* – based on a set of arcs and their corresponding suitability values, this function chooses the arc to be traversed next.
- *isCompleteSolution* – given a partial solution, this function returns a boolean value indicating whether the partial solution represents a complete solution.
- *updateLocalPheromoneTrail* – this function updates the pheromone concentration on the arc determined by the two components supplied as argument.
- *updateGlobalPheromoneTrails* – this function updates the pheromone trails at the end of a solution construction phase.

Our ACO meta-model contains the following global set of properties:

$\mathcal{P}$ = {*Type*, *PartialSolution*, *ArcComponents*, *PheromoneTrail*, *HeuristicInfo*}

The *Type* property admits two possible values: *type* $\in \{0, 1\}$. The value 0 corresponds to a mechanism of type *ant*, while the value 1 corresponds to a mechanism of type *arc*.

The values of the *PartialSolution* property are arrays of components. The last element of a *partialSolution* indicates the current location of the ant on the construction graph. The values of the *ArcComponents* property are arrays containing the two components that define an arc.

PheromoneTrail and *HeuristicInfo* are real-valued. The *pheromoneTrail* typically change as a result of the interactions between ants and arcs, while the *heuristicInfo* is typically not affected by interactions.

Since all mechanisms have identical sets of properties, ants must possess arc-specific properties and, conversely, arcs must possess ant-specific properties. The unused properties are initialized with default values and do not influence the evolution of the universe. An ACO meta-model can be described in MetFrEm as follows:

P_I = {*Type*, *PartialSolution*, *ArcComponents*, *PheromoneTrail*, *HeuristicInfo*}
P_O = {*Type*, *ArcComponents*, *PheromoneTrail*, *HeuristicInfo*}
$P_A = \emptyset$
P_R = {*Pheromone*, *HeuristicInformation*}
$\upsilon_O = \pi_{P_O}$

$\upsilon_A = empty$
$\upsilon_R = \pi_{P_R}$
η = *getNeighborComponents*
ψ = *getComponentSuitability*
σ = *chooseNextComponent*
$\varphi = id$
f_A – pseudocode description using the variables referred in Figure 4-3:

```
if targets = ∅ or isCompleteSolution (initiator.partialSolution) then
    initiator.partialSolution ← chooseStartComponent ()
else
    currComp ← initiator.partialSolution.last
    target ← idsel (targets)
    nextComp ← idsel ({c | c ∈ target.arcComponents, c ≠ currComp})
    initiator.partialSolution ← initiator.partialSolution ∪ nextComp
end if
```

f_R – pseudocode description using the variables referred in Figure 4-3:

```
updateLocalPheromoneTrail (target)
```

executeFinalActions = updateGlobalPheromoneTrails

In the pseudocode of the active interaction function f_A, the operator ∪ is used to indicate the addition of an element to an array. Also note that in the pseudocode of the above functions, the *initiator* mechanism is of type *ant*, while the *targets* mechanisms are of type *arc*.

MetFrEm model of Ant Colony System for the Traveling Salesman Problem

The Traveling Salesman Problem (TSP) was the first combinatorial optimization problem solved using ACO and is frequently used to assess the performance of new ACO algorithms. Ant Colony System (ACS) performs both local and global pheromone trail update, it introduces an aggressive choice rule and it uses candidate lists in order to restrict the number of available choices. The reader is referred to subsection ??? for a detailed description of this algorithm.

In the case of the TSP, the components that constitute the nodes of the construction graph are the cities to be visited. The cost of an arc in the construction graph represents the distance between the cities identifying the given arc.

In order to apply our ACO meta-model to the ACS algorithm for the TSP, we must instantiate the meta-model's generic functions. In the pseudocode below, N denotes the number of cities. The meaning of the parameters β, q_0, ξ, τ_0 and ρ is described in subsection ???.

```
function chooseStartComponent ()
    return random c ∈ C
end function
```

```
function getNeighborComponents (initiator)
    currentCity ← initiator.partialSolution.last
    return {c | c ∈ candidate_list[currentCity], c ∉ initiator.partialSolution }
end function
```

```
function getCost (partialSolution)
    cost ← 0
    previousCity ← partialSolution.first
    foreach c ∈ partialSolution do
        cost ← cost + distance[previousCity, c]
        previousCity ← c
    end foreach
    return cost
end function
```

```
function getComponentSuitability (initiator, target)
    return target.pheromoneTrail * target.heuristicInfo^β
end function
```

```
function chooseNextComponent (targets, suitabilities)
    q ← random ∈ [0,1]
    if q ≤ q0 then
        target ← argmax_{t∈targets}{suitabilities[t]}
        return target
    else
        target ← random proportional choice based on suitabilities
        return target
    end if
end function
```

```
function isCompleteSolution (partialSolution)
    if size(partialSolution) = N then
        return true
    else
        return false
    end if
end function
```

```
function updateLocalPheromoneTrail (target)
    target.pheromoneTrail ←(1 − ξ) * target.pheromonTrail + ξτ0
end function
```

```
function updateGlobalPheromoneTrail (M)
    search all mechanisms in M to determine bestSoFarAnt
    costBSF = getCost (bestSoFarAnt.partialSolution)
    foreach arc ∈ bestSoFarAnt.partialSolution do
        arc.pheromoneTrail ← (1 − ρ) * arc.pheromoneTrail + ρ / costBSF
    end foreach
end function
```

Some operations in the pseudocode above are described in plain text, because their implementation is not relevant to our discussion about MetFrEm.

4.7.3. Case study: Predator-Prey models

In the previous two case studies, the transformation function φ is the identity function, because the set of mechanisms does not change over time. In the predator-prey models considered in this subsection, this is no longer the case. Predators and prey can die and they can reproduce. One of the first and best known models of predation is the Lotka-Volterra model [77][116], which uses differential equations in order to describe the dynamic of interacting populations of predators and prey. Many other models are available in the literature, most of them with a higher complexity.

We consider in this subsection a very simple model, inspired by an article of Wilensky and Reisman [118]. This allows us to show that MetFrEm can be used to describe not only meta-models, but also ordinary models. In our model, predators and prey are wolves and sheep, which live on a rectangular grid of dimensions W x H.

Wolves need energy to live and they are able to gain it only by eating sheep. At each time step, a wolf moves to a randomly chosen location on the grid and it eats any sheep found at the new location. Each change of location costs the wolf an amount E1 of energy, and its energy increases by an amount E2, each time it eats a sheep. If the energy of a wolf decreases below 0, it dies. Wolves reproduce with a probability R1.

The sheep behavior is even simpler. At each time step, a sheep moves to a randomly chosen location on the grid and it reproduces with a probability R2.

In the considered model, wolves can start eating only after all animals have changed their locations. Therefore, in the corresponding MetFrEm model, we divide the interactions in two phases. In the first phase, which we call the *move* phase, the new locations of each animal are determined. In the second phase, which we call the *eat* phase, wolves are allowed to eat sheep and they may also die of starvation.

Our MetFrEm model contains the following global set of properties:

$\mathcal{P}$ = {*Phase*, *Type*, *X*, *Y*, *Energy*, *Dead*}

The *Phase* property admits two possible values: $phase \in \{0, 1\}$. The value 0 corresponds to the *move* phase, while the value 1 corresponds to the *eat* phase.

The *Type* property also admits two possible values: $type \in \{0, 1\}$. The value 0 corresponds to a mechanism of type *wolf*, while the value 1 corresponds to a mechanism of type *sheep*.

The *Energy* property is real-valued, while the *Dead* property has boolean values. The *Dead* property is needed only to mark that a sheep has been eaten. The dead of a wolf can be determined by checking its *energy*.

Our model can be described in MetFrEm as follows:

P_I = $\mathcal{P}$ = {*Phase*, *Type*, *X*, *Y*, *Energy*, *Dead*}
P_O = {*Type*, *X*, *Y*}
$P_A = \emptyset$
$P_R = \emptyset$
$\upsilon_O = \pi_{P_O}$
$\upsilon_A = empty$
$\upsilon_R = empty$

η = pseudocode description using the variables referred in Figure 4-3:

```
if initiator.type = 0 then
    return { {m} | m ∈ M, m.type=1,m.X=initiator.X, m.Y=initiator.Y }
else
    return ∅
end if
```

$\psi = zero$

$\sigma = idsel$

f_A – pseudocode description using the variables referred in Figure 4-3:

```
if initiator.phase = 0 then
    initiator.X ← random ∈ [1, W]
    initiator.Y ← random ∈ [1, H]
    if initiator.type = 0 then
        initiator.energy ← initiator.energy - E1
    end if
else
    if initiator.type = 0 then
        foreach sheep ∈ targets do
            initiator.energy ← initiator.energy + E2
        end foreach
    end if
end if
```

f_R – pseudocode description using the variables referred in Figure 4-3:

```
if target.phase = 1 then
    if target.type = 1 then
        target.dead ← true
    end if
end if
```

φ – pseudocode description using the notation m to denote the argument of this function:

```
replacement ← {m}
if m.phase = 1 then
    if m.type = 0 then
        if m.energy < 0 then
            replacement ← ∅
        else
            p ← random ∈ [0, 1]
            if p ≥ R1 then
                replacement ← replacement ∪ {new wolf}
            end if
        end if
    else
        if m.dead = true then
            replacement ← ∅
        else
            p ← random ∈ [0, 1]
            if p ≥ R2 then
                replacement ← replacement ∪ {new sheep}
            end if
        end if
```

```
    end if
end if
return replacement
```

The optional procedures *executePreliminaryActions* and *executeFinalActions* are not needed in this MetFrEm model.

5. THE CONSULTANT-GUIDED SEARCH METAHEURISTIC

The meta-framework introduced in the previous chapter has been designed to allow modeling of highly heterogeneous systems, with agents that know how to interact with any type of agent, without needing to know what types of agents exist in the system. For this reason, an interaction in MetFrEm is performed in two steps. First, the initiator chooses a set of targets from a family of potential sets, and then it interacts with the chosen targets. For simplicity, let us consider that the set of targets contains only one element, that is, there is only one target mechanism with which the initiator interacts. The initiator changes its state as a result of the interaction. The new state is computed based on the values present in the reactive view exposed by the target. Since the reactive view provides values that are in general not available in the observable view, these additional values can be regarded as private information offered by the target mechanism. The initiator gets this information only because it has chosen to interact with this specific target. We can see the target mechanism as a consultant that has expert knowledge, which it makes available to the mechanisms that are willing to interact with it. Similarly, the initiator can be seen as a client that chooses one of the available consultants, in order to get useful information. Viewed from this perspective, the interaction in MetFrEm has led us to the idea of a new heuristic method, which we call the *Consultant-Guided Search*, and which constitutes the subject of this chapter.

5.1. Introduction

Many combinatorial optimization problems of both practical and theoretical importance are known to be NP-hard. Since exact algorithms are not feasible in such cases, heuristics are the main approach to tackle these problems. Metaheuristics are algorithmic templates used to specify problem-independent optimization strategies, which can be instantiated in order to define problem-specific heuristics. Some of the most successful metaheuristics conceived in the last two decades are swarm intelligence techniques [16] like Ant Colony Optimization (ACO) [45], Particle Swarm Optimization (PSO) [70] or Bee Colony Optimization (BCO) [110]. They are population-based methods that make use of the global behavior that emerges from the local interaction of individuals with one another and with their environment.
Consultant-Guided Search (CGS) is a population-based metaheuristic for combinatorial optimization problems that takes inspiration from the way people make decisions based on advice received from consultants. Human behavior is complex, but CGS uses virtual people that follow only simple rules. Furthermore, there is no centralized control structure in CGS, and the group behavior self-organizes. These properties allow us to regard CGS as a swarm intelligence metaheuristic.

Individuals in a swarm intelligence system can use direct communication or stigmergy to exchange information. Stigmergy is an indirect form of communication based on traces left in the environment by an individual. Currently, ACO algorithms are the most successful swarm intelligence techniques that use stigmergic communication. In contrast, CGS uses only direct communication between individuals, thus bearing some resemblance to BCO.

5.2. The CGS Metaheuristic

In this section, after describing the CGS method, we present its formalization as a metaheuristic and analyze its operation.

5.2.1. Method Description

CGS is a population-based method. An individual of the CGS population is a virtual person, which can simultaneously act both as a client and as a consultant. As a client, a virtual person constructs at each iteration a solution to the problem. As a consultant, a virtual person provides advice to clients, in order to help them construct a solution.

The goal of a combinatorial optimization problem is to find values for discrete variables in order to obtain an optimal solution with respect to a given objective function. In CGS, a client constructs a solution as a sequence of steps, choosing at each step a value for one of the discrete variables of the considered problem.

At the beginning of each iteration, a client chooses a consultant that will guide it during the construction of the current solution. The first factor considered when deciding which consultant to choose is its ***reputation***. The reputation of a consultant depends on the number of successes achieved by its clients. We say that a client achieves a ***success***, if it constructs a solution better than all solutions found up to that point by any client guided by the same consultant. In this case, the consultant's reputation will be incremented. Should a client constructs a solution that is better than all solutions previously found by any client, irrespective of the consultant used, the consultant's reputation will receive a supplementary bonus. On the other hand, consultant's reputation fades over time. To keep its reputation, a consultant needs that its clients constantly achieve successes.

For each consultant, the algorithm keeps track of the best result obtained by any client working under its guidance. We use the term ***result*** in this context to refer to the value of the objective function applied to the considered solution. Based on these results, CGS maintains a ranking of the consultants. For a small number of consultants appearing at the top of this ranking, the algorithm prevents their reputations from sinking below a predefined level. This is motivated by the fact that the records (or at least the very good results) set by them in the past are still current, even if they happened long ago.

Besides reputation, another factor contributing to the choice of a consultant is client's ***personal preference***. The concretization of this concept is specific to each application of CGS to a given class of optimization problems.

Each consultant has a ***strategy***, which is used in order to guide its clients during the solution construction. If the consultant's reputation sinks below a minimum value, it will take a ***sabbatical leave***, during which it will stop offering advice to clients and it will instead start searching for a new strategy to use in the future. At each iteration, a virtual person on sabbatical leave constructs a new strategy, based on some heuristic. After a predefined period of time, the sabbatical ends and the virtual person selects the best strategy found during this period. This best strategy will be subsequently used to guide clients. At the end of the sabbatical, the consultant's reputation is reset to a predefined value.

At each step of an iteration, a client receives from the consultant chosen for this iteration a suggestion regarding the next action to be taken. Solution construction is a stochastic process in CGS, therefore the client will not always follow the consultant's recommendation. Usually, at each step there are several variants the client can choose from. The variant recommended

by the consultant has a higher probability to be chosen, but the client may opt for one of the other variants, which it selects based on some heuristic.

When a client achieves a success, it means having obtained a result that is better than the result it would have obtained following the consultant's recommendations at each step. For this reason, the consultant adjusts its strategy in order to reflect the sequence of decisions taken by the client. This way, a consultant dynamically improves its strategy every time one of its clients achieves a success.

5.2.2. The Metaheuristic

The pseudocode that formalizes the CGS metaheuristic is shown in Figure 5-1.

```
 1 procedure CGSMetaheuristic()
 2     create the set 𝒫 of virtual persons
 3     foreach p ∈ 𝒫 do
 4         setSabbaticalMode(p)
 5     end foreach
 6     while (termination condition not met) do
 7         foreach p ∈ 𝒫 do
 8             if actionMode[p] = sabbatical then
 9                 currStrategy[p] ← constructStrategy(p)
10             else
11                 currCons[p] ← chooseConsultant(p)
12                 if currCons[p] ≠ null then
13                     currSol[p] ← constructSolution(p, currCons[p])
14                 end if
15             end if
16         end foreach
17         applyLocalOptimization()                    // optional
18         foreach p ∈ 𝒫 do
19             if actionMode[p] = sabbatical then
20                 if currStrategy[p] better than bestStrategy[p] then
21                     bestStrategy[p] ← currStrategy[p]
22                 end if
23             else
24                 c ← currCons[p]
25                 if c ≠ null and currSol[p] is better than all solutions
26                     found by a client of c since last sabbatical then
27                         successCount[c] ← successCount[c] + 1
28                         strategy[c] ← adjustStrategy(c, currSol[p])
29                 end if
30             end if
31         end foreach
32         updateReputations()
33         updateActionModes()
34     end while
35 end procedure
```

Figure 5-1. The CGS metaheuristic

A virtual person may be in one of the following modes: *normal* and *sabbatical*. During the initialization phase (lines 2-5), virtual people are created and placed in sabbatical mode. Based on its mode, a virtual person constructs at each iteration (lines 7-33) either a solution to

the problem (line 13) or a consultant strategy (line 9). Optionally, a local optimization procedure (line 17) may be applied to improve this solution or consultant strategy.

The consultant used to guide the solution construction (line 11) is chosen based on its reputation and on the personal preference of the client. Since a virtual person act simultaneously as a client and as a consultant, it is possible for a client to choose itself as a consultant. The exact details of how reputation and personal preference are used in order to select a consultant are specific to each application of CGS to a particular class of problems. Similarly, the metaheuristic does not detail how a solution is constructed based on the strategy promoted by the consultant or how a strategy is constructed by a virtual person in sabbatical mode.

After the construction phase, a virtual person in sabbatical mode checks if it has found a new best-so-far strategy (lines 20-22), while a virtual person in normal mode checks if it has achieved a success and, if this is the case, its consultant adjusts its strategy accordingly (lines 24-29). Again, the method used to adjust the strategy is specific to each instantiation of the metaheuristic.

At the end of each iteration, the reputation and action mode of each virtual person are updated (lines 32-33).

Figure 5-2 details how consultants' reputations are updated based on the successes achieved by their clients. A concrete application of the metaheuristic must specify how reputations fade over time (line 4) and how the reputations of top-ranked consultants are prevented from sinking below a predefined level (line 13).

```
 1 procedure updateReputations()
 2     foreach p ∈ 𝒫 do
 3         if actionMode[p] = normal then
 4             rep[p] ← applyReputationFading(rep[p])
 5             rep[p] ← rep[p] + successCount[p]
 6             if currSol[p] is better than best-so-far solution then
 7                 rep[p] ← rep[p] + bonus
 8             end if
 9             if rep[p] > maxReputation then
10                 rep[p] ← maxReputation
11             end if
12             if isTopRanked(p) then
13                 rep[p] ← enforceMinimumReputation(rep[p])
14             end if
15         end if
16     end foreach
17 end procedure
```

Figure 5-2. Procedure to update reputations

Figure 5-3 details how the action mode of each virtual person is updated: consultants whose reputations have sunk below the minimum level are placed in sabbatical mode, while consultants whose sabbatical leave has finished are placed in normal mode.

```
1  procedure updateActionModes()
2      foreach p ∈ 𝒫 do
3          if actionMode[p] = normal then
4              if rep[p] < minReputation then
5                  setSabbaticalMode(p)
6              end if
7          else
8              sabbaticalCountdown ← sabbaticalCountdown – 1
9              if sabbaticalCountdown = 0 then
10                 setNormalMode(p)
11             end if
12         end if
13     end foreach
14 end procedure
```

Figure 5-3. Procedure to update action modes

Figure 5-4 and Figure 5-5 show the actions taken to place a virtual person in sabbatical or normal action mode.

```
1 procedure setSabbaticalMode(p)
2     actionMode[p] ← sabbatical
3     bestStrategy[p] ← null
4     sabbaticalCountdown ← sabbaticalDuration
5 end procedure
```

Figure 5-4. Procedure to set the sabbatical mode

```
1 procedure setNormalMode(p)
2     actionMode[p] ← normal
3     rep[p] ← initialReputation
4     strategy[p] ← bestStrategy[p]
5 end procedure
```

Figure 5-5. Procedure to set the normal mode

5.2.3. Method Analysis

The goal of a metaheuristic is to guide the solution search toward promising regions of the search space, where high-quality solutions are expected to be found. In CGS, the strategy of a consultant can be seen as a region of the search space that is advertised by this consultant. Because consultants with a higher reputation are more likely to be chosen by clients, it is important to ensure that the reputation of a consultant is in concordance with the probability to find high-quality solutions in the region of the search space that it advertises.

As the strategy of a consultant approaches a local or global optimum, the rate of achieving new successes decreases. This leads to a decrease in the consultant's reputation. As seen, CGS maintains a ranking of the consultants. This ranking is not based on reputation, but on the best result obtained until then by any client guided by the considered consultant. It is very likely that the global optimum lies in one of the regions advertised by consultants appearing at the top of the ranking. Preventing the reputations of these consultants to sink below a specified level guarantees that the search will continue in the regions that most likely contain the global optimum.

A heuristic should keep a balance between the exploration of new regions in the search space and the exploitation of promising regions already found. In CGS, the sabbatical leave allows to abandon regions of the search space that are no longer promising and to start exploring new regions. During the sabbatical, at each iteration a new region of the search space is visited. At the end, the best region found is chosen to be advertised as the consultant's strategy. Because this region had not been exploited before, it is expected that its corresponding results are initially rather modest. If clients would choose consultants based on their results, a consultant that has just finished its sabbatical leave would have only a small chance to be chosen. Fortunately, consultants are chosen based on their reputation and the reputation of a consultant is reset at the end of the sabbatical. The reset value should be low enough to prevent the overexploitation of a region whose potential is still unknown, but high enough to allow a few clients to exploit this region. If the region selected during the sabbatical leave really has potential, a great number of successes will be achieved in the next period for this region, leading to a rapid increase in the consultant's reputation. This way, the consultant is likely to rise to the top of the ranking, thus ensuring that its reputation will not fade below a specified level and, consequently, that the region of the search space it advertises will be further exploited by clients.

Two aspects should be taken into account when choosing the rate at which the reputation fades over time. On the one hand, a too high rate leads to the premature termination of the exploitation of promising regions of the search space, thus preventing the convergence of the algorithm. On the other hand, a too low rate leads to stagnation, because it keeps reputations at high values for a long time, thus preventing consultants from taking a sabbatical leave in order to explore new regions of the search space.

5.3. Modeling CGS in MetFrEm

The solution construction in CGS can be seen as the result of an interaction between a client and a consultant. Therefore, a meta-model of CGS in MetFrEm contains virtual persons as mechanisms. The client is the initiator of the interaction and the consultant is the target.

In CGS, a consultant is chosen based on its reputation and on client's personal preference. The reputation of a consultant is incremented each time one of its clients achieves a success. This increase in reputation can also be seen as the result of an interaction between a client and a consultant and it can be modeled by means of the reactive interaction function in MetFrEm. A consultant receives a reputation bonus when one of its clients constructs a best-so-far solution. In order to determine that a solution is best-so-far, global knowledge about the solutions constructed under the guidance of any consultant is needed. Therefore, we make use of the optional procedure *executeFinalActions* in order to describe the offering of a reputation bonus. In addition, we implement the reputation fading as another operation performed by the *executeFinalActions* procedure.

Our CGS meta-model contains the following global set of properties:

$\mathcal{P}$ = {*ActionMode*, *Solution*, *BestSolution*, *BestResult*, *Strategy*, *BestStrategy*, *Reputation*}

The *ActionMode* property admits two possible values: $actionMode \in \{0, 1\}$. The value 0 corresponds to a mechanism in *normal* action mode, while the value 1 corresponds to a mechanism in *sabbatical* action mode.

The Solution property identifies the solution constructed during the current iteration. The *BestSolution* property is relevant when a mechanism acts as a consultant and it represents the best solution constructed by any client working under the guidance of this consultant. The

BestResult property represents the cost of the best solution. It can be computed from the value of the *BestSolution* and therefore it is not considered an internal property of a mechanism in the meta-model. Instead, *BestResult* is one of the observable properties of a mechanism.

The *Strategy* property represents the strategy used by a consultant in normal mode. During the sabbatical mode, *Strategy* identifies the strategy constructed during the current iteration, while the BestStrategy property indicates the best strategy constructed since the beginning of the sabbatical leave.

We do not impose any constraints on the value types permitted for the *Solution*, *BestSolution*, *Strategy* and *BestStrategy* properties, because their structure is not specified at this level of description. The *Reputation* properties are relevant only for mechanisms in normal mode.

In order to create a meta-model of CGS in MetFrEm, we introduce a number of generic functions that, at this level, are only described in terms of their purpose. A concrete representation of these functions must be provided when instantiating the meta-model in order to describe a specific CGS algorithm. The generic functions are:

- *constructSolution* – given a client and a strategy, this function constructs a solution biased by the given strategy. It may also improve the solution by applying a local optimization procedure.
- *getCost* – given a solution, this function computes its cost. It represents the objective function of the considered optimization problem.
- *constructStrategy* – given a consultant in sabbatical mode, this function constructs a new strategy. It may also improve the strategy by applying a local optimization procedure.
- *getStrategyCost* – given a strategy, this function computes its cost. The result returned by this function must represent the cost of a solution constructed by a client that follows at each construction step the recommendations of a consultant using the strategy given as argument.
- *getConsultantSuitability* – given a client and a consultant, this function returns a value proportional to the probability that the client chooses this consultant.
- *chooseConsultant* – based on a set of potential consultants and their corresponding suitability values, this function chooses the consultant for the next iteration.
- *adjustStrategy* – based on a set of potential consultants and their corresponding suitability values, this function chooses the consultant for the next iteration.
- *isSabbatical* – based on a set of potential consultants and their corresponding suitability values, this function chooses the consultant for the next iteration.

We describe now the CGS meta-model in MetFrEm using the notation conventions introduced in section 4.7:

P_I = {*ActionMode*, *Solution*, *BestSolution*, *Strategy*, *BestStrategy*, *Reputation*}
P_O = {*ActionMode*, *BestResult*, *Reputation*}
$P_A = \emptyset$
P_R = {*Strategy*, *BestResult*}
$\upsilon_O = \pi_{\{ActionMode,Reputation\}} \cup (getCost \circ \pi_{\{BestSolution\}})$
$\upsilon_A = empty$
$\upsilon_R = \pi_{\{Strategy\}} \cup (getStrategyCost \circ \pi_{\{Strategy\}})$
η = pseudocode description using the variable M referred in Figure 4-3:

> **return** $\{\{m\} \mid m \in M,\ m.actionMode = 0\}$

ψ = *getConsultantSuitability*
σ = *chooseConsultant*

$\varphi = id$

f_A – pseudocode description using the variables referred in Figure 4-3:

```
if initiator.actionMode = 0 then
    consultant ← idsel (targets)
    if consultant ≠ null then
        initiator.solution ← constructSolution (initiator, consultant.strategy)
    end if
else
    initiator.strategy ← constructStrategy (initiator)
    strategyCost ← getStrategyCost (initiator.strategy)
    if strategyCost < getStrategyCost(initator.bestStrategy) then
        initiator.bestStrategy ← initiator.strategy
    end if
end if
```

f_R – pseudocode description using the variables referred in Figure 4-3:

```
if getCost(initiator.solution) < target.bestResult then
    target.strategy ← adjustStrategy (initiator.solution)
    target.reputation ← target.reputation + 1
end if
```

executeFinalActions:

update reputations and action modes

The *executeFinalActions* procedure is described in plain text, because a pseudocode description would be very similar to that given in Figure 5-2 and Figure 5-3 for the procedures *updateReputations* and *updateActionModes*.

5.4. Positioning of CGS

CGS introduces a new metaphor, but it is not obvious whether it represents a novel metaheuristic or rather a reformulation of a known method, using new names for existing concepts. In this section, we try to place CGS in the context of heuristic optimization methods and we argue that it represents a hybrid metaheuristic, which combines new ideas with concepts found in other optimization techniques.

CGS can be classified as a model-based search (MBS) algorithm [129]. In MBS, candidate solutions are constructed using some parameterized probabilistic model. These solutions are then used to modify the probabilistic model in order to bias future sampling toward high quality solutions. CGS mainly differs from other MBS algorithms like ACO and estimation of distribution algorithms (EDA) [74] in the way it bias probabilities with respect to past experience in order to intensify the search around the best combinations. Instead of using an indirect pheromone-based learning mechanism (like in ACO), or estimation of distributions (like in EDA), a set of combinations is used in CGS to directly bias probabilities.

Somewhat surprisingly, CGS can even be cast into the formal framework of ant programming (AP) [13]. AP is based on the use of an iterated Monte Carlo approach for the multi-stage solution of combinatorial optimization problems. A population of agents, called ants, is used in order to construct solutions. Each agent perceives the state of the system through a representation, which can be seen as a mental image and, in general, gives less information than the state description. During the solution construction, each ant moves on the state graph, but it represents its movement on the representation graph. At each step, a set of feasible

candidate actions is determined based on information pertaining to the system state. One of these actions is then selected based on a probabilistic policy parameterized in terms of a desirability function. The desirability function associates a real value to each edge of the representation graph. For both ACO and CGS, a partial solution is expressed as a sequence of components, while the AP representation of a state can be expressed as the last component added to the partial solution. In the case of ACO, the desirability is given by the amount of pheromone on the edge connecting the last component included and the candidate component. In the case of CGS, the desirability function can be expressed in terms of two elements: the reputations and the strategies of the consultants. The reputations bias the probability to use a given strategy, which in turn biases the probability to select a given component. In AP, the desirability information is updated on the basis of the cost of the generated solutions. CGS fits into the AP framework, because both reputations and strategies are updated in accordance with the successes achieved by clients. This shows that the AP framework is not restricted to pheromone-based algorithms, but it can also accommodate algorithms that use non-stigmergic communication.

In CGS, the information exchange is based on direct communication, thus bearing some resemblance to bee inspired algorithms [69]. These algorithms mimic the behavior of real bees, which perform a so-called waggle dance in order to transmit information about the direction and distance to a food source. In this way, a bee is able to recruit other nest mates to the discovered food source. In the CGS metaphor the consultants wait passively to be selected by clients, but we can consider that they actually try to recruit clients. The recruitment procedure in CGS differs though from the recruitment procedure in bee inspired algorithms in the way the probability to recruit an agent is biased: CGS uses the consultant reputation, while most bee inspired algorithms use the solution quality.

CGS does not specify how a strategy is represented and how a consultant uses this strategy in order to produce a recommendation at each step of the solution construction. These details are specific to each instantiation of CGS for a given class of problems. One possibility would be to represent the strategy as a solution advertised by the consultant to its clients. Because at some steps a client may choose to not follow the recommendation of the consultant, the solution constructed by a client could be interpreted as a perturbation of a guiding solution. This means that the solution construction process is in this case similar to an iterated local search [78].

CGS combines several concepts found in other optimization techniques. For example, the reputation fading is similar to the pheromone evaporation in ACO. Another example is the construction of a new strategy during the sabbatical leave. This process resembles the escape mechanism used in Reactive Tabu Search [8] when the system is trapped in a *complex attractor*, or the pheromone trail reinitialization performed in MAX-MIN Ant System [106] when the algorithm approaches the stagnation behavior. As a final example, we consider how CGS keeps information about promising solutions, by means of consultant strategies. This approach of maintaining a list of high quality solutions can also be found in some ACO variants like Population-Based ACO [54] or $ACO_{\mathbb{R}}$ [102].

5.5. CGS applied to the Traveling Salesman Problem

In this section, we illustrate the use of the CGS metaheuristic by applying it to the Traveling Salesman Problem (TSP). To this end, we propose a concrete CGS algorithm, called CGS-TSP, and report the results of its application to symmetric instances of the TSP. Most swarm intelligence techniques have been applied to TSP, but currently ACO algorithms outperform BCO [111][126] and PSO [23][97] for this class of problems. Therefore, we are

interested if our direct communication approach can compete with stigmergy-based methods and we compare the performance of CGS-TSP with that of ACO algorithms. Our experimental results show that the solution quality obtained by CGS-TSP is comparable with or better than that obtained by the Ant Colony System (ACS) [43] and MAX-MIN Ant System (MMAS) [106].

5.5.1. The CGS-TSP algorithm

The CGS-TSP algorithm introduced in this subsection exemplifies how the CGS metaheuristic can be used to solve a specific class of problems, in this case the TSP.

TSP [3] is the problem of a salesman, who is required to make a round-trip tour through a given set of cities, so that the total traveling distance is minimal. The problem can be represented by a graph $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$ is the set of nodes and $E = \{(i, j): i, j \in V\}$ is the set of edges. Each edge $(i, j) \in E$ has an associated cost measure d_{ij}, which we will refer to as the distance between cities i and j. If $d_{ij} = d_{ji}$ for all $i, j \in V$, the problem is a symmetric TSP, otherwise it is an asymmetric TSP. CGS-TSP can be applied to symmetric instances of TSP, but it can be easily adapted to also solve asymmetric instances.

In order to apply CGS to a particular class of problems, one must define the different concepts used by this metaheuristic (e.g. strategy, result, personal preference) in the context of the given class of problems. Then, one must decide how to implement the actions left unspecified by the CGS metaheuristic (e.g. *constructStrategy*, *constructSolution*, *chooseConsultant*).

Constructing a solution for the TSP means building a closed tour that contains each node of the graph only once. To avoid visiting a node several times, each virtual person in CGS-TSP keeps a list of the nodes already visited in the current iteration.

The ***strategy*** of a consultant is represented by a tour, which it advertises to its clients; the ***result*** of a tour is computed as the inverse of its length. Since both solution construction and strategy construction imply building a tour, the type of decision a virtual person has to make at each step is the same in both cases: it has to choose the next city to be visited. However, the rules used to make decisions in each of these two cases are different.

To restrict the number of choices available at each construction step, CGS-TSP uses candidate lists that contain for each city *i* the closest *cand* cities, where *cand* is a parameter. This way, the feasible neighborhood of a person *k* when being at city *i* represents the set of cities in the candidate list of city *i* that person *k* has not visited yet.

During the ***sabbatical leave***, consultants build strategies using a heuristic based only on the distances between the current city and the potential next cities. The rule used at each step to choose the next city is inspired by the pseudorandom proportional rule introduced by the ACS algorithm, but since CGS does not use pheromones (or other stigmergic communication), our rule involves only distances between cities. A virtual person *k* located at city *i* moves to a city *j* according to the following rule:

$$j = \begin{cases} argmin_{l \in \mathcal{N}_i^k}\{d_{il}\} & , if\ a \le a_0 \\ J & , otherwise \end{cases} \tag{5.1}$$

where:

- $\mathcal{N}_i^k$ is the feasible neighborhood of person *k* when being at city *i*.
- d_{il} is the distance between cities *i* and *l*.
- a is a random variable uniformly distributed in [0,1] and a_0 ($0 \le a_0 \le 1$) is a parameter.

- J is a random variable selected according to the probability distribution given by formula (5.2), where β is a parameter.

$$p_{ij}^{k} = \frac{(1/d_{ij})^{\beta}}{\sum_{l \epsilon \mathcal{N}_i^k}(1/d_{il})^{\beta}} \tag{5.2}$$

In other words, with probability a_0 the person moves to the closest city in its feasible neighborhood, while with probability $(1 - a_0)$ it performs an exploration of the neighbor cities, biased by the distance to the city *i*.

At each step of the solution construction, a client receives from its consultant a recommendation regarding the next city to be visited. This recommendation is based on the tour advertised by the consultant. Let *i* be the city visited by the client *k* at a construction step of the current iteration. To decide which city to recommend for the next step, the consultant finds the position at which the city *i* appears in its advertised tour and identifies the city that precedes *i* and the city that succeeds *i* in this tour. If neither of these two cities is already visited by the client, the consultant recommends the one that is closest to city *i*. If only one of these two cities is unvisited, this one is chosen to be recommended. Finally, if both cities are already visited, the consultant is not able to make a recommendation for the next step.

The client does not always follow the consultant's recommendation. Again, a pseudorandom proportional rule is used to decide which city to visit at the next step:

$$j = \begin{cases} v & ,if\ v \neq null\ \wedge q \leq q_0 \\ argmin_{l \in \mathcal{N}_i^k}\{d_{il}\} & ,if\ (v = null\ \vee q > q_0) \wedge b \leq b_0 \\ J & ,otherwise \end{cases} \tag{5.3}$$

where:

- v is the city recommended by the consultant for the next step.
- q is a random variable uniformly distributed in [0,1] and q_0 ($0 \leq q_0 \leq 1$) is a parameter.
- $\mathcal{N}_i^k$ is the feasible neighborhood of person *k* when being at city *i*.
- d_{il} is the distance between cities *i* and *l*.
- b is a random variable uniformly distributed in [0,1] and b_0 ($0 \leq b_0 \leq 1$) is a parameter.
- J is a random variable selected according to the probability distribution given by formula (5.2).

In other words, if a recommendation is available, the client moves with probability q_0 to the city recommended by its consultant; with probability $b_0(1 - q_0)$ it moves to the closest city in its feasible neighborhood; with probability $(1 - b_0)(1 - q_0)$ it performs an exploration of the neighbor cities, biased by the distance to the city *i*.

The two factors that influence the choice of a consultant are: consultant's ***reputation*** and client's ***personal preference***. In CGS-TSP the personal preference is given by the result of the consultant's advertised tour, that is, by the inverse of the advertised tour length. The probability to choose consultant *k* is given by formula (5.4):

$$p_k = \frac{reputation_k{}^{\alpha} result_k{}^{\gamma}}{\sum_{c \epsilon \mathcal{C}} reputation_c{}^{\alpha} result_c{}^{\gamma}} \tag{5.4}$$

where:

- $\mathcal{C}$ is the set of all available consultants, that is, the set of all virtual people that are not in sabbatical mode.
- α is a parameter that determines the influence of the reputation.

- γ is a parameter that determines the influence of the result.

A client is allowed to choose itself as a consultant. Because the probabilities given by formula (5.4) do not depend on the client making the choice, the client index does not appear in this formula.

Every time a client achieves a success (i.e., it finds a tour shorter than the tour advertised by its consultant), the consultant updates its strategy, replacing its advertised tour with the tour constructed by the client.

At each iteration, the consultant's *k* reputation fades as given by formula (5.5):

$$reputation_k \leftarrow reputation_k(1 - r) \tag{5.5}$$

where *r* is the reputation fading rate. CGS-TSP adjusts its reputation fading rate according to the total number *s* of successes achieved during the last *w* iterations by the best *fadingRanks* consultants, where *w* and *fadingRanks* are parameters:

$$r = r_0 \left(1 + \frac{s}{\sqrt{1 + \left(\frac{s}{f}\right)^2}} \right) \tag{5.6}$$

The parameter r_0 gives the reputation fading rate for the case where no successes where achieved by the best *fadingRanks* consultants during the last *w* iterations. The parameter *f* controls how the reputation fading rate increases with the number *s* of successes. In particular, we have:

$$\lim_{s \to \infty} r = r_0(1 + f) \tag{5.7}$$

Since it is difficult to estimate what values are appropriate for the parameter *f*, we compute its value based on the value of another parameter. Let us denote by *repdec(s)* the decrease in reputation after *w* iterations in the hypothetical case that *s* remains constant during this period:

$$repdec(s) = (1 - r_{(s)})^w \tag{5.8}$$

We introduce the parameter k_w that indicates how much greater is the decrease in reputation for a very high number of successes than the decrease in the case when no successes were achieved:

$$repdec(0) = k_w \cdot \lim_{s \to \infty} repdec(s) \tag{5.9}$$

From equations (5.6) to (5.9) we have:

$$f = \left(\frac{1}{r_0} - 1\right)\left(1 - \frac{1}{\sqrt[w]{k_w}}\right) \tag{5.10}$$

Using for example *w*=1000, k_w=10 and $r_0 = 10^{-5}$, we obtain $f \cong 230$.

In CGS-TSP the reputation value cannot exceed a maximum value specified by the parameter *maxReputation*, and the reputation of a top-ranked consultant cannot sink below the limit given by: initialReputation · bestSoFarTourLen/advertisedTourLen.

The value of the parameter $minReputation$ referred in Figure 5-3 is fixed in this algorithm at 1, because only the differences between $minReputation$ and the other reputation parameters are relevant for the algorithm.

The optional local optimization step referred in Figure 5-1 can be implemented in CGS-TSP by a local search procedure.

5.5.2. Implementation

As mentioned before, our goal is to find out if the direct communication approach used by CGS can compete with stigmergy-based methods like ACO. To allow a meaningful comparison between heuristics, we have created a software package containing Java implementations of CGS-TSP, ACS and MMAS algorithms. The software package is available as an open source project at http://swarmtsp.sourceforge.net/. At this address, we also provide all configuration files, problem instances and results files for the parameter tuning and for the experiments described in this section.

We have tried to share as much code as possible between the implementations of CGS-TSP, ACS and MMAS, and we have taken care to keep the same level of code optimization while implementing the code portions specific to each algorithm. This way, we ensure that the relative differences in performance observed in our experiments reflect only the characteristics of the algorithms and are not due to implementation factors.

The implementation of the ACO algorithms is basically a port to Java of Thomas Stützle's ACOTSP program available at http://www.aco-metaheuristic.org/aco-code/.

5.5.3. Parameter tuning

For all algorithms, the experiments presented in the following subsection have been performed without local search and using candidate lists of length 20. In the first series of experiments, each algorithm run has been terminated after constructing 500000 tours. In the preliminary tuning phase, we have used these basic settings to identify parameter values that produce good results. In order to make a fair comparison between algorithms, we have performed parameter tuning not only for CGS-TSP, but also for ACS and MMAS.

5.5.3.1. CGS-TSP tuning

Because CGS-TSP is a completely new algorithm, we have performed the parameter tuning in several steps. First, we have searched for a good configuration in a set of candidate configurations that span a large range of values. After this coarse tuning step, we have performed a fine tuning, by using candidate configurations in a closer range around the configuration selected in the previous step. Finally, we have identified dependencies between two parameters or between a parameter and the size of the problem.

We have excluded from tuning the parameters that configure the sabbatical leave, because the purpose of the sabbatical is only to provide a decent strategy to start with and, in our opinion, the algorithm should not be very sensitive to these parameters. We have chosen $a_0 = 0.9$ and, in order to keep the sabbatical leave relatively short, $sabbaticalDuration = 100$. We have also fixed at 1000 the value of the parameter w referred in equations

In the coarse and fine tuning steps we have used the paramILS configuration framework [59] to find good parameter settings. ParamILS executes an iterated local search in the parameter configuration space and it is appropriate for algorithms with many parameters, where a full factorial design becomes intractable. As training data for paramILS, we have generated a set of 500 Euclidean TSP instances with the number of cities uniformly distributed in the interval

[300, 500], and with coordinates uniformly distributed in a square of dimension 10000 x 10000.

In the next step, using as starting point the configuration found during the fine tuning, we have checked if there is a dependency between the optimum value for $protectedRanks$ and the number m of virtual persons used. For each value of m in the interval [4, 20], we have tuned the value of $protectedRanks$ using the F-Race method and the previous training set. F-Race [14] is a racing configuration method that sequentially evaluates candidate configurations and discards poor ones as soon as statistically sufficient evidence is gathered against them. It is appropriate when the space of candidate configurations is relatively small, because at the start all candidate configurations are evaluated. Applying the linear least squares method to the optimum configurations found by F-Race, we have obtained the equation:

$$protectedRanks = 0.8 \cdot m - 2.5 \tag{5.11}$$

In the final step, using as starting point the configuration found during the fine tuning, with the value of $protectedRanks$ given by the equation (5.11), we have checked if there is a dependency between the optimum value for the number m of virtual persons and the number n of cities of a TSP instance. To this end, we have generated 9 sets of Euclidean TSP instances, with coordinates uniformly distributed in a square of dimension 10000 x 10000, each set comprising 100 instances. Each instance in the first set has a number n of 100 cities, in the second set 200 cities, and so on, until the last set with 900 cities. Using again F-Race and the linear least squares method, we have obtained the equation:

$$m = 3 + 1400/n \tag{5.12}$$

The parameter settings used in the experiments presented in the following subsection are shown in Table 5-1.

5.5.3.2. ACS tuning

ACS is a well-known algorithm, for which good parameter values are already available (see [45], p.71): m=10, ρ=0.1, ξ=0.1, q_0=0.9, $\beta \in$[2, 5]. We have performed a fine tuning of these parameters using paramILS and the same training set used in the coarse and fine tuning steps of CGS-TSP. The best configuration found, which has been used in the experiments presented in the following subsection, is: m=6, ρ=0.9, ξ=0.1, q_0=0.5, β=6.

5.5.3.3. MMAS tuning

As in the case of ACS, good parameter values are already available for MMAS (see [45], p.71): m=n, ρ=0.02, α=1, $\beta \in$[2, 5]. We have performed a fine tuning of these parameters using paramILS and the same training set used in the coarse and fine tuning steps of CGS-TSP. The best configuration found, which has been used in the experiments presented in the following subsection, is: m=n, ρ=0.1, α=1, β=5.

Some of the values found by paramILS during the tuning of ACS and MMAS differ significantly from the standard values given in [45]. For this reason, for ACS and MMAS, we have repeated the experiments described in the next subsection, using the standard settings. The results obtained using the tuned parameters outperform those obtained with the standard settings, thus confirming the effectiveness of paramILS.

Table 5-1. Parameter settings for CGS-TSP

Parameter	Value	Description
m	3+1400/n	number of virtual persons
b_0	0.3	solution construction parameter *see formula (5.3)*
q_0	0.98	solution construction parameter *see formula (5.3)*
α	1	reputation's relative influence *see formula (5.4)*
β	10	heuristic's relative influence *see formula (5.2*)
γ	4	result's relative influence *see formula (5.4)*
maxReputation	70	maximum reputation value
initialReputation	10	reputation value after sabbatical
bonus	15	reputation bonus for best-so-far tour
protectedRanks	0.8*m-2.5	protected top consultants
r_0	10^{-5}	basic reputation fading rate *see formula (5.6)*
fadingRanks	2	top consultants for fading rate *see formula (5.6)*
k_w	20	reputation decrease factor *see formulas (5.9) and (5.10)*

5.5.4. Experimental results

To compare the performance of CGS-TSP with that of ACS and MMAS, we have applied these algorithms without local search to 15 symmetric instances from the TSPLIB benchmark library [92]. The number of cities of the TSP instances used in our experiments is between 150 and 1060. We have intentionally included TSP instances whose number of cities lies outside the range of values considered in the tuning phase (300 to 500), because a good algorithm is not problem dependent, and therefore it should not be very sensitive to the set of training instances used for tuning.

The experiments have been performed on an HP ProLiant with 8 x 2.33 GHz Intel(R) Xeon(R) CPUs and 16 GB RAM, running Red Hat Enterprise Linux 5.

In the first series of experiments performed to compare CGS-TSP with ACS and MMAS, we have stopped each run after constructing 500000 tours. Table 5-2 reports for each algorithm and TSP instance the best and mean percentage deviations from the optimal solutions over 25 trials, as well as the sample standard deviations of the means. The best results for each problem are in boldface. We also report for each problem the p-values of the one-sided Wilcoxon rank sum tests for the null hypothesis (H_0) that there is no difference between the solution quality of CGS-TSP and that of the considered algorithm, and for the alternative hypothesis (H_1) that CGS-TSP outperforms the considered algorithm. Applying the Bonferroni correction for multiple comparisons, we obtain the adjusted α–level: 0.05 / 15 = 0.0033. The p-values in boldface indicate the cases where the null hypothesis is rejected at this significance level.

Table 5-2. Comparison of CGS-TSP with ACS and MMAS. Runs are terminated after constructing 500000 tours.

Problem name	CGS-TSP			ACS				MMAS			
	Best (%)	**Mean (%)**	**Mean stdev**	**Best (%)**	**Mean (%)**	**Mean stdev**	**p-value**	**Best (%)**	**Mean (%)**	**Mean stdev**	**p-value**
kroA150	**0.098**	**0.659**	0.280	0.151	1.145	0.744	0.0067	0.222	0.759	0.253	0.2022
kroB150	**0.000**	0.510	0.304	**0.000**	0.881	0.512	0.0076	**0.000**	**0.140**	0.239	1.0000
si175	**0.000**	**0.074**	0.049	0.051	0.173	0.091	< **0.0001**	0.028	0.124	0.065	0.0045
kroA200	0.051	0.286	0.181	0.140	0.861	0.756	**0.0002**	**0.041**	**0.178**	0.113	0.9740
kroB200	**0.258**	**0.840**	0.345	0.265	1.141	0.740	0.1522	0.455	0.853	0.202	0.5019
pr299	**0.218**	**0.777**	0.491	0.826	1.805	0.574	< **0.0001**	0.494	0.790	0.267	0.0472
lin318	0.949	1.430	0.309	**0.305**	1.708	0.867	0.0462	0.404	**0.834**	0.302	1.0000
pr439	**0.632**	**1.518**	0.649	0.661	2.294	1.686	0.0880	1.054	2.216	0.432	< **0.0001**
pcb442	**0.825**	**1.610**	0.395	1.266	2.926	1.107	< **0.0001**	0.924	1.762	0.668	0.4257
att532	1.383	2.015	0.261	1.380	2.653	0.814	**0.0006**	**1.015**	**1.622**	0.338	1.0000
u574	**2.073**	**2.841**	0.396	2.412	5.651	2.025	< **0.0001**	2.078	3.205	0.541	0.0036
gr666	**1.705**	**2.933**	0.455	2.649	4.030	0.832	< **0.0001**	2.481	3.369	0.606	0.0045
u724	1.601	2.399	0.363	1.253	2.195	0.607	0.9467	**1.133**	**1.663**	0.326	1.0000
pr1002	3.406	**4.524**	0.608	**3.350**	5.263	1.721	0.0734	5.868	8.607	1.501	< **0.0001**
u1060	**3.461**	**4.352**	0.530	4.135	7.454	1.889	< **0.0001**	7.961	9.924	1.170	< **0.0001**

Using the one-sided Wilcoxon signed rank test, we compute the p-values for the null hypothesis (H_0) that there is no difference between the means of CGS-TSP and the means of the competing algorithm considered, and the alternative hypothesis (H_1) that the means of CGS-TSP are smaller than the means of the considered algorithm. The p-values obtained are: 0.00009 for ACS and 0.22287 for MMAS.

Therefore, the null hypothesis can be rejected for ACS at a high significance level, but cannot be rejected for MMAS. This means that CGS-TSP clearly outperforms ACS for runs terminated after constructing 500000 tours, but there is no statistically significant difference between CGS-TSP and MMAS for this kind of runs.

Using a given number of constructed tours as termination condition does not provide a fair comparison, because the algorithms have different complexities. A fairer approach is to stop the execution after a given CPU time interval. In our second series of experiments, we have limited to 60 seconds the CPU time allowed for each run. The results are presented in Table 5-3, which has the same structure as Table 5-2.

Again, using the one-sided Wilcoxon signed rank test, we compute the p-values for the null hypothesis (H_0) that there is no difference between the means of CGS-TSP and the means of the competing algorithm considered, and the alternative hypothesis (H_1) that the means of CGS-TSP are smaller than the means of the considered algorithm. The p-values obtained are: 0.00003 for ACS and 0.00269 for MMAS.

The null hypothesis can be rejected for both ACS and MMAS at a high significance level, which means that for runs terminated after 60 seconds CPU time, CGS-TSP clearly outperforms both ACS and MMAS.

Table 5-3. Comparison of CGS-TSP with ACS and MMAS. Runs are terminated after 60 seconds CPU time.

Problem name	CGS-TSP			ACS				MMAS			
	Best (%)	Mean (%)	Mean stdev	Best (%)	Mean (%)	Mean stdev	p-value	Best (%)	Mean (%)	Mean stdev	p-value
kroA150	**0.207**	**0.430**	0.177	0.256	1.344	0.676	< **0.0001**	0.211	0.558	0.228	0.0076
kroB150	**0.000**	0.287	0.319	**0.000**	0.641	0.549	0.0113	**0.000**	**0.010**	0.037	1.0000
si175	**0.000**	**0.036**	0.033	**0.000**	0.168	0.103	< **0.0001**	**0.000**	0.090	0.047	< **0.0001**
kroA200	**0.000**	**0.127**	0.099	**0.000**	0.585	0.549	**0.0001**	0.065	0.151	0.102	0.0434
kroB200	**0.000**	**0.573**	0.320	0.034	1.131	0.661	< **0.0001**	0.340	0.699	0.245	0.0460
pr299	**0.174**	**0.602**	0.350	0.583	1.647	0.698	< **0.0001**	0.291	0.643	0.283	0.1304
lin318	**0.385**	1.068	0.340	0.578	1.847	0.815	< **0.0001**	0.452	**0.940**	0.441	0.9339
pr439	**0.354**	**1.133**	0.531	0.850	2.382	1.542	**0.0002**	1.582	2.472	0.591	< **0.0001**
pcb442	**0.441**	**1.261**	0.363	1.266	2.953	0.923	< **0.0001**	1.321	2.967	1.096	< **0.0001**
att532	**1.358**	**1.862**	0.228	1.806	3.196	0.814	< **0.0001**	1.611	2.920	0.840	< **0.0001**
u574	**1.775**	**2.485**	0.336	2.834	6.004	2.033	< **0.0001**	3.325	5.659	1.094	< **0.0001**
gr666	**1.994**	**2.757**	0.358	2.778	6.149	2.294	< **0.0001**	5.848	7.480	0.971	< **0.0001**
u724	**1.090**	**2.232**	0.378	1.341	2.655	0.992	0.0174	7.860	9.584	0.952	< **0.0001**
pr1002	**3.010**	**4.277**	0.574	4.689	8.188	2.253	< **0.0001**	15.056	16.753	0.685	< **0.0001**
u1060	**3.603**	**4.644**	0.590	7.371	12.316	3.202	< **0.0001**	17.607	19.582	0.876	< **0.0001**

In our last series of experiments, we compare the development of the mean percentage deviation over 25 trials as a function of the CPU time for instances lin318 and u1060, over 10000 seconds.

In our previous experiments, CGS-TSP has achieved relatively poor results for lin318 and, as seen in Figure 5-6, MMAS is able to produce better solutions than CGS-TSP after less than 20 seconds. Nevertheless, at the end of our 10000 seconds interval, CGS-TSP succeeds to outperform MMAS, albeit by a very small margin.

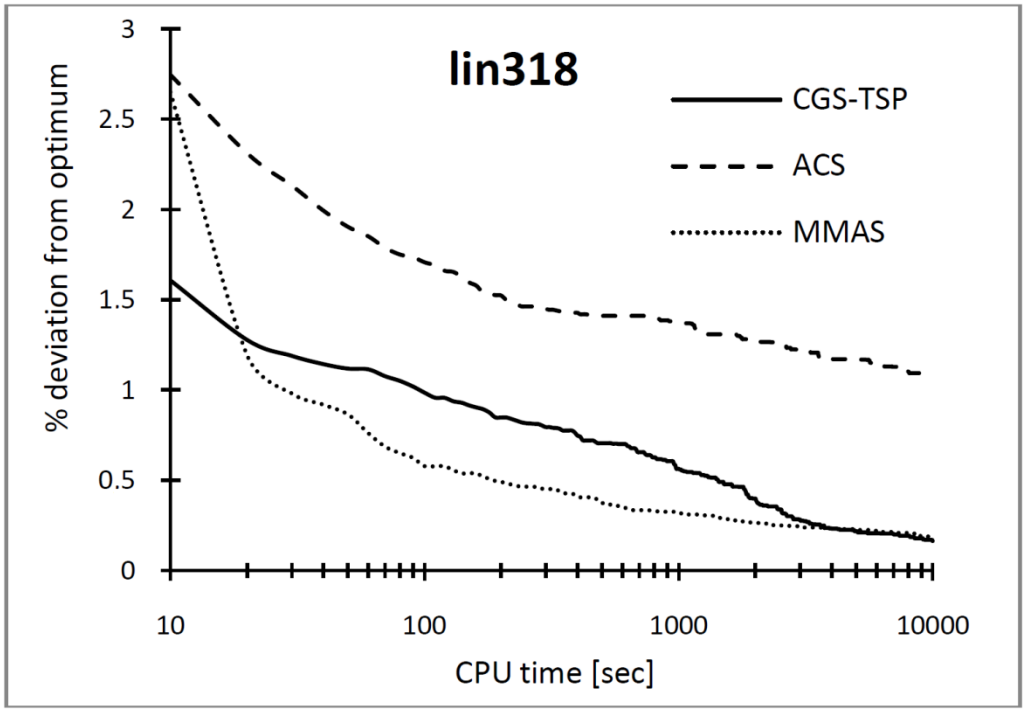


Figure 5-6. Mean percentage deviations for instance lin318.

In Figure 5-7, it can be observed that for u1060, CGS-TSP outperforms the other algorithms during the entire interval. MMAS is not able to reach the solution quality of CGS-TSP, although its performance improves significantly over time.

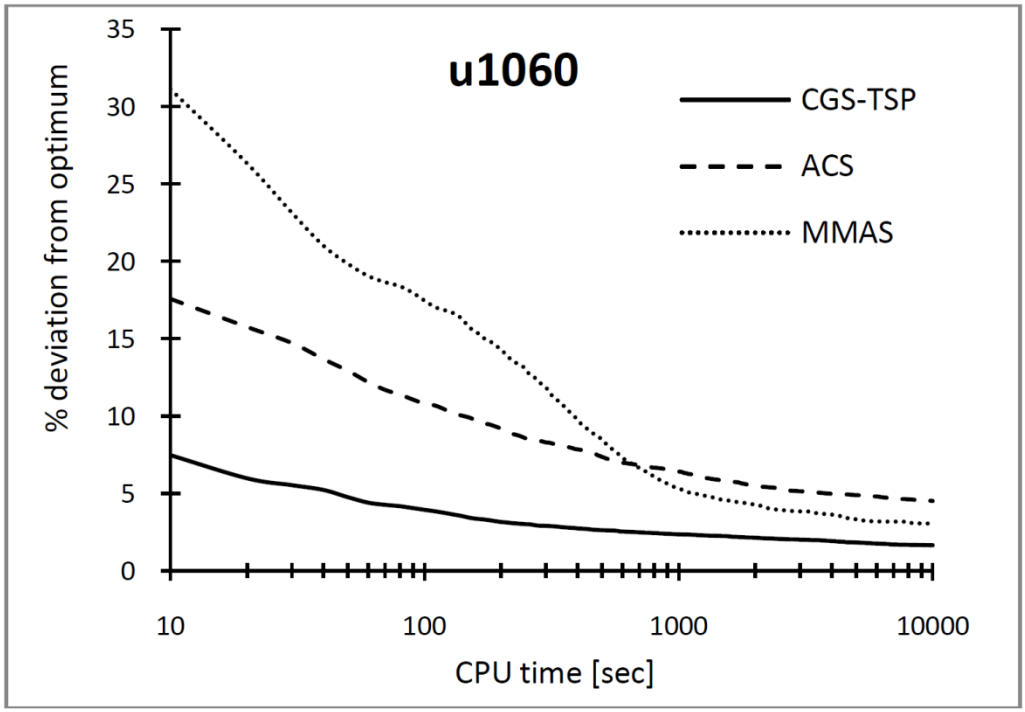


Figure 5-7. Mean percentage deviations for instance u1060.

5.6. CGS combined with local search for the Traveling Salesman Problem

As shown in the previous section, the CGS-TSP algorithm is able to outperform some of the best Ant Colony Optimization algorithms. However, these results can be misleading, because the experiments have been performed without local search and, in practice, ACO algorithms for the TSP are always combined with local search. While most metaheuristics work better when combined with local search, the performance improvement can vary significantly from one algorithm to another. Therefore, we investigate in this section whether CGS is still able to compete with ACO when the algorithms are combined with local search.

5.6.1. Local search

Local search [1] is a general technique that starts with a candidate solution and tries to improve it by iteratively moving to a neighbor solution. Typically, the exploration of the neighborhood is performed by applying local changes to the current solution. In the case of TSP, where a candidate solution is represented by a tour in the problem graph, a local change could consist in modifying a group of arcs, by interchanging their corresponding nodes.

The simplest variant of local search for the TSP is the 2-opt algorithm [28], which considers only groups composed of two arcs. These two arcs are initially eliminated, thus breaking the tour in 2 separate paths. After that, the two arcs are reconnected in the other possible way. In the example in Figure 5-8, the arcs (2,3) and (4,5) are eliminated and replaced by two new arcs, obtained by interchanging the nodes corresponding to the initial arcs, that is: (2,4) and (3,5).

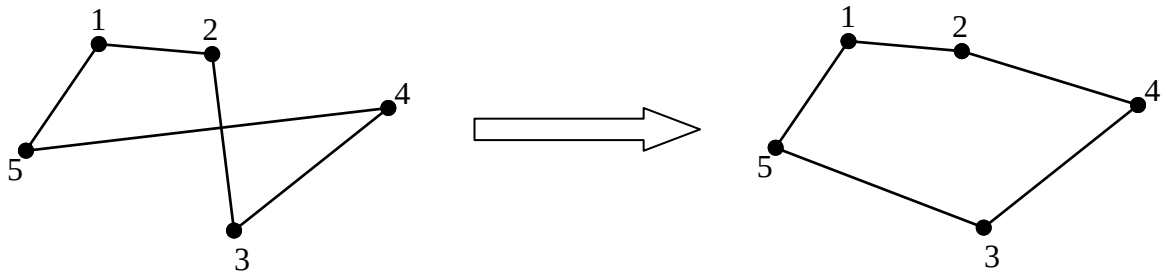


Figure 5-8. Modifying a tour by using a 2-opt move.

Another variant of local search is 3-opt, which considers groups of 3 arcs and analyses the 8 ways in which these arcs can be recombined. For the general case, called k-opt, the method is applied to groups comprising k arcs. The performance increases with the number k of arcs, but the complexity grows exponentially with this number. Therefore, in our experiments we combine CGS with 3-opt, which provides a good compromise between performance and complexity.

5.6.2. Applying local search to CGS-TSP

Since the CGS metaheuristic provides an optionally local optimization step, combining CGS-TSP with local search is a straightforward process. Local search improves the results, but it is a time-consuming procedure. Therefore, significantly fewer iterations can be performed in the same amount of time, when the algorithm is combined with local search. For this reason, different parameter settings are appropriate in this case. The standard CGS-TSP algorithm fixes the value of the parameter w to 1000 and the *sabbaticalDuration* to 100 iterations. In our experiments with local search we fix the value of w to 100 and the *sabbaticalDuration* to $500/m$ iterations, where m is the number of virtual persons. For the other parameters with fixed values in the standard CGS-TSP, we preserve the original values when combining the algorithm with local search: $a_0 = 0.9$ and $minReputation = 1$.

5.6.3. CGS-TSP with confidence

In addition to the algorithm described in the previous subsection, we propose a variant of the CGS-TSP algorithm, which we refer to as CGS-TSP-C, where each arc in the tour advertised by a consultant has an associated strength. Strengths are updated each time the consultant adjusts its strategy. If an arc in the new advertised tour was also present in the old advertised tour, its strength will be incremented; otherwise, its strength is set to 0. The strength of an arc could be interpreted as the consultant's confidence in recommending this arc to a client. A client is more likely to accept recommendations made with greater confidence. This idea is expressed in CGS-TSP-C by allowing the value of the parameter q0 to vary in a given range, at each construction step:

$$q_0 = \begin{cases} q_{min} + s \cdot \dfrac{q_{max} - q_{min}}{s_{max}} & ,if\ s < s_{max} \\ q_{max} & ,otherwise \end{cases} \tag{5.11}$$

where s is the strength of the recommended arc and q_{min}, q_{max} and s_{max} are parameters.

5.6.4. Experimental setup

We run a series of experiments in order to compare the performance of CGS-TSP and CGS-TSP-C with that of Ant Colony System (ACS)[43] and MAX-MIN Ant System (MMAS) [106]. We combine all algorithms used in our experiments with 3-opt local search and we use candidate lists of length 20 for all algorithms. Each run is terminated after n / 50 seconds CPU time, where n is the problem size (i.e., the number of cities).

In order to make a fair comparison, we have tuned the parameters of all algorithms considered, through the ParamILS [59] and F-Race [14] procedures. As training set, we have used 600 generated Euclidean TSP instances, with the number of cities uniformly distributed in the interval [1000, 2000].

For CGS-TSP and CGS-TSP-C, the parameter settings are given in Table 5-4.

Table 5-4. Parameter settings for CGS-TSP and CGS-TSP-C.

Parameter	**CGS-TSP**	**CGS-TSP-C**	**Description**
m	max(3,21-n/125)	max(3, 16-n/250)	number of virtual persons
b_0	0.98	0.95	*see formula (5.3)*
q_0	0.98	$q_{min} = 0.8$ $q_{max} = 0.99$ $s_{max} = 3$	*see formulas (5.3), (5.11)*
α	7	7	reputation's relative influence
β	12	12	heuristic's relative influence
γ	7	8	result's relative influence
maxReputation	40	50	maximum reputation value
initialReputation	6	3	reputation after sabbatical
bonus	8	6	best-so-far reputation bonus
protectedRanks	$0.7 \cdot m$	$0.6 \cdot m$	protected top consultants
r_0	$3 \cdot 10^{-7}$	$3 \cdot 10^{-6}$	basic reputation fading rate
fadingRanks	2	10	top consultants for fading rate
k_w	3	30	reputation decrease factor

The best configuration found for ACS is: m=12, ρ=0.6, ξ=0.4, q_0=0.98, β=2. For MMAS, the best configuration found is: $m = \max(10, 44 - 0.175 \cdot n)$, ρ=0.15, α=2, β=2. Some of these values differ significantly from the standard values given in [45], p.96, which are: m=10, ρ=0.1, ξ=0.1, q_0=0.98, β=2 for ACS and: m=25, ρ=0.2, α=1, β=2 for MMAS. For this reason, for ACS and MMAS, we have performed our experiments using both the tuned and the standard values. As shown in the next subsection, the results obtained using the tuned parameters outperform those obtained with the standard settings.

5.6.5. Experimental results

We have applied the algorithms to 27 symmetric instances from the TSPLIB benchmark library [92]. The number of cities of the TSP instances used in our experiments is between 654 and 3038. We have intentionally included TSP instances whose number of cities lies outside the range of values considered in the tuning phase (1000 to 2000), because a good algorithm is not problem dependent, and therefore it should not be very sensitive to the set of training instances used for tuning. The experiments have been performed on an HP ProLiant with 8 x 2.33 GHz Intel(R) Xeon(R) CPUs and 16 GB RAM, running Red Hat Enterprise Linux 5.

Table 5-5 reports for each algorithm and TSP instance the best and mean percentage deviations from the optimal solutions over 25 trials. The best mean results for each problem are in boldface. We also report for each problem the p-values of the one-sided Wilcoxon rank sum tests for the null hypothesis (H_0) that there is no difference between the solution quality of CGS-TSP-C and that of the competing ACO algorithm, and for the alternative hypothesis (H_1) that CGS-TSP-C outperforms the considered algorithm. Applying the Bonferroni correction for multiple comparisons, we obtain the adjusted α–level: 0.05 / 27 = 0.00185. The p-values in boldface indicate the cases where the null hypothesis is rejected at this significance level.

Table 5-5. Performance over 25 trials. Runs are terminated after n/50 CPU seconds.

Problem instance	ACS		MMAS		CGS-TSP		CGS-TSP-C			
	Best (%)	Mean (%)	Best (%)	Mean (%)	Best (%)	Mean (%)	Best (%)	Mean (%)	p-value (ACS)	p-value (MMAS)
p654	0.000	0.023	0.000	0.062	0.000	**0.009**	0.000	**0.009**	**0.0004**	**0.0000**
d657	0.002	0.167	0.031	0.133	0.002	0.129	0.002	**0.097**	0.0029	0.0326
gr666	0.056	0.159	0.000	**0.066**	0.040	0.069	0.000	0.109	0.0142	0.8059
u724	0.026	0.102	0.045	0.151	0.005	0.128	0.005	**0.101**	0.5932	0.0070
rat783	0.000	0.252	0.023	0.185	0.000	0.215	0.000	**0.147**	0.0157	0.1564
dsj1000	0.038	0.403	0.133	0.316	0.030	0.296	0.000	**0.224**	0.0042	0.1315
pr1002	0.000	0.273	0.034	0.201	0.000	0.189	0.000	**0.162**	0.0203	0.0863
si1032	0.000	**0.023**	0.000	0.035	0.000	0.029	0.000	0.024	0.6031	0.1758
u1060	0.116	0.331	0.104	0.359	0.026	**0.117**	0.026	0.119	**0.0000**	**0.0000**
vm1084	0.000	**0.064**	0.001	0.120	0.000	0.066	0.000	0.071	0.5096	**0.0012**
pcb1173	0.002	0.325	0.021	**0.219**	0.185	0.449	0.002	0.297	0.2456	0.9376
d1291	0.000	0.111	0.000	**0.105**	0.000	0.108	0.000	0.186	0.3690	0.4518
rl1304	0.000	0.196	0.000	0.229	0.000	0.200	0.000	**0.189**	0.5291	0.0243
rl1323	0.077	0.243	0.041	0.245	0.000	**0.131**	0.010	0.152	**0.0005**	**0.0000**
nrw1379	0.088	**0.256**	0.305	0.486	0.083	0.286	0.152	0.264	0.5668	**0.0000**
fl1400	0.020	0.247	0.298	0.668	0.000	0.190	0.000	**0.177**	0.0046	**0.0000**
u1432	0.218	**0.389**	0.250	0.601	0.292	0.481	0.153	0.426	0.8654	**0.0000**
fl1577	0.031	0.293	0.220	0.597	0.004	**0.060**	0.004	0.172	**0.0001**	**0.0000**
d1655	0.006	0.435	0.019	0.325	0.064	0.332	0.000	**0.322**	0.0754	0.4446
vm1748	0.113	0.272	0.154	0.471	0.061	0.191	0.004	**0.159**	**0.0000**	**0.0000**
u1817	0.192	0.480	0.080	**0.295**	0.156	0.386	0.107	0.347	0.0043	0.9048
rl1889	0.345	0.719	0.270	0.545	0.004	0.237	0.000	**0.217**	**0.0000**	**0.0000**
d2103	0.017	0.332	0.040	0.127	0.000	0.345	0.000	**0.047**	**0.0000**	**0.0000**
u2152	0.112	0.426	0.254	0.488	0.115	0.356	0.131	**0.352**	0.0472	**0.0015**
u2319	0.287	**0.372**	0.427	0.599	0.707	0.863	0.742	1.052	1.0000	1.0000
pr2392	0.235	0.507	0.214	0.542	0.019	0.441	0.051	**0.340**	**0.0001**	**0.0012**
pcb3038	0.243	**0.499**	0.671	0.940	0.704	1.056	0.428	0.810	1.0000	**0.0003**

For a few pairs of algorithms, we use the one-sided Wilcoxon signed rank test to compute the p-values for the null hypothesis (H_0) that there is no difference between the means of the first and the means of the second algorithm considered, and the alternative hypothesis (H_1) that the means of the first algorithm are smaller than the means of the second algorithm considered. The p-values are given in Table 5-6.

The null hypothesis can be rejected at a high significance level when CGS-TSP-C is compared with the two ACO algorithms, which means that for runs terminated after n/50 seconds CPU time, CGS-TSP-C clearly outperforms both ACS and MMAS. In addition, CGS-TSP-C outperforms CGS-TSP, which means that the use of confidence in relation to the recommendations made by consultants can lead to better results.

As shown in the previous section, in experiments without local search CGS-TSP clearly outperforms ACS and MMAS. Combined with 3-opt local search, the CGS-TSP algorithm still outperforms ACS and MMAS, but only at a moderate significance level. This means that ACS and MMAS benefit more than CGS-TSP from the hybridization with local search. One explanation for this discrepancy could be that the solution construction mechanism of CGS-TSP already bears some resemblance to a local search procedure: a client builds a

solution in the neighborhood of the solution promoted by a consultant and the consultant iteratively updates its recommended solution each time one of its clients finds a better one.

As mentioned in the previous subsection, some of the parameter values found during the tuning phase for ACS and MMAS differ significantly from the values recommended in [45], p.96. The last two lines of Table 5-6 compare the performance obtained using the tuned parameters and the standard settings. The tuned algorithms clearly outperform the algorithms that use standard settings, thus confirming the effectiveness of the tuning procedures.

Table 5-6. Performance comparison using the one-sided Wilcoxon signed rank test.

First algorithm	Second algorithm	p-value
CGS-TSP-C	ACS	0.00472
CGS-TSP–C	MMAS	0.00148
CGS-TSP–C	CGS-TSP	0.02004
CGS-TSP (without local search)	ACS (without local search)	0.00003
CGS-TSP	ACS	0.05020
CGS-TSP (without local search)	MMAS (without local search)	0.00269
CGS-TSP	MMAS	0.03051
ACS	ACS (standard settings)	0.00256
MMAS	MMAS (standard settings)	< 0.00001

In the following series of experiments, we compare the development of the mean percentage deviations from the optimum for our competing algorithms over 25 trials as a function of the CPU time, over 10000 seconds. We consider two TSP instances: u1060, for which the CGS algorithms have achieved good results in our previous experiments, and u2319, for which the CGS algorithms have obtained poor results.

As shown in Figure 5-9, for u1060 the CGS algorithms outperform the ACO algorithms during the entire interval. Although CGS-TSP-C performs best in the initial phases, it is outperformed by CGS-TSP in the long run. In case of u2319, the CGS algorithms are not able to reach the performance of the ACO algorithms.

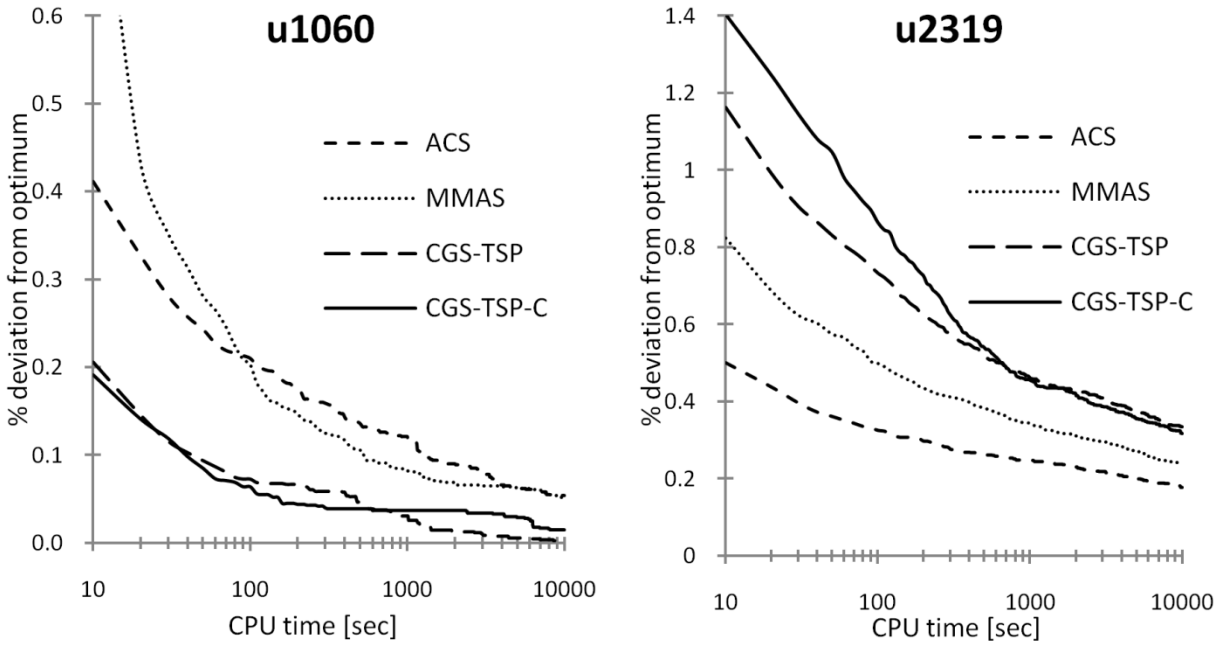


Figure 5-9. The development of the mean percentage deviations from the optimum.

Although for our experimental setup the CGS algorithms generally outperform the ACO algorithms, there are a few cases where the performance of CGS is relatively poor. The most striking example in our experiments is the TSP instance u2319. It is therefore worthwhile to investigate which characteristics of the TSP instances influence the performance of the CGS algorithms and how these characteristics relate to the values of the different parameters used by these algorithms. In the case of ACO, it has been shown [94] that an increase in the standard deviation of the cost matrix of TSP instances leads to a decrease in the performance of the algorithm. Like in ACO, the decisions taken in the solution construction phase of CGS algorithms depend on the relative lengths of edges in the TSP. Therefore, we expect that the standard deviation of the cost matrix also affects the performance of these algorithms.

The experimental results show that CGS is still able to compete with ACO when the algorithms are combined with local search, although the performance improvement due to local search is not as significant in the case of CGS as in the case of ACO algorithms. Moreover, the CGS-TSP-C algorithm introduced in this section shows that correlating the consultants' recommendations with a level of confidence may improve the results. Still, more research is needed in order to determine in which cases CGS-TSP-C should be preferred to CGS-TSP.

5.7. CGS applied to the Quadratic Assignment Problem

In this section, we discuss how the CGS metaheuristic can be applied to the Quadratic Assignment Problem (QAP), and we introduce the CGS-QAP algorithm, which hybridizes CGS with a local search procedure. Then, we describe the experimental setting and we report the experimental results, using MAX-MIN Ant System (MMAS) [106] as a yardstick to compare the performance of the proposed algorithm.

5.7.1. The CGS-QAP algorithm

Given n facilities and n locations, a flow matrix $F = \{f_{ij}\}$ and a distance matrix $D = \{d_{ij}\}$, the Quadratic Assignment Problem [71] consists in finding an assignment ϕ of facilities to locations, which minimizes the cost:

$$c_\phi = \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)} \tag{5.12}$$

The QAP is one of the most difficult combinatorial problems. Currently, exact algorithms are not able to solve in reasonable time instances with size n > 30.

An instantiation of the CGS for the QAP must define the different concepts and actions left unspecified by the CGS metaheuristic. In CGS-QAP, the ***strategy*** is implemented as a solution advertised by the consultant. It is represented by an assignment of facilities to locations, which is constructed during the ***sabbatical leave***. In the proposed algorithm, the sabbatical leave lasts only one iteration. In order to construct a new strategy, a consultant generates a random assignment and improves it by using a local search procedure.

The ***personal preference*** for a consultant is determined by the cost of its advertised assignment. Together with the ***reputation***, it gives the *suitability* of a consultant *k*:

$$suitability_k = \frac{reputation_k}{\beta + \frac{cost_k - cost_{bsf}}{cost_{bsf}}} \tag{5.13}$$

where the parameter β determines the influence of personal preference, $cost_k$ is the cost of the assignment advertised by consultant k and $cost_{bsf}$ is the cost of the best-so-far assignment. The probability to choose consultant k is:

$$p_k = \frac{suitability_k}{\sum_{c \epsilon \mathcal{C}} suitability_c} \tag{5.14}$$

where $\mathcal{C}$ is the set of all available consultants. A client is allowed to choose itself as a consultant. Because the probabilities given by formula (5.14) do not depend on the client making the choice, the client index does not appear in this formula.

At each construction step, a client places a not yet assigned facility to a free location. In CGS-QAP, the order in which facilities are assigned to locations is random. At each step, a client receives from its consultant a recommendation regarding the location to be chosen. The recommended location is the one corresponding to the given facility in the assignment advertised by the consultant. In order to decide whether to follow the recommendation, the client uses a method inspired by the pseudorandom proportional rule introduced by the Ant Colony System [43]: with probability q_0, a client places the given facility to the location recommended by its consultant; with probability $(1 - q_0)$ it randomly places the facility to one of the free locations. The value of the parameter q_0 is critical for the performance of CGS-QAP. A large value for q_0 leads to an aggressive search, focused around the assignment advertised by the consultant. A small value for q_0 favors the exploration of the search space, allowing the algorithm to escape from local optima.

Every time a client achieves a success (i.e., it finds an assignment better than that advertised by its consultant), the consultant updates its strategy, replacing its advertised assignment with the assignment constructed by the client.

At each iteration, the consultant's k reputation fades as given by formula (5.15):

$$reputation_k \leftarrow reputation_k(1 - r) \tag{5.15}$$

where the parameter r represents the reputation fading rate.

CGS-QAP prevents the reputation from sinking below the limit given by: $\text{initialReputation} \cdot \text{bestSoFarCost}/\text{advertisedCost}$. The value of the parameter $minReputation$ referred in Figure 5-3 is fixed in this algorithm at 1, because only the differences between $minReputation$ and the other reputation parameters are relevant for the algorithm.

At the end of each iteration, the algorithm applies a local search procedure in order to improve the assignments constructed by clients. Since the CGS metaheuristic provides an optional local optimization step, hybridizing CGS with a local search procedure is a straightforward process. Similar to other algorithms for the QAP, CGS-QAP can use 2-opt local search, short runs of tabu search [108] or simulated annealing [26] as the local search procedure.

5.7.2. Experimental setup

We run a series of experiments in order to compare the performance of CGS-QAP with that of MAX-MIN Ant System (MMAS). The choice of MMAS as a yardstick is motivated by the fact that it currently represents one of the best performing heuristics for the QAP.

To allow a meaningful comparison between heuristics, we have created a software package containing Java implementations of the algorithms considered in our experiments. The software package is available as an open source project at http://swarmqap.sourceforge.net/.

At this address, we also provide all configuration files, problem instances and results files for the parameter tuning and for the experiments described in this section.

Taillard [107] groups the QAP instances in four categories:

(1) unstructured instances with uniform random distances and flows.
(2) unstructured instances with random flows on grids.
(3) structured, real-life problems.
(4) structured, real-life like problems.

The instances in the first group are the most difficult to solve optimally. According to [104], in the case of MMAS the instance type has a strong influence on the local search procedure that should be used. Our preliminary experiments have shown that this is also true for CGS-QAP: as in the case of MMAS, 2-opt local search gives better solutions for structured instances, while short runs of tabu search are preferred for unstructured instances. In this section, we concentrate on the unstructured QAP instances and we combine all algorithms with short robust tabu search runs of length $4n$, where n is the problem size.

The parameters used for MMAS are those recommended in [104]: $m = 5$ ants; $\rho = 0.8$; $\tau_{max} = \frac{1}{1-\rho} \cdot \frac{1}{J_{\psi}^{gb}}$, where J_{ψ}^{gb} is the objective function value of the global best solution; $\tau_{min} = \frac{\tau_{max}}{2 \cdot n}$. In MMAS the pheromone trails are updated using either the iteration-best solution ψ^{ib}, or the global best solution ψ^{gb}. As suggested in [104], when applying tabu search we use ψ^{gb} every second iteration.

We have tuned the parameters of CGS-QAP using the ParamILS configuration framework [59]. ParamILS executes an iterated local search in the parameter configuration space and it is appropriate for algorithms with many parameters, where a full factorial design becomes intractable. As training data for paramILS, we have used a set of 500 QAP instances with sizes uniformly distributed in the interval [30, 90]. The training instances have random distances and flows and have been generated based on the method described in [105]. The parameter settings are given in Table 5-7.

Table 5-7. Parameter settings for CGS-QAP

Parameter	Value	Description
m	10	number of virtual persons
β	0.002	influence of the advertised cost
q_0	1-10/n	probability to follow consultant's recommendation
maxReputation	40	maximum reputation value
initialReputation	15	reputation after sabbatical
bonus	6	best-so-far reputation bonus
protectedRanks	2	protected top consultants
r	0.1	reputation fading rate

5.7.3. Experimental results

In our experiments we use 17 unstructured instances taken from QAPLIB [19]. For each run, we allow a total of 500 applications of tabu search. The experiments have been performed on an HP ProLiant with 8 x 2.33 GHz Intel(R) Xeon(R) CPUs and 16 GB RAM, running Red Hat Enterprise Linux 5.

Table 5-8 reports for each algorithm and QAP instance the mean percentage deviations from the best known solutions over 25 trials. The best mean results for each problem are in boldface. We also report for each problem the p-values of the one-sided Wilcoxon rank sum

tests for the null hypothesis (H_0) that there is no difference between the solution quality of CGS-QAP and that of MMAS, and for the alternative hypothesis (H_1) that CGS-QAP outperforms MMAS. Applying the Bonferroni correction for multiple comparisons, we obtain the adjusted α–level: 0.05 / 17 = 0.00294. The p-values in boldface indicate the cases where the null hypothesis is rejected at this significance level.

Using the one-sided Wilcoxon signed rank test, we compute the p-value for the null hypothesis (H_0) that there is no difference between the means of CGS-QAP and the means of MMAS, and the alternative hypothesis (H_1) that the means of CGS-QAP are smaller than the means of MMAS. The p-value obtained is 0.00019, which means that the null hypothesis can be rejected at a high significance level. Although the results are statistically significant, they do not seem to be important from a practical point of view.

Table 5-8. Algorithm performance for unstructured QAP instances, averaged over 25 trials. Runs are terminated after 500 applications of tabu search.

Problem instance	Best known	MMAS %	CGS-QAP % (p-value)
tai20a	703482	0.302	**0.097** (**0.0005**)
tai25a	1167256	0.361	**0.288** (0.1671)
tai30a	1818146	0.436	**0.364** (0.0441)
tai35a	2422002	0.556	**0.470** (0.0739)
tai40a	3139370	0.719	**0.585** (0.0122)
tai50a	4938796	1.089	**0.999** (0.1400)
tai60a	7205962	1.257	**1.051** (**0.0004**)
tai80a	13511780	1.380	**0.964** (**0.0000**)
tai100a	21052466	1.420	**0.917** (**0.0000**)
nug30	6124	0.013	**0.008** (0.3510)
sko42	15812	0.014	**0.004** (0.0108)
sko49	23386	0.060	**0.044** (0.3551)
sko56	34458	0.046	**0.029** (0.9131)
sko64	48498	0.036	**0.023** (0.5728)
sko72	66256	0.104	**0.098** (0.5058)
sko81	90998	0.077	**0.074** (0.9365)
sko90	115534	**0.086**	0.120 (0.9509)

In the previous experiment, CGS-QAP has outperformed MMAS on all instances except sko90. In the following experiment, we compare for this QAP instance the development of the mean percentage deviations from the best known solution for our competing algorithms over 10 trials and 10000 applications of tabu search. As shown in Figure 5-10, although CGS-QAP initially produces poorer results for the sko90 instance, it is able to outperform MMAS in the long run.

The experimental results show that combining CGS with a local search procedure leads to an efficient algorithm for the QAP, which performs better than MMAS for unstructured QAP instances. Our future research will investigate if CGS-QAP is still able to compete with MMAS for structured QAP instances.

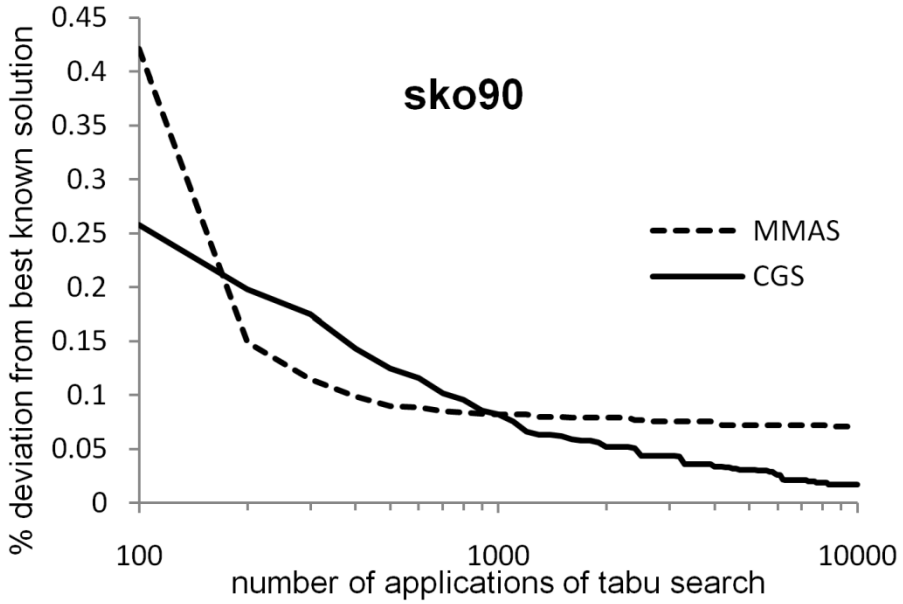


Figure 5-10. Mean percentage deviations for instance sko90 averaged over 10 trials.

5.7.4. Variants of the CGS-QAP algorithm

In this subsection, we propose a few variants of the CGS-QAP algorithm, which could potentially improve its solution quality.

5.7.4.1. The CGS_V-QAP algorithm

This algorithm is a variant of CGS-QAP, where the parameter q_0 referred in Table 5-7 is no longer constant. Instead, for each client, the value of q_0 is chosen at the beginning of each iteration, within an interval $[q_{min}, q_{max}]$, where q_{min} and q_{max} are constant parameters.

A value of q_0 close to q_{max} leads to an aggressive search, focused around the assignment advertised by the consultant. A value of q_0 close to q_{min} favors an exploration of the search space, allowing the algorithm to escape from local optima. There are many ways a client can choose the value of q_0 at the beginning of an iteration. The simplest one is to randomly pick a value in the interval $[q_{min}, q_{max}]$. Another possibility would be to correlate the value of q_0 with the number z of iterations since the last success achieved by a client working under the guidance of the consultant selected for the current iteration. For example, a client can use for $q_0(z)$ a formula such that:

$$\begin{cases} q_0(0) = q_{max} \\ \lim_{z \to \infty} q_0(z) = q_{min} \end{cases} \tag{5.16}$$

Because the value of q_0 controls the size of the neighborhood taken into consideration by a client, this variant of the algorithm could be seen as a hybridization of CGS-QAP with Variable Neighborhood Search [85].

5.7.4.2. The CGS_2-QAP algorithm

The idea behind this algorithm is that a client may ask for a second opinion in order to decide to which location to place a facility. For this purpose, at the beginning of each iteration, the client randomly selects a second consultant from the remaining ones. The rule used to choose a location for the current facility is based on two constant parameters, q_1 and q_2: with probability q_1, the client places the given facility to the location recommended by its main consultant; with probability $q_2(1 - q_1)$, it places the given facility to the location

recommended by the second consultant; with probability $(1-q_2)(1-q_1)$, it randomly places the given facility to one of the free locations.

The solution construction bears some resemblance to the approach used by genetic algorithms: choosing a location recommended by either the main or the second consultant is similar to a recombination operator; the perturbation produced when none of the recommendations was followed can be seen as a mutation.

5.7.4.3. The CGS_{2V}-QAP algorithm

This variant combines CGS_V-QAP and CGS_2-QAP. The parameters q_1 and q_2 used by CGS_2-QAP are no longer constants. Instead, their values are chosen at the beginning of each iteration, such that: $q_1 \in [q_{1min}, q_{1max}]$, $q_2 \in [q_{2min}, q_{2max}]$.

Our preliminary results suggest that the variants proposed in this subsection cannot outperform CGS-QAP for short runs, due probably to the additional overhead. However, for long runs, they seem to perform better than the standard CGS-QAP algorithm, because they are more effective in escaping from local optima. The CGS_{2V}-QAP variant appears to be the best choice for long runs with unstructured QAP instances.

5.8. CGS applied to the Generalized Traveling Salesman Problem

In this section, we discuss the application of CGS to the Generalized Traveling Salesman Problem (GTSP). The generalized traveling salesman problem (GTSP) is an NP-hard problem that extends the classical traveling salesman problem by considering a related problem given a partition of the nodes of a graph into clusters. The problem consists in finding the shortest closed tour visiting exactly one node from each cluster. T he existence of several applications of the GTSP and the difficulty of obtaining optimum solutions for the problem has led to the development of several heuristics and metaheuristics, see for example [55], [99], [101], [109].

We propose a hybrid algorithm that combines the consultant-guided search technique with a local-global approach for solving the GTSP. Most GTSP instances of practical importance are symmetric problems with Euclidean distances, where the clusters are composed of nodes that are spatially close one to the other. Our algorithm takes advantage of the structure of these instances.

5.8.1. The local-global approach to the Generalized Traveling Salesman Problem

Let $G = (V, E)$ be an n-node undirected complete graph whose edges are associated with non-negative costs and let $V_1, \ldots, V_m$ be a partitioning of V into p subsets called *clusters* (i.e. $V = V_1 \cup V_2 \cup \ldots \cup V_m$ and $V_l \cap V_k = \emptyset$ for all $l, k \in \{1, \ldots, m\}$).

Then, the *generalized traveling salesman problem* asks for finding a minimum-cost tour H spanning a subset of nodes such that H contains *exactly* one node from each cluster V_i, $i \in \{1, \ldots, m\}$. We call such a cycle a *generalized Hamiltonian tour*.

Based on the way the generalized combinatorial optimization problems are defined as extensions of the classical variants, a natural approach that takes advantage of the similarities between them is the *local-global approach* introduced by Pop [90] in the case of the generalized minimum spanning tree problem.

In the case of the GTSP, the local-global approach aims at distinguishing between *global connections* (connections between clusters) and *local connections* (connections between nodes from different clusters). This approach was already pointed out and exploited by Hu *et al.* in [58].

Given a sequence in which the clusters are visited (i.e. a global Hamiltonian tour), there are several generalized Hamiltonian tours corresponding to it. The best corresponding (with respect to cost minimization) generalized Hamiltonian tour can be determined either by using a layered network as we will describe next or by using integer programming.

We denote by G' the graph obtained from G after replacing all nodes of a cluster V_i with a supernode representing V_i. We will call the graph G' the *global graph*. For convenience, we identify V_i with the supernode representing it. Edges of the graph G' are defined between each pair of the graph vertices $V_i, \dots, V_m$.

Given a sequence $V_{k_1}, \dots, V_{k_m}$ in which the clusters are visited, we want to find the best feasible Hamiltonian tour H^* (with respect to cost minimization), visiting the clusters according to the given sequence. This can be done in polynomial time by solving $|V_{k_1}|$ shortest path problems, as we describe below.

We construct a layered network, denoted by LN, having $m+1$ layers corresponding to the clusters $V_{k_1}, \dots, V_{k_m}$ and in addition we duplicate the cluster V_{k_1}. The layered network contains all the nodes of G plus some extra nodes v' for each $v \in V_{k_1}$. There is an arc (i,j) for each $i \in V_{k_l}$ and $j \in V_{k_{l+1}}$ $(l = 1, \dots, m-1)$, having the cost c_{ij}. Moreover, there is an arc (i,j') for each $i \in V_{k_m}$ and $j' \in V_{k_1}$ having the cost $c_{ij'}$.

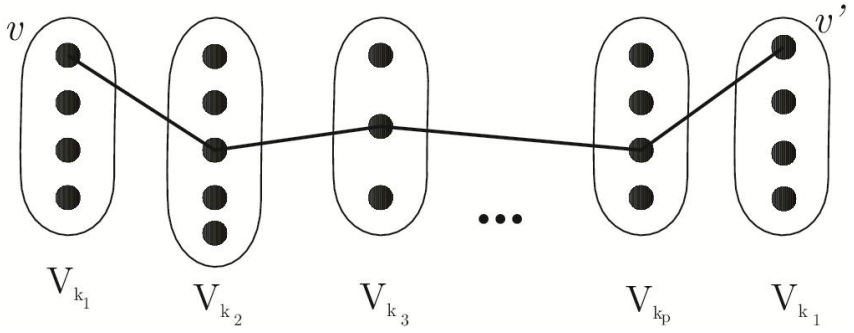


Figure 5-11. Example showing a Hamiltonian tour in the constructed layered network LN

For any given $v \in V_{k_1}$, we consider paths from v to v', $v' \in V_{k_1}$, that visits exactly one node from each cluster $V_{k_2}, \dots, V_{k_m}$, hence it gives a feasible Hamiltonian tour.

Conversely, every Hamiltonian tour visiting the clusters according to the sequence $(V_{k_1}, \dots, V_{k_m})$ corresponds to a path in the layered network from a certain node $v \in V_{k_1}$ to $v' \in V_{k_1}$.

Therefore, it follows that the best (with respect to cost minimization) Hamiltonian tour H^* visiting the clusters in a given sequence can be found by determining all the shortest paths from each $v \in V_{k_1}$ to the corresponding $v' \in V_{k_1}$ with the property that it visits exactly one node from each of the clusters $V_{k_2}, \dots, V_{k_m}$.

The overall time complexity is then $|V_{k_1}|O(|E| + \log n)$, i.e. $O(n|E| + \mathrm{n} \log n)$, in the worst case, where by $|E|$ we denote the number of edges. We can reduce the time by choosing V_{k_1} as the cluster with minimum cardinality.

Notice that the above procedure leads to an $O((m-1)!\,(n|E| + \mathrm{n} \log n))$ time exact algorithm for the GTSP, obtained by trying all the $(m-1)!$ possible cluster sequences.

Clearly, the algorithm presented is an exponential time algorithm, unless the number of clusters m is fixed.

5.8.2. The Hybrid Algorithm for the GTSP

We propose in this subsection an algorithm for the GTSP that combines the consultant-guided search technique with a local-global approach and improves the solutions using a local search procedure. Most GTSP instances of practical importance are symmetric problems with Euclidean distances, where the clusters are composed of nodes that are spatially close one to the other. We design our algorithm to take advantage of the structure of these instances.

At each iteration, a client constructs a global tour, that is, a Hamiltonian cycle in the global graph. The strategy of a consultant is also represented by a global tour, which the consultant advertises to its clients. The algorithm applies a local search procedure in order to improve the global tour representing either the global solution of a client or the strategy of a consultant in sabbatical mode. Then, using the cluster optimization procedure described in the previous subsection, the algorithm finds the best generalized tour corresponding to the global tour returned by the local search procedure.

In order to compare the strategies constructed during the sabbatical leave, a consultant uses the cost of the generalized tour corresponding to each strategy. Similarly, the success of a client is evaluated based on the cost of the generalized solution. The pseudocode of our algorithm is shown in Figure 5-12.

A virtual person may be in one of the following modes: *normal* and *sabbatical*. During the initialization phase (lines 2-5), virtual people are created and placed in sabbatical mode. Based on its mode, a virtual person constructs at each iteration of the algorithm (lines 7-31) either a global solution to the problem (line 19) or a global consultant strategy (line 9). In subsection 5.8.3, we describe the operations involved by the construction of a global solution or strategy, as well as the method used by a client in order to choose a consultant for the current iteration (line 17).

Global strategies and global solutions are improved by applying a local search procedure (lines 10 and 20). The *clusterOptimization* procedure described in section 5.8.1 is then used to find the best generalized strategy (line 11) corresponding to the current global strategy or to find the best generalized solution (line 21) corresponding to the current global solution.

After constructing a global strategy, a virtual person in sabbatical mode checks if the corresponding generalized strategy is the best generalized strategy found since the beginning of the sabbatical (lines 12-15). Similarly, after constructing a global solution, a client checks the corresponding generalized solution in order to decide if it has achieved a success and, if this is the case, it updates the strategy of its consultant (lines 22-26).

At the end of each iteration, the reputation and action mode of each virtual person are updated (lines 30-31).

```
 1 procedure CGS-GTSP()
 2     create the set 𝒫 of virtual persons
 3     foreach p ∈ 𝒫 do
 4         setSabbaticalMode(p)
 5     end foreach
 6     while (termination condition not met) do
 7         foreach p ∈ 𝒫 do
 8             if actionMode[p] = sabbatical then
 9                 currStrategy[p] ← constructStrategy(p)
10                 applyLocalSearch(currStrategy[p])
11                 genStrategy ← clusterOptimization(currStrategy[p])
12                 if cost(genStrategy) < bestStrategyCost then
13                     bestStrategy[p] ← currStrategy[p]
14                     bestStrategyCost[p] ← cost(genStrategy)
15                 end if
16             else
17                 c ← chooseConsultant(p)
18                 if c ≠ null then
19                     currSol[p] ← constructSolution(p, c)
20                     applyLocalSearch(currSol[p])
21                     currGenSol[p] ← clusterOptimization(currSol[p])
22                     if currGenSol[p] is better than all solutions found
23                                 by a client of c since last sabbatical then
24                         successCount[c] ← successCount[c] + 1
25                         strategy[c] ← currSol[p]
26                     end if
27                 end if
28             end if
29         end foreach
30         updateReputations()
31         updateActionModes()
32     end while
33 end procedure
```

Figure 5-12. The CGS-GTSP algorithm.

Figure 5-13 details how consultants' reputations are updated based on the successes achieved by their clients.

Reputations fade over time at a constant rate, given by the parameter *fadingRate* (line 4). The reputation of a consultant is incremented with each success achieved by one of its clients (line 5) and it receives an additional bonus of 10 for finding a best-so-far solution (lines 6-9). The reputation of a consultant cannot exceed a maximum value (lines 10-12) and the algorithm prevents the reputation of the best consultant, that is, the consultant that has found the best-so-far solution, from sinking below a given value (lines 13-17). The constant parameter *initialReputation* represents the reputation assigned to a consultant at the end of the sabbatical leave.

Figure 5-14 details how the action mode of each virtual person is updated: consultants whose reputations have sunk below the minimum level are placed in sabbatical mode, while consultants whose sabbatical leave has finished are placed in normal mode.

Figure 5-15 shows the actions taken to place a virtual person in sabbatical or in normal action mode.

```
 1 procedure updateReputations()
 2     foreach p ∈ 𝒫 do
 3         if actionMode[p] = normal then
 4             rep[p] ← rep[p] * (1 - fadingRate)
 5             rep[p] ← rep[p] + successCount[p]
 6             if cost(currGenSol[p]) < cost(bestSoFarSol) then
 7                 bestSoFarSol ← currGenSol[p]
 8                 rep[p] ← rep[p] + 10          // reputation bonus
 9             end if
10             if rep[p] > 10 * initialReputation then
11                 rep[p] ← 10 * initialReputation
12             end if
13             if p is the best consultant then
14                 if rep[p] < initialReputation then
15                     rep[p] ← initialReputation
16                 end if
17             end if
18         end if
19     end foreach
20 end procedure
```

Figure 5-13. Procedure to update reputations.

```
 1 procedure updateActionModes()
 2     foreach p ∈ 𝒫 do
 3         if actionMode[p] = normal then
 4             if rep[p] < 1 then
 5                 setSabbaticalMode(p)
 6             end if
 7         else
 8             sabbaticalCountdown ← sabbaticalCountdown – 1
 9             if sabbaticalCountdown = 0 then
10                 setNormalMode(p)
11             end if
12         end if
13 end procedure
```

Figure 5-14. Procedure to update action modes.

```
 1 procedure setSabbaticalMode(p)
 2     actionMode[p] ← sabbatical
 3     bestStrategy[p] ← null
 4     bestStrategyCost[p] ← ∞
 5     sabbaticalCountdown ← 20
 6 end procedure

 7 procedure setNormalMode(p)
 8     actionMode[p] ← normal
 9     rep[p] ← initialReputation
10     strategy[p] ← bestStrategy[p]
11 end procedure
```

Figure 5-15. Procedures to set sabbatical and normal mode.

5.8.3. Strategy and solution construction

The heuristic used during the sabbatical leave in order to build a new strategy is based on virtual distances between the supernodes in the global graph. We compute the virtual distance between two supernodes as the distance between the centers of mass of the two corresponding clusters. The choice of this heuristic is justified by the class of problems for which our algorithm is designed: symmetric instances with Euclidean distances, where the nodes of a cluster are spatially close one to the other.

By introducing virtual distances between clusters, we have the possibility to use candidate lists in order to restrict the number of choices available at each construction step. For each cluster i, we consider a candidate list that contains the closest $cand$ clusters, where $cand$ is a parameter. This way, the feasible neighborhood of a person k when being at cluster i represents the set of clusters in the candidate list of cluster i that person k has not visited yet. Several heuristic algorithms for the TSP use candidate lists during the solution construction phase (see [45] for examples of their use with Ant Colony Optimization algorithms), but candidate lists have not been widely used to construct solutions for the GTSP. Our algorithm uses candidate lists during both strategy construction and solution construction.

The use of candidate lists may significantly improve the time required by an algorithm, but it could also lead to missing good solutions. Therefore, the choice of appropriate sizes and elements of the candidate lists is critical for the working of an algorithm. In the case of TSP, candidate lists with size 20 are frequently used, but other values between 10 and 40 are also usual [68]. For GTSP instances with clusters composed of nodes spatially close to each other, appropriate sizes for the candidate lists are considerably smaller. Our experiments show that values of 4 or 5 are adequate in this case.

During the sabbatical leave, a consultant uses a random proportional rule to decide which cluster to visit next. For a consultant k, currently at cluster i, the probability to choose cluster j is given by formula (5.17):

$$p_{ij}^{k} = \frac{(1/d_{ij})^{\beta}}{\sum_{l \epsilon \mathcal{N}_i^k} (1/d_{il})^{\beta}} \tag{5.17}$$

where:

- $\mathcal{N}_i^k$ is the feasible neighborhood of person *k* when being at cluster *i*.
- d_{il} is the virtual distance between clusters *i* and *l*.
- β is a constant parameter.

As mentioned before, the feasible neighborhood $\mathcal{N}_i^k$ contains the set of clusters in the candidate list of cluster i that person k has not visited yet. If all the clusters in the candidate list have already been visited, the consultant can choose one of the clusters not in the candidate list, using a random proportional rule similar to that given by formula (5.17).

Using virtual distances between clusters as a heuristic during the sabbatical leave, leads to reasonably good initial strategies. In general, however, a global tour that is optimum with respect to the virtual distances between clusters does not produce the optimum generalized tour after applying the cluster optimization procedure. Therefore, during the solution construction phase, the algorithm does not rely on the distances between clusters, although it still uses candidate lists in order to determine the feasible neighborhood of a cluster.

At each step, a client receives a recommendation regarding the next cluster to be visited. This recommendation is based on the global tour advertised by the consultant. Let i be the cluster visited by the client k at a construction step of the current iteration. To decide which cluster to

recommend for the next step, the consultant finds the position at which the cluster i appears in its advertised global tour and identifies the cluster that precedes i and the cluster that succeeds i in this tour. If neither of these two clusters is already visited by the client, the consultant randomly recommends one of these two clusters. If only one of these two clusters is unvisited, this one is chosen to be recommended. Finally, if both clusters are already visited, the consultant is not able to make a recommendation for the next step.

The client does not always follow the consultant's recommendation. The rule used to choose the next cluster j to move to is given by formula (5.18):

$$j = \begin{cases} v & ,if\ v \neq null \wedge q \leq q_0 \\ random(\mathcal{N}_i^k) & ,otherwise \end{cases} \tag{5.18}$$

where:

- v is the cluster recommended by the consultant for the next step.
- q is a random variable uniformly distributed in [0,1] and q_0 ($0 \leq q_0 \leq 1$) is a parameter.
- $\mathcal{N}_i^k$ is the feasible neighborhood of person k when being at cluster i.
- $random$ is a function that randomly chooses one element from the set given as argument.

Again, if all the clusters in the candidate list have already been visited, the feasible neighborhood $\mathcal{N}_i^k$ is empty. In this case, a client that ignores the recommendation of its consultant can choose one of the clusters not in the candidate list, using a random proportional rule similar to that given by formula (5.17).

The personal preference of a client for a given consultant is computed as the inverse of the cost of the generalized tour corresponding to the global tour advertised by the consultant. In conjunction with the reputation, the personal preference is used by clients in order to compute the probability to choose a given consultant k:

$$p_k = \frac{(reputation_k \cdot preference_k)^2}{\sum_{c \epsilon \mathcal{C}} (reputation_c \cdot preference_c)^2} \tag{5.19}$$

where $\mathcal{C}$ is the set of all available consultants.

5.8.4. An algorithm variant using confidence

In this subsection, we propose a variant of our algorithm based on the approach introduced in subsection 5.6.3, which correlates the recommendation of a consultant with a level of confidence. Each arc in the global tour advertised by a consultant has an associated strength. Strengths are updated each time the consultant adjusts its strategy. If an arc in the new advertised tour was also present in the old advertised tour, its strength will be incremented; otherwise, its strength is set to 0. The strength of an arc could be interpreted as the consultant's confidence in recommending this arc to a client. A client is more likely to accept recommendations made with greater confidence. This idea is expressed in this algorithm variant by allowing the value of the parameter q_0 from formula (5.18) to vary in a given range, at each construction step:

$$q_0 = \begin{cases} q_{min} + s \cdot \dfrac{q_{max} - q_{min}}{s_{max}} & ,if\ s < s_{max} \\ q_{max} & ,otherwise \end{cases} \tag{5.20}$$

where s is the strength of the recommended arc and q_{min}, q_{max} and s_{max} are constant parameters. The use of confidence compensates somewhat for the absence of a heuristic during the solution construction phase.

5.8.5. Local Search

The global tours built during the strategy construction and solution construction phase are improved using a local search procedure generically described in Figure 5-16.

```
 1  procedure applyLocalSearch(H_G)
 2        H ← clusterOptimization(H_G)
 3        foreach H'_G ∈ tourNeighborhood(H_G) do
 4           if quickCheck(H'_G) then
 5              H' ← partialClusterOptimization(H'_G, H)
 6              if cost(H') < cost(H) then
 7                 H_G ← H'_G
 8                 H ← H'
 9              end if
10           end if
11        end foreach
12  end procedure
```

Figure 5-16. The local search procedure.

H_G and $H_G^{'}$ denote global Hamiltonian tours, that is, tours in the graph of clusters, while H and H' denote generalized Hamiltonian tours. Our algorithm can be combined with any local search procedure conforming to the above algorithmic structure. The working of the *clusterOptimization* function (line 2) is explained in section 5.8.1. The *cost* function (line 6) computes the cost of a generalized Hamiltonian tour. The other functions referred in Figure 5-16 are only generically specified and they must be implemented by each concrete instantiation of the local search procedure.

The *tourNeighborhood* function (line 3) should return a set of global tours representing the neighborhood of the global tour H_G provided as argument. The *quickCheck* function (line 4) is intended to speed up the local search by quickly rejecting a candidate global tour from the partial cluster optimization, if this tour is not likely to lead to an improvement.

The *partialClusterOptimization* function (line 5) starts with the generalized tour obtained by traversing the nodes of H in accordance with the ordering of clusters in the global tour $H_G^{'}$. Then, it reallocates some vertices in the resulting generalized tour, trying to improve its cost. Typically, this function considers only a limited number of vertices for reallocation and it has a lower complexity than the *clusterOptimization* function.

The generalized tour constructed by the function *partialClusterOptimization* is accepted only if its cost is better than the cost of the current generalized tour (lines 6-9).

We provide two instantiations of the generic local search procedure shown in Figure 5-16: one based on a 2-opt local search and one based on a 3-opt local search. We describe here only the 2-opt based variant. Except from the fact that it considers exchanges between 3 arcs, the 3-opt based local search is very similar to the 2-opt based variant.

In the 2-opt based local search, the *tourNeighborhood* function returns a set of global tours obtained by replacing a pair of arcs (C_α, C_β) and (C_γ, C_δ) in the original global tour with the pair of arcs (C_α, C_γ) and (C_β, C_δ). In order to reduce the number of exchanges taken into consideration, the set returned by our *tourNeighborhood* function includes only tours for which γ is in the candidate list of α. In other word, a pair of arcs is considered for exchange only if the center of mass of the cluster γ is close to the center of mass of the cluster α.

The *partialClusterOptimization* function used in this case is similar to the RP1 procedure introduced in [47]. Let (C_α, C_β) and (C_γ, C_δ) be the two arcs from the original global tour H_G that have been replaced with (C_α, C_γ) and (C_β, C_δ) in the neighbor global tour H_G', as shown in Figure 5-17.

The vertices in clusters C_α, C_β, C_γ and C_δ can then be reallocated, in order to minimize the cost of the generalized tour. For this purpose, we have to determine the two node pairs (u', w') and (v', z') such that:

$$\begin{aligned} d_{iu'} + d_{u'w'} + d_{w'h} &= min\{d_{ia} + d_{ab} + d_{bh} \mid a \in C_\alpha, b \in C_\gamma\} \\ d_{jv'} + d_{v'z'} + d_{z'k} &= \min\{d_{ja} + d_{ab} + d_{bk} \mid a \in C_\beta, b \in C_\delta\} \end{aligned} \tag{5.21}$$

This computation requires $|C_\alpha||C_\gamma| + |C_\beta||C_\delta|$ comparisons.

The *quickCheck* function permits the application of the partial cluster optimization only if the following inequality holds:

$$\begin{aligned} d_{min}(C_\rho, C_\alpha) + d_{min}(C_\alpha, C_\gamma) + d_{min}(C_\gamma, C_\sigma) + d_{min}(C_\pi, C_\beta) + d_{min}(C_\beta, C_\delta) \\ + d_{min}(C_\delta, C_\tau) < d_{iu} + d_{uv} + d_{vj} + d_{hw} + d_{wz} + d_{zk} \end{aligned} \tag{5.22}$$

where $d_{min}(C_\alpha, C_\beta)$ is the minimum distance between each pair of vertices from clusters C_α and C_β. These minimum distances are computed only once, at algorithm startup.

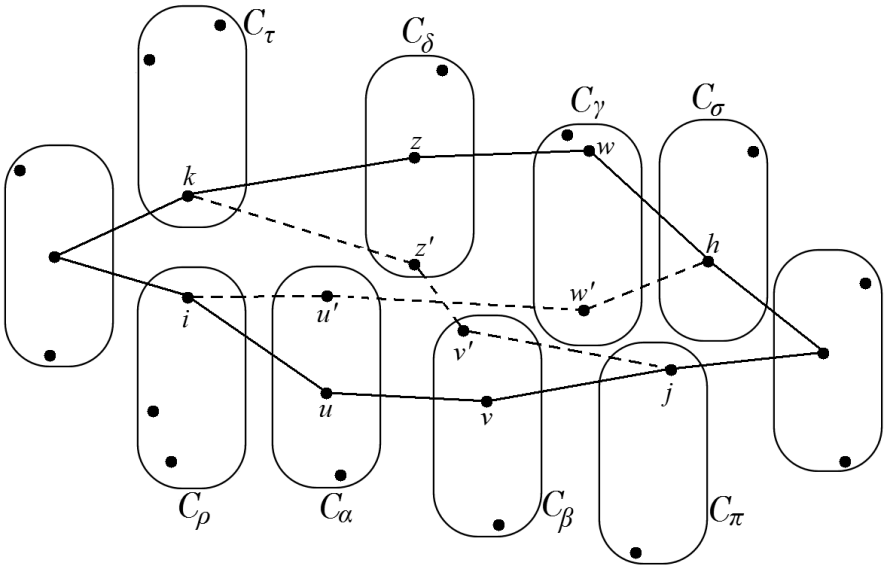


Figure 5-17. The 2-opt partial cluster optimization.

5.8.6. Experimental setup

We have implemented our algorithm as part of a software package written in Java, which is available online at http://swarmtsp.sourceforge.net/. At this address we provide all information necessary to reproduce our experiments.

The parameters of the algorithm have been tuned using the paramILS configuration framework [59]. ParamILS executes an iterated local search in the parameter configuration space and it is appropriate for algorithms with many parameters, where a full factorial design becomes intractable. We have generated a set of 100 Euclidean TSP instances with the number n of cities uniformly distributed in the interval [200, 500] and with coordinates uniformly distributed in a square of dimension 10000 x 10000. These instances have been

then converted to GTSP by applying the CLUSTERING procedure introduced in [47]. This procedure sets the number of clusters $s = \lceil n/5 \rceil$, identifies the s farthest nodes from each other and assigns each remaining node to its nearest center. We have used the resulting GTSP instances as training data for paramILS.

Before starting the tuning procedure, we have run our algorithm 10 times on each instance in the training set, using a default configuration. Each run has been terminated after $n/10$ seconds and we have stored the best result obtained for each GTSP instance. During the tuning procedure, these best known results are used as termination condition for our algorithm. Each time paramILS evaluates a parameter configuration with respect to a given instance, we determine the mean time (averaged over 10 trials) needed by our algorithm in order to obtain a result at least as good as the best known result for this instance, using the given parameter configuration.

The best parameter configuration found after 10 iterations of paramILS is given in Table 5-9.

Table 5-9. Parameter configuration for the standard algorithm.

Parameter	Value	Description
m	8	number of virtual persons
q_0	0.8	*see formula (5.18).*
initialReputation	6	reputation after sabbatical; *see Figure 5-13 and Figure 5-15.*
reputationFadingRate	0.003	reputation fading rate; *see Figure 5-13.*
candidateListSize	5	number of clusters in the candidate list.

For the algorithm variant using confidence, we have used the same procedure as for the standard algorithm, but we have tuned only the values of the parameters q_{min}, q_{max} and s_{max}. For the parameters m, initialReputation, reputationFadingRate and candidateListSize we have used the values from Table 5-9. The best parameter configuration found for the algorithm variant with confidence after 10 iterations of paramILS is given in Table 5-10:

Table 5-10. Parameter configuration for the algorithm variant with confidence.

Parameter	Value	Description
q_{min}	0.7	parameters used to compute the value of q_0; *see formulas (5.18) and (5.20).*
q_{max}	0.98	
s_{max}	3	

5.8.7. Computational results

The performance of the proposed algorithm has been tested on 18 GTSP problems generated from symmetric Euclidean TSP instances. These TSP instances, containing between 198 and 442 nodes, are drawn from the TSPLIB [92] benchmark library. The corresponding GTSP problems are obtained by applying the CLUSTERING procedure introduced in [47]. For 16 of the considered GTSP instances, the optimum objective values have been determined by Fischetti *et al.* [47]. For the remaining 2 instances (45tsp225 and 56a280), the best known results from the literature are conjectured to be optimal.

Currently, the memetic algorithm of Gutin and Karapetyan [55] clearly outperforms all published GTSP heuristics. Therefore, we use this algorithm as a yardstick to evaluate the performance of the different variants of our algorithm. We use the following acronyms to identify the algorithms used in our experiments:

- GK: the memetic algorithm of Gutin and Karapetyan [55].
- CGS-2: the standard variant of our algorithm combined with 2-opt local search.

- CGS-3: the standard variant of our algorithm combined with 3-opt local search.
- CGS-C-2: the variant of our algorithm using confidence combined with 2-opt local search.
- CGS-C-3: the variant of our algorithm using confidence combined with 3-opt local search.

For each GTSP instance, we run each algorithm 25 times and we report the average time needed to obtain the optimal solution. For the GK algorithm, we use the C++ implementation offered by its authors. The running times for GK differ from the values reported in [55], because we run our experiments on a 32-bit platform using an Intel Core2 Duo 2.2 GHz processor, while the results presented in [55] have been obtained on a 64-bit platform and using a faster processor (AMD Athlon 64 X2 3.0 GHz).

The computational results are shown in Table 5-11. The name of each problem is prefixed by the number of clusters and it is suffixed by the number of nodes. Average times that are better than those obtained by the GK algorithm are in boldface. For each problem and for each CGS algorithm variant, we also report the p-values of the one-sided Wilcoxon rank sum tests for the null hypothesis (H_0) that for the given problem there is no difference between the running times of the considered algorithm variant and the running times of the GK algorithm, and for the alternative hypothesis (H_1) that the considered algorithm outperforms the GK algorithm for the given problem. Applying the Bonferroni correction for multiple comparisons, we obtain the adjusted α-level: 0.05 / 18 = 0.00278. The p-values in boldface indicate the cases where the null hypothesis is rejected at this significance level.

Table 5-11. Times (in seconds) needed to finde the optimal solutions, averaged over 25 trials.

Problem instance	Optimal cost	GK time	CGS-C-3		CGS-C-2		CGS-3		CGS-2	
			time	p-value	time	p-value	time	p-value	time	p-value
40d198	10557	0.46	**0.36**	**0.0004**	**0.33**	**0.0012**	0.47	0.0034	**0.45**	0.0050
40kroA200	13406	0.38	**0.33**	**0.0000**	**0.25**	**0.0000**	**0.37**	0.5711	**0.30**	**0.0001**
40kroB200	13111	0.48	0.60	0.9460	**0.37**	**0.0008**	0.59	0.9689	0.60	0.6156
41gr202	23301	0.71	**0.64**	0.0141	0.91	0.4674	1.35	1.0000	1.10	0.9101
45ts225	68340	0.61	3.32	1.0000	4.06	0.9957	1.92	0.9999	2.67	1.0000
45tsp225	1612	0.51	4.83	1.0000	3.25	0.9994	4.07	1.0000	2.28	0.9967
46pr226	64007	0.28	**0.13**	**0.0000**	**0.07**	**0.0000**	**0.13**	**0.0000**	**0.09**	**0.0000**
46gr229	71972	0.81	**0.36**	**0.0000**	**0.33**	**0.0000**	**0.39**	**0.0000**	**0.37**	**0.0000**
53gil262	1013	0.83	1.22	0.1071	2.63	0.9999	1.63	1.0000	3.49	1.0000
53pr264	29549	0.67	**0.57**	0.0070	**0.49**	**0.0005**	0.94	0.9482	1.08	0.9406
56a280	1079	0.94	1.79	0.8215	3.71	0.9999	2.02	0.9998	4.46	1.0000
60pr299	22615	1.10	3.54	0.9992	2.91	0.9992	3.23	1.0000	4.74	0.9999
64lin318	20765	1.16	**0.85**	**0.0000**	2.68	0.9929	1.28	0.8946	3.81	1.0000
80rd400	6361	2.57	10.30	0.9996	13.27	1.0000	87.96	1.0000	270.04	1.0000
84fl417	9651	1.91	**1.10**	**0.0000**	**1.59**	**0.0001**	**1.51**	**0.0012**	2.27	0.0512
87gr431	101946	6.01	8.16	0.8361	12.86	0.9916	477.38	1.0000	866.53	1.0000
88pr439	60099	4.07	**1.56**	**0.0000**	**1.32**	**0.0000**	**3.68**	0.0104	10.71	0.9999
89pcb442	21657	4.24	11.11	0.9980	13.53	1.0000	395.93	1.0000	1430.13	1.0000

It can be observed that CGS-C-3 outperformed GK for 9 of the 18 instances and in 7 cases these results are significantly better. CGS-C-2 outperformed GK for 8 of the 18 instances and in all these 8 cases the results are significantly better. The algorithm variants without confidence perform poorer and for a few instances they need considerably more time to find the optimal solution.

For several pairs of algorithms, we use the one-sided Wilcoxon signed rank test to compute the p-values for the null hypothesis (H_0) that there is no difference between the running times of the first and the running times of the second algorithm, and the alternative hypothesis (H_1)

that the running times of the first algorithm are better than the running times of the second algorithm. The p-values are given in Table 5-12, where the significant values ($p < 0.05$) are in boldface.

Table 5-12. Performance comparison using the one-sided Wilcoxon signed rank test.

First algorithm	Second algorithm	p-value
GK	CGS-C-3	0.1061
GK	CGS-C-2	**0.0368**
GK	CGS-3	**0.0069**
GK	CGS-2	**0.0005**
CGS-C-3	CGS-C-2	0.0708
CGS-C-3	CGS-3	**0.0152**
CGS-C-2	CGS-2	**0.0028**

It can be observed that GK outperforms our algorithms, but in the case of CGS-C-3, the differences are not statistically significant. Similarly, CGS-C-3 outperforms CGS-C-2, but not statistically significant. The fact that 3-opt local search does not significantly improve the results obtained with 2-opt local search could be a consequence of the greater complexity of 3-opt. There are, however, significant differences between the running times of CGS variants with confidence and those without confidence. Due to the very poor results obtained in some cases by the algorithm variants without confidence, these differences are not only statistically, but also practically significant, thus indicating the importance of the *confidence* component.

Figure 5-18 shows how the candidate list size affects the time needed by CGS-C-3 to find the optimal solution of the problem instance 64lin318. The results are averaged over 25 trials.

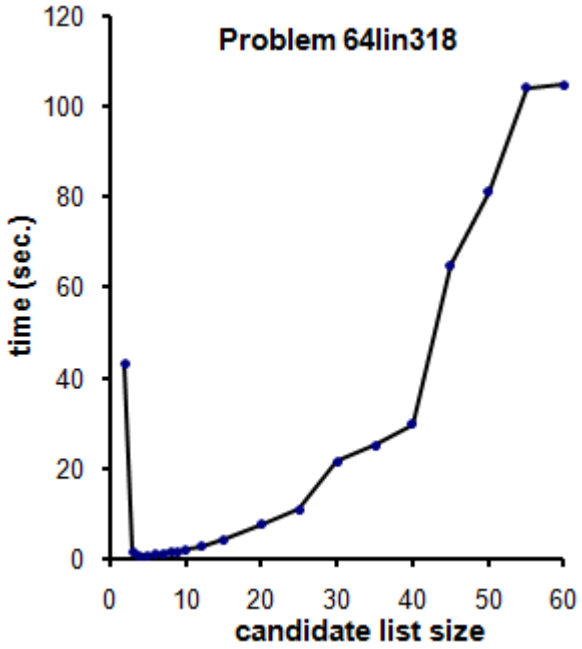


Figure 5-18. The influence of the candidate list size on the time needed to find the optimal solution for problem 64lin318.

It can be observed that the size of the candidate list has a huge influence on the time needed by the algorithm to find the optimal solution. Therefore, the use of candidate lists is a key component contributing to the success of our algorithm.

The best results are obtained for candidate lists of size 4 or 5, but we should note that the algorithm is able to find the optimum even for candidate lists with only 2 elements. However, in this case the time needed increases considerably. This is due to the fact that the probability to find the next cluster of the optimal tour in the candidate list of the current cluster is significantly smaller when using a candidate list with only 2 elements. For the 64lin318 instance, only 44 of the 64 clusters are present in the candidate list of their precedent cluster when using candidate lists with 2 elements. In contrast, 59 of the 64 clusters are present when using candidate lists with 5 elements. The algorithm is able to find the optimal solution even for very small sized candidate lists, because during the construction phase a client may visit clusters not contained in the current candidate list, if all clusters in this candidate list are already visited or when the consultant recommends it.

For candidate lists with a large number of elements, the algorithm performance in terms of running time worsens, due to the increase in the number of exchanges performed by the local search procedure.

As seen, although there are no statistically significant differences, our algorithm variant with confidence is outperformed by the memetic algorithm of Gutin and Karapetyan (GK), which is currently the best published heuristic for the GTSP. The GK algorithm uses a sophisticated local improvement strategy that combines many local search heuristics. One goal of our future research is to adopt a similar approach for the local improvement part of our algorithm, but still using candidate lists for each local search heuristic considered.

5.9. Conclusions and future work

In this chapter, we have introduced Consultant-Guided Search (CGS), a novel metaheuristic for combinatorial optimization problems, based on the direct exchange of information between individuals in a population. The metaphor used by CGS is that of clients that receive advice from consultants in order to solve a given problem. This metaphor was inspired by the way in which agent interaction is modeled by the MetFrEm framework.

We have applied the CGS metaheuristic to three classes of combinatorial optimization problems: the Traveling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP) and the Generalized Traveling Salesman Problem (GTSP). For the TSP, we have devised the CGS-TSP algorithm, which is able to compete with the best ACO algorithms. Moreover, a variant of CGS-TSP that attaches confidence levels to the recommendations made by consultants, yields even better results. For the QAP, we have introduced the CGS-QAP algorithm and we have also proposed a few variants of this algorithm, which combine the basic variant with other techniques. The experimental results show that CGS-QAP outperforms MAX-MIN Ant System for unstructured QAP instances. For the GTSP, we have used a local-global approach and we have defined the virtual distance between two supernodes as the distance between the centers of mass of the two corresponding clusters. We have introduced candidate lists based on these virtual distances, in order to reduce the search space. This has led to an efficient algorithm, able to compete with the best GTSP algorithms.

A drawback of the CGS algorithms proposed in this chapter is the large number of parameters to be tuned. We plan to devise variants of these algorithms that use only a small number of parameters. To this end, we will study the influence of the different components of CGS on its performance and we will try to identify parameters whose optimal value is not problem specific. These parameters can be subsequently removed by hard-coding their values in the algorithm.

An important goal of our future research is to apply CGS to other combinatorial optimization problems in order to identify classes of problems for which CGS could achieve state of the art performance. Furthermore, we plan to analyze the viability of a hybrid approach that combines CGS and ACO, thus benefiting of both direct communication and stigmergy.

Another research goal is to adapt the CGS metaheuristic to continuous optimization problems. One possible way to achieve this is to follow the approach used by the $ACO_{\mathbb{R}}$ [102] algorithm, which is an extension of ant colony optimization to continuous domains. In both CGS and ACO, at each construction step, a solution component is chosen from a finite set of available components, based on the probabilities associated with each of the components. For continuous optimization problems, the set of components to choose from is infinite. The idea of $ACO_{\mathbb{R}}$ is to replace the discrete probability distribution with a probability density function, constructed as a weighted sum of several one-dimensional Gaussian functions. In order to compute the parameters of this probability density function, $ACO_{\mathbb{R}}$ maintains an archive of good solutions found during the search. To each Gaussian function, there is a corresponding solution in the archive. Based on these solutions, $ACO_{\mathbb{R}}$ determines the weights of the Gaussian functions and the values of their parameters. Having computed the probability density function, $ACO_{\mathbb{R}}$ can sample a value that will be used as the solution component for the current construction step. The CGS algorithms presented in this chapter implement the strategy of a consultant as a solution advertised to its clients, which means that these algorithms already keep track of a number of good solutions. Therefore, it seems easy and natural to apply the ideas of $ACO_{\mathbb{R}}$ to CGS in order to adapt it for continuous optimization problems. Of course, thorough research is needed to test the viability of this approach.

As mentioned before, in our current instantiations of CGS, we have represented the strategy of a consultant as a solution recommended to its clients. This is a somewhat disappointingly way to implement a strategy, because it is very basic and it is probably not what most people would expect when thinking of a strategy. It is therefore interesting to fully use the possibilities offered by CGS, by conceiving algorithms where the strategy of a consultant has a different representation than that of the solution. One could for example represent the strategy as a set of rules, as a finite state machine or even as a genotype. In each case, the challenge is to find the right method to adjust the strategy each time one of the clients achieves a success.

The results presented in this chapter are mainly of experimental nature. This is not surprising, taking into account that CGS is a new metaheuristic. For almost all existing metaheuristics, the initial work has been purely experimental. Theoretical studies were carried out only after empirical evidence had shown the merits of a metaheuristic. CGS has been applied until now to only three classes of problems and more experimental research is needed in order to assess the full potential of this metaheuristic. However, the successes already obtained by CGS show that the theoretical study of this metaheuristic is an important future research direction. This implies investigating the computational properties of CGS in order to obtain answers to questions concerning convergence, expected optimization time, importance of different components of this metaheuristic or impact of the parameters' values on its performance. As in the case of other metaheuristics [86], a theoretical analysis of CGS is not amenable in the general case. Instead, it will be necessary to derive theoretical results for particular CGS algorithms and, most probably to algorithm variants simpler than those presented in this chapter.

6. CONCLUSIONS

The main motivation for the research presented in this thesis was the problem of engineering emergent behavior. This is a difficult and very general problem, for which no universally applicable methodology is currently known. We have explored in this dissertation a few aspects of this issue, starting from theoretical approaches and gradually moving toward more practical ones.

In order to build our research on a scientific foundation, we have begun by developing a mathematical formalism of emergence. This task was complicated by the absence of a unique definition of emergence and by the fact that most definitions are based on subjective factors such as the surprise element or the perceived novelty. We have been able to offer a definition of emergence that is both objective and suitable for practical purposes by taking an unconventional perspective on emergence. Instead of focusing on the properties of emergence, we have argued that a definition should rely only on the processes that produce emergent phenomena. Although the properties of emergence are not part of our definition, they represent a main subject of interest. One of the main reasons for developing our mathematical formalism of emergence was the possibility to derive truths concerning the properties of emergence in a rigorous way. The approach proposed in this thesis is an empirical one: such truths should be first identified by performing computational experiments and they can be then expressed, analyzed and proved using our mathematical formalism, in order to establish theoretical results.

The above mentioned approach implies the existence of a general setting for the study of emergence, which must be compatible with our mathematical formalism. Therefore, we have designed a meta-framework called MetFrEm, which can be used to describe various algorithmic frameworks comprising a population of interacting agents. We have identified a number of open questions that could be addressed by performing computational experiments with agent-based systems modeled in our meta-framework. One of the hypotheses guiding our research was that, in order to engineer a desired emergent behavior, it is preferable to use heterogeneous systems with agents that follow only simple rules instead of considering homogeneous systems with complex agents. Consequently, we have designed MetFrEm to allow describing systems comprising a great number of agent types without the need to explicitly state the interaction rules between each pair of agent types. In our framework, an agent of a given type may know how to interact with agents of other types without even being aware of the existence of these other types.

The goal of being able to describe agent rules in a generic manner has led to a particular way in which agent interactions should be modeled in MetFrEm. In turn, this particular way of interacting has led us to the idea of a new metaheuristic for combinatorial optimization problems, which we have called the Consultant-Guided Search (CGS). This metaheuristic has proved very successful, being able to produce state-of-the art results for all classes of problems to which we have applied it: the traveling salesman problem, the quadratic assignment problem and the generalized traveling salesman problem.

Currently, there are many open questions related to the problem of engineering emergent behavior. Although our research does not give definitive answers to these questions, we believe that it points the way toward a theory of emergence. An argument in favor of this assumption is given by the state-of-the-art results obtained by our CGS algorithms. Since the CGS metaheuristic reflects the way in which agent interactions take place in MetFrEm, and since MetFrEm is compatible with our mathematical formalism of emergence, we believe that the work presented in this thesis represents a step in the right direction. We view the mathematical formalism introduced in this thesis as the theoretical foundation for the problem of engineering emerging behavior, and the MetFrEm meta-framework as a major tool for the experimental study of emergence. Our future work will further concentrate on investigating appropriate methods to tackle this open-ended research topic.

APPENDIX A. AGSYSLIB - A SOFTWARE TOOL FOR AGENT-BASED PROBLEM SOLVING

Many complex phenomena that arise in physical and biological systems can be naturally described using agent-based models [115][112][46][21]. These models are able to capture the complex behavior that emerges from the interactions between agents governed by simple rules. It is tempting to use agent-based approaches not only to describe existing phenomena, but also to solve complex problems that cannot be tackled with conventional techniques. Consequently, in recent years, there has been a growing interest in designing algorithms that take advantage of the emergent behavior exhibited by systems composed by interacting agents [103].

While conceiving a new agent-based algorithm, one can benefit from the various software libraries and frameworks available nowadays for agent-based modeling and simulation (ABMS) [80]. However, ABMS software is not able to assist in all aspects related to the design, implementation and tuning of agent-based algorithms. In this chapter, we identify the difficulties encountered during these activities and we discuss how a software tool can help in overcoming them. We illustrate our ideas by presenting AgSysLib, a Java tool that we have developed in order to facilitate the tasks associated with agent-based problem solving.

The main difficulty in designing agent-based algorithms is to find the set of rules that produce the desired emergent behavior. At present, there is no generally accepted methodology for designing agent-based algorithms. Some of these algorithms are nature inspired. They adopt mechanisms found in biological systems such as colonies of ants [45] and bees [69], flocks of birds [88] or immune systems [32]. When no inspiration is found in nature, the task of finding an appropriate set of rules for the agents becomes more difficult, because the emergent behavior is often surprising and counterintuitive. Therefore, the design of an agent-based algorithm is frequently a trial-and-error process that could benefit from the help provided by a software tool.

While ABMS offers mainly qualitative insights, agent-based problem solving is concerned with producing high performance results in terms of both solution quality and running time. Turning a proof-of-concept simulation into a state-of-the-art implementation for solving a real-world problem is a very tedious task, for which ABMS software does not typically provide support. Since currently no tools are offering assistance with this task, we have developed AgSysLib in order to help algorithm designers and engineers in implementing agent-based solutions for complex problems.

A.1. The AgSysLib framework and library

AgSysLib is both a framework and a library. It is implemented in Java and it is available as an open source project at http://agsyslib.sourceforge.net/. AgSysLib offers an API (Application Programming Interface) that should be implemented by any agent-based algorithm and it provides a set of utility classes that help performing various tasks associated with agent-based

problem solving. For many of the interfaces specified by the API, AgSysLib offers default implementations or abstract classes that can be easily instantiated, extended and customized.

AgSysLib features a component-based architecture, which permits to build algorithms in a modular way and facilitates the experimentation and analysis of different variants of an algorithm. A new variant of an algorithm can be obtained by simply replacing a particular component of a base implementation with another component. AgSysLib also allows executing batches of runs, in order to apply repeatedly an algorithm to the same problem, to different problems, or using different parameter values.

As mentioned before, currently available ABMS software packages offer many features that are also useful for agent-based problem solving. AgSysLib does not try to provide yet another implementation of these features. Instead, it is concerned with those aspects of designing, implementing and tuning of agent-based algorithms that are not covered by ABMS software. However, in order to give users the possibility to still benefit from features available in ABMS software, AgSysLib can act as a wrapper for such libraries. This way, AgSysLib can transform an ABMS package into a tool able to assist in agent-based problem solving.

A plethora of ABMS packages has been developed in the last years, differing in their purposes and capabilities. In a recent survey, Allan [2] discusses 31 of the most commonly used toolkits for agent-based modeling and simulation, while in another survey, Nikolai and Madey [87] consider 53 such toolkits. AgSysLib is able to wrap around many of these packages, but in its default configuration, it integrates the MASON library. MASON (Multi-Agent Simulator of Neighborhoods) [79] is a discrete-event multi-agent simulation environment implemented in Java, allowing models with a large number of agents to be executed fast a large number of times. Some of its main features include:

- checkpointing – simulations can be serialized to disk. A serialized simulation can be later recovered with or without visualization, and it can be migrated on a different platform.
- model / view decoupling – models are completely independent of their visualizations. Different types of visualizations can be defined for the same model and they can be dynamically added and removed.
- media – various 2D and 3D visualizations, charts and graphs are available, with the possibility to save them as snapshots (in PNG format) or as Quicktime movies.
- duplicability – MASON simulations are able to produce identical results across different platforms.
- high quality random number generator – many agent-based algorithm are stochastic and they need a robust random number generator in order to produce valid results. MASON provides an efficient implementation of Mersenne Twister [81], which is a high quality random number generator.

A.2. Agent-based problem solving with AgSysLib

In this section, we identify a series of issues related to agent-based problem solving and we show how AgSysLib can assist in overcoming them. We introduce first a simple problem that will help us illustrate the use of AgSysLib.

A.2.1. The “Strange Collisions” problem

The problem considers a number of particles in a two-dimensional universe represented as a toroidal grid. These particles move horizontally, vertically or diagonally and they randomly change their direction from time to time. The time is discrete and at each step, each particle

moves one position in its current direction. At start, all particles have the same initial energy E_0. When two or more particles collide, their energies multiply according to the number of particles involved in the collision. In the absence of collisions, the energy of a particle varies in time according to a linear function. We let the universe evolve for 1000 steps and then we count the number of particles with energies in the interval $[0.5 \cdot E_0, 1.5 \cdot E_0]$. A pseudocode description of this scenario is given below:

```
 1  foreach agent do
 2      pos[agent] ← random position
 3      dir[agent] ← random direction
 4      energy[agent] ← E0
 5  end foreach
 6  for step = 1 .. 1000 do
 7      foreach agent do
 8          pos[agent] ← move one position according to dir[agent]
 9          with probability p: dir[agent] ← random direction
10          k ← total number of agents at pos[agent]
11          if k > 1 then
12              energy[agent] ← k * energy[agent]
13          else
14              energy[agent] ← a + b * energy[agent]
15          end if
16      end foreach
17  end for
18  count ← | {agent | energy[agent] ∈ [0.5·E0, 1.5·E0]} |
```

Fig. A.1. Pseudocode of the "strange collisions" problem.

For a given initial energy E_0, a given dimension of the toroidal grid, a given number of particles and a given probability p of changing the move direction (line 9), the problem consists in finding the coefficients a and b (line 14) that maximize the value of $count$ (line 18). In order to exclude some trivial solutions (such as $a=0.5 \cdot E_0$, $b=0$) and for exemplification purposes, we add the constraints: $a \in [-10, 10]$, $b \in [0.7, 1.2]$.

This is not a real-world problem and the law governing the modification of energy during collisions (line 12) bears no similarity with the laws governing real physical systems (hence the name "strange collisions"). However, this simple problem allows us to exemplify the difficulties encountered in solving real-world problems and to illustrate the use of AgSysLib in order to overcome them.

The source code of the classes implementing the universe described by this problem, along with all other classes and configuration files mentioned in this paper, is available in the AgSysLib package. In the next paragraphs, we present only fragments of code or configuration necessary to understand the working of AgSysLib.

A.2.2. Configuration

Agent-based problem solving is typically a trial-and-error process requiring experimentation at various levels. One type of experiments involves assessing different variants of an algorithm. Frequently, switching to a new algorithm variant is achieved by commenting out portions of the original code and replacing them with new code, or by using flags in various parts of the program in order to activate the portions of code corresponding to the given algorithm variant. Such a practice clutters the source code, making it hard to read and to maintain. A component-based architecture allows a clear separation of the portions of code

specific to each algorithm variant, but it usually still requires some changes in the original code, in order to specify which component should be used. AgSysLib allows specifying all components of an agent-based system in a configuration file. This way, no code changes or additional flags are needed in order to switch between different algorithm variants.

Another type of experiments involves comparing the results obtained by a given algorithm with different sets of parameters or even with different formulas. If parameter values or algorithm formulas are hard-coded into the program, this leads again to code cluttering. Therefore, AgSysLib provides a large set of utility functions for reading various types of values and lists of values, as well as mathematical formulas from a configuration file. Moreover, AgSysLib allows providing lists of values for parameters and executing batches of runs for each combination of these parameter values. Finally, it is possible to specify in the configuration file that the value of a parameter should be computed based on a given formula using the values of other parameters.

In order to exemplify the use of configuration files with AgSysLib, let us consider that we already have implemented the classes needed to emulate the "strange collisions" problem and we want to configure a universe consisting of 100 particles with the initial energy E_0=100, moving on a 50 x 50 toroidal grid and having at each step a probability of 0.03 to change their direction. For our first experiment we consider that in the absence of collisions, the energy of a particle changes according to the formula: 7.77 + 0.77 * energy. The corresponding configuration file is presented below:

```
initializer.class = net.sourceforge.agsyslib.collision.ParticleSystemInitializer
system.evolution.class =
net.sourceforge.agsyslib.collision.ParticleSystemEvolution
maxTickCount = 1000
gridWidth = 50
gridHeight = 50
particleCount = 100
normalEnergy = 100
directionChangeProbability = 0.03
energyFormula = 7.7 + 0.77 * energy
```

Fig. A.2. AgSysLib configuration file (variant 1).

An AgSysLib configuration file is a Java properties file containing entries for at least the following two mandatory properties: initializer.class and system.evolution.class. These properties specify the names of the Java classes implementing two high-level interfaces of the AgSysLib API: Initializer and AgentSystemEvolution. The Initializer interface has a single, parameterless method, called getNextAgentSystem, which is called before starting a new run in a batch, in order to retrieve the agent-based system for the new run. In the case of our "strange collisions" problem, this agent-based system is represented by the universe of the colliding particles. The AgentSystemEvolution interface declares methods that should implement the actions performed at the beginning and end of each run in a batch, and at the beginning, at the end and during each step in a run. It also declares a method that should implement the termination condition of a run. For both interfaces, AgSysLib offers abstract methods implementing the base functionality usually needed by any concrete implementation.

Instances of the classes declared in the configuration file for the two interfaces mentioned before are created through Java reflection. Since this operation is done only once, at program start, it has no impact on the performance of the application.

In its basic formulation, the "strange collisions" problem specifies that the result is obtained after a number of 1000 steps in the evolution of the universe. However, as observed in Fig. A.2, we do not hard code this number in our algorithm. Instead, we make it configurable by means of the parameter maxTickCount, in order to be able to generalize the problem for an arbitrary number of steps. Similarly, we do not hard code the rule governing the energy modification in the absence of collision as a linear function. We offer instead the possibility to specify a formula for the energy modification rule, in order to allow using more complex, non-linear functions. This way, we are able to experiment with any formula that can be expressed using the functions provided by java.lang.Math, without needing to make changes in our code. A few such possible formulas are given for exemplification below:

```
energyFormula = 100 - log(1 + energy)
energyFormula = 30 + 40 * exp(1 + 1 / energy)
energyFormula = 70 - 20 * cos(sqrt(abs(100 - energy)))
```

A number of other features related with parameters and formulas in configuration files are presented in subsection A.2.4, where we also describe the AgSysLib utility classes and methods used to retrieve the associated values, and we provide technical details about their implementation.

A.2.3. Listeners

During experimentation with an agent-based algorithm, it is frequently necessary to inspect the evolution of various quantities handled by the application, or aggregate values of them, in order to gain insight about the behavior emerging in the system. Similar information is required during tuning and debugging activities. The statements needed to output this information usually clutter the source code. Moreover, they may affect the performance of the algorithm, although they are usually no longer needed in the final application. The AgSysLib API introduces evolution listeners in order to allow keeping these statements separated from the main source code, while also permitting the quick activation or deactivation of these portions of code. An evolution listener provides methods that can be triggered by the following events:

- the start of the processing for a batch of runs
- the end of the processing for a batch of runs
- the start of the processing for a run in a batch
- the end of the processing for a run in a batch
- the start of an evolution step in a run
- the end of an evolution step in a run
- the end of the operations performed by an agent during a step.

The evolution listeners active during the execution of an application can be specified in a configuration file. This way, there is no need to make changes in the application code in order to add or remove a listener. In Fig. A.3, we show how a listener can be used in our "strange collisions" application. Differences from the configuration file presented in Fig. A.2 are in boldface.

The purpose of the listener used in this example is to aggregate the results obtained by the algorithm in a batch of runs. It writes its output to a separate file, in order to keep it apart from the application's standard output. The size of the batch is specified by the parameter repeat.

```
initializer.class = net.sourceforge.agsyslib.collision.ParticleSystemInitializer
system.evolution.class = net.sourceforge.agsyslib.collision.ParticleSystemEvolution
system.evolution.listener.class.1 = \
                        net.sourceforge.agsyslib.collision.NormalEnergyCountListener
result.file = result.txt
repeat = 50
seed = 1234567654321
maxTickCount = 1000
gridWidth = 50
gridHeight = 50
particleCount = 100
normalEnergy = 100
directionChangeProbability = 0.03
energyFormula = 7.7 + 0.77 * energy
```

Fig. A.3. AgSysLib configuration file (variant 2).

The application uses a stochastic algorithm, because the particles are initialized with random positions and directions and they may also randomly change their direction with a given probability. Therefore, the results differ from run to run and they should be averaged over a number of runs in order to be statistically meaningful. By default, AgSysLib's random generator is initialized with a seed based on the current system time, but a particular value for the seed can be specified in the configuration file, in order to make the experiments duplicable. Running the algorithm with the configuration from Fig. A.3, produces the following output:

```
energyFormula: 7.7 + 0.77 * energy, averageCount:  12.66 (12.66 %),
                        bestCount:  27, worstCount:   5, stdev: 4.27, particleCount: 100
```

A.2.4. Experimentation and tuning

As mentioned in subsection A.2.2, AgSysLib allows specifying lists of values in the configuration files, in order to run an algorithm with all possible combinations of these parameter values. An example of using value lists is given in Fig. A.4, where differences from the configuration file presented in Fig. A.3 are in boldface. In the configuration from Fig. A.4, value lists are specified for parameters prm.a and prm.b. The parameter names must be prefixed with the text "value.list" followed by a number indicating the order in which the value lists have to be handled. For each of the 5 x 6 = 30 possible combination of values for the parameters prm.a and prm.b, AgSysLib will execute a batch of 50 runs and it will report the aggregate results of each batch, by means of the configured listener.

```
initializer.class = net.sourceforge.agsyslib.collision.ParticleSystemInitializer
system.evolution.class = net.sourceforge.agsyslib.collision.ParticleSystemEvolution
system.evolution.listener.class.1 = \
                                net.sourceforge.agsyslib.collision.NormalEnergyCountListener
result.file = result.txt
repeat = 50
seed = 1234567654321
maxTickCount = 1000
gridWidth = 50
gridHeight = 50
particleCount = 100
normalEnergy = 100
directionChangeProbability = 0.03
value.list.1.prm.a = -10, -5, 0, 5, 10
value.list.2.prm.b = 0.7, 0.8, 0.9, 1.0, 1.1, 1.2
energyFormula = a + b * energy
```

Fig. A.4. AgSysLib configuration file (variant 3).

AgSysLib reads the content of a configuration file in a variable of type MultivaluedProperties. This utility class provides methods for accessing parameters of various types, with values that can be simple, computed or taken from a list. Let us consider that the configuration file from Fig. A.4 has been read in a variable named props. Then, the value of the particleCount parameter can be retrieved using: props.getInt("particleCount"). Similarly, the value of the parameter prm.a for the current batch is given by: props.getDouble("prm.a").

AgSysLib also offers a number of utility classes that permit the evaluation of mathematical formulas appearing in a configuration file. The code needed to create an evaluator object for the energyFormula from Fig. A.4 is:

```
String energyFormula = props.get("energyFormula");
this.energyEvaluator = new ExtPropsEvaluatorDoubles(
                                        props, energyFormula, "prm", "energy");
```

In the above code, the constructor is instructed to create an evaluator that retrieves the coefficients of its formula from parameters prefixed with "prm" and accepts an argument called "energy". It can be observed in Fig. A.4 that in the energyFormula, the parameters are referred simply as a and b, but they should be prefixed with "prm" in the entry lines specifying their values.

The energyEvaluator can be used then to update the energy of a particle after a move not involving a collision:

```
double updatedEnergy = this.energyEvaluator.evaluate(energy);
```

AgSysLib does not use an interpreter to evaluate formulas, because this would have a negative impact on the performance of an algorithm. Instead, it generates on-the-fly a Java class for each formula and creates an instance of it. The dynamically generated class contains a method that accepts as arguments the variables present in the given formula and returns the value corresponding to its evaluation. The on-the-fly class generation takes place only once, at program start. This way, the evaluation of a formula is performed as fast as if it would have been hard-coded in the algorithm code. In a series of experiments aimed to assess the

performance of our evaluator implementation, we have determined that the time needed to evaluate a formula using a Groovy interpreter is in average about 15 times longer than that needed by our evaluator.

AgSysLib also uses evaluators in order to obtain the values of computed parameters. These are parameters whose values depend on the values of other parameters. Examples of computed parameters are given later in this subsection.

Comparing the aggregate results obtained for each batch of 50 runs executed by AgSysLib using the configuration presented in Fig. A.4, it is possible to select the best performing combination and use it as a solution to our problem. While solutions obtained using this approach are of reasonably good quality, they are usually not close to the optimum solution. In order to obtain high-quality solutions, a more elaborated tuning procedure should be applied. One such tuning procedure offered by AgSysLib is based on genetic algorithms and it uses the JGAP library [82]. An example configuration file for this type of tuning is shown in Fig. A.5, where the differences from the configuration file presented in Fig. A.4 are in boldface.

```
initializer.class = net.sourceforge.agsyslib.collision.ParticleSystemInitializer
system.evolution.class = net.sourceforge.agsyslib.collision.ParticleSystemEvolution
system.evolution.listener.class.1 = \
                                net.sourceforge.agsyslib.collision.NormalEnergyCountListener
result.file = result.txt
repeat = 50
seed = 1234567654321
maxTickCount = 1000
gridWidth = 50
gridHeight = 50
particleCount = 100
normalEnergy = 100
directionChangeProbability = 0.03
fitness.provider.class = net.sourceforge.agsyslib.collision.ParticleFitnessProvider
ga.max.evolutions = 10
ga.population.size = 1000
gene.0 = prm.a, double, -10, 10
gene.1 = prm.b, double, 0.7, 1.2
energyFormula = a + b * energy
```

Fig. A.5. AgSysLib configuration file (variant 4).

Instead of using lists of values, the tuning configuration specifies value ranges for the parameters to be tuned. Each parameter is associated with a gene. In our example, the gene associated with the prm.a parameter has alleles of type double, with values between -10 and 10. Similarly, the gene associated with the prm.b parameter has alleles of type double, with values between 0.7 and 1.2. The configuration also instructs the genetic algorithm to use a population of 1000 potential solutions and to evolve for 10 generations.

There is only one interface provided by the AgSysLib API that must be implemented in order to perform the genetic algorithm tuning. This is the FitnessProvider interface, which offers two methods: getCurrentResult and getFitness. The first one is called at the end of each run and should return a value representing the result of the current run. The second one is called at the end of each batch of runs and it should return a fitness value computed by aggregating the results of all runs in the given batch.

In our example, the result of a run is the number of particles having energy in the range $[0.5 \cdot E_0, 1.5 \cdot E_0]$ at the end of this run. The fitness of a batch is computed as the average number of particles with energy in this "normal" range. The implementation of the methods declared by the FitnessProvider interface is given below:

```
public double getCurrentResult(AgentSystem system) {
        return ((ParticleSystem) system).getNormalEnergyCount();
}

public double getFitness(AgentSystem system, double[] results) {
    double fitness = 0;
    for(double result : results) {
        fitness += result;
    }
    return fitness / results.length;
}
```

Using the configuration from Fig. A.5, AgSysLib has run the genetic algorithm for 10 generations and it has found the following combination of parameters: prm.a=9.748942613581587; prm.b=0.8064227099345613. We have then adjusted the configuration from Fig. A.3 based on these parameter values and AgSysLib has produced the following result:

```
energyFormula: 9.748942613581587 + 0.8064227099345613 * energy,
        averageCount:  96.40 (96.40 %), bestCount: 100, worstCount:  89,
        stdev: 2.97, particleCount: 100
```

The result above shows that the tuning procedure has been able to find a high-quality solution to our problem.

We change now the configuration from Fig. A.5, in order to instruct the tuning procedure to take into consideration only values having up to two decimal points while searching for a good combination of parameters. This example also allows us to show the use of computed parameter values. The fragment of the configuration file relevant for the tuning procedure is shown in Fig. A.6, where differences from the configuration file presented in Fig. A.5 are in boldface.

```
fitness.provider.class = net.sourceforge.agsyslib.collision.ParticleFitnessProvider
ga.max.evolutions = 10
ga.population.size = 1000
gene.0 = xa, int, -1000, 1000
gene.1 = xb, int, 70, 120
value.computed.1.prm.a = xa / 100.0
value.computed.2.prm.b = xb / 100.0
energyFormula = a + b * energy
```

Fig. A.6. Fragment of an AgSysLib configuration file (variant 5).

In this case, the alleles have integer values (xa and xb), representing the value of their corresponding parameter multiplied by 100. The parameters prm.a and prm.b are then computed based on the values of xa and xb. Using the configuration from Fig. A.6, AgSysLib has found the following combination of parameters: prm.a=9.5; prm.b=0.81. We have then

adjusted the configuration from Fig. A.3 based on these parameter values and AgSysLib has produced the following result:

```
energyFormula: 9.5 + 0.81 * energy, averageCount:  96.40 (96.40 %),
                    bestCount: 100, worstCount:  89, stdev: 2.97, particleCount: 100
```

It can be observed that this solution leads to the same average number of particles with energy in the required range as the solution found using the configuration from Fig. A.5, but the current solution uses more accessible parameter values.

The question arises whether the above solution also produces high-quality results for other dimensions of the toroidal grid. In Fig. A.7 we show a configuration file used to experiment with square grids having sizes between 20 and 100. Differences from the configuration file presented in Fig. A.3 are in boldface. The widths of the grid are given as a list of values, while the grid height is a computed parameter with a value identical to that of the grid width. The number of particles is also a computed parameter, its value depending on the grid dimensions.

```
initializer.class = net.sourceforge.agsyslib.collision.ParticleSystemInitializer
system.evolution.class = net.sourceforge.agsyslib.collision.ParticleSystemEvolution
system.evolution.listener.class.1 = \
                        net.sourceforge.agsyslib.collision.NormalEnergyCountListener
result.file = result.txt
repeat = 50
seed = 1234567654321
maxTickCount = 1000
value.list.1.gridWidth = 20, 30, 40, 50, 60, 70, 80, 90, 100
value.computed.1.gridHeight = gridWidth
value.computed.2.particleCount = round(gridWidth * gridHeight / 25)
normalEnergy = 100
directionChangeProbability = 0.03
energyFormula = 9.5 + 0.81 * energy
```

Fig. A.7. AgSysLib configuration file (variant 6).

The results obtained by AgSysLib using the above configuration are presented in Table A.1. It can be observed that the combination of parameters found by the tuning procedure produces high-quality results for all considered grid sizes.

Table A.1. Results of the "strange collisions" problem for different grid sizes.

grid size	number of particles	average number of particles with energy in the required range
20 x 20	16	14.98 (93.63 %)
30 x 30	36	34.42 (95.61 %)
40 x 40	64	61.50 (96.09 %)
50 x 50	100	95.80 (95.80 %)
60 x 60	144	138.90 (96.46 %)
70 x 70	196	187.86 (95.85 %)
80 x 80	256	246.26 (96.20 %)
90 x 90	324	311.76 (96.22 %)
100 x 100	400	385.08 (96.27 %)

The genetic algorithm tuning is able to find good combinations of parameters for many problems. However, in some cases, more sophisticated tuning procedures are required in order to obtain high-quality results. A number of software packages for automatic parameter configuration are currently available, for example: ParamILS [59], F-Race [14] or SPOT [6]. AgSysLib enables an easy integration with these tools, thus offering many possibilities to tune the performance of an algorithm.

A.2.5. Debugging

Agent-based applications are usually very adaptive, thus being the best choice for solving real-world problems in complex, dynamic environments. However, detecting flaws in such applications is more difficult, because in many cases, even a buggy implementation is able to solve the problem, although not as efficient as possible. In addition, when designing a new agent-based heuristic, one does not know in advance how efficient an implementation could be, therefore bugs that only affects the algorithm performance may remain unnoticed, since the developer has only limited knowledge about what to expect from the algorithm. Because many agent-based algorithms are stochastic, reproducing an unusual behavior may also prove difficult.

Investigating problems related to agent-based applications often requires a detailed analysis of the dynamic of the internal program state. AgSysLib offers a GUI component that allows connecting via RMI (Remote Method Invocation) to a running application and interrogating, watching or changing its internal state. This GUI component is called the remote control console. Since it can establish a connection not only at program start-up, but at any moment, the remote control console can be also used to investigate unexpected behavior appearing in a program not actually under debugging.

Access to the internal state of a program is provided by means of scripts written in the Groovy programming language. Therefore, it is possible to make complex queries on the internal program state or to make multiple state changes, such as altering in some way the state of each agent, by using only a few lines of code. The scripts can be also registered, in order to be executed at the end of each step. In addition, the results of internal state queries can be used to set complex conditional breakpoints.

We use the "strange collisions" problem to illustrate the working of the remote control console. Suppose we want to check whether the combination of parameters found by the tuning procedure still produces high-quality results after a large number of steps. Therefore, we adjust the energyFormula in the configuration from Fig. A.3 and we set maxTickCount to a large value, for example 1000000. Since in this case the execution needs a long time to finish, we may want to check from time to time how the system evolves. To this end, we start the remote control console and query the internal program state, as shown in Fig. A.8.

After introducing the script code in the upper text area and pressing the "Evaluate" button, the results are displayed in the lower text area of the console. Scripts can be also saved to and loaded from a file. We can change the script in order to log the output into a file and register it for execution. By pressing the "Run" button, the information queried by the string is written to the log file at the end of each step.

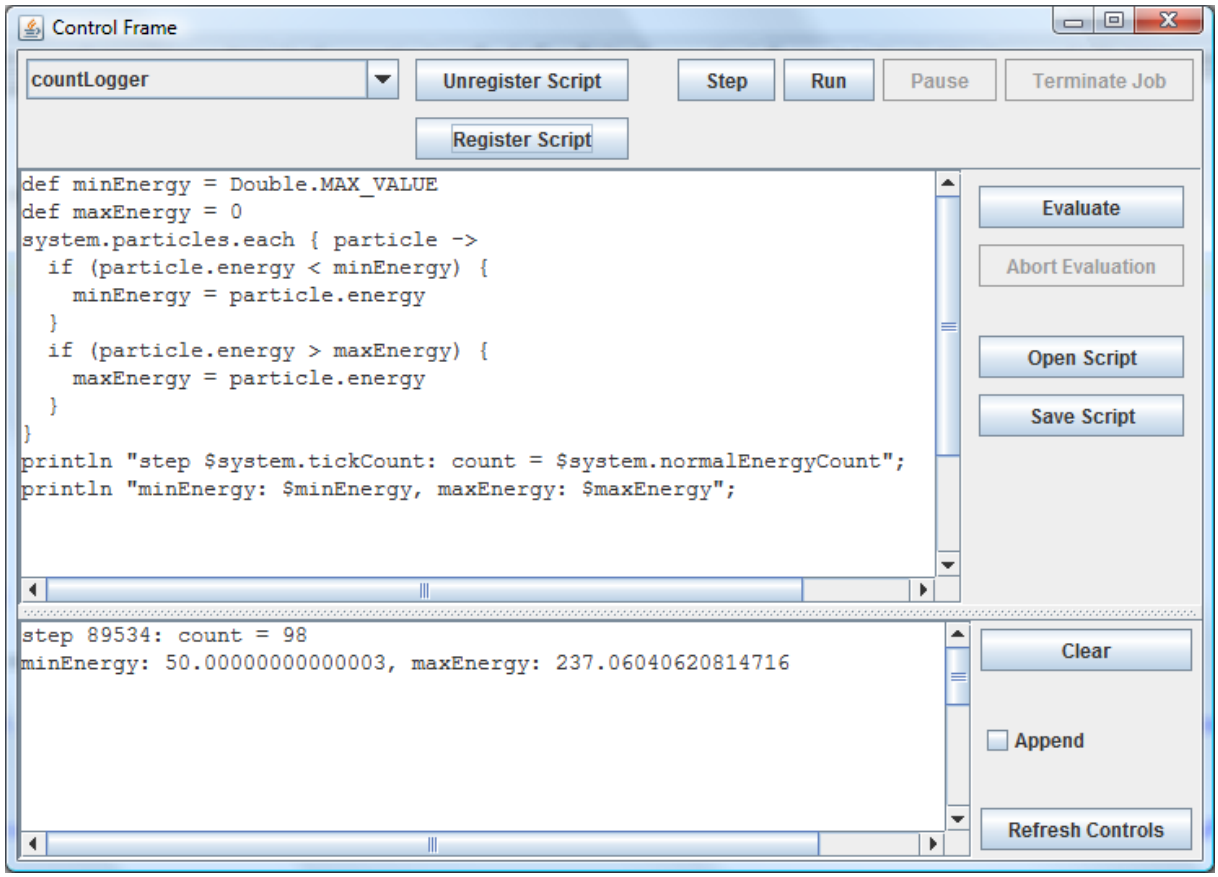


Fig. A.8. The remote control console.

Suppose now that we want to interrupt the execution of the program if the energy of a particle exceeds a given value, for example 1000. To do this, we add the following statements to our script, in order to set a conditional breakpoint:

```
if(maxEnergy > 1000) {
   control.pause()
}
```

When the breakpoint condition is satisfied, the program is interrupted and its internal state can be queried in the remote control console.

A.3. Conclusions

We have identified a number of issues involved in the process of solving a real-world problem using agent-based techniques and we have introduced AgSysLib, a software tool developed by us in order to overcome these issues. AgSysLib is both a framework and a library and it facilitates many tasks related with agent-based problem solving. It can act as a wrapper around existing ABMS software packages, thus seamless integrating their capabilities.

The various examples of AgSysLib usage presented in this paper show that it is a valuable tool for all people involved in agent-based problem solving.

REFERENCES

[1] Emile Aarts and Jan K. Lenstra, Eds., *Local Search in Combinatorial Optimization*.: John Wiley & Sons, Inc., 1997.

[2] R.J. Allan, "Survey of Agent Based Modelling and Simulation Tools," Science and Technology Facilities Council, Daresbury Laboratory, Warrington, Technical Report DL-TR-2010-007, 2010.

[3] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*.: Princeton University Press, 2007.

[4] Philip Ball, "For sustainable architecture, think bug," *New Scientist*, no. 2748, pp. 35-37, February 2010.

[5] E.R. Banks, "Information processing and transmission in cellular automata," Massachusetts Institute of Technology, Cambridge, MA, USA, PhD thesis 1971.

[6] Thomas Bartz-Beielstein, Christian Lasarczyk, and Mike Preuss, "The Sequential Parameter Optimization Toolbox," in *Experimental Methods for the Analysis of Optimization Algorithms*, Thomas Bartz-Beielstein et al., Eds. Berlin, Heidelberg, New York: Springer, 2010, pp. 337-360.

[7] J. Bates, "The role of emotion in believable agents," *Communications of the ACM*, no. 37(7), pp. 122-125, 1994.

[8] Roberto Battiti and Giampietro Tecchiolli, "The Reactive Tabu Search," *INFORMS Journal on Computing*, vol. 6, pp. 126-140, 1994.

[9] M A Bedau, "Downward causation and the autonomy of weak emergence," *Principia*, vol. 6, pp. 5-50, 2002.

[10] Mark A Bedau, "Weak Emergence," *Philosophical Perspectives 11: Mind, Causation, and World*, pp. 375--399, 1997.

[11] E.R. Berlekamp, J.H. Conway, and R. Guy, *Winning Ways for Your Mathematical Plays, vol. 2*.: Academic Press, 1982.

[12] William A Beyer, Peter H Sellers, and Michael S Waterman, "Stanislaw M. Ulams contributions to theoretical theory," *Letters in Mathematical Physics*, vol. 10, no. 2, pp. 231-242, 1985.

[13] Mauro Birattari, Gianni D Caro, and Marco Dorigo, "Toward the Formal Foundation of Ant Programming," in *ANTS '02: Proceedings of the Third International Workshop on*

Ant Algorithms, 2002, pp. 188--201.

[14] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp, "A Racing Algorithm for Configuring Metaheuristics," , 2002, pp. 11--18.

[15] Eric Bonabeau and Jean-Louis Dessalles, "Detection and emergence," *Intellectica*, vol. 2, no. 25, pp. 85--94, 1997.

[16] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. USA: Oxford University Press, 1999.

[17] Fabio Boschetti and Randall Gray, "A Turing Test for Emergence," in *Advances in Applied Self-organizing Systems*, Lakhmi Jain, Xindong Wu, and Mikhail Prokopenko, Eds.: Springer London, 2008, pp. 349-364.

[18] B Bullnheimer, R F Hartl, and C Strauss, "A new rank based version of the Ant System," Institute of Management Science, University of Vienna, Austria, Technical report 1997.

[19] R E Burkard, S Karisch, and F Rendl, "QAPLIB-A quadratic assignment problem library," *European Journal of Operational Research*, vol. 55, pp. 115-119, November 1991.

[20] Cosmin Carabelea, Olivier Boissier, and Adina Florea, "Autonomy in Multi-agent Systems: A Classification Attempt," *Agents and Computational Autonomy: Potentials, Risks and Solutions. Lecture Notes in Computer Science*, vol. 2969, pp. 59-70, 2004.

[21] Lars-Erik Cederman, "Endogenizing geopolitical boundaries with agent-based modeling," *Proceedings of The National Academy of Sciences*, vol. 99, no. suppl. 3, pp. 7296-7303, 2002.

[22] David J Chalmers, "Strong and weak emergence," in *The Re-Emergence of Emergence*.: Oxford University Press, 2006, pp. 244-254.

[23] Maurice Clerc, "Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem," in *New Optimization Techniques in Engineering*, G.C., Babu, B.V. Onwubolu, Ed.: Springer, 2004, pp. 219-239.

[24] E.F. Codd, *Cellular Automata*. New York: Academic Press, 1968.

[25] A Colorni, Marco Dorigo, and Vittorio Maniezzo, "Distributed Optimization by Ant Colonies," in *Proceedings of the First European Conference on Artificial Life*, Cambridge, MA, 1992, pp. 134--142.

[26] David T Connolly, "An improved annealing scheme for the QAP," *European Journal of Operational Research*, vol. 46, pp. 93-100, May 1990.

[27] M. Cook, "Universality in elementary cellular automata," in *Complex Systems, Vol. 15, Issue 1.*, 2004, pp. 1-40.

[28] A Croes, "A method for solving traveling salesman problems," *Operations Research*, vol. 5, pp. 791-812, 1958.

[29] James P Crutchfield, "The calculi of emergence: computation, dynamics and induction," *Physica D*, vol. 75, no. 1-3, pp. 11--54, August 1994.

[30] Vince Darley, "Emergent phenomena and complexity," *Artificial Life*, vol. 4, pp. 411--416, 1994.

[31] Paul Davidsson, "Agent Based Social Simulation: A Computer Science View," *Journal of Artificial Societies and Social Simulation*, vol. 5, no. 1, 2002.

[32] Leandro N. de Castro and Jonathan Timmis, *Artificial Immune Systems: A New Computational Intelligence Approach*.: Springer-Verlag London, 2002.

[33] J.-L. Deneubourg, S. Aron, and J.M. Pasteels, "The self-organizing exploratory pattern of the Argentine ant," *Journal of Insect Behavior*, no. 3, pp. 159-168, 1990.

[34] D.C. Dennett, *Consciousness Explained*. Boston: Back Bay Books, 1991.

[35] D.C. Dennett, *Darwin's Dangerous Idea: Evolution and the Meanings of Life*. New York: Simon & Schuster, 1995.

[36] D.C. Dennett, *Freedom Evolves*. New York: Penguin Books, 2003.

[37] Gianni di Caro and Marco Dorigo, "AntNet: Distributed Stigmergetic Control for Communications Networks," *J. Artif. Intell. Res. (JAIR)*, vol. 9, pp. 317-365, 1998.

[38] M d'Inverno and M. Luck, *Understanding Agent Systems*, 2nd ed. Berlin Heidelberg New York: Springer-Verlag, 2004.

[39] Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf, "Artificial chemistries - A review," *Artificial Life*, vol. 7, no. 3, pp. 225--275, June 2001.

[40] Kevin Dooley, "Complex adaptive systems: A nominal definition," *The Chaos Network*, no. 8, pp. 2-3, 1996.

[41] Marco Dorigo, "Optimization, Learning and Natural Algorithms [in Italian]," Dipartimento di Elettronica, Politecnico di Milano, Milano, PhD thesis 1992.

[42] M Dorigo, A Colorni, and V Maniezzo, "Positive feedback as a search strategy," Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, Technical Report 91-016, 1991.

[43] Marco Dorigo and Luca Maria Gambardella, "Ant Colony System: A cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53-66, 1997.

[44] Marco Dorigo, Luca Maria Gambardella, and Gianni Di Caro, "Ant algorithms for discrete optimization," *Artificial Life*, vol. 5, no. 2, pp. 137-172, April 1999.

[45] M. Dorigo and T. Stützle, *Ant Colony Optimization*.: The MIT Press, 2004.

[46] Joshua M Epstein, "Agent-based computational models and generative social science," *Complexity*, vol. 4, no. 5, pp. 41-60, May 1999.

[47] Matteo Fischetti, "A Branch-and-Cut Algorithm for the Symmetric Generalized Travelling Salesman Problem," *Operations Research*, vol. 45, no. 3, pp. 378--394, 1997.

[48] Walter Fontana, "Algorithmic Chemistry," in *Artificial Life II*, C.G. Langton et al., Eds. Redwood City, CA: Addison-Wesley, 1992, pp. 159--210.

[49] E Fredkin, "Digital mechanics: an informational process based on reversible universal cellular automata," *Physica D*, vol. 45, no. 1-3, pp. 254-270, October 1990.

[50] Luca Maria Gambardella and Marco Dorigo, "Ant-Q: A reinforcement learning approach to the traveling salesman problem," in *Proceedings of the Twelfth International Conference on Machine Learning (ML-95)*, Palo Alto, CA, 1995, pp. 252-260.

[51] M. Gardner, "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life"," *Scientific American*, no. 223, pp. 120-123, October 1970.

[52] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels, "Self-organized shortcuts in the Argentine ant," *Naturwissenschaften*, no. 76, pp. 579-581, 1989.

[53] P.P. Grassé, "La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes Natalensis et Cubitermes sp. La théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs," *Insectes Sociaux*, no. 6, pp. 41-81, 1959.

[54] Michael Guntsch and Martin Middendorf, "A Population Based Approach for ACO," in *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002*, 2002, pp. 72--81.

[55] Gregory Gutin and Daniel Karapetyan, "A memetic algorithm for the generalized traveling salesman problem," *Natural Computing*, vol. 9, no. 1, pp. 47--60, March 2010.

[56] John H Holland, "Complex adaptive systems," in *A new era in computation*. Cambridge, MA, USA: MIT Press, 1993, pp. 17--30.

[57] J.H. Holland, *Emergence: From Chaos to Order*. Oxford: Oxford University Press, 1998.

[58] Bin Hu and Günther R. Raidl, "Effective neighborhood structures for the generalized traveling salesman problem," in *Proceedings of the 8th European conference on Evolutionary computation in combinatorial optimization, EvoCOP'08*, Naples, Italy, 2008, pp. 36--47.

[59] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle, "ParamILS: an automatic algorithm configuration framework," *J. Artif. Int. Res.*, vol. 36, pp. 267--306, 2009.

[60] Serban Iordache, "A Framework for the Study of the Emergence of Rules in Multiagent Systems," in *Proceedings of the 20th International DAAAM Symposium*, Vienna, Austria, 2009, pp. 1285-1286.

[61] Serban Iordache, "Consultant-Guided Search Algorithms for the Quadratic Assignment Problem," in *Hybrid Metaheuristics - 7th International Workshop, HM 2010. LNCS*, vol. 6373, Vienna, Austria, 2010, pp. 148-159.

[62] Serban Iordache, "Consultant-guided search algorithms for the quadratic assignment problem," in *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, ortland, Oregon, USA, 2010, pp. 2089-2090.

[63] Serban Iordache, "Consultant-Guided Search Algorithms with Local Search for the Traveling Salesman Problem," in *11th International Conference Parallel Problem Solving from Nature - PPSN XI. LNCS*, vol. 6239, Krakow, Poland, 2010, pp. 81-90.

[64] Serban Iordache, "Consultant-guided search combined with local search for the traveling salesman problem," in *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, Portland, Oregon, USA, 2010, pp. 2087-2088.

[65] Serban Iordache, "Consultant-guided search: a new metaheuristic for combinatorial optimization problems," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation, GECCO 2010. ACM Press*, Portland, Oregon, USA, 2010, pp. 225--232.

[66] Serban Iordache and Florica Moldoveanu, "AgSysLib - A Software Tool for Agent-Based Problem Solving," *Scientific Bulletin of "Politehnica" University of Bucharest, C Series (Electrical Engineering)*, vol. 73, no. 2, 2011.

[67] Serban Iordache and Petrica C. Pop, "An Efficient Algorithm for the Generalized Traveling Salesman Problem," in *Proceedings of the 13-th International Conference on Computer Aided Systems Theory (EUROCAST 2011)*, Las Palmas de Gran Canaria, Spain, 2011, pp. 264-266.

[68] D.S. Johnson and L. McGeoch, "Experimental Analysis of Heuristics for STSP," in *The Traveling Salesman Problem and Its Variations (Combinatorial Optimization)*, Gregory Gutin and Abraham Punnen, Eds.: Springer, 2002, pp. 369-443.

[69] Dervis Karaboga and Bahriye Akay, "A survey: algorithms simulating bee swarm intelligence," *Artif. Intell. Rev.*, vol. 31, pp. 61--85, 2009.

[70] J. Kennedy and R.C. Eberhart, *Swarm Intelligence*.: Morgan Kaufmann, 2001.

[71] T.C. Koopmans and M.J. Beckmann, "Assignment problems and the location of economic activities," *Econometrica*, no. 25, pp. 53-76, 1957.

[72] Ales Kubík, "Toward a formalization of emergence," *Artificial Life*, vol. 9, no. 1, pp. 41--65, December 2002.

[73] Olivier Labarthe, Bernard Espinasse, Alain Ferrarini, and Benoit Montreuil, "Toward a methodological framework for agent-based modelling and simulation of supply chains in a mass customization context," *Simulation Modelling Practice and Theory*, vol. 15, no. 2, pp. 113-136, 2007.

[74] Pedro Larranaga and Jose A. Lozano, Eds., *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation (Genetic Algorithms and Evolutionary*

Computation).: Springer, 2001.

[75] Jiming Liu, XiaoLong Jin, and Kwok C Tsui, *Autonomy Oriented Computing: From Problem Solving to Complex Systems Modeling (Multiagent Systems, Artificial Societies, and Simulated Organizations)*.: Springer-Verlag New York, Inc., 2004.

[76] Ming Li and Paul Vitányi, *An introduction to Kolmogorov complexity and its applications*, 2nd ed.: Springer-Verlag New York, Inc., 1997.

[77] A.J. Lotka, *Elements of physical biology*. New York: Dover Publications, 1925.

[78] Helena Lourenço, Olivier Martin, and Thomas Stützle, "Iterated Local Search," in *Handbook of Metaheuristics*, Frederick S. Hillier, Fred Glover, and Gary Kochenberger, Eds.: Springer New York, 2003, vol. 57, ch. 11, pp. 320-353.

[79] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel C Balan, "MASON: A Multiagent Simulation Environment," *Simulation*, vol. 81, no. 7, pp. 517-527, 2005.

[80] C M Macal and M J North, "Agent-based Modeling and Simulation," in *Winter Simulation Conference*, Austin, TX, USA, 2009, pp. 86 -98.

[81] Makoto Matsumoto and Takuji Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3-30, January 1998.

[82] K. Meffert. (2011) JGAP - Java Genetic Algorithms and Genetic Programming Package. [Online]. http://jgap.sourceforge.net/

[83] John H Miller and Scott E Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*.: Princeton University Press, 2007.

[84] M. Mitchell, J. P. Crutchfield, and R. Das, "Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work," in *Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*, Moscow, Russia, 1996.

[85] Nenad Mladenovic and Pierre Hansen, "Variable neighborhood search.," *Computers & OR*, vol. 24, pp. 1097-1100, 1997.

[86] Frank Neumann and Carsten Witt, *Bioinspired Computation in Combinatorial Optimization : Algorithms and Their Computational Complexity*.: Springer, 2010.

[87] Cynthia Nikolai and Gregory Madey, "Tools of the Trade: A Survey of Various Agent Based Modeling Platforms," *Journal of Artificial Societies and Social Simulation*, vol. 12, no. 2, 2009.

[88] Riccardo Poli, James Kennedy, and Tim Blackwell, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, no. 1, pp. 33-57, 2007.

[89] Ioannis Politopoulos, "Review and Analysis of Agent-based Models in Biology," University of Liverpool, Technical Report 2007.

[90] Petrica C. Pop, "The Generalized Minimum Spanning Tree Problem," The Netherlands, PhD thesis 2002.

[91] Petrica C. Pop and Serban Iordache, "A Hybrid Heuristic Approach for Solving the Generalized Traveling Salesman Problem," in *GECCO 2011: Proceedings of the Genetic and Evolutionary Computation Conference*, Dublin, Ireland, 2011.

[92] Gerhard Reinelt, "TSPLIB—A Traveling Salesman Problem Library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376-384, 1991.

[93] P. Rendell, "Turing Universality of the Game of Life," in *Collision-Based Computing*.: Springer, 2002, pp. 513-539.

[94] Enda Ridge and Daniel Kudenko, "Determining Whether a Problem Characteristic Affects Heuristic Performance.," *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, vol. 153, pp. 21-35, 2008.

[95] Edmund M Ronald, Moshe Sipper, and Mathieu S Capcarrère, "Design, Observation, Surprise! A Test of Emergence," *Artificial Life*, vol. 5, pp. 225-239, 1999.

[96] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[97] X H Shi, Y C Liang, H P Lee, C Lu, and Q X Wang, "Particle swarm optimization-based algorithms for TSP and generalized TSP," *Inf. Process. Lett.*, vol. 103, pp. 169-176, 2007.

[98] Y. Shoham, "Agent-oriented programming," *Artificial Intelligence*, no. 60(1), pp. 51-92, 1993.

[99] John Silberholz and Bruce Golden, "The Generalized Traveling Salesman Problem: A New Genetic Algorithm Approach," *Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies*, pp. 165--181, 2007.

[100] A. Smith. (2007) Universality of Wolfram's 2, 3 Turing Machine. [Online]. http://www.wolframscience.com/prizes/tm23/TM23Proof.pdf

[101] Lawrence V Snyder and Mark S Daskin, "A random-key genetic algorithm for the generalized traveling salesman problem," *European Journal of Operational Research*, vol. 174, pp. 38-53, October 2006.

[102] Krzysztof Socha and Marco Dorigo, "Ant colony optimization for continuous domains," *European Journal of Operational Research*, vol. 185, pp. 1155-1173, 2008.

[103] Susan Stepney, Fiona Polack, and Heather R Turner, "Engineering Emergence," in *11th International Conference on Engineering of Complex Computer Systems (ICECCS 2006)*, Stanford, CA, USA, 2006, pp. 89-97.

[104] Thomas Stützle and Marco Dorigo, "ACO algorithms for the quadratic assignment problem," *New ideas in optimization*, pp. 33-50, 1999.

[105] Thomas Stützle and Susana Fernandes, "New Benchmark Instances for the QAP and the

Experimental Analysis of Algorithms.," in *EvoCOP*, vol. 3004, 2004, pp. 199-209.

[106] Thomas Stützle and Holger H Hoos, "MAX-MIN Ant system," *Future Gener. Comput. Syst.*, vol. 16, pp. 889--914, 2000.

[107] Eric Taillard, "Comparison of iterative searches for the quadratic assignment problem," *Location Science*, no. 3, pp. 87-105, 1995.

[108] Eric D Taillard, "Robust taboo search for the quadratic assignment problem.," *Parallel Computing*, vol. 17, pp. 443-455, 1991.

[109] Mehmet F Tasgetiren, P N Suganthan, and Quan-Qe Pan, "A discrete particle swarm optimization algorithm for the generalized traveling salesman problem," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07*, 2007, pp. 158--167.

[110] Dusan Teodorovic, "Bee Colony Optimization (BCO)," *Innovations in Swarm Intelligence*, vol. 248, pp. 39--60, 2009.

[111] Dusan Teodorovic and Mauro D Orco, "Bee Colony Optimization - A cooperative learning approach to complex transportation problems," *Advanced OR and AI Methods in Transportation*, pp. 51-60, 2005.

[112] Leigh Tesfatsion, "Agent-Based Computational Economics: Growing Economies From the Bottom Up," *Artificial Life*, vol. 8, no. 1, pp. 55-82, 2002.

[113] The Object Management Group. [Online]. http://www.omg.org

[114] G Theraulaz and E Bonabeau, "A brief history of stigmergy," *ARTIFICIAL LIFE*, vol. 5, no. 2, pp. 97-116, 1999.

[115] A Troisi, V Wong, and M A Ratner, "An agent-based approach for modeling molecular self-organization.," *Proc. National Academy of Sciences*, vol. 102, no. 2, pp. 255-260, 2005.

[116] V. Volterra, "Fluctuations in the Abundance of a Species Considered Mathematically," *Nature*, no. 188, pp. 558-560, 1926.

[117] J. von Neumann, *Theory of Self-Reproduction Automata*, A.W. Burks, Ed. Urbana, Illinois, USA: University of Illinois Press, 1966.

[118] Uri Wilensky and Kenneth Reisman, "Thinking Like a Wolf, a Sheep, or a Firefly: Learning Biology Through Constructing and Testing Computational Theories - An Embodied Modeling Approach," *Cognition and Instruction*, vol. 24, pp. 171--209, 2006.

[119] S. Wolfram, *A New Kind of Science*.: Wolfram Media, 2002.

[120] Stephen Wolfram, "Origins of Randomness in Physical Systems," *Physical Review Letters*, no. 55, pp. 449-452, 1985.

[121] S Wolfram, "Random sequence generation by cellular automata," *Advances in Applied*

Mathematics, vol. 7, no. 2, pp. 123-169, 1986.

[122] S. Wolfram, "Statistical mechanics of cellular automata," *Reviews of Modern Physics*, vol. 55, no. 3, pp. 601-644, July 1983.

[123] S. Wolfram, *The Mathematica Book*.: Cambridge University Press; 4 edition, 1999.

[124] Stephen Wolfram, *Theory and Applications of Cellular Automata*.: dvanced series on complex systems, World Scientific, 1986, vol. 1.

[125] S. Wolfram, "Universality and Complexity in Cellular Automata," *Physica D*, no. 10, pp. 1-35, 1984.

[126] L.P. Wong, M.Y.H. Low, and C.S. Chong, "A Bee Colony Optimization Algorithm for Traveling Salesman Problem," in *Proceedings of Second Asia International Conference on Modelling & Simulation (AMS 2008)*, 2008, pp. 818-823.

[127] Michael Wooldridge, *An Introduction to MultiAgent Systems*.: John Wiley & Sons, 2002.

[128] M Wooldridge and N. Jennings, "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review*, no. 10(2), pp. 115-152, 1995.

[129] Mark Zlochin, Mauro Birattari, Nicolas Meuleau, and Marco Dorigo, "Model-Based Search for Combinatorial Optimization: A Critical Survey," *Annals OR*, vol. 131, pp. 373-395, 2004.

[130] Konrad Zuse, "Calculating Space (translation of "Rechnender Raum")," Massachusetts Institute of Technology, Technical Translation AZT-70-164-GEMIT, 1970.