University „POLITEHNICA“ of Bucharest

Faculty of Automatic Control and Computers

Computer Science Department

Emergent Phenomena in Agent-Based Systems

Extended Abstract of the PhD Dissertation

Author: Serban Iordache

Supervisor: Professor Florica Moldoveanu

BUCHAREST 2011

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 [1]. 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 [11], 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. It has been successfully applied to many combinatorial optimization problems and it has been also adapted for continuous optimization problems.

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 [10] and MAX-MIN Ant System [32].
- *the Quadratic Assignment Problem (QAP)* - Our CGS algorithm for the QAP is significantly better than MAX-MIN Ant System [31], 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 [12], 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 [19] (**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 [17].

Iordache, S. *Consultant-Guided Search Algorithms for the Quadratic Assignment*. In: Hybrid Metaheuristics - 7th International Workshop, HM 2010. LNCS 6373, Vienna, Austria. Springer, 2010, pp. 148-159 [15].

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 [20].

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 [14].

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. 264-267 [21].

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 [16].

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 [18].

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) [28].

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

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.

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 [30] 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".

In the context of *Agent-Based Modeling and Simulation* (ABMS), Macal and North [24] consider that 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.

Most systems considered in ABMS are *complex adaptive systems* (CAS) [25]. These are systems that "change and reorganize their component parts to adapt themselves to the problems posed by their surroundings" [13]. In the context of CAS, "*agents* are semi-autonomous units that seek to maximize their fitness by evolving over time" [8].

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 [34] 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.

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 pheromone trails corresponding to the shortest paths between nest and food sources emerge from the collective behavior of individual ants.

Different people ascribe different meanings to this term, but, in general, they fall into two distinct classes: strong emergence and weak emergence. For Chalmers [4], 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, and most scientists reject this kind of emergence. Chalmers [4] 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”.

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 are mainly concerned with weak emergence, but we attach a broader meaning to this term, 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.

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.

Elementary cellular automata are the simplest cellular automata. They 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. Although extremely simple, elementary cellular automata can exhibit complex behavior, and it has been proved [5] that one of the 256 possible elementary cellular automata is computation universal.

An example of engineering emergent behavior with cellular automata is offered by Mitchell *et al*. [26], which use genetic algorithms that evolve a population of cellular automata in order to solve a particular problem.

2.4. Ant colony optimization

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. One example is offered by ant colony optimization (ACO), which takes inspiration from 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* [33]. 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 this technique. 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. At the end of each iteration, 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 first ACO algorithm, called Ant System, has been introduced by Marco Dorigo [9]. Many extensions of this algorithm have been proposed, among which Ant Colony System [10] and MAX-MIN Ant System [32] are currently the best-performing.

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.

There are many definitions for emergence in the literature, ranging from intuitive to formal and differing 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 [7], 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 [2]. 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 [3] or the use of an observer that relies on the surprise element [29]. There are, nevertheless, several attempts to provide an objective definition of emergence. One example is the approach taken by Crutchfield [6], 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 *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, because it also considers "uninteresting" behavior as emergent. In order to defend our position, we 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. In our view, defining emergence in terms of the characteristics of the exhibited behaviors is like arguing that only "interesting" 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.

One of the most interesting aspects of emergence is that even very simple rules are able to produce complex behavior. The question is how to express the notion of simple rules in a formal way. Using algorithmic complexity concepts such as Kolmogorov complexity [22] 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 3.2 *An agent-based model is a discrete-time dynamical system described by a tuple* $(S, \mathrm{L}, \mathrm{A}_0)$, *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 **3.1**:

Definition 3.3 *An agent-based model is decentralized iff:*

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

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 3.3 can be alternatively expressed as:

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

Based on this observation we provide the following definition:

Definition 3.4 *The centralization level of an agent-based model is a quantity γ given by:*

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

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

Definition 3.5 *An agent-based model is strongly decentralized iff* $\gamma \ll 1$.

Emergent phenomena can also appear in agent-based models that are only partially decentralized. We offer the following definition for this concept:

Definition 3.6 *An agent-based model is partially decentralized iff:*

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

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 3.7 *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 3.8 *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 3.9 *The dynamic of a decentralized agent-based model, expressed in terms of system-level properties, is called pure emergent behavior.*

Definition 3.10 *The dynamic of a partially decentralized agent-based model, expressed in terms of system-level properties, is called partially emergent 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?

One of the main reasons for developing our mathematical formalism of emergence is 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. 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. The main design objectives of MetFrEm are:

- to allow the modeling of virtually any agent-based algorithmic framework.
- to favor the modeling of strongly decentralized systems.
- to favor the modeling of systems with agents that follow only simple rules.
- to facilitate the modeling of algorithmic frameworks with highly heterogeneous agents.
- to offer a unitary approach to the modeling of direct communication and stigmergy.

4.1. Concepts and structure of MetFrEm

Universe
We denote as *universe* an agent-based model represented in MetFrEm. It 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
We use the term *mechanism* to denote an agent in MetFrEm. 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*. The set of properties that characterizes a mechanism is a subset of a finite *global set of properties* defined in the given universe. 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: 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* υ_O, the *active view function* υ_A and the *reactive view function* υ_R. We call *observable state* of a mechanism the set of values returned by the observable view function υ_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.

4.2. 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 $(P_I, P_O, P_A, P_R, s_0, v_o, v_A, v_R, \eta, \psi, \sigma, \varphi, f_A, f_R)$, 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. 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:

$$\mathcal{N}_t^{\mu} = \eta^{\mu}\Big(\bigcup_{m \in M_t} v_O(s_t^m)\Big) \tag{4.1}$$

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:

$$B_t^\mu = \bigcup_{N \in \mathcal{N}_t} \psi^\mu(\bigcup_{m \in N} \upsilon_O^\mu(s_t^m)) \tag{4.2}$$

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

$$V_t^\mu = \sigma^\mu(B_t^\mu) \tag{4.3}$$

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:

$$W_t^\mu = V_t^\mu \cup \{m \in M_t \mid \mu \in V_t^m\} \tag{4.4}$$

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:

$$s'^\mu_{t+1} = f_A^\mu(\{s_t^\mu\} \cup \{\upsilon_R^m(s_t^m) \mid m \in V_t^\mu\}) \tag{4.5}$$

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

$$s'^\lambda_{t+1} = f_R^\lambda(\{s_t^\lambda\} \cup \{\upsilon_A^\mu(s_t^\mu)\} \cup \{\upsilon_R^m(s_t^m) \mid m \in V_t^\mu, m \neq \lambda\}) \tag{4.6}$$

The definitive configuration of the universe at time step $t + 1$ is:

$$M_{t+1} = \bigcup_{m \in M_t} \varphi^m(s'^m_{t+1}) \tag{4.7}$$

4.3. 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 4-1. 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 4.1. 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 4.2. The *selectTargets* procedure (line 8) uses the selection function σ in order to choose the set of target mechanisms, in accordance with formula 4.3. 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 4.5. 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 4.6. 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 4.7.

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* (CGS), and which constitutes the subject of this chapter.

5.1. The CGS Metaheuristic

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. 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 accordance with its ***strategy***. Usually, at each step of the solution construction, there are several variants a 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 will be selected based on some heuristic.

At the beginning of each iteration, a client chooses a consultant based on its ***personal preference*** and on the consultant's ***reputation***. The reputation of a consultant increases with the number of successes achieved by its clients. A client achieves a ***success***, if it constructs a solution better than all solutions found until that point by any client guided by the same consultant. Each time a client achieves a success, the consultant adjusts its strategy in order to reflect the sequence of decisions taken by the client. Because the reputation fades over time, a consultant needs that its clients constantly achieve successes, in order to keep its reputation. 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.

The pseudocode that formalizes the CGS metaheuristic is shown in Fig. 5.1. 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 either a solution to the problem (line 13) or a consultant strategy (line 9). A local optimization procedure (line 17) may be applied to improve this solution or consultant strategy.

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).

```
1  procedure CGSMetaheuristic()
2      create the set $\mathcal{P}$ of virtual persons
3      foreach $p \in \mathcal{P}$ do
4          setSabbaticalMode(p)
5      end foreach
6      while (termination condition not met) do
7          foreach $p \in \mathcal{P}$ do
8              if actionMode[p] = sabbatical then
9                  currStrategy[p] $\leftarrow$ constructStrategy(p)
10             else
11                 currCons[p] $\leftarrow$ chooseConsultant(p)
12                 if currCons[p] $\neq$ null then
13                     currSol[p] $\leftarrow$ constructSolution(p, currCons[p])
14                 end if
15             end if
16         end foreach
17         applyLocalOptimization()                    // optional
18         foreach $p \in \mathcal{P}$ do
19             if actionMode[p] = sabbatical then
20                 if currStrategy[p] better than bestStrategy[p] then
21                     bestStrategy[p] $\leftarrow$ currStrategy[p]
22                 end if
23             else
24                 c $\leftarrow$ currCons[p]
25                 if c $\neq$ null and currSol[p] is better than all solutions
26                                 found by a client of c since last sabbatical then
27                         successCount[c] $\leftarrow$ successCount[c] + 1
28                         strategy[c] $\leftarrow$ adjustStrategy(c, currSol[p])
29                 end if
30             end if
31         end foreach
32         updateReputations()
33         updateActionModes()
34     end while
35 end procedure
```
Fig. 5.1. The CGS Metaheuristic.

Reputations are updated based on the ***results*** obtained by clients (line 32): the reputation of a consultant is incremented each time one of its clients achieves a success and it receives an additional bonus when a client obtains a best-so-far result. Each consultant is ranked based on the best result obtained by any client working under its guidance. For a number of top-ranked consultants, CGS prevents their reputations from sinking below a predefined level.

Finally, the action mode of each virtual person is updated (line 33): 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.

5.2. CGS applied to the Traveling Salesman Problem

In this section, we propose a CGS algorithm for the traveling salesman problem, which we refer to as CGS-TSP. 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: with probability a_0 (where a_0 is a constant parameter), 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. A pseudorandom proportional rule based on two constant parameters q_0 and b_0 is used to decide which city to visit at the next step: 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.

Each 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.

In experiments with and without local search, CGS-TSP has been able to outperform the best-performing Ant Colony Optimization algorithms.

5.3. 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. Given a set of facilities and a set of locations, a flow matrix specifying the flows between each pair of facilities and a distance matrix specifying the distances between each pair of locations, the Quadratic Assignment Problem consists in finding an assignment of facilities to locations, which minimizes the sum of the flows multiplied by the corresponding distances.

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 is used to compute the probability to choose a given consultant.

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 pseudorandom proportional: 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 the end of each iteration, the algorithm applies a local search procedure in order to improve the assignments constructed by clients. Similar to other algorithms for the QAP, CGS-QAP can use 2-opt local search, short runs of tabu search or simulated annealing as the local search procedure.

Experimental results show that CGS-QAP performs better than MAX-MIN Ant System (MMAS) for unstructured QAP instances. Our future research will investigate if CGS-QAP is still able to compete with MMAS for structured QAP instances.

5.4. CGS applied to the Generalized Traveling Salesman Problem

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. 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.

The *local-global approach* has been introduced by Pop [27] 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). 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 or by using integer programming. We call *global graph* the graph obtained by replacing all nodes of a cluster with a supernode representing it.

In our proposed algorithm, a client constructs at each iteration 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 a cluster optimization procedure, 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 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.

Experimental results show that our algorithm is able to compete with the best GTSP algorithms.

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. 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.

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) [24]. 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 a set of rules that produce the desired emergent behavior. At present, there is no generally accepted methodology for designing agent-based algorithms. Therefore, the design of such algorithms 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.

AgSysLib is both a framework and a library. It is implemented in Java and it offers an API 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. AgSysLib is able to wrap around many of these packages, but in its default configuration, it integrates the MASON library. MASON [23] 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.

Configuration

Agent-based problem solving requires 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.

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.

Experimentation and tuning

As mentioned before, AgSysLib allows specifying lists of values in the configuration files, in order to run an algorithm with all possible combinations of these parameter values. This way, it is possible to perform a basic form of tuning. In addition, AgSysLib offer a more elaborate tuning procedure, based on genetic algorithms and it also enables an easy integration with software packages for automatic parameter configuration.

AgSysLib offers a number of utility classes that permit the evaluation of mathematical formulas appearing in a configuration file. 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.

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 limited knowledge about what to expect from the algorithm. Because many agent-based algorithms are stochastic, reproducing 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, called the remote control console, which allows connecting via RMI (Remote Method Interface) to a running application and interrogating, watching or changing its internal state. 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.

REFERENCES

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

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

[3] 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.

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

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

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

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

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

[9] 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.

[10] 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.

[11] M. Dorigo and T. Stützle, *Ant Colony Optimization*.: The MIT Press, 2004.

[12] 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.

[13] John H Holland, "Complex adaptive systems," in *A new era in computation*. Cambridge, MA, USA: MIT Press, 1993, pp. 17--30.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Ming Li and Paul Vitányi, *An introduction to Kolmogorov complexity and its applications*, 2nd ed.: Springer-Verlag New York, Inc., 1997.

[23] 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.

[24] C M Macal and M J North, "Agent-based Modeling and Simulation," in *Winter Simulation Conference*, Austin, TX, USA, 2009, pp. 86 -98.

[25] John H Miller and Scott E Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life.*: Princeton University Press, 2007.

[26] 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.

[27] Petrica C. Pop, "The Generalized Minimum Spanning Tree Problem," The Netherlands, PhD thesis 2002.

[28] 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.

[29] Edmund M Ronald, Moshe Sipper, and Mathieu S Capcarrère, "Design, Observation, Surprise! A Test of Emergence," *Artificial Life*, vol. 5, pp. 225-239, 1999.

[30] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[31] Thomas Stützle and Marco Dorigo, "ACO algorithms for the quadratic assignment problem," *New ideas in optimization*, pp. 33-50, 1999.

[32] Thomas Stützle and Holger H Hoos, "MAX-MIN Ant system," *Future Gener. Comput. Syst.*, vol. 16, pp. 889--914, 2000.

[33] G Theraulaz and E Bonabeau, "A brief history of stigmergy," *ARTIFICIAL LIFE*, vol. 5, no. 2, pp. 97-116, 1999.

[34] Michael Wooldridge, *An Introduction to MultiAgent Systems*.: John Wiley & Sons, 2002.