AGENTS: INVESTIGATING ENVIRONMENT PERCEPTION

LUIZ FILIPE POLIMENO ABRAHAO

MSc Project Report

Engineering Department

King's College London

University of London

September 2012 – Draft 2

ABSTRACT

The natural world offer examples in which very limited agents when put together can create very sophisticated overall behaviour. Social insects have been incredibly successful in solving complex problems. This project proposes an agent oriented computational model that simulates certain aspects of social insects living in a colony. A brief introduction is given on emergence and the relevant aspects of distributed complex systems to this project, as well as some background information on social insects and their communication techniques.

Two experiments are proposed and run. The first experiment investigates the relationship between agents and pheromone concentration present in the environment. The second experiment investigates the effect of changing the radius of action of chemical stimuli on the colony's foraging capability.

The resulting conclusions taken from the data collected point out the issues risen by the node selection process implemented. Another two experiments are proposed for further investigation on the node selection as well as possible improvements on the model related to the issues found during the simulations.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ABC	Ant-Based Control
ABM	Agent-based Model
ACO	Ant Colony Optimisation
ACS	Ant Colony System
API	Application Public Interface
BDI	Beliefs Desires Intentions
CLT	Central Limit Theorem
LIFO	Last-In-First-Out
OOP	Object Oriented Programming
UML	Unified Modelling Language

Part I

SETTING THE CONTEXT

In this first part, an introduction on the project is presented in Chapter 1. Social insects and other background information, such as emergence and agents are presented in Chapter 2. Some of the applications of the concepts presented are highlighted at the end of the same chapter. The last chapter in this part, Chapter 3, introduces the generic computational model proposed by this paper and some parts of its implementation.

1

INTRODUCTION

Recent studies have estimated that there are 8.7 million eukaryote species on the planet [45] [43]. Most of them are still to be discovered. Dawkins [16] argues that the ultimate goal of every single one of them is survival, and to achieve that they do extraordinary things to solve the most varied range of challenges that the environment they live in imposes on them. Birds migrate thousands of miles in search of food, fish spend most of their energy swimming up against the stream in order to reach the perfect location to lay their eggs. One could argue that these animals are intelligent, but here are numerous ways to define intelligence. Kennedy et al. [36] is part of the group of people who believe that intelligence is mostly about the capacity to adapt to new situations that have not been foreseen [36], and one shares this same point of view.

We are surrounded by examples of animals with a wide variety of complexity, ranging from simple organisms with limited intelligence to us humans beings. All meet the challenged of surviving in an ever changing environment in different and ingenious ways. It could be argued that humans are the most successful creatures on this planet. It is very easy to blindly accept this statement, for the vast majority of us grow up being told that is the case. Nevertheless, there are different ways to think about 'success'. If survival is all that matters, the number of individuals in a certain species could be used as a good benchmark for success, and in this case humans are losing emphatically to insects [42].

So how can these, as far as we are concerned, 'limited' creatures accomplish very complex tasks? How can they react to the highly dynamic environments they live in?

For example, when worker ants leave the nest to forage in the morning, the environment around the nest can be completely different from what it was last time they went out. Leaves might have fallen, thus changing the route to access food in the rain forest; the wind can blow the sand in the desert, covering food sources. Yet despite these unpredictable factors, the ants face the necessity to locate food, and indeed they do. As a complex decentralised system, the colony agents are able to overcome the disruptions introduced by the environment and keep going on, meeting the challenge of survival. These questions are very hard to answer, and indeed this project has no intention to do so.

Systems of social insects are difficult to study due to the very nature of the structures they create. These structures are decentralised and behaviours emerge from interactions between the system's agents - the individual insects of which there are a great number. Accordingly, understanding of the agents' behaviour in isolation does not guarantee and understanding of the overall picture. Also, small changes in the way agents interact and react to external stimuli may introduce large changes in the resulting overall behaviour; the complexity of these interactions and the amount of possible variables that can affect the system as a whole are so large that we easily fail to appreciate the incredible achievement that these complex systems are by themselves.

This project has as objective to study how agents and the ant colony as a whole react to changes in some properties of the pheromones that the ants use to communicate indirectly to each other. In simple terms, ants use different pheromones to transmit different types of messages. As chemical substances, pheromones interact with the environment differently from one another. As a byproduct, agents are affected in different ways also. For instance, molecules that compose Pheromone A are heavier than the ones that compose Pheromone B. In this case Pheromone B is likely to diffuse through space faster than Pheromone A. How does this area of spread affect the colony's forage capability? The experiment in section 4.3 investigates that.

In order to carry out these experiments, a flexible computational model that enables studies on social multi-agent complex systems is proposed. This model needs to be extensible and flexible enough to give users the freedom to create complex computational simulations. As in nature, the model formalises a decentralised set of entities, taking advantage of the tools computer models provide. The model also formalises an hierarchical infrastructure that can be used to describe different type of systems and track its components throughout simulations in order to acquire data.

In this model each agent is defined by its type and a list of tasks associated with them. The model is inspired in the subsumption architecture proposed by Brooks [7] [8]. Brooks' robots are composed by reactive interconnected modules, in a way that effectively gives different priorities to different modules. In the computational model proposed by this project, agents are composed by tasks, each task is completely independent from one another as far their execution goes. There is nothing from prevening a task to use another task to achieve a desired goal though. Differently from the subsumption architecture, the agents in the proposed model are not purely reactive. They are able to reason what they should do next and then select which task to execute if they are programmed to do so. The users have the freedom to implement the agents as they wish. Simple Beliefs Desires Intentions (BDI) [6] [55] agents can be implemented by extending the abstract infrastructure with minimum effort.

The environment in which these agents are placed is formed by a set of *Nodes*. A node can be thought as a infinitesimal are of the environment. Nodes themselves are composed by a list of agents currently in the node and a list of communication stimuli, which represent any stimulus left by the agents that have been to the node. All simulations run for this paper used environments composed by $250,000$ interconnected nodes, but simulations using up to $750,000$ nodes have also been successfully run.

Agent-based modelling is at the core of the computational model. Even though more formal agent technology, such as communication protocols, is not used by the

computational model, the very fact that agents are autonomous offers the ideal toolset to be used to create decentralised complex systems.

The model is presented in Unified Modelling Language (UML) as class diagrams and activity diagrams and it is implemented using the computer language Java. UML has become the standard modelling language within the Object Oriented Programming (OOP) world as well as it has become a popular popular technique outside the OOP circle [27]. Among other features it provides a set of diagrams that can be used to describe classes, objects, action sequences and other parts of larger systems, such as packages. As far this project is concerned, class diagrams and activity diagrams supply all the tools needed to describe and document the model.

Java is a high level object-oriented programming language. It has been released publicly in 1995 and has developed to a powerful platform since then. Historically, Java has had great success in the enterprise environment, due to its flexibility, robustness and an entire ecosystem created around the core language that allows users to concentrate on the important parts of their work, leaving the language frameworks to deal with low level issues such as transaction management. In recent years, Java has been increasingly adopted by users from the academic background. With the introduction of libraries such as JScience and the development in hardware, it is possible to take advantage of features that are exclusive to Java to facilitate research in a variety of areas.

Most important for this project is the fact that Java offers a comprehensive, high level, shared-memory management and threading facilities. If not coding low level libraries users do not need to use complex synchronisation techniques such as semaphores. The language also offers key words, context blocks and utility libraries ready to be used when multi-threading is required.

In this project each agent is run as an isolated thread in any available CPU. Agents are completely autonomous from each other, and only share the same information about the environment around them, which means that, there is no access to global

information, but only the local context is available to decision making and task execution. The simulations executed for this project use from 10 up to 300 agents. There is no theoretical limit for the number of agents (that is threads) that could be used, only physical restrictions such as memory availability will impose a limit on the number of agents or nodes used in the simulations.

2

BACKGROUND INFORMATION

2.1 SOCIAL INSECTS

Social insects are a great example of how limited agents can tackle very challenging problems when working in unison. Complex behaviour emerges from the interaction between the individual parts of the colony and the environment. They are extremely successful, and Fittkau and Klinge [26] argue that even representing only around 2 percent of the known species, they represent approximately half of the biomass in the central Amazonian rain forest. In his research Erwin [25] has also shown that they constitute 69 percent of all individuals in the canopy of Peruvian rain forest.

This begs the question, how come social insects are so successful? Because the colony is a non-centralised structure, the same task is be done by a large number of individuals that in fact can easily switch from one task to another. Large amounts of individuals can forage a relatively large amount of food when compared to solitary insects. Additionally, the genetic loss in the case of a worker which gets lost during forage or a predator attack is zero to the colony; and in case of serious danger to the colony, its members are capable of deploying coordinated actions that take nest defence to the next level.

Colonies that present reproductive division of labour within the same generation of individuals are called semi-social. Now when there is reproductive division of labour with overlap of generations, the colony fulfils the two requirements for *eusociality*, that is, social organisation with different hierarchical levels.

Arguably, colonies of social insects are one *superorganism*. This idea was first introduced by Wheeler in his essay *"The ant-colony as an organism"* [53]. There are many parallels between organism and superorganism. Organisms have cells and organs, while their counterparts in superorganisms are colony members and castes respectively.

Castes themselves are critical for the division of labour, and therefore for the formation of superorganisms. As Hölldobler and Wilson [34] puts it:

> The superorganism exists in the separate programmed responses of the organisms that compose it. The assembly instructions the organisms follow are the developmental algorithms, which create the castes, together with the behavioural algorithms, which are responsible for moment-to-moment behaviour of the caste members.

These behavioural algorithms are full sets of *decision rules* and *decision points* that define an individual's behaviour, that is, its caste. Each cast has different thresholds for different stimuli associated with tasks, which results in specialised work because individuals of different castes will respond differently to the environment around them. For example, a worker picks up food as soon as it encounters it; on the other hand an ant belonging to the *Soldier* caste will not pick up any food until it comes across too much food that it is impossible to ignore it. Even at this point, if it senses any other stimuli that it has low response threshold to, it is very likely that this ant will drop the food that it has collected and will respond to the stimulus executing a different task.

An ideal division of labour system would have a specialist for each type of task. However, in reality it does not happen [54] [46]. This is because the challenges the environment imposes to the colony require that ants must change from one role to another as fast as possible in order to be efficient.

2.2 COMMUNICATION

Communication is a basic requirement for emergence of complex social systems [48]. Honeybees are known for their extraordinary communication method - they use a vocabulary in which messages are expressed in form of dances [52]. But, in more than 90 percent of cases, social insects use some form of chemical signals when communicating [34]. These chemical signals, pheromones, are laid onto the environment by a different set of glands located throughout the insect's body.

Another very popular communication method is physical contact - in this case there can be a direct communication between the individuals. For instance, honeybees can get hold of their nest mates using their forearms and vibrate their body to transmit information [1]. There can also be indirect communication; that is an individual transmits information just by interacting with other individuals or/and the environment around them, but with no intention to transmit information directly. For example, workers of some types of ants can interpret the encounter of ants of another caste as a signal of danger. There is no direct transmission of information in this case, but it does happen, indirectly.

2.2.1 *Ants*

In order to forage on the ground efficiently, sometimes even underneath its surface, ants rely heavily on the use of chemicals for communicating, but they also make use of tactile signals and vibrations [32]. There are at least 12 functional categories of communication deployed by social insects [34]:

1. Alarm
2. Attraction
3. Recruitment

4. Grooming
5. Trophallaxis, exchange of fluids
6. Exchange of solid food
7. Group effect, induce or inhibit a particular action
8. Recognition
9. Caste determination
10. Reproduction control
11. Territorial and home orientation
12. Sexual communication

A more in-depth discussion is presented by Holldobler and Wilson in *The Ants* [33].

Ants have evolved to a very high level of sophistication as far as chemical communications is concerned. The fire ant (*Solenopsis invicta*) for example, is known to use around 20 different signals to communicate, from which only 2 are not chemical [34] [51].

As chemical structures, pheromones are volatile and have different diffusion rates. How volatile a chemical signal is affects how much agents will interact with that signal before it fades out. The diffusion rate is also of high importance because it affects the size of the *active space*, which is the zone in which the intensity of a chemical signal is above the threshold concentration necessary to trigger action from the agents [34].

Warning signals, for example, should be highly volatile and have a large range, as it guarantees that enough ants will be recruited to fight. However, because it fades out quickly, if the ants that have already been recruited do not lay more pheromone onto the environment, it will disappear fast enough to avoid over-recruitment. Regarding the direction and forage signals for example, it is important that they have a long lasting effect, so that ants can use them as guides to food sources. However if they

were to have a large active zone, they would impede the colony of exploring new areas of the space, 'trapping' the ants only in the area within the signal trails of known food sources.

In addition to chemical signals, ants use a variety of techniques to find their way back to the nest. Forest ants, for example, can memorise the canopy structure above the foraging area [24]. Klotz and Reid [37] have shown that black carpenter ants (*Camponotus pennsylvanicus*) use the moon at night as reference to find their nest, and Graham and Cheng [30] claim that the Australian desert ant makes use of landmarks to that end.

2.3 EMERGENCE

Emergence is a concept that is difficult to define. It is present in many disciplines, such as science, arts and philosophy. It is seen throughout nature in phenomena such as patterns on the sand in the desert and flocks of birds.

A general concept of emergence can be defined as decentralised, local behaviour which, when seen from a higher perspective, aggregates into a global behaviour. As local behaviour is not directly connected to the global behaviour, it does not play any role in the aggregate outcome. Accordingly, the agents, that have local behaviour, do not share a global behaviour as a target.

There are multiple layers of emergence, and this is a crucial concept in understanding complex systems [44]. As an illustration to that, one could use any of most of the multi-cellular animals, including ourselves. The theoretical biologist Kauffman [35] argues that life is an emergent event itself. There are trillions of cells in our body, each of them concerned only with its own very specific context. Many of them are replaced daily. In a couple of years we are very likely to not have any cell that is in our body today, but we will continue to be what we are today, at least physically.

Emergence is not a new concept; it has been around for a long time. A good example of this is the Central Limit Theorem (CLT), which was first postulated in 1733 by the mathematician Abraham de Moivre [50]. In few words the CLT states that, if certain conditions are satisfied, the mean of a large number of independent random variables will be distributed following a normal distribution.

2.3.1 *Types of emergence*

Typically emergence is split into weak and strong. It is possible to say that a phenomenon is strong emergent when it arises from a low-level domain, but the new qualities that these phenomenon bring to the system are irreducible to the system's constituent parts [38]. For an example of strong emergence we can turn to ant colonies. Some ant colonies when defending their queen, recruit workers to create a semisphere of ants around the queen, keeping it safe. The resulting global behaviour cannot be traced back to any individual worker. On the other hand, a weak emergence describes properties that can be reducible to its individual constituents.

2.3.2 *Feedback*

When agents are interacting with each other, and these interactions are not independent, feedback becomes a very important part of complex systems. If the feedback is positive, disturbance on the system gets amplified, leading to instability. A good example of positive feedback can be borrowed from Chemistry. If case a chemical reaction happens faster at higher temperatures, but the reaction itself releases heat, it is very likely a positive feedback loop will be created and the reaction could lead to explosion very quickly.

On the other hand, if the feedback is negative, any disturbance on the system is absorbed, taking the system to a state of stability. There are many examples of negative feedback in our own body, such as secretion of sweat to regulate body temperature and secretion of a variety of hormones in order to regulate water absorption, salt absorption and so forth.

2.3.3 *Decentralised Systems*

Systems that lack a central authority are called decentralised. In their most common form they are self-regulated, they are present in a vast range of domains, from nature to our society. The stock market is one example of such a system. Although there are regulatory instruments in place to avoid abuse, the large number of dealers controls the market as for as the value of shares are concerned. If the case, for some reason, a share is particularly attractive people are likely to buy it. Following the high demand for the share, its price will rise. After a certain point, due its high price, the share will not be as attractive anymore and agents involved in trading will go after other options. In time the demand for the share will get weaker and its price is likely to go down.

Of course this is an oversimplified version of what actually happens, but the important point here is that the agents involved in process of buying and selling the share determine its price themselves. They are autonomous in with respect to making the decision to buy or sell.

Arguably this property is the foundation of emergent systems. The autonomy of the system's components allows complex behaviour to emerge in a way that in centralised system it would not occur, while in cases of centralised systems co-ordination is key. Indeed complex behaviour is capable of emerging in such systems, but only as a byproduct of this co-ordination. Ballerini et al. [3] has shown that a bird, that belongs to a flock, follows the movements of 6 or 7 another birds around it in order to decide how to move.

2.4 AGENT-BASED OBJECT MODELS

Agent-based modelling has proved to be one of the most relevant research areas in computing in the last decade. Nowadays we are overwhelmed by the amount of information available to us, and the improvements on hardware in the last two decades introduced some of the tools to make use of available information in ways that were not possible before. Agent technology enters the scene taking advantage of these improvements and opens up a whole new world for new technologies to be created and put in use, whether for research or in commercial applications. Different domains like biology, game theory, stock market and evolutionary computing are using agents extensively nowadays, from simulation on animal populations [10] to predicting market patterns [2].

An agent can be defined as a computational entity that is autonomous and exhibits flexible behaviour. Agents are also responsible over their own internal state. Usually agents are placed in environments that are dynamic and unpredictable. By flexible behaviour three main aspects are of most importance [55]:

1. Reactive: In most of the applications in which agent technology is deployed, the environment is not static. That is, it changes over time. A reactive system is capable of responding to the changes in the environment in the best way possible in order for the system to continue operating as it was before the changes had been introduced.

2. Pro-active: Reacting only to a dynamic environment most of the times is not enough. Agents have a reason to be, something to achieve, a goal. So it is crucial that agents do not just react to stimuli from the environment, but also take the initiative to achieve their goals.

3. Social: Agents are likely to be deployed in a multi-agent environment. In some cases, there can be some goals that are achieved only if agents cooperate with

each other. Thus, social ability, that is, being capable of interaction with other agents, is vital.

Agent technology provides a variety of standards and tools empowering designers and developers to structure applications around autonomous and communicative components from their concept to their implementation [39].

All in all, agent-oriented modelling offers the best methodology for representing complex dynamic systems.

2.4.1 *Agents and Objects*

It is important to make the distinction between agents and objects. Objects are all about encapsulating state and providing methods to execute operation upon it. Objects do communicate, through messaging or method invocation, but they are passive. Objects and OOP are merely the means to build agent systems. Agent-oriented modelling is a whole new programming paradigm.

2.4.2 *Agents as a Theoretical Tool*

Another way to see Agent-oriented modelling is as a new kind of tool that empowers us to touch questions that are very difficult to be addressed with traditional tools such as mathematical methods; a tool that is particularly suited to deal with complex social systems.

In comparison with traditional tools, computational models are placed at the other side of the spectrum. Traditional tools are static, precise and timeless; computational models are dynamic, flexible and timely. Even more, computational models are flexible enough in a way that we can add complexity to them in order to gain precision.

So it is as if the computational model precision is a variable that can controlled as needed.

At present, it is hard to look at computational models as a scientific tool on the same level as mathematical methods that have been used for centuries to build our knowledge on the phenomena that surround us. But the problems we are committed to tackle today are different from the ones people were working on in the past and Agent-oriented modelling provides a powerful framework, which by its nature is well suited against this new class of complex problems.

Miller and Page [44] presents an in-depth discussion on the contrasts of traditional methods and computational models, as well as a list of the advantages of Agent-oriented modelling over traditional methods.

2.5 JAVA AND CONCURRENT PROGRAMMING

Computational models define entities that are instanced in memory as objects. These entities usually hold state variables that represent a determined state of that object in time. If the state of an object can change over time and this state is somehow shared by more than one thread *synchronisation* is vital. Alternatively, objects that hold state variables can be *immutable*, their state does not change after their creation. In this case *synchronisation* is not necessary.

2.5.1 *Synchronisation*

If a state variable is going to be accessed by more than one thread, it must be protected in a way that all accesses are coordinated. This is necessary to avoid reading invalid or inconsistent states of the shared variable. This is a basic rule that one must follow

when designing objects which are going to be used in shared environments that make use of multiple threads.

The Java language offers many different tools to tackle the problem of publishing objects safely in the case they will be accessed by more than one thread simultaneously. At the core of this toolset is the *synchronized* keyword. It is used to define high-level exclusive locking. Atomic types, standard libraries, such as synchronised lists and *volatile* variables can also be used to implement coordination when accessing shared states at a high-level. The use of low-level technics such as semaphores and custom synchronisers are avoidable in the vast majority of cases. Concurrent programming is at the heart of the Java language and unless the user is creating a library or working in very specialised contexts, all the necessary tools are provided by the language at high-level, making it straight forward to implement very complex concurrent systems.

A very common problem in concurrent systems is the so-called *race conditions*. Goetz and Peierls [29] defines race conditions as follows:

> A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing.

Check-then-act conditions serve as the classic illustration of the problem. Given that thread A checks if a variable X is 0, if that is true, thread A changes the value of that variable to 1. In case of a situation when thread A reads the value of variable X and gets 0 as answer, but a moment after this thread B changes the value of X to -1. As far as thread A is concerned the value of X is still 0, therefore it should be changed to 1, but we know that it is not the case anymore. This problem can lead very complex models to an inconsistent state generating errors that are very difficult to be isolated and corrected.

Thread coordination can solve the problem, and the most common way to synchronise threads access of shared state is by the usage of *intrinsic locks*. They are defined by the *synchronized* keyword and a reference to an object that will be the lock. *In-*

trinsic locks are also called *monitor locks* and they act as *mutual exclusive (mutex)* locks. Therefore a thread might end up holding a lock forever, making other threads to wait indefinitely, creating a *deadlock*.

The fundamental characteristic of intrinsic locks is that they are *reentrant*, meaning that if a thread that already holds a lock and tries to acquire it again it will be successful. In contrast to *pthreads* that grant locks on per-invocation basis [9], Java locks are granted on a per-thread basis. This greatly facilitates the development of object-oriented concurrent systems [29].

2.5.2 *Threads And Task Execution*

In Java terms, threads are the mechanisms that are used to run tasks asynchronously. And it is a common mistake to think that the *Thread* class is the primary abstraction for task execution in the language, but in fact the *Executor* interface is. It is the base of a powerful task execution framework. It is important to note though that the executors follow the producer-consumer pattern.

Java provides the *Executors* factory to create thread pools for task execution. Using thread pools has many advantages over manually managing threads' lifecycle. It is possible to reuse threads to execute more than one task, which minimises the cost of creating and stopping threads, so speeding up task execution. There are four methods provided by the *Executors* factory for creating thread pools. For this project the most important are: *newFixedThreadPool* and *newScheduledThreadPool*. The former create a fixed-sized thread pool, tasks are executed as soon as submitted. When the fixed limit number is reached the tasks have to wait until a thread is available to execute. The latter also allows the creation of fixed-size thread pool, but in this case the pool supports delayed and periodic task execution.

The basic representation of a task in Java is the *Runnable* interface, but tasks implementing *Runnable* are not able to return a value or throw checked exceptions.[29] Now

when the *Callable* interface is a richer abstraction of tasks, they allow the task to return a value and to throw checked exceptions.

Tasks executed by an *Executor* can have various states. As far as this project is concerned, these states are not critical, because all the simulations are run to investigate the colony at a point in time only, so in all the cases, the thread pools will be created to the size of the number of agents necessary to run the simulation.

As the *Future* interface is an abstraction of the state of a task moving forward, it provides useful methods to manage and retrieve results of tasks. Perhaps for social insects, the most popular of them is search optimisation.

So the natural way of executing tasks is to create a thread pool using an *Executor* such as *ExecutorService*, and then submitting tasks to it - any classes that implement *Runnable* or *Callable*. The methods used for task submission are likely to return *Future* objects that represent the task state. These objects can be used for a variety of things, e.g. exception checking.

2.6 APPLICATIONS

The applications of the knowledge gained by researching social insects and agents are present in solutions for a wide range of problems. Perhaps, for social insects, the most popular application is search optimisation. Dorigo [19] in partnership with Colorni et al. [14, 15, 20] introduced the idea for ants in particular. Dorigo and Gambardella [22, 23] execute a comparison analysis on the application of Ant Colony System (ACS) against other nature-inspired methods as well as to more traditional algorithms for the traveling salesman problem.

Another example of application of Ant Colony Optimisation (ACO) is routing in telecommunication networks. Schoonderwoerd et al. [47] introduced Ant-Based Control (ABC) for telephone networks. Further development on ABC has been proposed by Heusse et al. [31] and Subramanian et al. [49]. Another algorithm for routing has

been proposed by Caro and Dorigo [12, 11, 18], in contrast to ABC these new algorithm, called *AntNet*, can be applied to networks that are either connection-oriented or connectionless, such as packet-switching networks.

Based on corpse clustering and larval sorting Deneubourg et al. [17] has proposed two clustering models. Lumer and Faieta [40] have extended Deneubourg et al.'s work to apply it to data analysis.

Some lessons learned from social insects on self-assembly have been applied on the creation of self-assembling robots. Fukuda et al. [28] was the first person to propose the idea of self-assembly to the robotic field [5]. *Metamorphic* robots [13] are one of the main research lines on self-assembly robotics. Although this area of study is not directly inspired by social insects, they share many common features such as the simplicity of the agents and access only to local information.

More recently Dorigo [21] has experimented with self-assembling robots against tasks that require physical cooperation and coordination. An overview of the *Swarm-bot* [21] is presented by Mathews et al. [41].

3

MODEL OVERVIEW

The model's objective is to describe features present on natural systems in a flexible and robust way. By flexible this means the capability of being easily extendable to a wide variety of applications. By robust, this means meant that the model implementation must be always consistent. If the same simulation is carried many times using the same parameters, one would expect to have the same results for every run. Bonabeau et al. [5] describe the task of modelling as:

> Modelling is very different from designing an artificial system, because in modelling one tries to uncover what actually happens in the natural system - here an insect colony. Not only should a model reproduce some features of the natural system it is supposed to describe, but its formulation should also be consistent with what is known about the considered natural system: parameters cannot take arbitrary values, and the mechanisms and structures of the model must have some biological plausibility. Furthermore, the model should make testable predictions, and ideally all variables and parameters should be accessible to experiment.

The proposed computational model includes all the proprieties described above.

3.1 ENVIRONMENT

The computational model describes the environment as a set of connected nodes, objects that implement the *Node* interface. These nodes can be connected as the user requeires, creating one, two or three dimensional environments. Agents are capable

to navigate from nodes to their neighbours, read communication stimuli deposited in the nodes by another agents and alter the environment themselves by depositing communication stimuli if they require.

As we will see the *Node* interface does not formalise how the nodes are connected, so users have the freedom to create the environments that suit their experiments. The standard implementation uses 2 dimensional grids, with each node connected to a maximum of 4 neighbours, one in each of the directions listed in the enumeration *Direction*.

3.1.1 *Nodes*

The fundamental entity to represent the environment is the *Node* interface. A node can be seen as an infinitesimal piece of the environment. By linking nodes together it is possible to create complex network of objects that will describe the space where agents can navigate.

The *Node* interface does not specify how these connections need to be made nor how many neighbours each node has. This gives a great degree of freedom to users. Hence, it is easy to declare different types of nodes, that can describe different types of environment. For example, the *BasicNode* class is used to describe two dimensional environments, as we will see later on. A three dimensional environment could be easily described as well, by implementing the *Node* interface with 6 neighbours, one in each direction in space - north, south, east, west, above and below. Agents would be able to navigate through nodes in three different axis. For more details on the methods that operate upon the node's neighbours, please see Listing 1 in Section B.2.1.

Each node has an unique string that is used to identify it. The method *String getId()* returns this string. Note that the *Node* interface does not have any instrument to guarantee that nodes have unique identifiers, it is up to the users of the interface to guarantee that nodes do not have duplicated identifiers. Failing to do so may cause the

framework to operate inconsistently, for there would no way to distinguish nodes that have the same identifier.

Nodes also have a list of agents (see section 3.2), being the list of agents that are currently in the node. The node interface provides the necessary methods to operate upon these agents:

- void addAgent(Agent): Verify if the agent can be added to the node, if yes, adds the agent to the node, if not ignores.
- List<Agent> getAgents(): Return the list of agents currently in the node
- void addAgentStartingHere(Agent): This method is used to place agents into the environment. *addAgent(Agent)* method might have to check for a few conditions before allowing the agent into the node. Agents that are starting their lifecycle could fail to pass this condition. So this method is provided, and it does nothing else but add the agent to the node.

For the agents list is published, that is, it is accessible to many threads, any implementation of the *Node* interface must be thread-safe.

Agents are able to communicate with each other indirectly through the use of communication stimuli (see subsection 3.1.2) that they add, or manipulate existing ones in their environment.

The *Node* interface specifies the following methods to manipulate communication stimuli:

- List<CommunicationStimulus> getCommunicationStimuli(): returns a list of all communication stimulis present in the node
- void addCommunicationStimulus(CommunicationStimulus): add a new communication stimulus to the node.

- CommunicationStimulus getCommunicationStimulus(CommunicationStimulus Type): returns a communication stimulus of the type specified, if one is present in the list.

Each communication stimulus has a type. The type determines which proprieties are present for a stimulus. The *Node* interface enforces no limitation on having more than one stimulus of the same type in the communication stimulus list. There will be cases that make no sense in having two stimuli of the same type in the list. For example, imagine that we have two ants that deposit chemical substances (pheromones) in the environment to communicate to each other. Let say the first ant deposit 0.1 and the second ant another 0.1 of the same pheromone. As far the other ants are concerned the total of that pheromone in that node is 0.2, it does not matter to know when they were deposited and by which ant. In this case it makes sense to have only one type of communication stimulus in the list to represent that type of pheromone, each ant can update that stimulus if necessary. That is exactly what the *ChemicalStimulusType* class (see subsection 3.1.4.1) and its implementations do. There could be cases that having two stimulus of the same type in the list is valid, for instance when a stimulus expires after some time or when it is necessary to know the agent that issued a particular stimulus.

In case more than one communication stimulus of the same type is allowed in the list, the *Node* interface does not specify which one should be returned when the method *getCommunicationStimulus(CommunicationStimulusType)* is called. It is up to the user to decide how to handle the request.

3.1.1.1 *BasicNode Class*

The *BasicNode* class is the reference implementation of the *Node* interface for a 2 dimensional space, with nodes connected to 4 neighbours.

It is a pseudo-thread-safe class. The methods *getNeighbour(Direction)*, *getNeighbour (Direction, Node)* and *setNeighbours(Direction, Node)* do not make use of synchronisa-

tion mechanisms that would guarantee thread-safety. The reason for that is that these methods are exhaustively used during simulations, and the lack of synchronisation mechanisms here remove any overhead introduced by them. It is safe to not use synchronisation in these methods as long as the environment does not change during the simulations. Note that this is documented in the code by the use of the annotations *@PseudoThreadSafe* and *@ThreadSafetyBreaker*, see listings in section B.2.4 for more details.

Each basic node can have a maximum of four neighbours, one in each direction defined by the Direction enumeration: North, East, South and West.

The *BasicNode* implementation of *getCommunicationStimulus* method returns the first stimulus of the requested type found in the list. If none is found, *null* is returned.

3.1.1.2 *PheromoneNode Class*

The *PheromoneNode* class is a specialisation of the *BasicNode*, exclusive for ant simulations. In addition to everything present in the super class, it declares two new methods that come in handy when dealing with chemical communication stimuli (see subsection 3.1.4.1):

- ChemicalCommStimulus getCommunicationStimulus(ChemicalCommStimulus Type): This method returns a chemical communication stimulus of the requested type if any is present in the stimuli list. If there are more than one stimulus of the same type, the first one found is returned. If none is present, it creates a new chemical stimulus of that type, adds it to the stimuli list and returns the newly created stimulus.

- void increaseStimulusIntensity (ChemicalCommStimulus, double): Increment the intensity of the chemical stimulus of the requested type. If no stimulus of the type is present, it adds one to the list and increment by the specified amount.

The first method is just an overload of the generic *getCommunicationStimulust (CommunicationStimulustType)* method from *Node* interface. Clients do not necessarily need to use this method when creating ants experiments, it is just a shortcut to be used in order to avoid casting.

The same is valid for the second method. Agents could increase a stimulus intensity directly, but by using the *increaseStimulusIntensity* method clients take advantage of existing infrastructure, simplifying their code.

3.1.2 *Communication*

3.1.3 *Communication Stimulus*

Agents have their own lifecycle, their own intentions and goals. Communication is a vital part of the process of achieving their goals, for in most cases agents cannot deal with all challenges imposed by the problems they are trying to solve by themselves.

There are plenty of examples that illustrate that around us. If someone is going to build a house, they most certainly will need help do be able to do it. Not only physical help, they will need another people's skills. Animals that live in packs, such as african lions and arctic whales, are also good examples. They are able to confront challenges in which it would be possible to be done if not in group.

The *CommunicationStimulus* interface is an abstraction of any communication interaction. Implementations of the interface are required to implement only one public method:

- CommunicationStimulusType getType(): Returns what type of communication stimulus the object is.

The class *BasicCommunicationStimulus* offer a simple reference implementation of the *CommunicationStimulus* interface and is the perfect class to extend when creating complex communication stimuli.

3.1.4 *Communication Stimulus Type*

We have many ways to communicate, we can talk to other people, we can write them an e-mail or use sign language to transmit any information we find useful to transmit. Everyone knows that a dog barks, what information exactly it is transmitting we are not able to decode yet, and maybe we never will. Anyway some information is being transmitted in the process. Dogs also use another way to communicate. They urinate in order to mark territory.

The *CommunicationStimulusType* interface is an high level abstraction of a specific way to communicate. As such it formalises only one public method:

- String getName(): Returns the name of the type of communication stimulus.

By extending this interface users can formalise different types of communication. Adding extra parameters necessary to their specific communication types.

Implementations of *CommunicationStimulusType*, and any of its specialisations, must be singletons in order to save computational resources.

3.1.4.1 *Chemical Communication Stimulus*

As seen in section 2.2, ants use pheromones as a way to communicate indirectly to fellow ants from the same nest. The interface *ChemicalCommStimulusType* extends *CommunicationStimulusType* to formalise pheromone as a type of communication. It declares two public methods:

- double getDecayFactor(): Returns the decay factor of the chemical type.
- int getRadius(): Returns the radius of action of the pheromone.

The class *ChemicalCommStimulus* is the base implementation for using this new communication type. It adds a new propriety to the communication: *intensity*. This new field is a double that represents the intensity of the pheromone in that communication stimulus. The class is thread-safe, and *intensity* is guarded by the class.

The key methods of the class are:

- void synchronised increaseIntensity(double): Increases the intensity of the chemical stimulus by the amount passed as parameter.
- void synchronised decayIntensity(): Decays the intensity of the stimulus depending on its type.

It is important to note that *ChemicalCommStimulus* limits the maximum amount of intensity to 1.0. If an agent tries to increment a stimulus that is already 1.0 in intensity, the request is ignored. Another noteworthy point is how *ChemicalCommStimulus* implements the stimulus intensity decay. It uses the following:

$$\mathtt{intensity}_{new} = \mathtt{intensity}_{\mathrm{current}} * (1 - \mathtt{decayFactor}) \tag{1}$$

The *decayFactor* parameter above comes from the chemical communication type declared by the users. See section 3.1.4.2 for an example of a type implementation.

The *int getRadius()* method retrieves how many neighbour nodes are actually affected when agents deposit the particular chemical type. It is analogue to the *active space* seen in section 2.2. A details explanation how the *radius* property affects how pheromone is deposited onto the environment see section 3.2.4.

Equation 1 represents an exponential decay mechanism that models how intensity would decay in reality.

3.1.4.2 *Forage Communication Stimulus*

This communications stimulus type is used by ants when foraging. It is deposited either when the ants are searching for food or going back to the nest carrying food. The amount deposited should be higher if ants are carrying food and going back to their nest. This is a form of positive feedback (see 2.3.2), since it reinforces 'success'.

One should expect for this type of pheromone to have a small decay factor, as it needs to have a long lasting effect on the agents in order to them to be able to collect food from sources far from the nest. In addition it should have a short action radius, for large radius would 'confuse' agents within the trail.

The enumeration *ForageStimulusType* is an extension of *ChemicalCommStimulusType* and declares this stimulus type.

3.1.4.3 *Warning Communication Stimulus*

WarningStimulusType is also an extension of *ChemicalCommStimulusType*, but it represents a very different type of information. It is an analogy to the pheromone laid by soldier ants to warn their fellows of any danger around. Different types of agents react differently to the same communication stimulus (see 3.2.2), but one would expect this type of stimulus to have a high decay factor and a large radius of action. The former will avoid too many soldiers being recruited to attack a predator for instance. The latter would allow the stimulus to reach a larger number of ants around the danger faster.

3.1.5 *Environment Package*

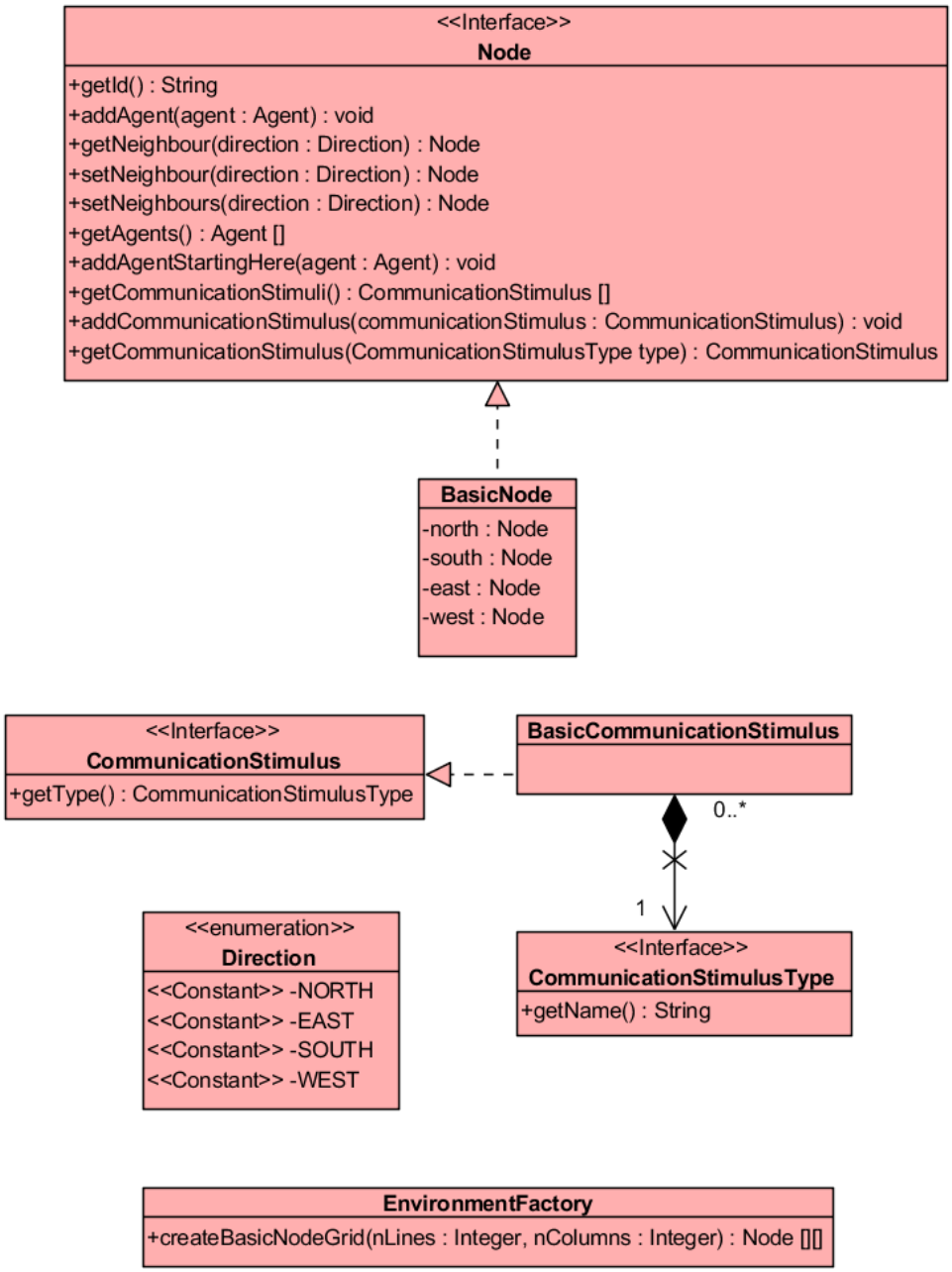


Figure 1: Generic model of environment package

3.2 AGENTS

To enable the agents defined by the model to enjoy the properties that define agents in general, it is critical that they have their own lifecycle. The most flexible way to

achieve that is if each agent is run in an isolated thread in one of the available CPUs of the machine running the simulation. This is guaranteed by the *AbstractAgent* which implements the *Callable* interface (see section 2.5.2 for more details).

The abstraction of any agent is the *Agent* interface though. It formalises the basic Application Public Interface (API) that is available to any agent in the model. The most relevant are:

- String getId(): Each agent must have an unique identifier string to be used for identification. This method returns it. Note that is up to the user to make sure this string is unique.
- AgentType getAgentType(): Returns the type associated with the agent.
- void setCurrentNode(Node): Places the agent in a specific node of the environment.

There are another methods that are related to tracking the nodes the agent has been in the environment during the simulation, more information on these methods can be found in the appendix B.

The *AbstractAgent* is the base abstract implementation of the *Agent* interface. It implements all the public methods declared by the interface, plus declares the implementation of the *Callable* interface, returning the type *Void*, which is a convention to tell the compiler that no value is expected to be returned. By being abstract and declaring the implementation of *Callable*, the class effectively forces any concrete specialisation class to implement the methods that define a task in Java, therefore allowing any agent to be executed in an isolated thread.

3.2.1 *Task Agents*

The smallest unit of work in the computational model is a task, and they are the means to agents to get things done. Please note that this task is different from a Java task discussed in section 2.5.2.

In order to enable agents to execute tasks the concrete specialisation of the *AbstractAgent* class, the *TaskAgent* class has a synchronised field called *currentTask* that contains a reference to the task that the agent is executing at the moment. The second difference from the super class, is that task agents have a different *type* associated to them, a *TaskAgentType*. This is important because it is in the *TaskAgentType* that the tasks that an agent is capable of executing are defined, see section 3.2.2.1 for more details.

3.2.2 *Agent Types*

In nature, individuals use different ways to differentiate themselves from other individuals; in the proposed computational model each agent belongs to a specific type, something that define them as a class. The *AgentType* interface formalises the most basic abstraction of a type of agents. It has only one method:

- String getName(): Returns the name of that type, must be unique.

As in the previous cases, the interface does not have any mechanism to make sure a type has a unique name, it is up to the user to make sure that is the case.

An implementation of *AgentType* provides the means for agents to distinguish themselves. If necessary, one agent can check another agentâs type when they are in the same node. This is very important, because one agent can react differently if it comes across another agent of the same type or if it encounters an agent that might impose some sort of danger.

In the sake of performance, agent types must implement the Singleton Pattern, which guarantees only one object instance of a type only [4]. If agent types were not to be singletons, every time a new agent is instanced a new agent type and the other objects that compose it, like tasks, are going to be created as well, consuming large amounts of memory.

3.2.2.1 *The TaskAgentType type*

The *TaskAgent* interface is simple and has only one method:

- List<Task> getTasks(): Returns a list of tasks that the agents of this type are able to execute.

The enumeration *BasicTaskAgentType* is a reference implementation of the interface. Agents of that type are able to execute only one task, *WandererTask*, see section 3.3.1 for more details on the task.

3.2.2.2 *FoodSource Type*

Food of sources are modelled as agents, and the *FoodSourceAgentType* interface formalises their type. The *FoodSourceAgent* class is the concrete implementation of the *AbstractAgent* class which defines a guarded field named *foodAmount*. This field indicates how much food is present in a food source.

The method *collectFood(double amountToCollect)* is called by agents when requesting to collect food. The method returns the amount of food the source was able to deliver to the agent.

3.2.3 *The Ant Interface*

The ant implementation of the model starts from the *Ant* interface. It declares the public API for any agent that is an ant. The most relevant methods are:

- Direction getMovingDirection(): Returns what direction the ant is moving in relation to the environment grid.
- void incrementStimulusIntensity(ChemicalStimulusType): Lay pheromone onto the current node that the agent is at.
- double collectFood(Agent, double): Tries to collect the specified amount of food from the food source specified as parameter, returns the amount of food that the agent was able to collect.
- Boolean isCarryingFood(): Returns true if the agent is carrying food, false otherwise.
- FoodSourceAgent findFoodSource(): Checks if any agent in the current node is a food source, if any is found it is returned otherwise *null* is returned.
- void invertDirection(): Inverts the direction the agent is traveling, for example, if the agent is traveling East, after this method is called the agent's moving direction will be West.
- void depositFood(AntNestAgent): Deposit the food the agent is carrying into the nest passed as parameter.

For the full API and more detail explanations of all the methods, please see Listing 18 in Section B.2.2.

3.2.4 *Ant Agents*

The *AntAgent* class offer an implementation of the *Ant* interface. Mostly important it has three private methods that are used by *incrementStimulusIntensity (ChemicalCommStimulusType)* for laying pheromone onto the environment. As seen in section 3.1.4.1 chemical communication stimulus types have a propriety called *radius* that defines

the reach of that particular stimulus around the node where it was deposited, these three methods use this information when depositing pheromone. The intensity of the stimulus decreases with the distance away from the node where the agent is:

$$intensity_{new} = intensity_{current} + increase * (1/distance) \tag{2}$$

Figure 2 shows how the intensity of the deposited amount of chemical stimulus varies according to the distance to the node the agent is executing the increment. The amount of stimulus that each agent increments (the *increase* parameter from Equation 2) is defined by the agent type, see section 3.2.2 for a detailed explanation. Note that the *distance* parameter is always greater or equal to 1, nodes next to the agent's current node have distance equal to 1.

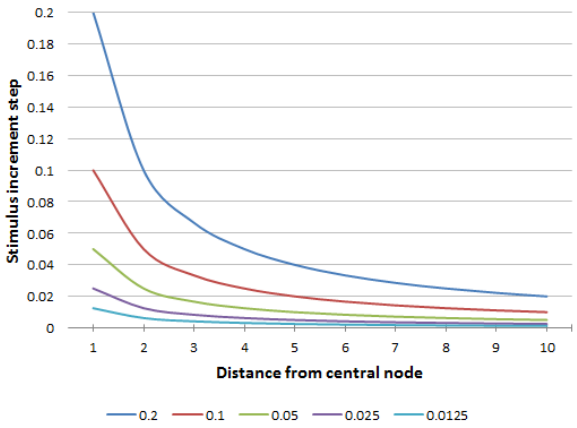


Figure 2: Variation of increment according to distance to central node

Figure 3 illustrates how the pheromone is laid depending on the radius of the stimulus. The strong the increment is the brighter the red. Each node is represented by a single square. In the image three different radius are demonstrated, form left to right we have 2, 4 and 5 respectively. It is clear that he increment falls as the distance from the central node gets bigger.

Figure 3: Pheromone deposit for stimulus with different radius sizes

3.2.5 *Ant types*

The interface *AntType* and its implementations are used to represent any specific type of ant in the computational model. The interface declares the following methods:

- void execute(Agent): Method executed when the agent is executed by the thread pool.
- double getStimulusIncrement(String): Returns the amount of stimulus the agent is able to lay.
- int getMemorySize(): The number nodes the agent can remember it has been.
- double getAmountOfFoodCapableToCollect(): Returns the amount of food the agent is capable of caring

By far the most important method in the list is *void execute(Agent)*, because this method is the method that defines the agent's behaviour. It is in this method the agent can analyse the environment state around it and decide which tasks to execute.

As seen, different type of ants are able to deposit different types of pheromone and in different quantities. So every type of ant has a map that tells by name, what is the increment for each stimulus the ant is able to deposit. the method *double getStimulus-Increment(String)* is used to query these amounts.

3.2.5.1 *Ant Agent's memory*

The ant's memory is implemented as a *linked list*, and it is used in a Last-In-First-Out (LIFO) fashion. As we have seen above, each agent type has the public method *getMemorySize()*. This method returns the amount of nodes the agent is capable of remembering being in.

Just before moving to the a new node the agent can be requested to *push* to its memory the node it is in at the moment. If the number of nodes it has already in its 'memory' is greater than the maximum it is supposed to have as a maximum, it *removes* the newst node from its 'memory'.

3.2.5.2 *WorkerType*

The enumeration *WorkerType* defines the type of ant that represent workers in a colony. The type is capable of executing three tasks, *ForageTask*, *FindHomeTask* and *FindAndHideInNestTask*. Ants of this type are sensitive can lay *ForageStimulusType* and low quantities of *WarningStimulusType*.

The goal of this type of ant is to explore the environment, find food, collect as much as possible and take it back to the nest. Figure 4 fully describes the algorithm implemented by the *WorkerType*.

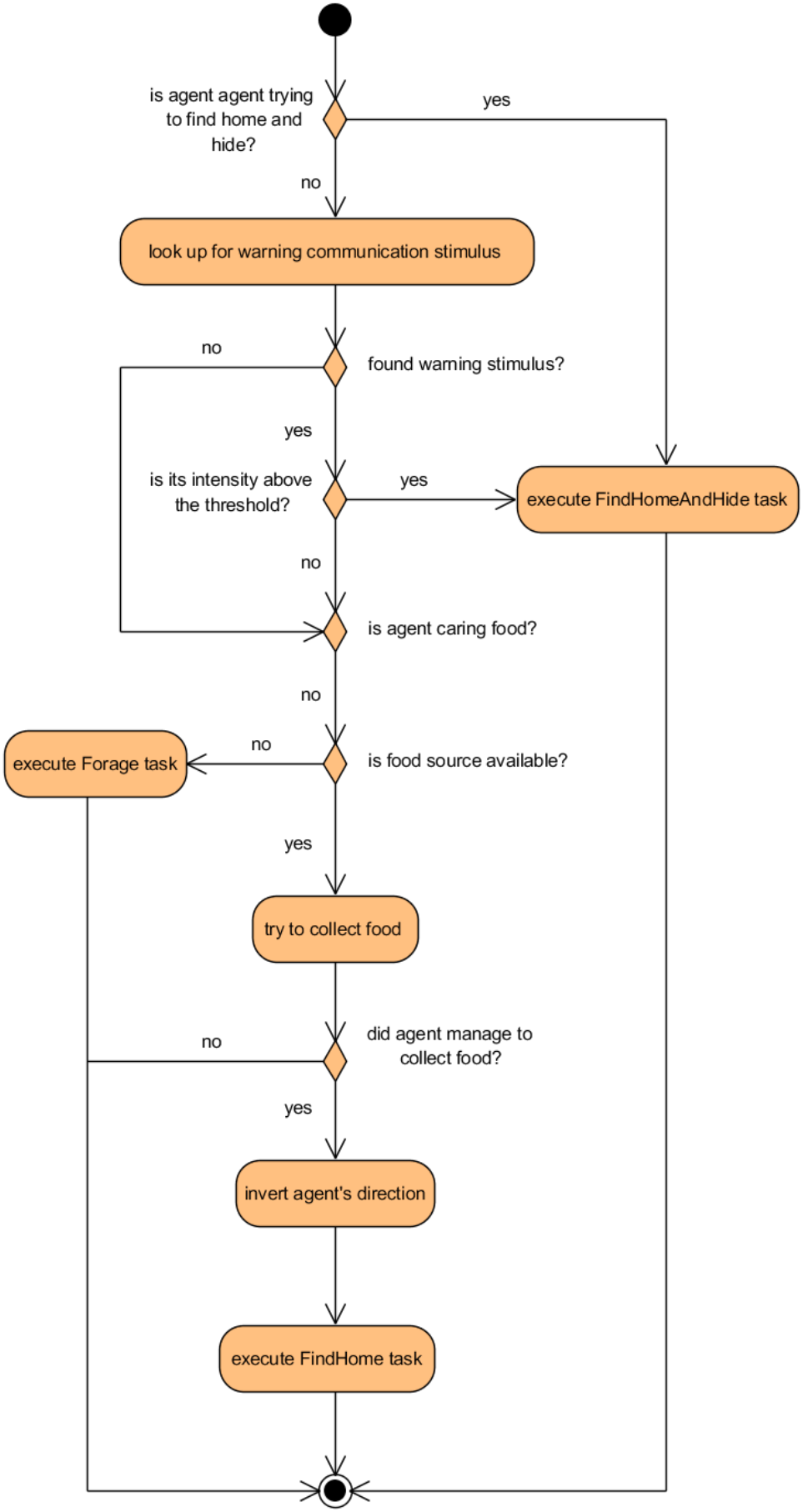


Figure 4: Worker Type ant algorithm

3.2.5.3 *Ant Nests*

The enumeration *AntNestType* declares a type of agent that represents an ant nest. It implements the base *AgentType* interface, so as one would expect, agents of this type are not able to execute tasks.

The class *AntNestAgent* is an specialisation of *AbstractAgent*, it has an extra synchronised field that works as a counter of how much food is available in the nest, *amountOfFoodHeld*. It also declares a public method called *void addPortionOfFood(AntAgent, double)* which allows ants to deposit food in the nest.

3.2.6 *Pheromone Decay and the StaticPheromoneUpdaterAgent*

As a decentralised system, the model lacks a central authority that could handle the chemical stimulus decay question. As seen in section 2.2.1, pheromones are volatile. Section 3.1.4.1 explains the pheromone decay rule implemented by the model.

However, to decay a particular stimulus intensity in a node, the method *decayIntensity()* from *ChemicalCommStimulus* has to be triggered. With no central authority to trigger this method for every node present in the environment, two alternative ways could be deployed; make each update itself or create an agent that would navigate through the entire environment, triggering the decay method for every node.

The first option is more attractive initially, but after some consideration, for practical reasons it would not be viable. To enable nodes to update themselves they would have to effectively be tasks run by threads in the same as the agents are. With environments normally containing around $250,000$ nodes, it is obvious that this is not an option at all.

The second option is the creation of a specialised agent to trigger the necessary method to execute the stimuli update. *StaticPheromoneUpdaterAgent* does exactly that. Its implementation also allows us to divide the environment in slices and have an

agent to update each of these slices, improving performance in large environments. This technique is used in the experiment in section 4.3.

3.2.7 *Agents Package*

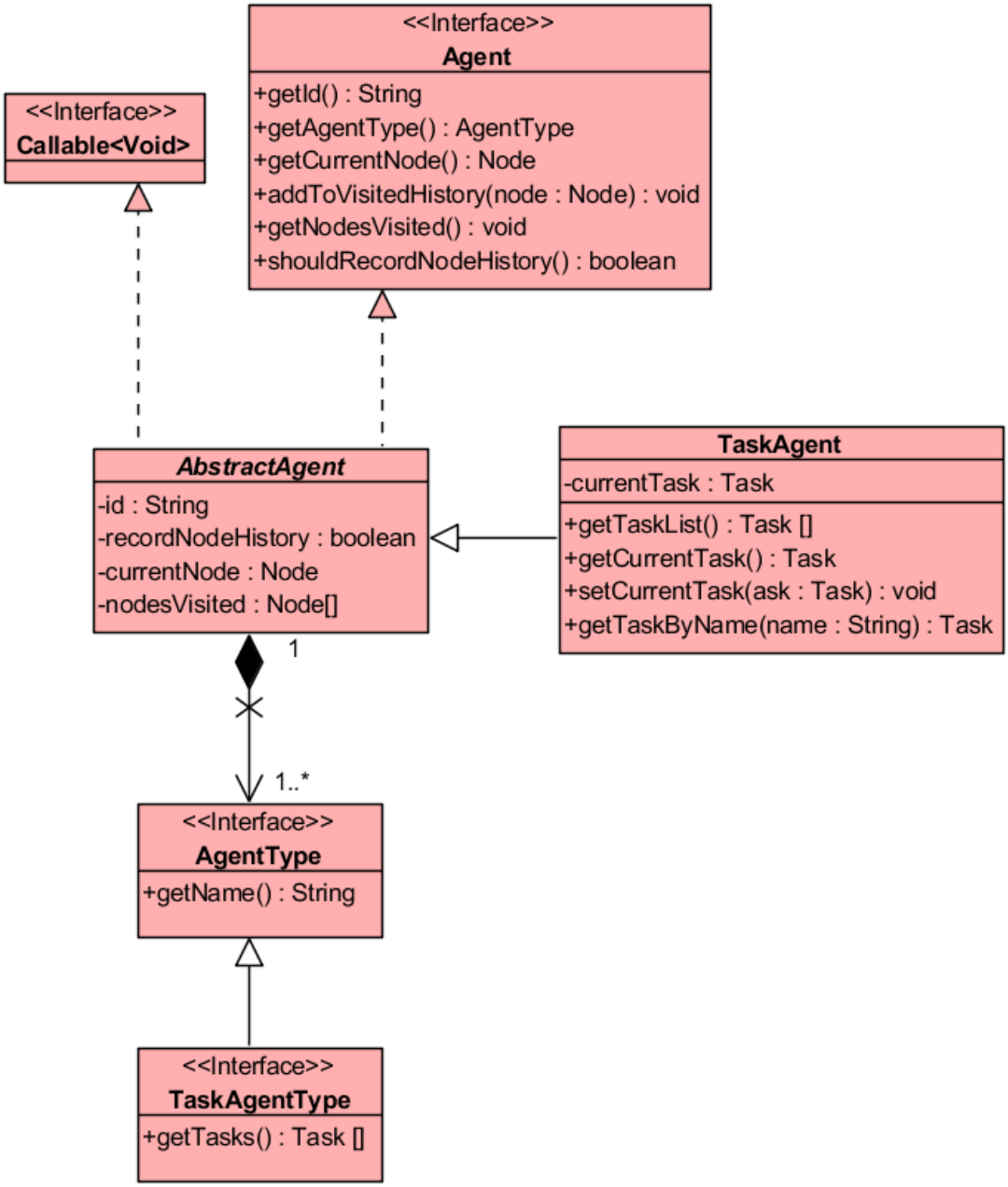


Figure 5: Generic model of the agent package

3.3 TASKS

Tasks are a unit of specialised work. They are the means by which agents do something they need to do. Different type of agents can share the same task, but it is up to each of them how to use it.

The *Task* interface formalises two methods:

- String getName(): Returns the unique name that identifies the task.
- void execute(Agent): Runs the task algorithm for the agent used as parameter.

Tasks are a convenient way to process unit of work that are isolated one from another and done by more than one type of agent. The *void execute(Agent)* method contains the actions that the task is to perform, the agent that is requesting the task execution is passed as a parameter so the task is able to access the agent's context, such as neighbours and communication stimuli. The *AbstractTask* class provides the base implementation for concrete tasks.

3.3.1 *Wanderer Task*

This task is a simple task that agents can use when they are exploring the environment, it causes the agent to move from the current node to one of the available neighbours. It selects which neighbour to move to completely at random.

In the case of ants, for example, they use this task when looking for a new source of food or when they reach one of the environment boundaries and need to switch their moving direction.

3.3.2 *Ant Tasks*

Ants execute tasks that implement the *AntTask* interface. For the list of methods of the interface, please see Listing 29 in section B.2.3.

3.3.2.1 *Forage Task*

The *ForageTask* is also used by agents to find food sources and collect food. Differently from the *WandererTask*, it does not pick a node to move to at random, it follows the forage chemical stimulus trail.

Although the pheromone trail is the most important part when the agents are deciding where to go next, another effects should be considered, for instance an agent could be taken away from the trail by the wind or if some information, like chemical landmarks, that the ants use to locate themselves is modified, should somehow be considered. It is virtually impossible to model all these cases though, more, it would be wrong to try to model all the possible interferences in the colony.

The agents present a stochastic behaviour as far as moving through the space is concerned. The process of choosing the node to move to consists of:

1. Each neighbour node is associated with a weight.
2. The weights are multiplied by the actual pheromone trail found in each of the neighbour nodes.
3. The results of each multiplication in the previous step are summed up to a total.
4. Each multiplication of step 2 is divided by the total, in order to find their ratio.
5. A Monte Carlo like selection on the neighbour nodes is executed using these rates.

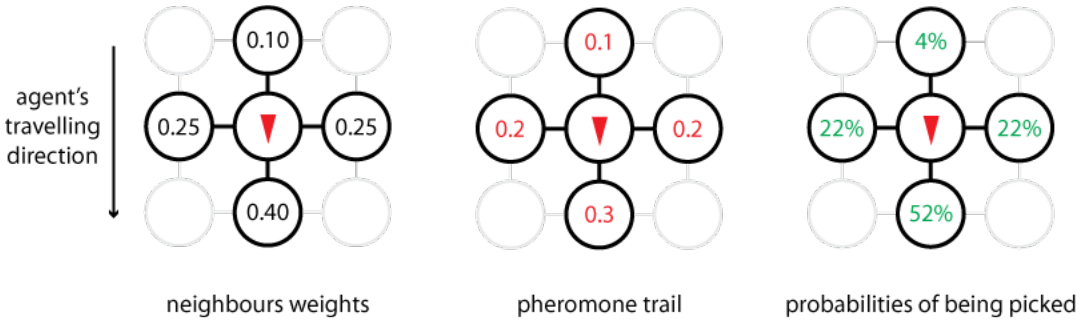


Figure 6: Elements used to select next node to move to

Figure 6 illustrates these steps. It is important to understand that when applying the weights to the neighbours the task uses the direction of travel of agent as reference not the grid. As an example, if we imagine a agent with the moving direction set to East in relation to the grid, so the east neighbour from the node that the agent sits is pointing to the north direction of the agent. Figure 7 illustrates that.

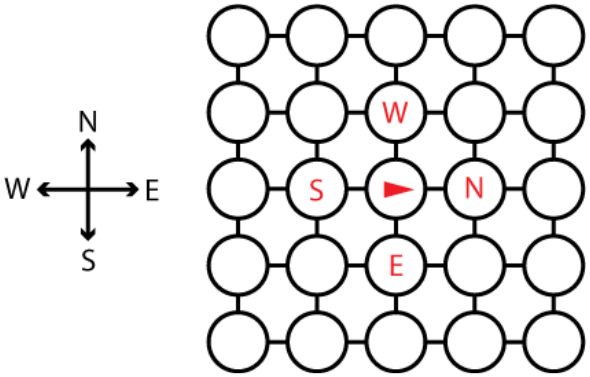


Figure 7: Reference of directions

It is necessary to know what direction the agent is travelling to and perform this transformation between the grid direction reference to the agent's in order to apply the right selection weights to the neighbour nodes.

This method of selection is implemented by the utility class *AntTaskUtil* and is used in other tasks as well. The method is flexible enough to allow tasks to set their own weights for the neighbour nodes and ask any type of chemical stimulus to be used in the calculation.

Figure 8 is the active diagram of the algorithm implemented by the *void execute(Agent)* method of the *ForageTask*.

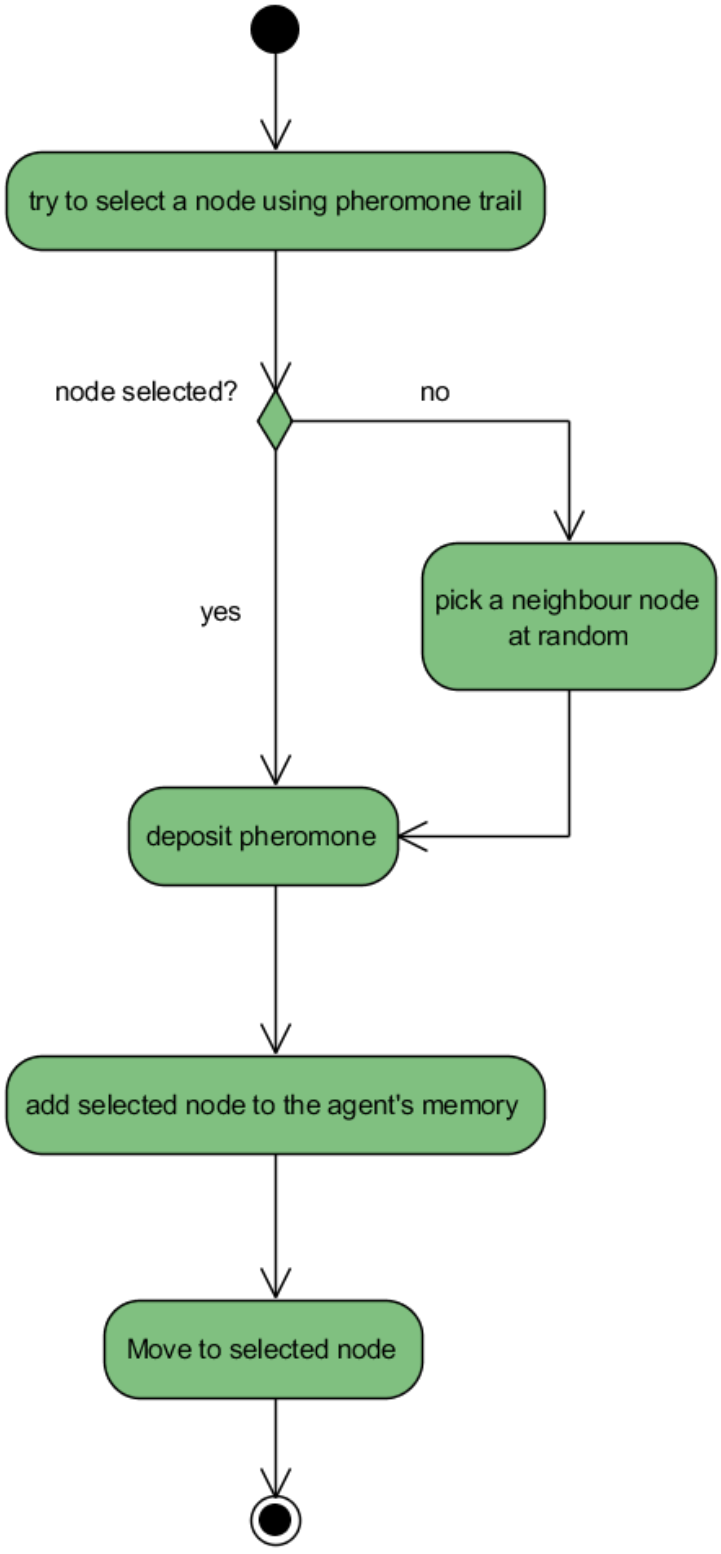


Figure 8: Forage task execution activity diagram

3.3.2.2 *Find Home Task*

The *FindHomeTask* is useful for ants that have collected food and need to deposit it in their nest. It uses the same procedure that *ForageTask* to select which node to move next. The biggest difference is that it tries to use the agent's memory before following the specified pheromone trail. The task's algorithm is illustrated in Figure 9.

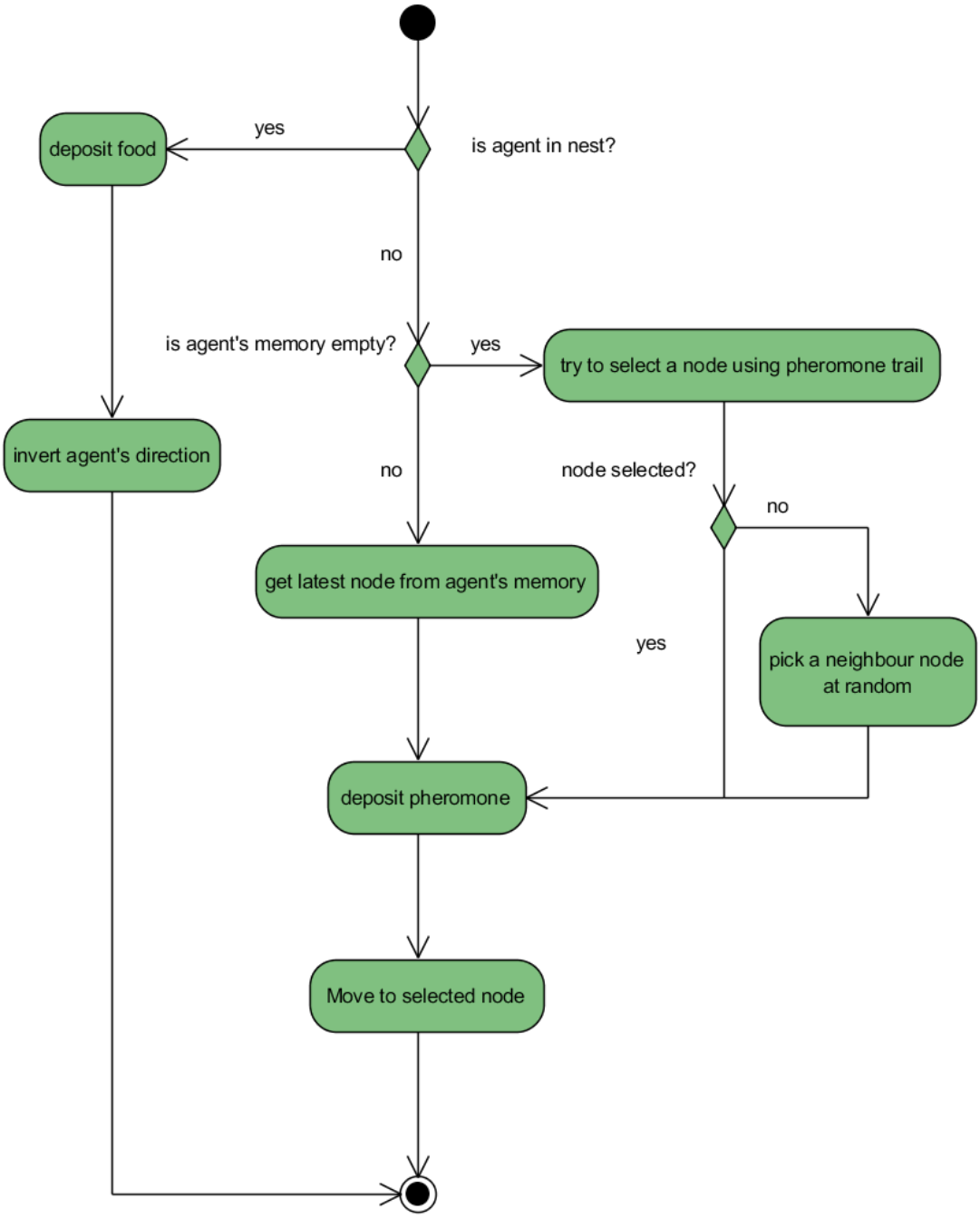


Figure 9: Find home task execution activity diagram

3.3.2.3 *Find Home And Hide*

The *FindHomeAndHideTask* is very similar to *FindHomeTask* the only difference is that if the agent is in a node that contains a nest the task does not try to do anything else, it leaves the agent there.

It is useful for ants that have collected food and need to deposit it in their nest. It uses the same procedure that *ForageTask* to select which node to move next. The biggest difference is that it tries to use the agent's memory before following the specified pheromone trail. The task's algorithm is illustrated in Figure 10.

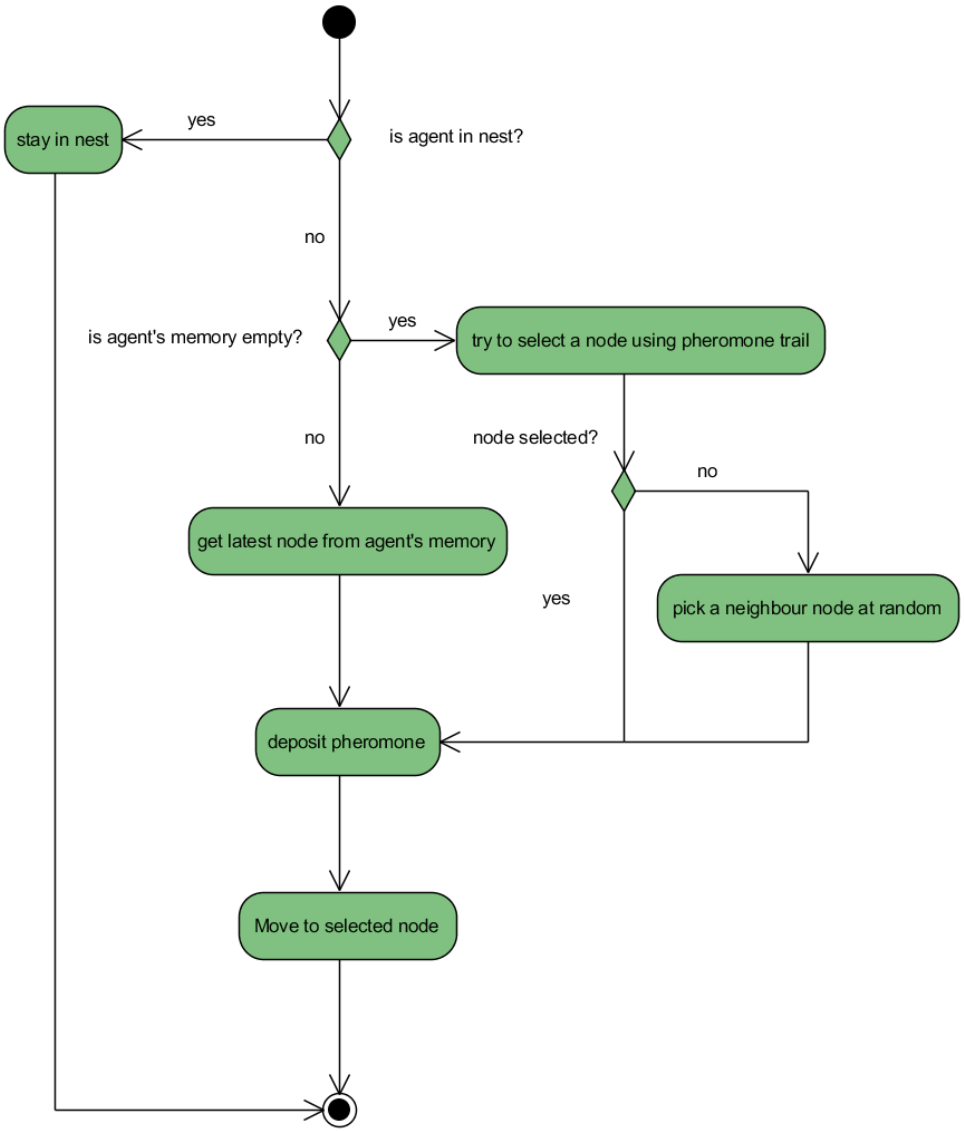


Figure 10: Find home and hide task execution activity diagram

3.4 GENERIC COMPUTATIONAL MODEL DIAGRAM

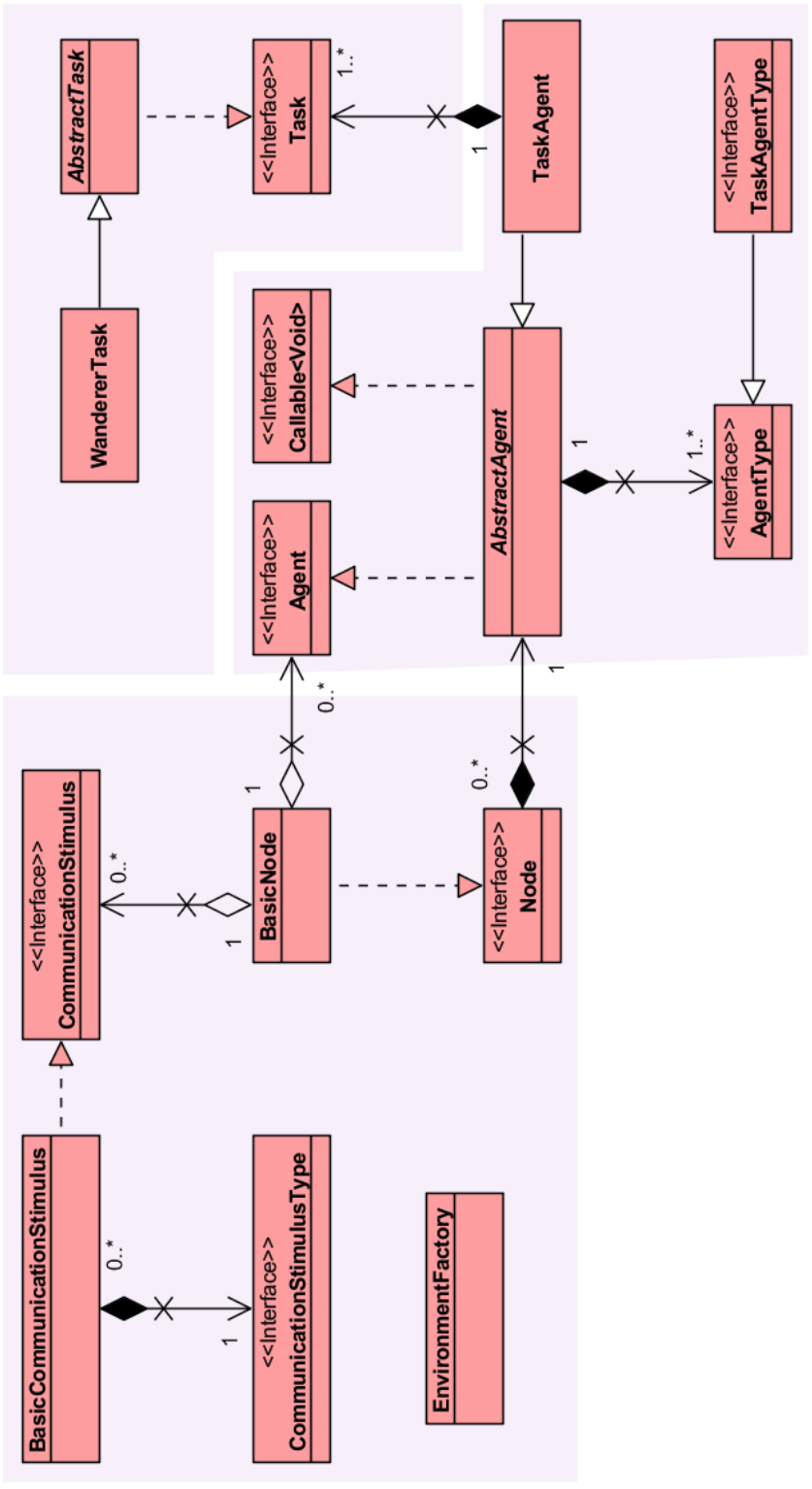


Figure 11: Generic Computational Model

Part II

EXPERIMENTS AND OBSERVATIONS

In this part of the report two experiments are proposed and executed. Firstly an investigation on the colony's sensitivity in relation to present pheromone in the environment is carried. The objective of this experiment is to study how the agent's navigation capacity is affected by different concentrations of pheromone presented in the environment.

The second experiment is a study on the affects that different diffusion rates of the forage pheromone will have on the colony's capability of foraging. The objective of this experiment is to determine if the colony's forage capacity is directly related to the increase of the pheromone radius. The results of each experiment are discussed in Chapter 4

Improvements to the model, to its implementation and further studies are proposed in Chapter 5, followed by the conclusion of this paper.

4

EXPERIMENTS AND OBSERVATIONS

4.1 SIMULATIONS AND DATA COLLECTION

The computational model does not formalise any resources to be used to create and run simulations, neither does the model formalise any way to collect data from the simulations.

JUnit is the reference unit test framework for Java, and the basic implementation of the proposed computational model is tested against more than 20 unit tests. However, unit tests also provide the perfect environment to run simulations. If resources such as initial node grid, initial pheromone concentrations and agents are going to be the same for more than one simulation, one can take advantage of test *fixture* with the *@Before* annotation to initialise these objects without repeating code. One could create a class for each simulation with a *Main* method also, however it is more convenient to use the test framework.

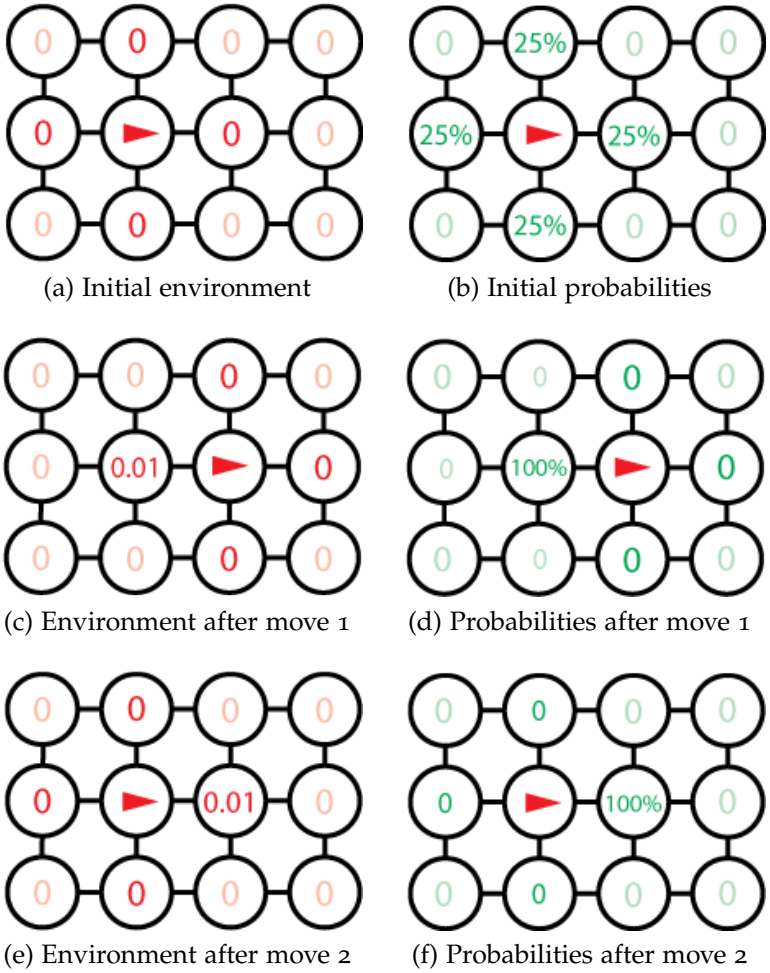
Another critical aspect of the simulations is data collection. The model does not describe how data can be acquired during and after the simulation. For the simulations run for this paper logging was used for this purpose. Log4j is a thread-safe logging service maintained by the Apache foundation and it was used throughout the implementation and the simulations. In most of the simulations the logger was setup to write into files that are processed after the simulation is finished. Also, it is possible to configure the logger service to write on the console, so information on the agents can be easily visualised during the execution of the simulations.

4.2 PHEROMONE CONCENTRATION SENSITIVITY

Due to the model of node selection, agents are very sensitive to the pheromone concentration presented by the environment around them. This happens because the probability of choosing a neighbour node swings from 0% to 100% very swiftly, causing the agents to get trapped.

This experiment investigates how different initial pheromone concentrations in the environment and the amount of pheromone each agent is capable of depositing in each interaction affect the agents navigation through the space. It also has an objective to determine what values of these two parameters are acceptable to use in the other experiments.

Firstly, let's examine the case when all the nodes of the environment are created with no initial concentration of *Forage* pheromone (Figure 12a). When the agent is deciding which node it is going to make the first move to, all neighbours have the same probability of being picked, because of the 0 of pheromone concentration (Figure 12b). So as it was programmed to, the agent picks one node at random. But before moving to the next node it lays a bit of pheromone, suppose the concentration deposited is 0.1. When the move is done, the environment around the agent should look like the one pictured in Figure 12c.



(a) Initial environment (b) Initial probabilities

(c) Environment after move 1 (d) Probabilities after move 1

(e) Environment after move 2 (f) Probabilities after move 2

Figure 12: Effect of initial pheromone concentration at zero

Now it is time for the agent to move again. Firstly the agent reads the pheromone intensity in each of the neighbour nodes, after it computes where it is more likely to move. For all the nodes around the agent apart from the one it has just moved from have 0 pheromone intensity in them, the agent is certain to move back to the previous node (Figure 12d). Before completing the move, it lays pheromone in the current node. When the agent is back to the node it first started from, the scenario repeats and it becomes a cycle. The agent go back and forth, trapped in these two nodes forever, with no changes to explore the environment further.

It is clear that the environment cannot be initialised with no *Forage* pheromone in its nodes. The question that arises from this is: Is there any pair of values for these two parameters that will allow the agents to navigate in an acceptable fashion?

To answer this question, the experiment was setup with the following configuration:

PROPERTY	VALUE
Number of lines	500
Number of columns	500
Total number of nodes	250,000
Duration of each simulation	10 s
Number of agents	50
Agent Type	WorkerType
Task executed	ForageTask
Agent sleep time	5 ms

Table 1: Experiment setup for investigation of initial pheromone concentration

In Table 1 the property *Agent sleep time* means how long the agent waits after a task is executed to choose another task to run. This is necessary to slow down agents (in this case the threads that are running the agent), otherwise they would cover the entire space in this 10 seconds only by the fact that they can do it very fast not because they are actively foraging.

The initial pheromone concentration and the amount of pheromone deposited in each interaction by the agents were varied from 0.001 to 0.04 in 5 steps. Each of the possible pair of values for the two parameters has been simulated:

1	2	3	4	5
0.001, 0.001	0.001, 0.005	0.001, 0.01	0.001, 0.02	0.001, 0.04
0.005, 0.001	0.005, 0.005	0.005, 0.01	0.005, 0.02	0.005, 0.04
0.01, 0.001	0.01, 0.005	0.01, 0.01	0.01, 0.02	0.01, 0.04
0.02, 0.001	0.02, 0.005	0.02, 0.01	0.02, 0.02	0.02, 0.04
0.04, 0.001	0.04, 0.005	0.04, 0.01	0.04, 0.02	0.04, 0.04

Table 2: Variations for initial concentration and amount of pheromone deposited by agents

In each case, the colony containing 50 agents is created at the north of the environment, horizontally centred. The agents execute the *ForageTask* since their creation, they do not analyse any contextual parameters such as other agents, they only try to move through the space using the rules defined by the task.

In order to compare how each possible value pairs in Table 3 affect the resulting navigation of the agents, two samples of the pheromone trail left by the agents are analysed. The first one is from close to the nest that will enable us to check how is the agents' response to the initial pheromone concentration shortly they have left the nest. The second sample is taken further ahead in the environment, far from the nest. This sample is a good way to test how the agents' own deposit of pheromone will affect the system behaviour, Figure 13 illustrates how and where the samples are made.

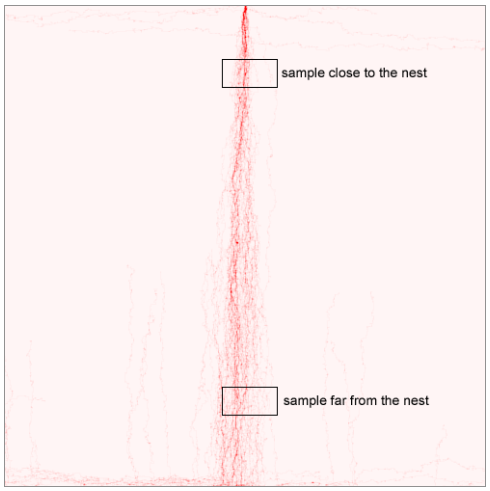


Figure 13: How the two samples of the environment is made

Figures 14 and 15 show the resulting sampling for all possible combination for the initial pheromone concentration and the amount of pheromone deposited by each agent in each interaction.

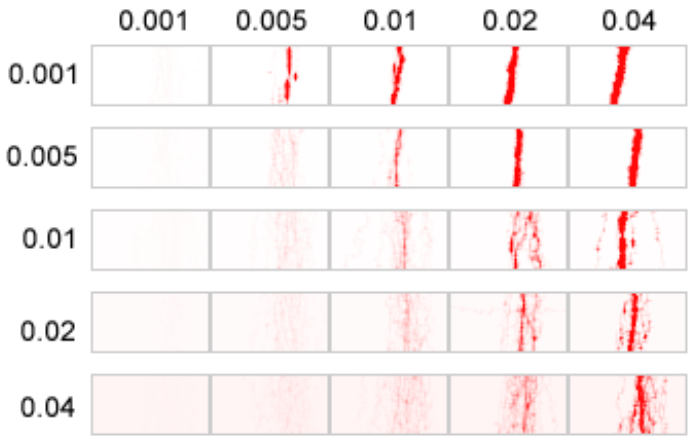


Figure 14: Resulting pheromone trail close to the nest

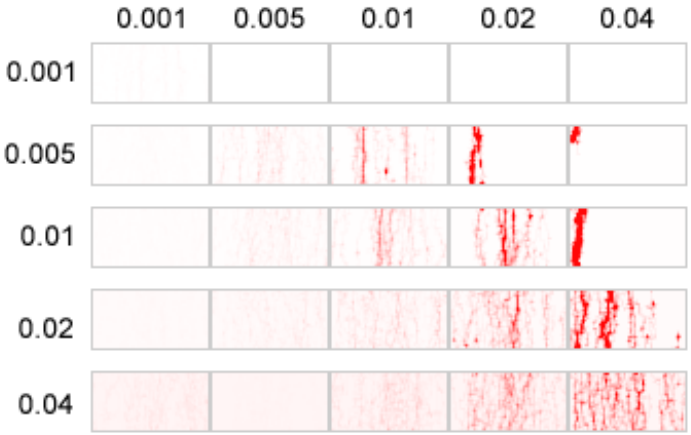


Figure 15: Resulting pheromone trail far from the nest

The trails left by the colonies can be compared in relation on how strong they are, how the agents are able to 'scape' them to explore the environment and how it shaped when agents get further from the nest.

Starting from 0.001 as the amount of pheromone to be deposited by agents in each interaction; it is possible to observe from the figures that two very contrasting behaviours emerge, firstly because the environment has so little pheromone and the update is so small, they weight assigned to each of the neighbour nodes count considerably more than the pheromone deposited by the agents, so the agents end up very dispersed, thus no chemical trail is formed at all. This phenomena is actually seen in

many other combination of the parameters, all the cases when the update is 0.001 in fact.

When the amount of pheromone deposited by the agents is increased the behaviour of the colony could not be more different than what was seen previously. The agents switch from exploring a large area to be 'trapped' into the pheromone trail. This impedes the agents of exploring the space, what is not desirable for any colony. This behaviour is also seen in other values for the parameters. What seems to be the rule is that if the update is considerable larger than the concentration of pheromone in the environment the agents will start to create a 'bubble' of high pheromone concentration and as consequence they are very unlikely to move to any node outside this area.

It rises the question, why does it happen? The answer is similar to one of the problem in initialising the environment with no pheromone at all. In this case, the critical point is the rate between the initial concentration and the amount deposited by the agents in each interaction.

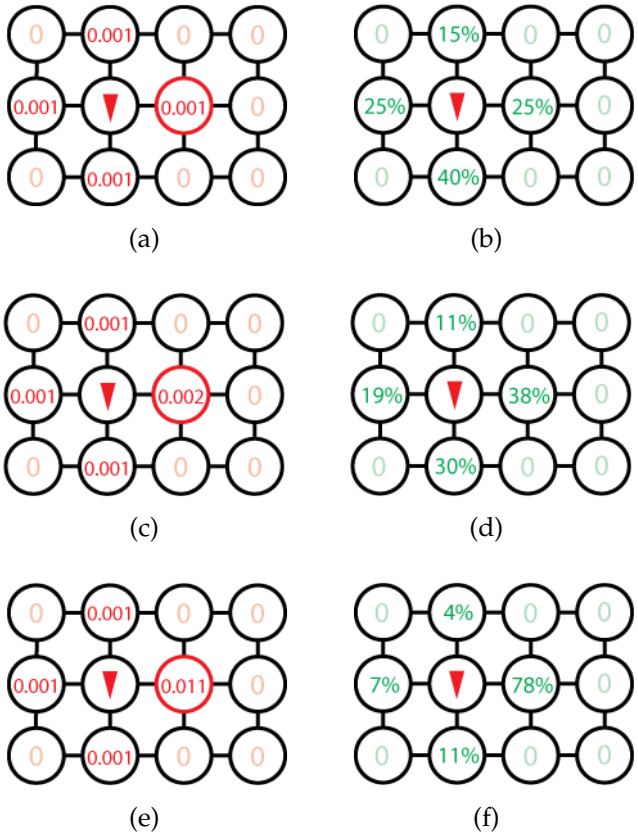


Figure 16: Shift of probability depending on agent update

Figure 16 illustrates how swiftly the probabilities can change depending on the amount of pheromone deposited in the node by agents. The node in red in the picture represents a node that has been updated previously by another agent. In the first scenario (Figure 16c) the update was 0.001, in the second (Figure 16e) the pheromone deposited was 0.01. The probability of the node in red being picked up to be the next node the agent will move to more than doubled. (Figure 16d and Figure 16f).

Further investigation revealed that the probability of selection increases in a logarithmic fashion. Figure 17 shows how the increase in probability progress when the amount of pheromone deposited by the agents increases by multiples of the initial concentration in the environment.

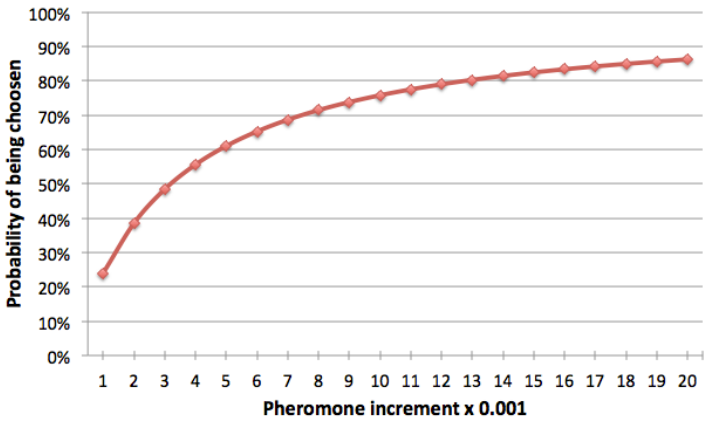
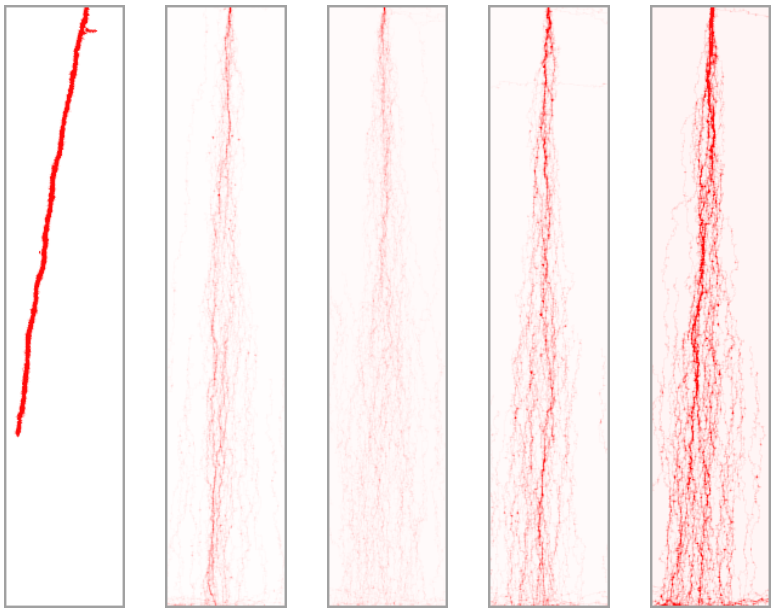


Figure 17: Increase of probability selection according to the increase of the deposit increment

It is very tempting to conclude that if the right ratio between the initial pheromone concentration and the increment step used by the agents is found the, problems of the lack of trail formation and the agent confinement in the trail will be solved. However from figures 14 and 15 it is observed that the same ratio (initial concentration/increase step) present different results for different values. In the case 0.001/0.001 no trail is formed at all, for 0.01/0.01 a solid trail is formed.

The experimental results are that the initial pheromone concentration should not be too low that avoid trail formation or saturation. It has to be an intermediate value that allows the agents to converge to a specific path, generating the trail, but not too fast, allowing the agents to explore other parts of the environment as well. As far as the update step goes, it also needs to be an intermediate value, so it starts actually being part of the node selection process but not big enough to quickly saturate the pheromone trail.



(a) (b) (c) (d) (e)

Figure 18: Complete trail pheromone trails left by simulation using 0.001,0.04 (a), 0.01,0.01 (b), 0.02,0.01 (c), 0.02,0.02 (d) and 0.4,04 (e) for initial pheromone concentration and update step respectively

One can conclude from this analysis pair 0.01/0.01 (Figure 18b) for the initial pheromone concentration and increment step present the best results. It allows pheromone trail to *emerge as a result of the agents' interactions with the environment*. It could also be argued that another pairs such 0.01/0.02 and 0.02/0.02 set the right conditions for agent navigation. However, in this experiment the agents are executing the *ForageTask* without actually collecting food, to no agent tries to deposit food in the nest and start foraging again - the trail is not reinforced. Taking that in consideration the other pairs tested could also be considered to lead to a premature saturation of the trail, which is the case of 0.02/0.02.

The pair 0.01/0.01 also proved to be less sensitive to the number of agents used in this and the other experiments.

4.3 FORAGE RADIUS INVESTIGATION

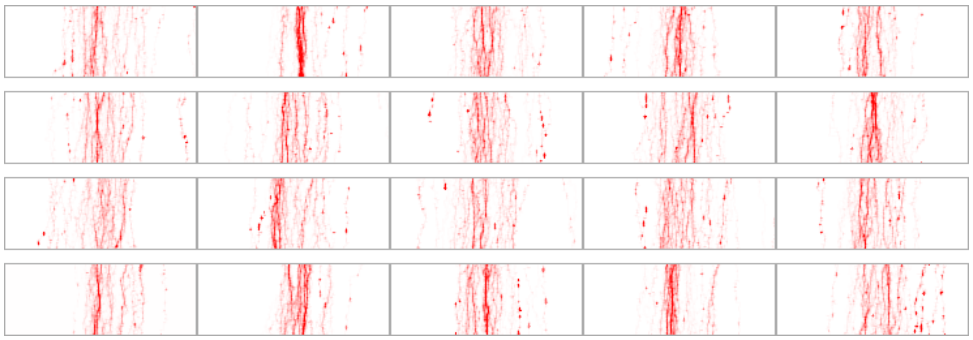
This experiment investigates the affect of the radius variation of the *Forage* chemical stimulus (section 3.1.4.2) on the capacity of the colony to collect food. Four values for the stimulus' radius were tried - 0, 1, 2. The experiment was setup as follows:

PROPERTY	VALUE
Number of lines	500
Number of columns	500
Total number of nodes	250,000
Initial grid pheromone intensity	0.01
Pheromone radius variations	0, 1, 2
Duration of each simulation	30 s
Number of agents	50
Agent Type	WorkerType
Pheromone increment step	0.01
Task executed	ForageTask and FindHome
Agent sleep time	5 ms
Pheromone decay frequency	every 3 seconds

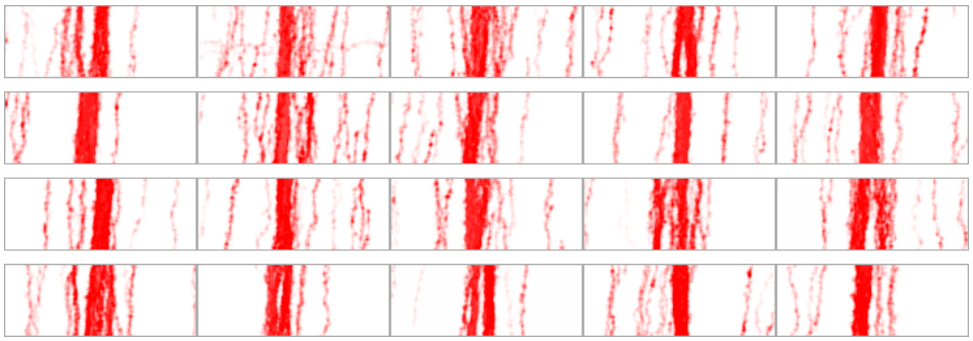
Table 3: Experiment setup for pheromone radius variation study

In this experiment the radius of action of the forage pheromone is varied in order to check the effects in the amount of food the colony is capable of forage. At first one would expect that the increase on the radius of the forage pheromone would have a positive impact on the colony capacity of collecting food as trails that lead to food sources would are reinforced by agents caring food back to the nest and with a wider spread of the pheromone more workers would be fall in these trails. However the data from the simulations show a different picture. The colony collect less and less food as the stimulus' radius increase. Table 4 is a summary of the data collected from the study.

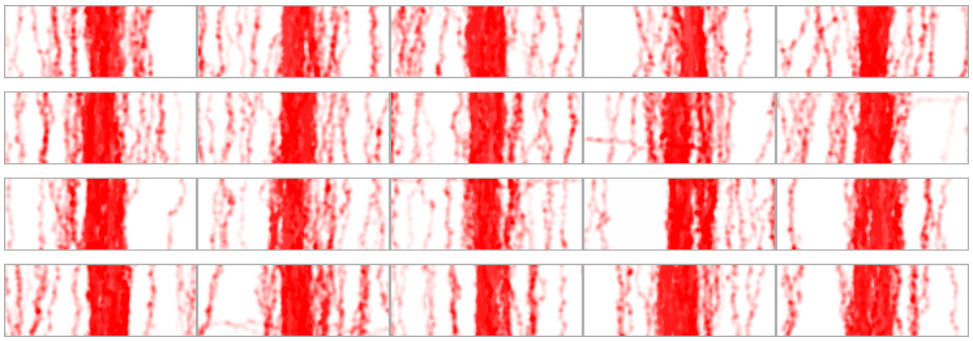
For this experiment a row of food sources was placed far away from the colony. Each source has a total amount of food of 30. The simulation was run 100 times for each radius variation. Figure 19 presents part of the middle section (see Figure 20a) of the trails created by the agents for the first 20 simulations for each value used for radius.



(a) radius = 0

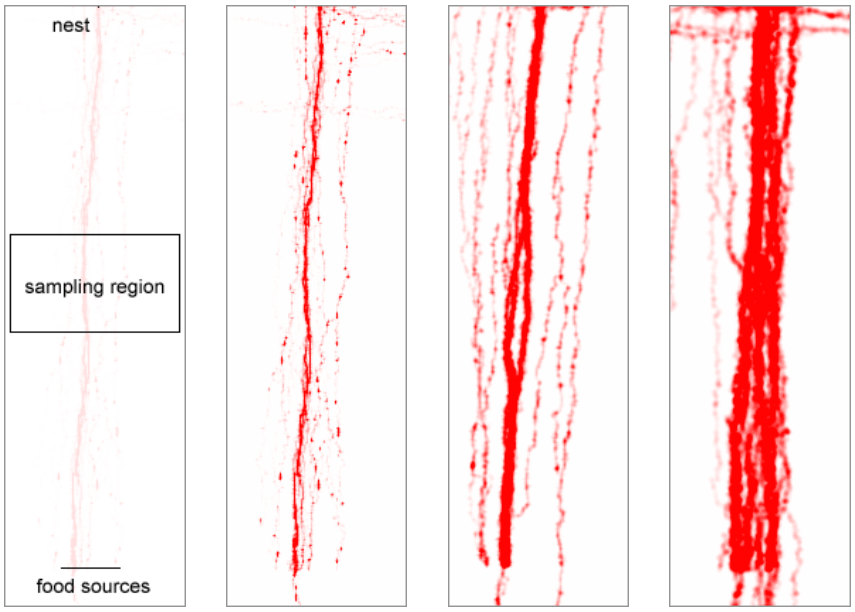


(b) radius = 1



(c) radius = 2

Figure 19: 20 graphical samples of the results in variating forage stimulus' radius



(a) setup (b) radius = 0 (c) radius = 1 (d) radius = 2

Figure 20: Example of trail formation for different radiuses

RADIUS	AVERAGE OF FOOD COLLECTED	STANDARD DEVIATION
0	6.0	1.21
1	4.9	0.72
2	3.3	0.53

Table 4: Forage radius simulations' outcomes

The averages that are presented in the Table 4 was calculated after statistically validating the dataset collected for each radius value, with any outlier sample was removed from the results collected from the simulations. The following pair of equation are used to define a sample point x_i as outlier or not:

$$\begin{cases} x_i \leqslant \bar{x} - 2 \times \sigma \\ x_i \geqslant \bar{x} + 2 \times \sigma \end{cases} \tag{3}$$

Where $\bar{x}$ is the average of the resulting dataset for each radius and σ is the standard deviation. The data collected proved to be consistent and only few samples were rejected. Figure 21 shows the samples spread for each radius and their probability density distribution.

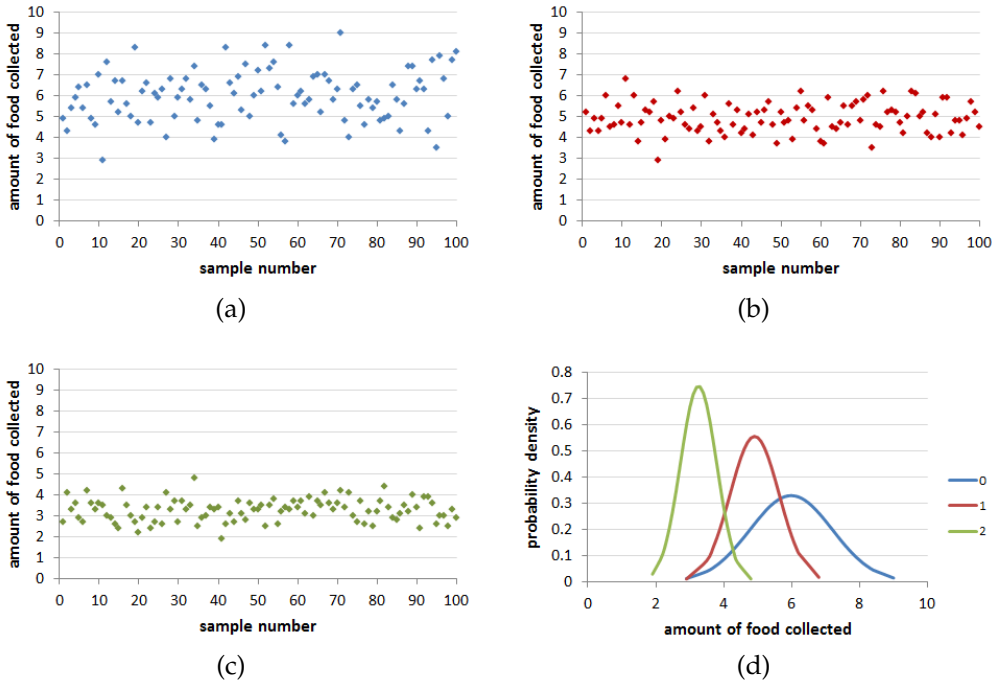


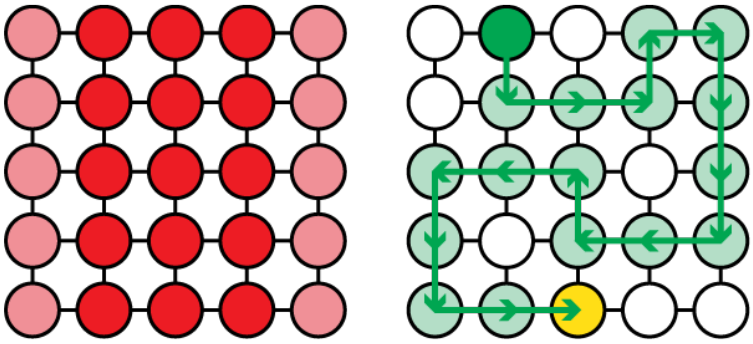
Figure 21: Samples distribution variating the pheromone radius - 0 (a), 1 (b) and 2 (c) - and the samples' probability density distribution (d)

Figure 21 helps to visualise what the standard deviations from Table 4 had already shown - the samples *variance* decreases when the radius increases. The reason for this phenomena is that the bigger the pheromone radius is the lesser agents are probable to 'escape' from the trail, thus they are not able to explore different areas of the environment. For the agents use virtually only the same pheromone trail to forage their outcomes are likely to be very close.

With a more diversified exploratory reach, agents in the simulations using radius 0 are likely to have different degrees of success in finding food and taking it back to the nest in each simulation run, justifying the greater variance in the colony outcome.

The study carries raises another question - why does smaller radiuses outperform bigger ones? From Table 4 we can see that the simulations with radius 0 outperforms the ones with radiuses 1 and 2 in 22 and 82 percent in average respectively. This occurs due to the formation of trails with large width, leading the agents to move from note to node in almost random fashion.

Figure 19 illustrates how the pheromone trails widen with the increase of the pheromone radius. It might seem contradictory at first that wider trails do not led to better forage performance, for more ants should be lead to food sources. However due to the model of node selection implemented the agents are too sensitive to the pheromone intensity of the direct neighbours of the node they are currently at, and this is translated to random node selection when agents are trapped in wide pheromone trails.



(a) (b)

Figure 22: Affect of wide pheromone trails on node selection

This random behaviour, as far node selection is concerned, has a profound impact on the colony's forage performance, as agents will struggle to find their way to the food sources, and when eventually some of them do, there still a good change that many will not reach the nest back in order to deposit the food they have collected.

This strongly suggests that ants use more than only local information available at the instant they are making decision where to move to. The other communication

methods employed by ants should play a vital role in assisting this decision to be made.

Two new possible experiments to better understand and improve node selection are proposed in section 5.3.

5

FUTURE WORK AND CONCLUSION

5.1 MODEL IMPROVEMENTS

The model proved to be flexible enough to enable easy implementation of the base classes and the declaration of different agent types and resources. The area that has been least developed is related with simulation. As explained in section 4.1 there is no formalisation related to simulations. This did not prove to be of any inconvenience during the creation and execution of the simulations for this paper.

However, for users who are less acquainted with the model and its implementation it could prove to be a challenge to write simulations and collected data from them. More important is the fact that managing all the elements that compose a simulation would be very challenging for the users less experienced to the Java language itself.

5.1.1 *Simulation handler*

Formalising the simulations and data acquisition by the use of interfaces is necessary. The would remove from the user the responsibility of creation and management of all elements necessary to run simulations. This could be done by the creation of a *simulation handler* that would contain all the necessary data objects, such as the environment (grid of nodes), list of agents, simulation renderers and the necessary methods to schedule data collection, execution of chemical stimulus renderers and so forth.

5.1.2 *Ant agent navigation improvements*

When it came to implement the different agents from the different casts, the biggest challenge faced was to create an algorithm to take the agents back to their nest. The method implemented in this paper (see sections 3.2.5.1, 3.3.2.3 and 3.3.2.3) should be improved in order to be used in more complex and longer simulations. In some cases the simulations would have a high rate of ants not finding their nest at all.

A possible improvement is to bring chemical landmarks to the model. An agent type could be created for this. This type could have properties that would point the ants to the right direction to the nest.

5.1.3 *Nests As Agents*

As seen in section 3.2.5.3 nest are represented as agents. On the one hand this facilitates their declaration and use as they enjoy all the infrastructure already in place for agents. On the other hand they are punctual, that is, they are placed in a node as any other agent and the other agents can see their nests only if they reach the same node at which the nest is placed. This is clearly a disadvantage and it does not reflect the reality of natural nests also.

An improvement to the model, that could be used to solve the problem of nests being punctual, could be an new interface to declare *node types*. A node would have a type in the same way agents have. This would open up opportunity to create far more complex environments that it is possible now. For instance, a node could be an *obstacle* type, agents would not be allowed in there, or even a node could be a chemical landmark. With nodes having types, nests could be modelled as a set of nodes of the *nest type*. This nodes would form a rectangular shape grid, in which if an agent reach any of the nodes they would be able to recognise they have reached their nest.

5.2 IMPLEMENTATION ISSUES

There is room from improvement on the basic implementation of the model. The comments made in the source files should be consolidated and in some cases improved. However a major possible improvement on the basic implementation is to change the ways the nodes are connected to from the environment grid.

Currently the nodes are connected in a *four-way* fashion, each node has 4 neighbours, one in each direction: *north, east, south, west*. This creates a major limitation related to agent navigation. Agents are very unlikely to move in diagonals, as it consists in two movements, first the agent move forward and after it moves sideways. It has been observed that agents hardly would do that because they have very low probability of selecting nodes in a diagonal sequence, e.g. north, east, north, east, north, east. Thus agents generally could not find any food sources in more complex environments where they were not directly aligned to the nest where the ants departed from.

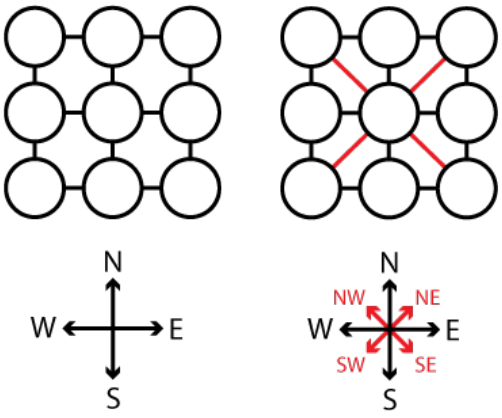


Figure 23: Node connection to its neighbours, four-way connected grid (a) and the eight-way connected grid (b)

An implementation of a grid connected differently would improve the problem considerably. If nodes had 8 neighbours instead of 4, agents would be able to navigate

directly in the diagonal. Figure 23 illustrates how the nodes would be connected and all the possible directions the agents could travel towards.

5.3 PROPOSED STUDIES

5.3.1 *Limited Number of Agents Per Node*

This experiment has the potential to explore how the formation and saturation of the pheromone trails would be affected by the limiting the number of agents that can be in a same node at the same time. From the experiment of section 4.3 it is clear that in the case of a chemical communication stimulus having radius greater than 0 agents do not follow a *rational* behaviour towards their goal of foraging. As explained in the experiment discussion the node selection model is responsible for that.

It would be interesting to investigate how the trails would emerge if the nodes had a maximum number of agents allowed in that node at the same time, this would force some agents to move to nodes they would not move to otherwise, it is clear that this is the actual case in nature, as two ants cannot occupy the same physical space.

5.3.2 *Improved pheromone sensing capability*

The experiment *Forage Radius Investigation* (ection 4.3) has showed that the agents are too sensitive to the pheromone intensity of the direct neighbours of the node they are currently at. Again, this is due to the node selection model.

This proposed experiment would investigate how the colony forage performance and trail formation would change if agents would take in consideration not only the pheromone intensity of the neighbours of its current node but also the neighbours of the neighbours would be part of the decision process. Different 'depths' of sensibility

could be experimented. This new feature would reduce the agent's sensibility to direct neighbours of its current node.

Another possible improvement on the agent's sensing capacity would be the introduction of some sort of memory that would play part of the node selection process.

5.4 CONCLUSION

The proposed computational model proposed by this paper proved to achieve the two objectives set for this project - flexibility and robustness. The generic model infrastructure provided all the necessary entities to define and implement a wide variety of agents, environments and tasks. More than that, it was possible to demonstrate how simple it is to extend the model, using the case of ant colonies and 2 dimensional environments. Layers of complexity were added as needed, taking advantage of all the previously existing infrastructure with minimal effort.

Also, simulations that were run repeatedly using the same parameters returned, within the expected variations, the same results. The simulations scaled well when the number of agents and the size of the environment varied greatly. Some test experiments used as few as 5 agents in small environments containing a few hundreds of nodes. Some simulations were run for minutes using up to 350 agents in environments containing up to 750,000 nodes with no problems at all.

As far as the implementation of the model is concerned, the model of node selection implemented by this paper presented to be inappropriate. Further study should be carried (see section 5.3 for proposed studies) in order to confirm that. The agents' over-sensitivity to the neighbours of the node it is currently in, is the most problematic effect of the node selection model. The necessity of having to initialise the environment with some pheromone intensity, for the smallest that it might be, is a strong argument against the node selection model implemented by itself. It does not reflect what actually happens in real ant colonies. However this did not affected the two proposed

experiments in anyway as their objective was to study how the agent's react to the environment changes, starting from a pre-defined setup, whatever it might be.

As for the agent implementation, the problem of taking the agents back to their nest proved to be by far the most challenging task to be implemented. The tasks *FindHomeTask* (section 3.3.2.2) and *FindHomeAndHide* (section 3.3.2.3) use the agent's limited memory to direct the agent on its way to the nest, however they proved to be very inefficient and other mechanisms such as chemical landmarks must be considered.

The first experiment demonstrated how agents react to different concentrations of pheromone, the most important outcome of the experiment was the contestation that the trails are a by product of the agents' interactions not only with the environment but also with other agents. They clearly *emerge* from these interactions. This is a case of *strong* emergence.

Another noteworthy conclusion taken from the first experiment is that the relationship between the quantity of pheromone that each agent deposit in each interaction and the amount of pheromone already present in the environment regulate the colony's capability to explore the environment around its nest. There is a fine line between having very successful colonies, as far space exploration is concerned, and colonies that could not even reach a source of food. The colony's behaviour switches from one state to another very swiftly depending on this rate.

The second experiment showed that the increase of the communication stimulus' radius did not mean an increase on the forage capacity of the ant colonies, as it was firstly expected. Smaller radiuses give rise to thin, better defined, pheromone trails that allow the agents to move towards the food source direction objectively. Whereas, larger radius 'confuse' the agents within the pheromone trail, seriously undermining the colony's capacity of foraging. The reason for this problem is that communication stimuli with larger radius of reach end up creating clouds of pheromone, and agents have little change to escape this region of high pheromone concentration. This creates a vicious cycle, for the agents in the trail continue to deposit pheromone,

over and over again around the same area, saturating the nodes within the trail with pheromone very quickly. Resulting in even smaller probability of 'breaking' from the trail. It is clear that this outcome is also closely related to the agents' over-sensitivity to the pheromone intensity in the neighbour nodes of the current node the agent is in. Another strong result against the model of node selection implemented.

Part III

APPENDIX

A

EXTRA EXPERIMENTAL RESULTS

I need to select which images to add here. They already have been generated, I just need to choose and crop some of them if necessary.

B

MODEL AND SIMULATION SOURCE CODE

B.1 MODEL IMPLEMENTATION DETAILS

I'm finalising all the diagrams for the ants package. I will add them here

B.2 SOURCE CODE

B.2.1 *Environment*

Listing 1: Node.java

```
1  package com.luizabrahao.msc.model.env;

   import java.util.List;

   import com.luizabrahao.msc.model.agent.Agent;
6  import com.luizabrahao.msc.model.annotation.FrameworkExclusive;
   import com.luizabrahao.msc.model.annotation.ThreadSafetyBreaker;

   /**
    * This class represents a piece of the environment, one could say it is a
11  * infinitesimal representation of the environment. A node as a representation
    * of space is able to accommodate agents.
    *
    * @author Luiz Abrahao <luiz@luizabrahao.com>
    *
16  */
   public interface Node {
           /**
            * Each agent has an unique identifier, this is going to be used during the
            * data analysis.
21          * @return
            */
           String getId();

           /**
26          * Adds a agent to this node, it takes care of both directions connecting
            * both node to the agent and the agent to the node.
            *
            * @param agent Agent that is going to be added to the node.
            */
31          void addAgent(Agent agent);

           /**
            * Returns the neighbour node of the specified direction. In case the node
            * is part of the boundary in a determined direction it will not have any
36          * neighbour in that direction, so null will be returned instead.
            *
            * IMPORTANT: This method is not thread-safe! This means that you must not
            * use it in simulations that the environment changes, that is, the nodes'
            * neighbours change. This method should be used only to transverse the
41          * nodes graph. The reason for not making this method thread-safe is
            * performance. This method is likely to be called very often during the
            * simulations, so it wouldn't make sense to make it thread-safe if there
```

```
         * are no plans to use dynamic environments.
         *
         * @param direction Direction of the neighbour node related to the current
         * object.
         * @return Node neighbour node.
         */
        @ThreadSafetyBreaker
        Node getNeighbour(Direction direction);

        /**
         * This method should not be called directly from your code. As the
         * neighbours node are not exposed explicitly, this method creates the
         * means to set a neighbour in a particular direction.
         *
         * DO NOT CALL this method in your code, use setNeighbours instead, it
         * makes sure both of your node objects are liked to each other.
         *
         * IMPORTANT: This method is not thread-safe! This means that you must not
         * use it in simulations that the environment changes, that is, the nodes'
         * neighbours change. This method is to be used only during the environment
         * setup.
         *
         * @param direction Direction of the neighbour node related to the current
         * object.
         * @param node Node neighbour node.
         */
        @FrameworkExclusive @ThreadSafetyBreaker
        void setNeighbour(Direction direction, Node node);

        /**
         * Firstly this set the node passed as argument as the neighbour of the
         * instance node, after that, it does the same for the neighbour node but
         * in the opposite direction.
         *
         * IMPORTANT: This method is not thread-safe! This means that you must not
         * use it in simulations that the environment changes, that is, the nodes'
         * neighbours change. This method should be used only to transverse the
         * nodes graph. The reason for not making this method thread-safe is
         * performance. This method is likely to be called very often during the
         * simulations, so it wouldn't make sense to make it thread-safe if there
         * are no plans to use dynamic environments.
         *
         * @param direction Direction of the neighbour node related to the current
         * object.
         * @param node Node neighbour node.
         */
        @FrameworkExclusive @ThreadSafetyBreaker
        void setNeighbours(Direction direction, Node node);

        List<Agent> getAgents();

        void addAgentStartingHere(Agent agent);

        List<CommunicationStimulus> getCommunicationStimuli();

        void addCommunicationStimulus(CommunicationStimulus communicationStimulus);

        CommunicationStimulus getCommunicationStimulus(CommunicationStimulusType communicationStimulusType);
    }
```

Listing 2: BasicNode.java

```
    package com.luizabrahao.msc.model.env;

    import java.util.ArrayList;
    import java.util.Collections;
    import java.util.List;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import net.jcip.annotations.GuardedBy;

import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.annotation.FrameworkExclusive;
import com.luizabrahao.msc.model.annotation.PseudoThreadSafe;
import com.luizabrahao.msc.model.annotation.ThreadSafetyBreaker;

/**
 * This class is the basic implementation of Node. It hold references to
 * neighbour nodes and utility methods to navigate through them. The node
 * shape is a square, and its neighbours are represented by the north, east,
 * south and west field variables.
 *
 * It is essential to keep this class as lightweight as possible, for it is
 * extensively used. To illustrate the point, a simulation with a reasonable
 * resolution like 300 lines by 100 columns will have 30 000 objects of this
 * class.
 *
 * Note that this class is thread—safe as far as the agents are concerned. The
 * methods getNeighbour, setNeighbour and setNeighbours do expose the neighbour
 * nodes, but they were deliberately left without synchronisation because they
 * must be used only at setup time, that is, the environment does not change
 * after the simulation starts. getNeighbour is extensively used throughout the
 * simulation, so the overhead added by the synchronisation would not pay off.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@PseudoThreadSafe
public class BasicNode implements Node {
	private static final Logger logger = LoggerFactory.getLogger(BasicNode.class);

	private final String id;

	private Node north = null;
	private Node east = null;
	private Node south = null;
	private Node west = null;
	@GuardedBy("this") private List<Agent> agents = null;
	@GuardedBy("this") private List<CommunicationStimulus> communicationStimuli = null;

	public BasicNode(String id) {
		this.id = id;
	}

	@Override public List<Agent> getAgents() { return agents; }
	@Override public String getId() { return id; }
	@Override public List<CommunicationStimulus> getCommunicationStimuli() { return communicationStimuli; }

	@Override
	public void addCommunicationStimulus(CommunicationStimulus communicationStimulus) {
		synchronized (this) {
			if (communicationStimuli == null) {
				communicationStimuli = Collections.synchronizedList(new ArrayList<
					CommunicationStimulus>());
			}
		}

		this.communicationStimuli.add(communicationStimulus);
	}

	/**
	 * This needs to be synchronised because the agents list is lazily
	 * initialised. This time it was chosen to pay the price the
	 * synchronisation adds in order to save memory allocation.
	 */
```

```
@Override
public void addAgent(Agent agent) {
	synchronized(this) {
		if (agents == null) {
			agents = Collections.synchronizedList(new ArrayList<Agent>());

		} else {
			// if the agent is in the node already, just ignore the call.
			if ((agent.getCurrentNode() == this)) {
				logger.info("Agent {} already in the node {}!", agent.getId(), this.getId());
				return;
			}
		}
	}

	synchronized(agents) {
		this.agents.add(agent);
		logger.debug("{}: agent {} moved here.", this.getId(), agent.getId());
	}

	// Let's remove the agent from the node's agent list, and after we set
	// the new current node to reflect the agent's new position. This was
	// decided to be done here because a agent cannot be in two places at
	// the same time, so adding a agent to a node, means removing from the
	// other one (agent's current node).
	agent.getCurrentNode().getAgents().remove(agent);

	// I'm not sure if I should leave this inside or outside the
	// synchronisation block above, I'm leaving outside now because I think
	// if I leave inside the method will use the wrong lock and Agent is
	// thread-safe so should be fine.
	agent.setCurrentNode(this);

	// it doesn't need to be in a synchronised block because the recording
	// flag is final and the history list is synchronised
	if (agent.shouldRecordNodeHistory()) {
		agent.addToVisitedHistory(this);
	}
}

/**
 * Returns the neighbour node in the specified direction. This method is
 * not thread-safe, but it was decided to leave so as it will not cause any
 * issue when used in environment that don't change during the simulation.
 */
@Override @ThreadSafetyBreaker
public Node getNeighbour(Direction direction) {
	switch (direction) {
		case NORTH:
			return this.north;
		case EAST:
			return this.east;
		case SOUTH:
			return this.south;
		case WEST:
			return this.west;
	}

	throw new RuntimeException("Direction '" + direction + "' is not valid.");
}

/**
 * Should not be called directly from user code. It is used to expose
 * neighbours indirectly and is not thread-safe.
 */
@Override @ThreadSafetyBreaker @FrameworkExclusive
public void setNeighbour(Direction direction, Node node) {
	switch (direction) {
		case NORTH:
			this.north = node;
```

```
                        break;
                case EAST:
                        this.east = node;
                        break;
                case SOUTH:
                        this.south = node;
                        break;
                case WEST:
                        this.west = node;
                        break;
        }
}

/**
 * Should be used only at environment setup time as it is not thread-safe
 */
@Override @ThreadSafetyBreaker @FrameworkExclusive
public void setNeighbours(Direction direction, Node node) {
        switch (direction) {
                case NORTH:
                        this.north = node;
                        node.setNeighbour(Direction.SOUTH, this);
                        break;

                case EAST:
                        this.east = node;
                        node.setNeighbour(Direction.WEST, this);
                        break;

                case SOUTH:
                        this.south = node;
                        node.setNeighbour(Direction.NORTH, this);
                        break;

                case WEST:
                        this.west = node;
                        node.setNeighbour(Direction.EAST, this);
                        break;
        }
}

@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((east == null) ? 0 : east.hashCode());
        result = prime * result + ((north == null) ? 0 : north.hashCode());
        result = prime * result + ((south == null) ? 0 : south.hashCode());
        result = prime * result + ((west == null) ? 0 : west.hashCode());
        return result;
}

@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        BasicNode other = (BasicNode) obj;
        if (east == null) {
                if (other.east != null)
                        return false;
        } else if (!east.equals(other.east))
                return false;
        if (north == null) {
                if (other.north != null)
                        return false;
        } else if (!north.equals(other.north))
```

```
                return false;
            if (south == null) {
                if (other.south != null)
                    return false;
            } else if (!south.equals(other.south))
                return false;
            if (west == null) {
                if (other.west != null)
                    return false;
            } else if (!west.equals(other.west))
                return false;
            return true;
        }

        @Override
        public String toString() {
            return "BasicNode [id=" + id + "]";
        }

        @Override
        public void addAgentStartingHere(Agent agent) {
            synchronized (this) {
                if (agents == null) {
                    agents = Collections.synchronizedList(new ArrayList<Agent>());
                }

                synchronized(agents) {
                    this.agents.add(agent);
                    logger.trace("{}: agent {} initialised here.", this.getId(), agent.getId());
                }

                // not synchronised for the same reason described in addAgent
                // method.
                agent.setCurrentNode(this);
            }
        }

        public CommunicationStimulus getCommunicationStimulus(CommunicationStimulusType communicationStimulusType)
            {
            if (communicationStimuli == null) {
                return null;
            }

            for (CommunicationStimulus stimulus : communicationStimuli) {
                if (stimulus.getType() == communicationStimulusType) {
                    return stimulus;
                }
            }

            return null;
        }
}
```

Listing 3: Direction.java

```
package com.luizabrahao.msc.model.env;

/**
 * These are the directions each node can use to look for a neighbour. The idea
 * is that a node can have a neighbour in each of the following directions.
 *
 * At this stage, as we only have the BasicNode implementation, which has a
 * square shape, we only have four possible neighbours, thus we have four
 * directions. Nothing stops us to create more complex nodes, hexagon shaped
 * for example, with 6 neighbours. In that case extra directions could be added
 * to describe the direction of a hexagon—node and its neighbour.
 *
```

```
 * @author Luiz Abrahao <luiz@abrahao.com>
 *
 */
public enum Direction {
        NORTH, EAST, SOUTH, WEST;
}
```

Listing 4: CommunicationStimulus.java

```
package com.luizabrahao.msc.model.env;

/**
 * A communication stimulus might represent any interaction between agents and
 * the environment. For example, when ant agents deposit pheromone on the
 * environment, the pheromone is a type of communication stimulus.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
public interface CommunicationStimulus {
        /**
         * Returns the type of communication stimulus the object has.
         * @return CommunicationStimulusType Type of communication stimulus.
         */
        CommunicationStimulusType getType();
}
```

Listing 5: CommunicationStimulusType.java

```
package com.luizabrahao.msc.model.env;

/**
 * Communication stimulus type allows to differentiate between stimulus that are
 * present in a node. This is important when agents update the present stimuli
 * in the nodes they operate upon.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
public interface CommunicationStimulusType {
        /**
         * A identifier that should be unique.
         *
         * @return String Type's identifier name
         */
        String getName();
}
```

Listing 6: BasicCommunicationStimulus.java

```
package com.luizabrahao.msc.model.env;

import net.jcip.annotations.Immutable;

/**
 * A communication stimulus might represent any interaction between agents and
 * the environment. For example, when ant agents deposit pheromone on the
 * environment, the pheromone is a type of communication stimulus. The
 * ChemicalCommStimulus class is an example of this case. This is just an
 * abstraction implementation that should be used as base.
```

```
 *
 * @see ChemicalCommStimulus
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@Immutable
public class BasicCommunicationStimulus implements CommunicationStimulus {
	private final CommunicationStimulusType communicationStimulusType;

	public BasicCommunicationStimulus(CommunicationStimulusType communicationStimulusType) {
		this.communicationStimulusType = communicationStimulusType;
	}

	@Override public CommunicationStimulusType getType() { return this.communicationStimulusType; }

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime
				* result
				+ ((communicationStimulusType == null) ? 0
						: communicationStimulusType.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		BasicCommunicationStimulus other = (BasicCommunicationStimulus) obj;
		if (communicationStimulusType == null) {
			if (other.communicationStimulusType != null)
				return false;
		} else if (!communicationStimulusType
				.equals(other.communicationStimulusType))
			return false;
		return true;
	}


}
```

Listing 7: FoodSourceAgentType.java

```
package com.luizabrahao.msc.ants.env;

import java.util.List;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.agent.TaskAgentType;
import com.luizabrahao.msc.model.task.Task;

@ThreadSafe
public enum FoodSourceAgentType implements TaskAgentType {
	TYPE;

	private final String name = "type:ant:food-source";

	@Override
	public String getName() { return name; }
```

```
19          @Override public List<Task> getTasks() { return null; }
}
```

Listing 8: FoodSourceAgent.java

```
package com.luizabrahao.msc.ants.env;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import net.jcip.annotations.GuardedBy;
import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.agent.AbstractAgent;
import com.luizabrahao.msc.model.env.Node;

@ThreadSafe
public class FoodSourceAgent extends AbstractAgent {
        private static final Logger logger = LoggerFactory.getLogger(FoodSourceAgent.class);
        private static final long CHECK_INTERVAL = 1000;
        @GuardedBy("this") private double foodAmount;

        public FoodSourceAgent(String id, Node currentNode, double amountOfFood) {
                super(id, FoodSourceAgentType.TYPE, currentNode, false);
                this.foodAmount = amountOfFood;

                logger.debug("Food source '{}' initialised with amount of '{}'", id, amountOfFood);
        }

        public synchronized double collectFood(double amountToCollect) {
                if (this.foodAmount == 0) {
                        logger.trace("{}: there is no more food to be collected.", this.getId());
                        return 0;
                }

                if (this.foodAmount > amountToCollect) {
                        this.foodAmount = this.foodAmount - amountToCollect;

                        logger.trace("{}: {} of food collected.", this.getId(), amountToCollect);
                        return amountToCollect;

                } else {
                        double amountAvailable = this.foodAmount;
                        this.foodAmount = 0;

                        logger.trace("{}: {} of food collected.", this.getId(), amountAvailable);
                        return amountAvailable;
                }
        }

        @Override
        public Void call() throws Exception {
                while (true) {
                        try {
                                Thread.sleep(FoodSourceAgent.CHECK_INTERVAL);
                        } catch (Exception e) {
                                // don't need to do anything... let it just stop...
                        }

                        synchronized (this) {
                                if (foodAmount == 0) {
                                        logger.info("Food source '{}' has 0 of food.", this.getId());
                                }
                        }
                }
        }
```

```
}
```

Listing 9: PheromoneNode.java

```
package com.luizabrahao.msc.ants.env;

import com.luizabrahao.msc.model.env.BasicNode;
import com.luizabrahao.msc.model.env.CommunicationStimulus;

public class PheromoneNode extends BasicNode {

	public PheromoneNode(String id) {
		super(id);
	}

	public synchronized ChemicalCommStimulus getCommunicationStimulus(ChemicalCommStimulusType
	      chemicalCommStimulusType) {
		if (this.getCommunicationStimuli() != null) {
			for (CommunicationStimulus stimulus : this.getCommunicationStimuli()) {
				if (stimulus.getType() == chemicalCommStimulusType) {
					return (ChemicalCommStimulus) stimulus;
				}
			}
		}

		ChemicalCommStimulus c = new ChemicalCommStimulus(chemicalCommStimulusType);
		this.addCommunicationStimulus(c);

		return c;
	}

	public void incrementStimulusIntensity(ChemicalCommStimulusType chemicalCommStimulusType, double amount) {
		ChemicalCommStimulus c = this.getCommunicationStimulus(chemicalCommStimulusType);
		c.increaseIntensity(amount);
	}
}
```

Listing 10: ForageStimulusType.java

```
package com.luizabrahao.msc.ants.env;

public enum ForageStimulusType implements ChemicalCommStimulusType {
	TYPE;

	private static final String name = "ant:env:stimulus:forage";
	private static final double decay_factor = 0.1;
	private static final int radius = 0;

	@Override public String getName() { return name; }
	@Override public double getDecayFactor() { return decay_factor; }
	@Override public int getRadius() { return radius; }
}
```

Listing 11: WarningStimulusType.java

```
package com.luizabrahao.msc.ants.env;

public enum WarningStimulusType implements ChemicalCommStimulusType {
	TYPE;
```

```
        private static final String name = "ant:env:stimulus:attack";
        private static final double decay_factor = 0.2;
        private static final int radius = 5;

        @Override public String getName() { return name; }
        @Override public double getDecayFactor() { return decay_factor; }
        @Override public int getRadius() { return radius; }
}
```

B.2.2 *Agent*

Listing 12: Agent.java

```
package com.luizabrahao.msc.model.agent;

import java.util.List;

import com.luizabrahao.msc.model.env.Node;

/**
 * The public API for every agent defined in the simulation.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
public interface Agent {
        /**
         * Every agent must have an identifier, and this should be used in the
         * hashCode() and equals().
         *
         * @return String a unique identifier.
         */
        String getId();

        /**
         * Returns agent's cast
         *
         * @return Cast agent's cast
         */
        AgentType getAgentType();

        /**
         * The node that the agent is currently sat on.
         * This must be thread safe
         *
         * @param node Node a node
         */
        void setCurrentNode(Node node);

        /**
         * Returns the node the agent is sat on.
         * This must be thread safe
         *
         * @return Node
         */
        Node getCurrentNode();

        /**
         * An agent might hold a list of nodes that it has visited, this method
         * allows to a node to be added to this list. Agents that do implement the
         * list should initialise it lazily.
         *
         * @param node Node to add to the list of nodes that have been visited.
         */
```

```
52          void addToVisitedHistory(Node node);

            /**
             * Returns the list of nodes that the agent has visited. Node that this
             * list must be unmodifiable to ensure thread safety.
57           *
             * @return List of nodes
             */
            List<Node> getNodesVisited();

62          /**
             * Agents should define a flag that is used to know if the history of nodes
             * visited by the agent should be recorded or not. This method returns the
             * status of that flag.
             *
67           * @return boolean the status of the flag.
             */
            boolean shouldRecordNodeHistory();
}
```

Listing 13: AbstractAgent.java

```
    package com.luizabrahao.msc.model.agent;

    import java.util.ArrayList;
    import java.util.Collections;
5   import java.util.List;
    import java.util.concurrent.Callable;

    import org.slf4j.Logger;
    import org.slf4j.LoggerFactory;
10
    import net.jcip.annotations.GuardedBy;
    import net.jcip.annotations.ThreadSafe;

    import com.luizabrahao.msc.model.env.Node;
15  /**
     * Base implementation of Agent interface. It also implements Runnable in order
     * to force subclasses to be able to be ran in different threads.
     *
     * @author Luiz Abrahao <luiz@luizabrahao.com>
20   *
     */
    @ThreadSafe
    public abstract class AbstractAgent implements Agent, Callable<Void> {
            private static final Logger logger = LoggerFactory.getLogger(AbstractAgent.class);
25
            protected final String id;
            protected final AgentType agentType;
            protected final boolean recordNodeHistory;
            @GuardedBy("this") protected Node currentNode;
30          @GuardedBy("this") protected List<Node> nodesVisited = null;

            public AbstractAgent(String id, AgentType agentType, Node currentNode, boolean recordNodeHistory) {
                    this.id = id;
                    this.agentType = agentType;
35                  this.recordNodeHistory = recordNodeHistory;
                    currentNode.addAgentStartingHere(this);
            }

            /**
40           * This method is not normally called directly form user code. In normal
             * circumstances the current node is set by the node.addAgent() method.
             * Which makes sure both sides of the relationship are defined.
             *
             * @see Node
45           * @param Node node is going to be set as current.
```

```
         */
        @Override public synchronized void setCurrentNode(Node node) { this.currentNode = node; }

        @Override public synchronized Node getCurrentNode() { return currentNode; }
        @Override public String getId() { return id; }
        @Override public AgentType getAgentType() { return agentType; }

        @Override
        public void addToVisitedHistory(Node node) {
                synchronized (this) {
                        if (nodesVisited == null) {
                                nodesVisited = Collections.synchronizedList(new ArrayList<Node>());
                        }
                }

                nodesVisited.add(node);
        }

        @Override
        public synchronized List<Node>getNodesVisited() {
                if (!this.recordNodeHistory) {
                        logger.error("Node {} wasn't asked to record the list of nodes it has been, but the recordHistoryNode
                              has tried to be accessed.");
                        return new ArrayList<Node>();
                }

                if (nodesVisited != null) {
                        return nodesVisited;
                }

                if (this.recordNodeHistory) {
                        logger.warn("{} has no node in the visited list, but was asked to recorcord its moving history", this.
                              getId());
                        return new ArrayList<Node>();
                }

                return null;
        }

        @Override public boolean shouldRecordNodeHistory() { return recordNodeHistory; }
}
```

Listing 14: TaskAgent.java

```
package com.luizabrahao.msc.model.agent;

import java.util.List;

import net.jcip.annotations.GuardedBy;
import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.env.Node;
import com.luizabrahao.msc.model.task.Task;

/**
 * This class is a task-oriented agent. It contains a list of tasks that the
 * agent is capable to execute. Only one task can be executed at time, the
 * currentTask variable holds the current task in execution.
 *
 * This class is thread-safe.
 *
 * I have decided not to protect the code throwing custom exceptions at this
 * stage, it should be an improvement to be considered if the model is to be
 * expanded.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
```

```
 */

// TODO I don't remember why it has a field that is a TaskAgentType... Shouldn't
// be needed, clients should use generic interface and cast when necessary...
@ThreadSafe
public abstract class TaskAgent extends AbstractAgent {
        @GuardedBy("this") private Task currentTask;
        protected final TaskAgentType agentType;

        public TaskAgent(String id, TaskAgentType agentType, Node currentNode, boolean recordNodeHistory) {
                super(id, agentType, currentNode, recordNodeHistory);
                this.agentType = agentType;
        }

        public List<Task> getTaskList() { return agentType.getTasks(); }
        public synchronized Task getCurrentTask() { return currentTask; }
        public synchronized void setCurrentTask(Task currentTask) { this.currentTask = currentTask; }

        public synchronized void setCurrentTask(String taskName) {
                this.currentTask = this.getTaskByName(taskName);
        }

        public Task getTaskByName(String name) {
                for (Task task : agentType.getTasks()) {
                        if (task.getName().equals(name)) {
                                return task;
                        }
                }

                throw new RuntimeException(this.getId() + " is not capable of performing task " + name);
        }
}
```

Listing 15: AgentType.java

```
package com.luizabrahao.msc.model.agent;

/**
 * AgentType is a basic data type that hold all the tasks agent of a certain
 * type can perform and extra information like the type's name.
 *
 * AgentTypes are advised to be implemented as singleton, in order to save
 * memory consumption.
 *
 * Classes that implement this interface must be thread−safe and have a public
 * static final field called NAME.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
public interface AgentType {
        String getName();
}
```

Listing 16: TaskAgentType.java

```
package com.luizabrahao.msc.model.agent;

import java.util.List;

import com.luizabrahao.msc.model.task.Task;


public interface TaskAgentType extends AgentType {
```

```
        List<Task> getTasks();
}
```

Listing 17: BasicTaskAgentType.java

```
package com.luizabrahao.msc.model.agent;

import java.util.ArrayList;
import java.util.List;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.task.Task;
import com.luizabrahao.msc.model.task.WandererTask;

/**
 * This is the reference implementation of a concrete TaskAgentType. Note that
 * it is implemented as an Enum to implement the Singleton pattern.
 *
 * This agent yype only has the WandererTask in its list and should be used in
 * unit tests and as a reference when building more complex and domain specific
 * agent types.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public enum BasicTaskAgentType implements TaskAgentType {
        TYPE;

        private final String name = "BaseTaskAgent Type";
        private final List<Task> tasks;

        BasicTaskAgentType() {
                tasks = new ArrayList<Task>();
                tasks.add(new WandererTask());
        }

        @Override public List<Task> getTasks() { return tasks; }
        @Override public String getName() { return name; }
}
```

Listing 18: Ant.java

```
package com.luizabrahao.msc.ants.agent;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.ants.env.ChemicalCommStimulusType;
import com.luizabrahao.msc.ants.env.FoodSourceAgent;
import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;

/**
 * Defines the basic API that must be implemented by any agent that represent
 * a ant.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public interface Ant {
        /**
```

```
     * Each agent moves towards to a direction, this effectively means that the
     * ant has a bigger probability of selecting a neighbour node on that
     * direction than in any other.
     *
     * @return Direction direction the ant is moving towards to.
     */
    Direction getMovingDirection();

    /**
     * Changes the direction the agent is moving towards to.
     * @param movingDirection Direction the agent is more likely to follow.
     */
    void setMovingDirection(Direction movingDirection);

    /**
     * An agent has a map of ChemicalStimulusType and how much they increment
     * each stimulus they are sensitive to. This method looks up that list and
     * increment the node's current chemical stimulus accordingly to each agent
     * type.
     *
     * @param chemicalCommStimulusType Chemical stimulus that will be updated.
     */
    void incrementStimulusIntensity(ChemicalCommStimulusType chemicalCommStimulusType);

    /**
     * Collects food from a food source.
     *
     * @param foodSource Food source that the food will be collected from
     * @param amountToCollect Amount of food the agent will try to collect.
     * @return Double the amount of food the agent actually collected.
     */
    double collectFood(Agent foodSource, double amountToCollect);

    /**
     * Checks if the agent is caring food.
     * @return boolean True if the agent is caring food, false otherwise.
     */
    boolean isCarryingFood();

    /**
     * Each agent should have a short term memory of where the agent has been.
     * This method adds a new method to the agent's memory.
     *
     * @param node Node where the agent has visited.
     */
    void addToMemory(Node node);

    /**
     * Scans all agents in the current node, if any agent that is a food source
     * is found, it is returned. If there are more than one food sources in the
     * same node, the first to be found is returned. But it should not happen.
     * There is no reason to have two food sources agents in the same node, as
     * ther is no different types of food.
     *
     * @return FoodSourceAgent food source
     */
    FoodSourceAgent findFoodSource();

    /**
     * Returns the most recent node that the agent contains in his memory.
     *
     * @return Node Latest node visited.
     */
    Node getLatestNodeFromMemory();

    /**
     * Sometimes a agent needs to reinforce something that it has learned, a way
     * to do that is to deposit big quantities of chemical stimulus to tell it
     * fellow agents something. For example, when a worker is caring food, it
     * deposits more ForageStimulus than when it is not caring food. That
```

```
         * indirectively tells other ants that they are on the right path that leads
         * to a food source.
         *
         * So this method deposits the normal amount of stimulus multiplied by the
         * factor that is passed as parameter.
         *
         * @param chemicalCommStimulusType Stimulus to be deposited
         * @param factor Factor that the stimulus increment will be multiplied by
         */
        void incrementStimulusIntensityMultipliedByFactor(final ChemicalCommStimulusType chemicalCommStimulusType,
                int factor);

        /**
         * Inverts the agents moving direction. If the agent is moving NORTH it
         * should be moving SOUTH after this method is called. The same for all the
         * other directions:
         *
         * NORTH -> SOUTH
         * SOUTH -> NORTH
         * EAST -> WEST
         * WEST -> EAST
         */
        void invertDirection();

        /**
         * If an agent is caring food, it will be able to deposit the food is caring
         * in a nest.
         *
         * @param nest AntNestAgent Nest the food will be deposited.
         */
        void depositFood(AntNestAgent nest);
}
```

Listing 19: AntAgent.java

```
package com.luizabrahao.msc.ants.agent;

import java.util.LinkedList;
import java.util.Queue;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import net.jcip.annotations.GuardedBy;
import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.ants.env.ChemicalCommStimulusType;
import com.luizabrahao.msc.ants.env.FoodSourceAgent;
import com.luizabrahao.msc.ants.env.FoodSourceAgentType;
import com.luizabrahao.msc.ants.env.PheromoneNode;
import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.agent.TaskAgent;
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;

/**
 * Defines an agent that represents a ant. The movingDirection field represents
 * the direction the agent is moving in relation to the grid of nodes. This
 * direction is used when the algorithm is calculating the probabilities of the
 * agent selecting what node it will move next.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 * @see ForageTask
 *
 */
@ThreadSafe
public class AntAgent extends TaskAgent implements Ant {
```

```
    private static final Logger logger = LoggerFactory.getLogger(AntAgent.class);

    @GuardedBy("this") private Direction movingDirection;
    @GuardedBy("this") private Queue<Node> memory;
    private double amountOfFoodCarring = 0;

    @Override public synchronized Direction getMovingDirection() { return movingDirection; }
    @Override public synchronized void setMovingDirection(Direction movingDirection) { this.movingDirection =
        movingDirection; }

    public AntAgent(String id, AntType agentType, Node currentNode, boolean recordNodeHistory) {
        super(id, agentType, currentNode, recordNodeHistory);

        memory = new LinkedList<Node>();
        this.movingDirection = Direction.SOUTH;
    }

    @Override public AntType getAgentType() { return (AntType) super.getAgentType(); }

    @Override
    public Void call() {
        while (!Thread.currentThread().isInterrupted()) {
            this.getAgentType().execute(this);
        }

        logger.debug("{} is stoping...", this.getId());
        return null;
    }

    @Override
    public double collectFood(Agent foodSource, double amountToCollect) {
        this.amountOfFoodCarring = ((FoodSourceAgent) foodSource).collectFood(amountToCollect);

        return this.amountOfFoodCarring;
    }

    @Override public boolean isCarryingFood() { return (amountOfFoodCarring != 0) ? true : false; }

    @Override
    public void incrementStimulusIntensity(final ChemicalCommStimulusType chemicalCommStimulusType) {
        PheromoneNode currentNode = (PheromoneNode) this.getCurrentNode();
        this.updateNeighbour(currentNode, chemicalCommStimulusType, 0);

        // if the chemical stimulus is punctual, that is, it does not spread
        // across to its nodes neighbours we don't need to to anything else.
        if (chemicalCommStimulusType.getRadius() == 0) {
            return;
        }

        // if radius == 1, only main rows and columns will be updated, so let's
        // call to only them to be updated and return after that, to save
        // computational resources as much as we can
        if (chemicalCommStimulusType.getRadius() == 1) {
            this.updateNeighbours(chemicalCommStimulusType, Direction.NORTH);
            this.updateNeighbours(chemicalCommStimulusType, Direction.EAST);
            this.updateNeighbours(chemicalCommStimulusType, Direction.SOUTH);
            this.updateNeighbours(chemicalCommStimulusType, Direction.WEST);

            return;
        }

        // updates the main rows and columns
        this.updateNeighbours(chemicalCommStimulusType, Direction.NORTH);
        this.updateNeighbours(chemicalCommStimulusType, Direction.EAST);
        this.updateNeighbours(chemicalCommStimulusType, Direction.SOUTH);
        this.updateNeighbours(chemicalCommStimulusType, Direction.WEST);

        this.updateNeighbours(chemicalCommStimulusType, Direction.NORTH, Direction.EAST);
        this.updateNeighbours(chemicalCommStimulusType, Direction.NORTH, Direction.WEST);
        this.updateNeighbours(chemicalCommStimulusType, Direction.SOUTH, Direction.EAST);
```

```
            this.updateNeighbours(chemicalCommStimulusType, Direction.SOUTH, Direction.WEST);
        }

        @Override
        public void incrementStimulusIntensityMultipliedByFactor(final ChemicalCommStimulusType
            chemicalCommStimulusType, int factor) {
            for (int i = 0; i < factor; i++) {
                this.incrementStimulusIntensity(chemicalCommStimulusType);
            }
        }

        private void updateNeighbours(final ChemicalCommStimulusType chemicalCommStimulusType, final Direction
            direction) {
            PheromoneNode currentNode = (PheromoneNode) this.getCurrentNode();

            for (int i = 0; i < chemicalCommStimulusType.getRadius(); i++) {
                if (currentNode == null) {
                    break;
                }

                currentNode = (PheromoneNode) currentNode.getNeighbour(direction);

                if (currentNode != null) {
                    this.updateNeighbour(currentNode, chemicalCommStimulusType, i + 1);
                }
            }
        }

        private void updateNeighbours(final ChemicalCommStimulusType chemicalCommStimulusType, final Direction
            verticalDirection, final Direction horizontalDirection) {
            PheromoneNode currentLineNode = (PheromoneNode) this.getCurrentNode().getNeighbour(verticalDirection);
            PheromoneNode currentNode = currentLineNode;

            for (int i = 0; i < chemicalCommStimulusType.getRadius() - 1; i++) {
                for (int j = 0; j < chemicalCommStimulusType.getRadius() - 1; j++) {
                    if (currentNode == null) {
                        break;
                    }

                    currentNode = (PheromoneNode) currentNode.getNeighbour(horizontalDirection);

                    if (currentNode != null) {
                        if (i + j < chemicalCommStimulusType.getRadius() - 1) {
                            this.updateNeighbour(currentNode, chemicalCommStimulusType, i + j + 1);
                        }
                    }
                }

                if (currentNode == null) {
                    break;
                }

                currentLineNode = (PheromoneNode) currentLineNode.getNeighbour(verticalDirection);
                currentNode = currentLineNode;
            }
        }

        private void updateNeighbour(final PheromoneNode node, final ChemicalCommStimulusType chemicalCommStimulusType,
            final int distanceFromCurrentNode) {
            if (node == null) {
                return;
            }

            if (distanceFromCurrentNode == 0) {
                node.getCommunicationStimulus(chemicalCommStimulusType).increaseIntensity(this.getAgentType().
                    getStimulusIncrement(chemicalCommStimulusType.getName()));
                logger.trace("Node {} updated with {}", node.getId(), this.getAgentType().getStimulusIncrement(
                    chemicalCommStimulusType.getName()));

                return;
```

```
                }

                node.getCommunicationStimulus(chemicalCommStimulusType).increaseIntensity(this.getAgentType().
                    getStimulusIncrement(chemicalCommStimulusType.getName()) / distanceFromCurrentNode);
169             logger.trace("Node {} updated with {}", node.getId(), this.getAgentType().getStimulusIncrement(
                    chemicalCommStimulusType.getName()) / distanceFromCurrentNode);
        }

        @Override
        public void addToMemory(Node node) {
174             memory.add(node);

                if (memory.size() > this.getAgentType().getMemorySize()) {
                        memory.poll();
                }
179     }

        @Override
        public FoodSourceAgent findFoodSource() {
                synchronized (currentNode.getAgents()) {
184                     for (Agent agent : currentNode.getAgents()) {
                                if (agent.getAgentType() == FoodSourceAgentType.TYPE) {
                                        return (FoodSourceAgent) agent;
                                }
                        }
189             }

                return null;
        }

194     public Node getLatestNodeFromMemory() {
                return this.memory.poll();
        }

        @Override
199     public void invertDirection() {
                if (this.getMovingDirection() == Direction.SOUTH) {
                        this.setMovingDirection(Direction.NORTH);
                        return;
                }
204
                if (this.getMovingDirection() == Direction.NORTH) {
                        this.setMovingDirection(Direction.SOUTH);
                        return;
                }
209
                if (this.getMovingDirection() == Direction.EAST) {
                        this.setMovingDirection(Direction.WEST);
                        return;
                }
214
                if (this.getMovingDirection() == Direction.WEST) {
                        this.setMovingDirection(Direction.EAST);
                        return;
                }
219     }

        @Override
        public void depositFood(AntNestAgent nest) {
                nest.addPortionOfFood(this, this.amountOfFoodCarring);
224             this.amountOfFoodCarring = 0;
        }

}
```

Listing 20: AntType.java

```java
package com.luizabrahao.msc.ants.agent;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.agent.TaskAgentType;

/**
 * Formalises the public API for ant agents. Ants are implemented having limited
 * memory size and limited capability of collecting food. Different types of
 * ants have different memory sizes and can collect different amounts of food.
 * Also the different types of ant interact differently with the environment so
 * each type of ants define a list of chemical stimulus and how much of each of
 * them the agent is able to lay in every interaction.
 *
 * Ant agent type that implements this interface should be declared as
 * enumeration, therefore they are singletons by definition.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public interface AntType extends TaskAgentType {
        /**
         * Returns the amount of chemical stimulus that the ant type is enable of
         * depositing in each time.
         *
         * @param chemicalCommStimulusTypeName Name of the chemical communications
         * stimulus
         * @return amount that agent type is capable of deposit for that particular
         * chemical stimulus
         */
        double getStimulusIncrement(String chemicalCommStimulusTypeName);

        /**
         * Returns the amount of nodes the agent type is capable of remembering
         * being in.
         * @return amount of nodes agent is able to remember.
         */
        int getMemorySize();

        /**
         * In this method the agent type uses the agent's local context to decide
         * which task the agent will run next.
         *
         * @param agent Agent that will execute the task.
         */
        void execute(Agent agent);

        /**
         * Returns the amount of food the agent type is capable of collecting when
         * the agent finds a food source.
         *
         * @return Amount of food the agent is capable of collecting from a food
         *   source
         */
        double getAmountOfFoodCapableToCollect();
}
```

Listing 21: WorkerAntType.java

```java
package com.luizabrahao.msc.ants.agent;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
7
import net.jcip.annotations.ThreadSafe;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
12
import com.luizabrahao.msc.ants.env.WarningStimulusType;
import com.luizabrahao.msc.ants.env.ChemicalCommStimulus;
import com.luizabrahao.msc.ants.env.FoodSourceAgent;
import com.luizabrahao.msc.ants.env.ForageStimulusType;
17 import com.luizabrahao.msc.ants.task.FindAndHideInNest;
import com.luizabrahao.msc.ants.task.FindHomeTask;
import com.luizabrahao.msc.ants.task.ForageTask;
import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.task.Task;
22
/**
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
27 @ThreadSafe
public enum WorkerAntType implements AntType {
        TYPE;

        private static final Logger logger = LoggerFactory.getLogger(WorkerAntType.class);
32
        private static final String name = "type:ant:worker";
        private final List<Task> tasks;
        private final Map<String, Double> stimulusIncrementList;
        private static final int memorySize = 50;
37      private final double amountOfFoodCapableToCollect = 0.1;
        private static final long miliseconds ToWait = 5;
        private static final double warningThreshold = 0.5;

        WorkerAntType() {
42              tasks = new ArrayList<Task>();
                tasks.add(new ForageTask());
                tasks.add(new FindHomeTask());
                tasks.add(new FindAndHideInNest());

47              // TODO need to add Task to the end of FindAndHideInNest.

                stimulusIncrementList = new HashMap<String, Double>();
                stimulusIncrementList.put(ForageStimulusType.TYPE.getName(), 0.01);
                stimulusIncrementList.put(WarningStimulusType.TYPE.getName(), 0.05);
52      }

        @Override public String getName() { return name; }
        @Override public List<Task> getTasks() { return tasks; }
        @Override public int getMemorySize() { return memorySize; }
57      @Override public double getAmountOfFoodCapableToCollect() { return amountOfFoodCapableToCollect; }

        @Override
        public double getStimulusIncrement(String chemicalCommStimulusTypeName) {
                for (Map.Entry<String, Double> entry : stimulusIncrementList.entrySet()) {
62                      if (entry.getKey().equals(chemicalCommStimulusTypeName)) {
                                return entry.getValue();
                        }
                }

67              throw new RuntimeException("WorkerType does not have an increment declared for '" +
                      chemicalCommStimulusTypeName + "'");
        }

        @Override
        public void execute(Agent agent) {
72              AntAgent ant = (AntAgent) agent;

                if (ant.getCurrentTask() != null) {
                        // if it is trying to hide, just continue...
```

```
                        if (ant.getCurrentTask().getName().equals(FindAndHideInNest.NAME)) {
                                ant.getTaskByName(FindAndHideInNest.NAME).execute(agent);

                                this.waitSomeTime();
                                return;
                        }
                }

                ChemicalCommStimulus warningStimulus = (ChemicalCommStimulus) ant.getCurrentNode().
                     getCommunicationStimulus(WarningStimulusType.TYPE);

                if ((warningStimulus != null) && (warningStimulus.getIntensity() > warningThreshold)) {
                        // if the ant is caring food it is likely it already is moving
                        // towards the nest, so there is no need to invert its direction.
                        if (!ant.isCarryingFood()) {
                                ant.invertDirection();
                        }

                        ant.setCurrentTask(ant.getTaskByName(FindAndHideInNest.NAME));
                        logger.debug("{} has switched to {}", agent.getId(), FindAndHideInNest.NAME);
                        ant.setCurrentTask(FindAndHideInNest.NAME);
                        ant.getTaskByName(FindAndHideInNest.NAME).execute(agent);

                        this.waitSomeTime();
                        return;
                }


                FoodSourceAgent foodSource = ant.findFoodSource();

                if ((foodSource != null) && (!ant.isCarryingFood())) {
                        ant.collectFood(foodSource, amountOfFoodCapableToCollect);
                        logger.debug("{} found a source food and will try to collect food.", agent.getId());

                        if (ant.isCarryingFood()) {
                                ant.invertDirection();
                        }
                }

                if (!ant.isCarryingFood()) {
                        ant.setCurrentTask(ForageTask.NAME);
                        ant.getTaskByName(ForageTask.NAME).execute(agent);
                } else {
                        ant.setCurrentTask(FindHomeTask.NAME);
                        ant.getTaskByName(FindHomeTask.NAME).execute(agent);
                }

                this.waitSomeTime();
        }

        private void waitSomeTime() {
                try {
                        Thread.sleep(milisecondsToWait);
                } catch (InterruptedException e) {
                        // don't need to do anything...
                }
        }
}
```

Listing 22: AntNestType.java

```
package com.luizabrahao.msc.ants.agent;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.agent.AgentType;
```

```
/**
 * Define the type of agent that identify ant nests.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public enum AntNestType implements AgentType {
        TYPE;

        private static final String name = "ant:agent:nest";

        @Override public String getName() { return name; }
}
```

Listing 23: AntNestAgent.java

```
package com.luizabrahao.msc.ants.agent;

import net.jcip.annotations.GuardedBy;
import net.jcip.annotations.ThreadSafe;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.model.agent.AbstractAgent;
import com.luizabrahao.msc.model.env.Node;


/**
 * Objects of this class represent ant nests. They have a special property named
 * amountOfFoodHeld, which contains the total amount of food that is held in a
 * particular nest.
 *
 * This agent should not be executed as a Java task. If a user tries to do so, a
 * RuntimeExecption is thrown.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public class AntNestAgent extends AbstractAgent {
        private static final Logger logger = LoggerFactory.getLogger(AntNestAgent.class);
        @GuardedBy("this") private double amountOfFoodHeld = 0;

        public AntNestAgent(String id, Node currentNode) {
                super(id, AntNestType.TYPE, currentNode, false);
        }

        public synchronized double getAmountOfFoodHeld() { return amountOfFoodHeld; }

        @Override
        public Void call() throws Exception {
                throw new RuntimeException("Nests are not to be used as threads... " +
                                "They just take advantage of the infrastructure of agents");
        }

        public void addPortionOfFood(final AntAgent agent, final double portion) {
                synchronized (this) {
                        amountOfFoodHeld = amountOfFoodHeld + portion;
                }

                logger.debug(this.getId() + ": {} deposited {} of food.", agent.getId(), portion);
        }
}
```

Listing 24: StaticPheromoneUpdaterAgentType.java

```
package com.luizabrahao.msc.ants.agent;

import java.util.ArrayList;
import java.util.List;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.ants.task.StaticPheromoneUpdateTask;
import com.luizabrahao.msc.model.agent.TaskAgentType;
import com.luizabrahao.msc.model.task.Task;

/**
 * Represents an agent that triggers the pheromone update method of the nodes
 * that compose a grid. As required it implements the singleton pattern.
 *
 * Workers have the following tasks:
 * — StaticPheromoneUpdateTask
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public enum StaticPheromoneUpdaterAgentType implements TaskAgentType {
        TYPE;

        private static final String name = "type:helper:pheromone—updater";
        private final List<Task> tasks;

        StaticPheromoneUpdaterAgentType() {
                tasks = new ArrayList<Task>();
                tasks.add(new StaticPheromoneUpdateTask());
        }

        @Override public String getName() { return name; }
        @Override public List<Task> getTasks() { return tasks; }
}
```

Listing 25: StaticPheromoneUpdaterAgent.java

```
package com.luizabrahao.msc.ants.agent;

import net.jcip.annotations.ThreadSafe;

import com.luizabrahao.msc.model.agent.TaskAgent;
import com.luizabrahao.msc.model.env.Node;

/**
 * This agent sweeps a grid, starting from its current node, it triggers the
 * pheromone update method for each node than it moves to the next node
 * (towards east) in the same line until it reaches the end of that line, when
 * that happens it moved to the next line and the process is repeated until it
 * reaches the last line of the grid or the number of lines the updater has
 * processed is the number of lines to be processed defined by the variable
 * numberOfLinesToProcess.
 *
 * The reason to have the numberOfLinesToProcess is that in that way we can
 * create more than one updater for the same grid, and assign slices of the
 * grid for each updater.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@ThreadSafe
public class StaticPheromoneUpdaterAgent extends TaskAgent {
        private final int numberOfLinesToProcess;
```

```
        public StaticPheromoneUpdaterAgent(String id, Node currentNode, int numberOfLinesToProcess) {
29              super(id, StaticPheromoneUpdaterAgentType.TYPE, currentNode, false);

                this.numberOfLinesToProcess = numberOfLinesToProcess;
        }

34      public int getNumberOfLinesToProcess() { return numberOfLinesToProcess; }

        @Override
        public Void call() throws Exception {
                this.getTaskList().get(0).execute(this);
39
                return null;
        }
}
```

B.2.3 *Task*

Listing 26: Task.java

```
package com.luizabrahao.msc.model.task;

3   import com.luizabrahao.msc.model.agent.Agent;

public interface Task {
        /**
         * Each task type has a name, an identifier that is unique and is used to
8        * tell one task to another when it is needed.
         *
         * @return String task type identifier.
         */
        String getName();
13
        /**
         * Executes a set of instruction that defines the agent's behaviour when
         * executing the task.
         *
18       * @param agent Agent that will perform the task.
         */
        void execute(Agent agent);
}
```

Listing 27: AbstractJava.java

```
package com.luizabrahao.msc.model.task;

import net.jcip.annotations.Immutable;
4
@Immutable
public abstract class AbstractTask implements Task {
        protected final String name;

9       public AbstractTask(String id) {
                this.name = id;
        }

        @Override
14      public String getName() { return name; }

        @Override
```

```
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + ((name == null) ? 0 : name.hashCode());
                return result;
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                AbstractTask other = (AbstractTask) obj;
                if (name == null) {
                        if (other.name != null)
                                return false;
                } else if (!name.equals(other.name))
                        return false;
                return true;
        }
}
```

Listing 28: WandererTask.java

```
package com.luizabrahao.msc.model.task;


import java.util.Random;

import net.jcip.annotations.Immutable;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.env.Direction;

/**
 * A simple implementation of a task. When executing this task the agent
 * wanders the environment without worrying about anything else. The task picks
 * up a random direction and the agent moves to that neighbour node.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@Immutable
public class WandererTask extends AbstractTask {
        private static final Logger logger = LoggerFactory.getLogger(WandererTask.class);
        public static final String NAME = "Wanderer";
        private static final Random rand = new Random();

        public WandererTask() {
                super(WandererTask.NAME);
        }

        @Override
        public void execute(Agent agent) {
                logger.info("agent {} started task: {}", agent.getId(), WandererTask.NAME);

                while (true) {
                        Direction directionToMove = WandererTask.getRandomDirection(agent);
                        agent.getCurrentNode().getNeighbour(directionToMove).addAgent(agent);
                }
        }
```

```
        /**
         * This method returns one of the neighbours of the agent's current node.
         * It chooses one direction at random, if there is no neighbour on that
         * direction, it tries again until one is found.
         *
         * Note that his method would fall into a deadlock if the agent's current
         * node is not connected to any other node.
         *
         * @param agent Agent that is performing the task
         * @return Node that the agent is going to move to.
         */
        public static Direction getRandomDirection(Agent agent) {
                Logger logger = LoggerFactory.getLogger(WandererTask.class);

                // 4 directions
                int randomDirection = WandererTask.rand.nextInt(4);
                Direction direction = null;

                switch (randomDirection) {
                        case 0:
                                direction = Direction.NORTH;
                                break;

                        case 1:
                                direction = Direction.EAST;
                                break;

                        case 2:
                                direction = Direction.SOUTH;
                                break;

                        case 3:
                                direction = Direction.WEST;
                                break;
                }

                return direction;
        }
}
```

Listing 29: AntTask.java

```
package com.luizabrahao.msc.ants.task;

public interface AntTask {
        double getNeighbourWeightNorth();
        double getNeighbourWeightEast();
        double getNeighbourWeightSouth();
        double getNeighbourWeightWest();

}
```

Listing 30: ForageTask.java

```
package com.luizabrahao.msc.ants.task;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.ants.agent.AntAgent;
import com.luizabrahao.msc.ants.env.ForageStimulusType;
import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.env.Direction;
```

```
import com.luizabrahao.msc.model.env.Node;
import com.luizabrahao.msc.model.task.AbstractTask;

public class ForageTask extends AbstractTask implements AntTask {
	private static final Logger logger = LoggerFactory.getLogger(ForageTask.class);
	public static final String NAME = "ant:task:forage";

	public static final double WEIGHT_NORTH = 0.1;
	public static final double WEIGHT_EAST = 0;
	public static final double WEIGHT_SOUTH = 0;
	public static final double WEIGHT_WEST = 0.0;

	public ForageTask() {
		super(ForageTask.NAME);
	}

	@Override public double getNeighbourWeightNorth() { return WEIGHT_NORTH; }
	@Override public double getNeighbourWeightEast() { return WEIGHT_EAST; }
	@Override public double getNeighbourWeightSouth() { return WEIGHT_SOUTH; }
	@Override public double getNeighbourWeightWest() { return WEIGHT_WEST; }

	@Override
	public void execute(Agent agent) {
		AntAgent ant = (AntAgent) agent;

		Direction d = AntTaskUtil.getDirectionToMoveTo(ant, ForageStimulusType.TYPE);
		Node nodeToMoveTo = agent.getCurrentNode().getNeighbour(d);

		if (nodeToMoveTo == null) {
			Direction newDirection = this.findRandomDirectionToMove(ant);
			nodeToMoveTo = ant.getCurrentNode().getNeighbour(newDirection);
			ant.setMovingDirection(newDirection);
		}

		ant.incrementStimulusIntensity(ForageStimulusType.TYPE);

		ant.addToMemory(ant.getCurrentNode());
		nodeToMoveTo.addAgent(agent);
	}

	private Direction findRandomDirectionToMove(AntAgent agent) {
		Direction d = AntTaskUtil.getDirectionToMoveTo(agent, ForageStimulusType.TYPE);

		Node n = agent.getCurrentNode().getNeighbour(d);

		if (n == null) {
			return findRandomDirectionToMove(agent);
		}

		logger.debug("{} has changed its direction to {}", agent.getId(), d);

		return d;
	}
}
```

Listing 31: FindHomeTask.java

```
package com.luizabrahao.msc.ants.task;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.ants.agent.AntAgent;
import com.luizabrahao.msc.ants.agent.AntNestAgent;
import com.luizabrahao.msc.ants.agent.AntNestType;
import com.luizabrahao.msc.ants.env.ForageStimulusType;
import com.luizabrahao.msc.model.agent.Agent;
```

```
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;
import com.luizabrahao.msc.model.task.AbstractTask;

public class FindHomeTask extends AbstractTask implements AntTask {
	private static final Logger logger = LoggerFactory.getLogger(FindHomeTask.class);
	public static final String NAME = "ant:task:find_home";

	public static final double WEIGHT_NORTH = 0.40;
	public static final double WEIGHT_EAST = 0.25;
	public static final double WEIGHT_SOUTH = 0.10;
	public static final double WEIGHT_WEST = 0.25;

	public FindHomeTask() {
		super(FindHomeTask.NAME);
	}

	@Override public double getNeighbourWeightNorth() { return WEIGHT_NORTH; }
	@Override public double getNeighbourWeightEast() { return WEIGHT_EAST; }
	@Override public double getNeighbourWeightSouth() { return WEIGHT_SOUTH; }
	@Override public double getNeighbourWeightWest() { return WEIGHT_WEST; }

	private AntNestAgent getNest(Node node) {
		synchronized (node.getAgents()) {
			for (Agent agent : node.getAgents()) {
				if (agent.getAgentType() == AntNestType.TYPE) {
					return (AntNestAgent) agent;
				}
			}
		}

		return null;
	}

	@Override
	public void execute(Agent agent) {
		AntAgent ant = (AntAgent) agent;
		AntNestAgent nest = this.getNest(agent.getCurrentNode());

		if (nest != null) {
			// The agent has reached nest... Do something...
			logger.debug("{} deposited food in the nest", agent.getId());
			ant.depositFood(nest);
			ant.invertDirection();

			return;
		}

		Node nodeToMoveTo = ant.getLatestNodeFromMemory();

		// if the agent runs out of memory
		if (nodeToMoveTo == null) {
			Direction d = AntTaskUtil.getDirectionToMoveTo(ant, ForageStimulusType.TYPE);
			nodeToMoveTo = agent.getCurrentNode().getNeighbour(d);

			if (nodeToMoveTo == null) {
				Direction newDirection = this.findRandomDirectionToMove(ant);
				nodeToMoveTo = ant.getCurrentNode().getNeighbour(newDirection);
				ant.setMovingDirection(newDirection);
			}
		}

		ant.incrementStimulusIntensityMultipliedByFactor(ForageStimulusType.TYPE, 2);
		nodeToMoveTo.addAgent(agent);
	}

	private Direction findRandomDirectionToMove(AntAgent agent) {
		Direction d = AntTaskUtil.getDirectionToMoveTo(agent, ForageStimulusType.TYPE);

		Node n = agent.getCurrentNode().getNeighbour(d);
```

```
82                  if (n == null) {
                            return findRandomDirectionToMove(agent);
                    }

                    logger.debug("{} has changed its direction to {}", agent.getId(), d);
87
                    return d;
            }
    }
```

Listing 32: FinAndHideInNest.java

```
    package com.luizabrahao.msc.ants.task;

    import org.slf4j.Logger;
    import org.slf4j.LoggerFactory;
5
    import com.luizabrahao.msc.ants.agent.AntAgent;
    import com.luizabrahao.msc.ants.agent.AntNestAgent;
    import com.luizabrahao.msc.ants.agent.AntNestType;
    import com.luizabrahao.msc.ants.env.ForageStimulusType;
10  import com.luizabrahao.msc.model.agent.Agent;
    import com.luizabrahao.msc.model.env.Direction;
    import com.luizabrahao.msc.model.env.Node;
    import com.luizabrahao.msc.model.task.AbstractTask;

15  public class FindAndHideInNest extends AbstractTask implements AntTask {
            private static final Logger logger = LoggerFactory.getLogger(FindAndHideInNest.class);
            public static final String NAME = "ant:task:find—home—and—hide";

            public static final double WEIGHT_NORTH = 0.40;
20          public static final double WEIGHT_EAST = 0.25;
            public static final double WEIGHT_SOUTH = 0.10;
            public static final double WEIGHT_WEST = 0.25;

            public FindAndHideInNest() {
25                  super(FindAndHideInNest.NAME);
            }

            @Override public double getNeighbourWeightNorth() { return WEIGHT_NORTH; }
            @Override public double getNeighbourWeightEast() { return WEIGHT_EAST; }
30          @Override public double getNeighbourWeightSouth() { return WEIGHT_SOUTH; }
            @Override public double getNeighbourWeightWest() { return WEIGHT_WEST; }

            private AntNestAgent getNest(Node node) {
                    synchronized (node.getAgents()) {
35                          for (Agent agent : node.getAgents()) {
                                    if (agent.getAgentType() == AntNestType.TYPE) {
                                            return (AntNestAgent) agent;
                                    }
                            }
40                  }

                    return null;
            }

45          @Override
            public void execute(Agent agent) {
                    AntAgent ant = (AntAgent) agent;
                    ant.setCurrentTask(this.name);

50                  AntNestAgent nest = this.getNest(agent.getCurrentNode());

                    if (nest != null) {
                            // The agent has reached nest... Do something...
                            logger.debug("{} is hidding in the nest", agent.getId());
```

```
55                      return;
                }

                Node nodeToMoveTo = ant.getLatestNodeFromMemory();

60              // if the agent runs out of memory
                if (nodeToMoveTo == null) {
                        Direction d = AntTaskUtil.getDirectionToMoveTo(ant, ForageStimulusType.TYPE);
                        nodeToMoveTo = agent.getCurrentNode().getNeighbour(d);

65                      if (nodeToMoveTo == null) {
                                Direction newDirection = this.findRandomDirectionToMove(ant);
                                nodeToMoveTo = ant.getCurrentNode().getNeighbour(newDirection);
                                ant.setMovingDirection(newDirection);
                        }
70              }

                ant.incrementStimulusIntensityMultipliedByFactor(ForageStimulusType.TYPE, 2);
                nodeToMoveTo.addAgent(agent);
        }
75
        private Direction findRandomDirectionToMove(AntAgent agent) {
                Direction d = AntTaskUtil.getDirectionToMoveTo(agent, ForageStimulusType.TYPE);

                Node n = agent.getCurrentNode().getNeighbour(d);
80
                if (n == null) {
                        return findRandomDirectionToMove(agent);
                }

85              logger.debug("{} has changed its direction to {}", agent.getId(), d);

                return d;
        }
}
```

Listing 33: AntTaskUtil.java

```
1   package com.luizabrahao.msc.ants.task;

    import com.luizabrahao.msc.ants.agent.AntAgent;
    import com.luizabrahao.msc.ants.env.ChemicalCommStimulus;
    import com.luizabrahao.msc.ants.env.ChemicalCommStimulusType;
6   import com.luizabrahao.msc.model.agent.Agent;
    import com.luizabrahao.msc.model.env.Direction;
    import com.luizabrahao.msc.model.env.Node;
    import com.luizabrahao.msc.model.task.WandererTask;

11  public class AntTaskUtil {


        public static double getNeighbourForagePheromone(Agent agent, Direction direction, ChemicalCommStimulusType
              chemicalCommStimulusType) {
                Node n = agent.getCurrentNode().getNeighbour(direction);
16
                if (n == null) {
                return 0;

            } else {
21              ChemicalCommStimulus foragePheromone = (ChemicalCommStimulus) n.getCommunicationStimulus(
                      chemicalCommStimulusType);

                if (foragePheromone == null) {
                        return 0;
                }
26
                return foragePheromone.getIntensity();
```

```
        }
    }

    /**
     * Uses chemical stimulus to decide which node the agent should move next.
     * Each neighbour node has a weight, which is multiplied by the node's
     * stimulus' intensity. The resulting rate WEIGHT * INTENSITY is normalised
     * by the sum of the rates to find the proportional probability of a node
     * being picked.
     *
     * This is necessary to add a stochastic behaviour to the agent, simulating
     * external forces like wind that can sometimes take agents away from the
     * optimal path.
     *
     * @param agent Agent that is going to move.
     * @return Node to move to.
     */
    public static Direction getDirectionToMoveTo(AntAgent agent, ChemicalCommStimulusType
        chemicalCommStimulusType,
                                                  double weightNort, double weightEast, double
                                                      weightSouth, double weightWest) {

        double pheromoneNorth = 0;
        double pheromoneEast = 0;
        double pheromoneSouth = 0;
        double pheromoneWest = 0;

        if (agent.getMovingDirection() == null) {
            agent.setMovingDirection(Direction.SOUTH);
        }

        synchronized (agent.getCurrentNode()) {
            // First we need to work out what are the neighbours nodes in
            // relation to the agent's current movement.
            Direction northOfTheAgent = Direction.NORTH;
            Direction eastOfTheAgent = Direction.EAST;
            Direction southOfTheAgent = Direction.SOUTH;
            Direction westOfTheAgent = Direction.WEST;

            // Transformation of direction from being related to the
            // grid to being related to the agent's moving direction.
            // If the agent is moving north, nothing is needed to be done.
            if (agent.getMovingDirection() == Direction.EAST) {
                northOfTheAgent = Direction.EAST;
                eastOfTheAgent = Direction.SOUTH;
                southOfTheAgent = Direction.WEST;
                westOfTheAgent = Direction.NORTH;

            } else if (agent.getMovingDirection() == Direction.SOUTH) {
                northOfTheAgent = Direction.SOUTH;
                eastOfTheAgent = Direction.WEST;
                southOfTheAgent = Direction.NORTH;
                westOfTheAgent = Direction.EAST;

            } else if (agent.getMovingDirection() == Direction.WEST) {
                northOfTheAgent = Direction.WEST;
                eastOfTheAgent = Direction.NORTH;
                southOfTheAgent = Direction.EAST;
                westOfTheAgent = Direction.SOUTH;
            }

            // Get the intensity of the pheromone of the neighbour nodes. The
            // directions are all in relation to the agent movement direction
            // and not to the grid.
            pheromoneNorth = AntTaskUtil.getNeighbourForagePheromone(agent, northOfTheAgent,
                chemicalCommStimulusType);
            pheromoneEast = AntTaskUtil.getNeighbourForagePheromone(agent, eastOfTheAgent,
                chemicalCommStimulusType);
            pheromoneSouth = AntTaskUtil.getNeighbourForagePheromone(agent, southOfTheAgent,
                chemicalCommStimulusType);
```

```
                        pheromoneWest = AntTaskUtil.getNeighbourForagePheromone(agent, westOfTheAgent,
                              chemicalCommStimulusType);

                        double rateNorth = pheromoneNorth * weightNort;
                        double rateEast = pheromoneEast * weightEast;
                        double rateSouth = pheromoneSouth * weightSouth;
                        double rateWest = pheromoneWest * weightWest;

                        final double sumRates = rateNorth + rateEast + rateSouth + rateWest;

                        rateNorth = rateNorth / sumRates;
                        rateEast = rateEast / sumRates;
                        rateSouth = rateSouth / sumRates;
                        rateWest = rateWest / sumRates;

                        final double randomPoint = Math.random();

                        if (rateNorth >= randomPoint) {
                                return northOfTheAgent;
                        }

                        if ((rateNorth < randomPoint) && (rateEast >= randomPoint)) {
                                return eastOfTheAgent;
                        }

                        if ((rateEast < randomPoint) && (rateSouth >= randomPoint)) {
                                return southOfTheAgent;
                        }

                        if ((rateSouth < randomPoint) && (rateWest >= randomPoint)) {
                                return westOfTheAgent;
                        }

                        return WandererTask.getRandomDirection(agent);
                }
        }

        public static Direction getDirectionToMoveTo(AntAgent agent, ChemicalCommStimulusType
              chemicalCommStimulusType) {
                return AntTaskUtil.getDirectionToMoveTo(agent, chemicalCommStimulusType, ForageTask.WEIGHT_NORTH,
                      ForageTask.WEIGHT_EAST, ForageTask.WEIGHT_SOUTH, ForageTask.WEIGHT_WEST);
        }

}
```

Listing 34: StaticPheromoneUpdateTask.java

```
package com.luizabrahao.msc.ants.task;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.ants.agent.StaticPheromoneUpdaterAgent;
import com.luizabrahao.msc.ants.env.ChemicalCommStimulus;
import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.env.CommunicationStimulus;
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;
import com.luizabrahao.msc.model.task.AbstractTask;

/**
 * Starting from its current node, it triggers the pheromone update method for
 * each node than it moves to the next node (towards east) in the same line
 * until it reaches the end of that line, when that happens it moved to the
 * next line and the process is repeated until it reaches the last line of the
 * grid or the number of lines the updater has processed is the number of lines
 * to be processed defined by the agents' field numberOfLinesToProcess.
```

```
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
public class StaticPheromoneUpdateTask extends AbstractTask {
	private static final Logger logger = LoggerFactory.getLogger(StaticPheromoneUpdateTask.class);
	public static final String NAME = "Static updater";

	public StaticPheromoneUpdateTask() {
		super(StaticPheromoneUpdateTask.NAME);
	}

	@Override
	public void execute(Agent agent) {
		Node initialNode = agent.getCurrentNode();

		logger.debug("[{}] Starting pherormone update run", agent.getId());
		final int maximumNumberOfLines = ((StaticPheromoneUpdaterAgent) agent).getNumberOfLinesToProcess();

		Node startNode = agent.getCurrentNode();
		Node nodeToBeUpdated = startNode;
		int numberOfRowsProcessed = 0;

		for (;;) {
			for (;;) {

				for (CommunicationStimulus stimulus : nodeToBeUpdated.getCommunicationStimuli()) {
					if (stimulus instanceof ChemicalCommStimulus) {
						((ChemicalCommStimulus) stimulus).decayIntensity();
					}
				}

				if (nodeToBeUpdated.getNeighbour(Direction.EAST) != null) {
					nodeToBeUpdated = nodeToBeUpdated.getNeighbour(Direction.EAST);

				} else {
					break;
				}
			}

			if (startNode.getNeighbour(Direction.SOUTH) != null) {
				startNode = startNode.getNeighbour(Direction.SOUTH);
				nodeToBeUpdated = startNode;

			} else {
				break;
			}

			numberOfRowsProcessed++;

			if (numberOfRowsProcessed == maximumNumberOfLines) {
				break;
			}
		}

		agent.setCurrentNode(initialNode);
		logger.debug("[{}] Finished pherormone update run", agent.getId());
	}
}
```

B.2.4 *Annotations*

Listing 35: FrameworkExclusive.java

```
1
package com.luizabrahao.msc.model.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
6 import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
11 * Tells users that the annotated method must not be used in their code, that is
 * it is of exclusive use of the framework itself.
 *
 * If you call any method that has this annotation in your code, you are doing
 * something wrong.
16 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@Documented
21 @Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface FrameworkExclusive {}
```

Listing 36: PseudoThreadSafe.java

```
/**
2 *
 */
package com.luizabrahao.msc.model.annotation;

import java.lang.annotation.Documented;
7 import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

12 /**
 * The class to which this annotation is applied is considered to be thread−safe
 * at runtime, but it contains methods that are not thread safe. For example,
 * for a performance optimisation point of view, a method that is likely to be
 * called all the time would gain from not having to synchronise as long as its
17 * users know the risks they are taking.
 *
 * Example of ussage: BasicNode
 *
 * @see BasicNode
22 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@Documented
27 @Target(value = ElementType.TYPE)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface PseudoThreadSafe {}
```

Listing 37: ThreadSafetyBreaker.java

```
1 package com.luizabrahao.msc.model.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;

/**
 * Indicates that the method breaks thread−safety. It is to be used with
 * PseudoThreadSafe annotation. This is useful to document what methods of the
 * class cause it to not be thread−safe.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
@Documented
@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface ThreadSafetyBreaker {}
```

B.2.5 *Utility*

Listing 38: EnvironmentFactory.java

```
package com.luizabrahao.msc.model.env;

/**
 * The EnvironmentFactory class is an utility class that holds the methods
 * to generate environments to be used for the simulation.
 *
 * @author Luiz Abrahao <luiz@luizabrahao.com>
 *
 */
public class EnvironmentFactory {

	private EnvironmentFactory() {}

	/**
	 * Initialises an environment based on BasicNode objects. This environment
	 * has rectangular shape and each node are assigned an identifier following
	 * the pattern: "n+lineNumber,colunmNumber", e.g. "n3,2" corresponds to the
	 * node at the third line and second column.
	 *
	 * @param nLines number of lines the grid will contain
	 * @param nColunms number of column the grid will contain.
	 * @return Node[][] A two dimensional array of interconnected BasicNode objects.
	 */
	public static Node[][] createBasicNodeGrid(int nLines, int nColunms) {
		Node[][] nodes = new Node[nLines][nColunms];

		for(int l = 0; l < nLines; l++) {
			for (int c = 0; c < nColunms; c++) {
				nodes[l][c] = new BasicNode("n" + l + "," + c);

				if (c != 0) {
					nodes[l][c].setNeighbours(Direction.WEST, nodes[l][c − 1]);
				}

				if (l != 0) {
					nodes[l][c].setNeighbours(Direction.NORTH, nodes[l − 1][c]);
				}
			}
		}

		return nodes;
	}
}
```

Listing 39: AntEnvironmentFactory.java

```
package com.luizabrahao.msc.ants.env;

import java.util.ArrayList;
import java.util.List;

import com.luizabrahao.msc.ants.agent.AntNestAgent;
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;

public class AntEnvironmentFactory {
	private AntEnvironmentFactory() {}

	/**
	 * Initialises an environment based on PheromoneNode objects.
	 * This environment has rectangular shape and each node are assigned an
	 * identifier following the pattern: "n+lineNumber,colunmNumber",
	 * e.g. "n3,2" corresponds to the node at the third line and second column.
	 *
	 * @param nLines number of lines the grid will contain
	 * @param nColunms number of column the grid will contain.
	 * @return Pheromone[][] A two dimensional array of interconnected
	 *   PheromoneNode objects.
	 */
	public static PheromoneNode[][] createPheromoneNodeGrid(int nLines, int nColumns) {
		PheromoneNode[][] nodes = new PheromoneNode[nLines][nColumns];

		for(int l = 0; l < nLines; l++) {
			for (int c = 0; c < nColumns; c++) {
				nodes[l][c] = new PheromoneNode("n" + l + "," + c);

				if (c != 0) {
					nodes[l][c].setNeighbours(Direction.WEST, nodes[l][c - 1]);
				}

				if (l != 0) {
					nodes[l][c].setNeighbours(Direction.NORTH, nodes[l - 1][c]);
				}
			}
		}

		return nodes;
	}


	public static List<FoodSourceAgent> placeRowOfFoodSources(Node initialNode, int numberOfSources, double
	    amountOfFoodInEachSource) {
		List<FoodSourceAgent> foodSources = new ArrayList<FoodSourceAgent>();
		Node currentNode = initialNode;

		for (int i = 0; i < numberOfSources; i++) {
			foodSources.add(new FoodSourceAgent("food-source-" + i, currentNode, amountOfFoodInEachSource
			    ));

			currentNode = currentNode.getNeighbour(Direction.EAST);

			if (currentNode == null) {
				break;
			}
		}

		return foodSources;
	}

	public static double sumFoodCollected(List<AntNestAgent> nests) {
		double result = 0;

		for (AntNestAgent nest : nests) {
			result = result + nest.getAmountOfFoodHeld();
```

```
67                  }

                    return result;
            }
    }
```

Listing 40: AntAgentFactory.java

```
    package com.luizabrahao.msc.ants.agent;

    import java.util.ArrayList;
4   import java.util.List;

    import net.jcip.annotations.ThreadSafe;

    import org.slf4j.Logger;
9   import org.slf4j.LoggerFactory;

    import com.luizabrahao.msc.model.env.Node;

    /**
14   * Utility class for creating populations of agents.
     *
     * @author Luiz Abrahao <luiz@luizabrahao.com>
     *
     */
19  @ThreadSafe
    public class AntAgentFactory {
            // avoid any other class to extend this one.
            private AntAgentFactory() {}

24          private static final Logger logger = LoggerFactory.getLogger(AntAgentFactory.class);

            /**
             * Sometimes it is necessary to have a reasonable large number of agents to
             * use in simulations. This method helps users to create populations of
29           * ants of the Worker type. Note that the entire population will be placed
             * at the same node.
             *
             * @param numberOfAgents The size of the population to be generated
             * @param namePrefix Each agent will receive a id that starts with the
34           * prefix plus a number, e.g. 'prefix-01'.
             * @param initialNode The node the population is going to be placed at.
             * @return List of ant agents.
             */
            public static List<AntAgent> produceBunchOfWorkers(final int numberOfAgents, final String namePrefix, final Node
                  initialNode) {
39                  List<AntAgent> agents = new ArrayList<AntAgent>();

                    for (int i = 0; i < numberOfAgents; i++) {
                            agents.add(new AntAgent(namePrefix + "-" + i, WorkerAntType.TYPE, initialNode, false));
                    }
44
                    logger.debug("{} agents created with prefix '{}'", agents.size(), namePrefix);
                    return agents;
            }

49          /**
             * In cases that the population produced should be sprang across a line of
             * the grid, this method allows the population to be initialised in chucks
             * spaced by the parameter horizonalSpacing.
             *
54           * Note that if the number of agents is not a multiple of the number of
             * agents per node, a rounded number of agents will be return.
             *
             * @param numberOfAgents Size of the population
             * @param namePrefix namePrefix Each agent will receive a id that starts
```

```
59          * with the prefix plus a number, e.g. 'prefix-01'.
            * @param grid Node grid
            * @param line The index of the line of the grid that the population will
            * be placed at
            * @param startColumn The index of the column that the population will
64          * start to be placed.
            * @param horizontalSpacing The number of nodes that separates the chucks
            * of the population
            * @param numberOfAgentsPerNode Number of agents at each chuck
            * @return List of agents.
69          */
            public static List<AntAgent> produceBunchOfWorkers(final int numberOfAgents, final String namePrefix, final Node[][]
                grid, final int line,
                                                               final int startColumn, final int horizontalSpacing, final int
                                                                   numberOfAgentsPerNode) {
              List<AntAgent> agents = new ArrayList<AntAgent>();
              int interations = numberOfAgents / numberOfAgentsPerNode;
74            int columm = startColumn;
              int nAgentsCreated = 0;

              // let's check if the grid has enough columns to allocate the population
              // nOfInterations = nOfAgents / nOfAgents per node, rounded
79            // nGaps = nOfInterations - 1, number of gaps of size horizontalSpacing
              // numberOfColumnsNeeded = nOfInterations + nGaps * gapSize
              // = nOfInterations + (nOfInterations - 1) * gapSize
              // = nOfInterations ( 1 + gapSize) - gapSize
              //
84            // index of last column needed = startColumn + numberOfColumnsNeeded - 1;
              final int numberOfColumnsNeeded = interations * (1 + horizontalSpacing) - horizontalSpacing;
              final int lastColumnIndexNeeded = startColumn + numberOfColumnsNeeded - 1;

              try {
89                    grid[line][lastColumnIndexNeeded].getId();

              } catch (ArrayIndexOutOfBoundsException e) {
                    throw new RuntimeException("The grid has not enough cloumns to create the population! Index of last
                        column needed =" + lastColumnIndexNeeded);
              }
94
              for (int i = 0; i < interations; i++) {
                    for (int j = 0; j < numberOfAgentsPerNode; j++) {
                          agents.add(new AntAgent(namePrefix + "-" + nAgentsCreated, WorkerAntType.TYPE, grid[line
                              ][columm], false));
                          nAgentsCreated++;
99                  }
                    columm += horizontalSpacing + 1;
              }

              logger.info("{} agents created with prefix '{}'", agents.size(), namePrefix);
104           return agents;
            }

            /**
             * Sometimes it is necessary to have a reasonable large number of agents to
109          * use in simulations. This method helps users to create populations of
             * ants of the Worker type. Note that the entire population will be placed
             * at the same node.
             *
             * @param numberOfAgents The size of the population to be generated
114          * @param namePrefix Each agent will receive a id that starts with the
             * prefix plus a number, e.g. 'prefix-01'.
             * @param initialNode The node the population is going to be placed at.
             * @return List of ant agents.
             */
119         public static List<AntAgent> produceBunchOfTactileWorkers(final int numberOfAgents, final String namePrefix, final
                Node initialNode) {
              List<AntAgent> agents = new ArrayList<AntAgent>();

              for (int i = 0; i < numberOfAgents; i++) {
                    agents.add(new AntAgent(namePrefix + "-" + i, TactileWorkerAntType.TYPE, initialNode, false));
```

```
124                 }

                    logger.debug("{} agents created with prefix '{}'", agents.size(), namePrefix);
                    return agents;
            }
129 }
```

B.2.6 *Simulation Renderers*

Listing 41: ExploredSpaceRenderer.java

```
1   package com.luizabrahao.msc.ants.render;

    import java.awt.Color;
    import java.awt.Graphics2D;
    import java.awt.image.BufferedImage;
6   import java.io.File;
    import java.io.IOException;
    import java.util.concurrent.Callable;

    import javax.imageio.ImageIO;
11
    import org.slf4j.Logger;
    import org.slf4j.LoggerFactory;

    import com.luizabrahao.msc.model.env.Node;
16
    public class ExploredSpaceRenderer implements Callable<Void> {
            private static final Logger logger = LoggerFactory.getLogger(ExploredSpaceRenderer.class);

            private final Node[][] grid;
21          private final String imagePath;
            private final int nColumns;
            private final int nLines;

            public ExploredSpaceRenderer(Node[][] grid, String imagePath, int nColumns, int nLines) {
26                  this.grid = grid;
                    this.imagePath = imagePath;
                    this.nColumns = nColumns;
                    this.nLines = nLines;
            }
31
            @Override
            public Void call() throws Exception {
                    logger.info("Starting to render explored space...");
                    BufferedImage bufferedImage = new BufferedImage(nColumns, nLines, BufferedImage.TYPE_INT_RGB);
36                  Graphics2D g2d = bufferedImage.createGraphics();

                    for (int l = 0; l < nLines; l++) {
                            for (int c = 0; c < nColumns; c++) {
                                    if (grid[l][c].getAgents() == null) {
41                                          g2d.setColor(Color.white);

                                    } else {
                                            g2d.setColor(Color.black);
                                    }
46
                                    g2d.fillRect(c, l, c + 1, l + 1);
                            }
                    }

51                  g2d.dispose();
                    logger.info("Rendering explored space completed...");
                    try {
```

```
                File file = new File(imagePath);
                ImageIO.write(bufferedImage, "png", file);

            } catch (IOException e) {
                logger.error("Error rendering explored space.");
                logger.error(e.getMessage());
            }

            return null;
        }

}
```

Listing 42: ExploredSpaceRenderer.java

```
package com.luizabrahao.msc.ants.render;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.concurrent.Callable;

import javax.imageio.ImageIO;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;

public class NodeHistoryRenderer implements Callable<Void> {
    private static final Logger logger = LoggerFactory.getLogger(NodeHistoryRenderer.class);

    private final Agent agent;
    private final String imagePath;
    private final int nColumns;
    private final int nLines;

    public NodeHistoryRenderer(Agent agent, String imagePath, int nColumns, int nLines) {
        this.agent = agent;
        this.imagePath = imagePath;
        this.nColumns = nColumns;
        this.nLines = nLines;
    }

    @Override
    public Void call() throws Exception {
        logger.info("Starting to render visited nodes history for: {} wirh name '{}'", agent.getId(), imagePath);
        List<Node> nodes = agent.getNodesVisited();

        BufferedImage image = new BufferedImage(nColumns, nLines, BufferedImage.TYPE_INT_RGB);
        Graphics2D g2d = image.createGraphics();

        g2d.setColor(Color.white);
        g2d.fillRect(0, 0, nColumns, nLines);
        g2d.setColor(Color.black);

        for (Node node : nodes) {
            int line = 0;
            int column = 0;

            Node currentNode = node.getNeighbour(Direction.NORTH);
```

```
                        while (currentNode != null) {
                                line++;
                                currentNode = currentNode.getNeighbour(Direction.NORTH);
                        }

                        currentNode = node.getNeighbour(Direction.WEST);

                        while (currentNode != null) {
                                column++;
                                currentNode = currentNode.getNeighbour(Direction.WEST);
                        }

                        image.setRGB(column, line, 0);
                }

                g2d.dispose();
                nodes = null;

                try {
                        File file = new File(imagePath);
                    ImageIO.write(image, "png", file);

                } catch (IOException e) {
                        logger.error("Error rendering population.");
                        logger.error(e.getMessage());
                }

                logger.info("Finished rendering nodes history for: {}", agent.getId());
                return null;
        }

}
```

Listing 43: PheromoneRenderer.java

```
package com.luizabrahao.msc.ants.render;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.concurrent.Callable;

import javax.imageio.ImageIO;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.ants.env.ChemicalCommStimulus;
import com.luizabrahao.msc.ants.env.ChemicalCommStimulusType;
import com.luizabrahao.msc.model.env.Node;

public class PheromoneRenderer implements Callable<Void> {
        private static final Logger logger = LoggerFactory.getLogger(PheromoneRenderer.class);

        private final Node[][] grid;
        private final String imagePath;
        private final int nColumns;
        private final int nLines;
        private final ChemicalCommStimulusType chemicalCommStimulusType;

        public PheromoneRenderer(Node[][] grid, String imagePath, int nColumns, int nLines, ChemicalCommStimulusType
              chemicalCommStimulusType) {
                this.grid = grid;
                this.imagePath = imagePath;
                this.nColumns = nColumns;
```

```
                this.nLines = nLines;
                this.chemicalCommStimulusType = chemicalCommStimulusType;
        }

        @Override
        public Void call() throws Exception {
                logger.info("Pheromone render for '{}' started.", this.chemicalCommStimulusType.getName());
                BufferedImage bufferedImage = new BufferedImage(nColumns, nLines, BufferedImage.TYPE_INT_RGB);
                Graphics2D g2d = bufferedImage.createGraphics();

                double pheromoneIntensity = 0;

                for (int l = 0; l < nLines; l++) {
                        for (int c = 0; c < nColumns; c++) {
                                ChemicalCommStimulus stimulus = (ChemicalCommStimulus) grid[l][c].
                                      getCommunicationStimulus(chemicalCommStimulusType);

                                if (stimulus == null) {
                                        pheromoneIntensity = 0;
                                } else {
                                        pheromoneIntensity =stimulus.getIntensity();
                                }

                                int colorRate = 255 - (int) (pheromoneIntensity * 255);
                                g2d.setColor(new Color(255, colorRate, colorRate));

                                g2d.fillRect(c, l, c + 1, l + 1);
                        }
                }

                g2d.dispose();

                try {
                        File file = new File(imagePath);
                    ImageIO.write(bufferedImage, "png", file);

                } catch (IOException e) {
                        logger.error("Error rendering pheromone.");
                        logger.error(e.getMessage());
                }

                logger.info("Pheromone render for '{}' finished.", this.chemicalCommStimulusType.getName());
                return null;
        }
}
```

Listing 44: PopulationRenderer.java

```
package com.luizabrahao.msc.ants.render;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;

import javax.imageio.ImageIO;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.model.agent.Agent;
import com.luizabrahao.msc.model.env.Direction;
import com.luizabrahao.msc.model.env.Node;
```

```
20
public class PopulationRenderer implements Callable<Void>{
        private static final Logger logger = LoggerFactory.getLogger(PopulationRenderer.class);

        private final List<Agent> agents;
25      private final String imagePath;
        private final int nColumns;
        private final int nLines;

        public PopulationRenderer(List<Agent> agents, String imagePath, int nColumns, int nLines) {
30              this.agents = agents;
                this.imagePath = imagePath;
                this.nColumns = nColumns;
                this.nLines = nLines;
        }
35
        @Override
        public Void call() throws Exception {
                List<Node> nodes = new ArrayList<Node>();

40              for (Agent agent : agents) {
                        nodes.add(agent.getCurrentNode());
                }

                BufferedImage image = new BufferedImage(nColumns, nLines, BufferedImage.TYPE_INT_RGB);
45              Graphics2D g2d = image.createGraphics();

                g2d.setColor(Color.white);
                g2d.fillRect(0, 0, nColumns, nLines);
                g2d.setColor(Color.black);
50
                for (Node node : nodes) {
                        int line = 0;
                        int column = 0;

55                      Node currentNode = node.getNeighbour(Direction.NORTH);

                        while (currentNode != null) {
                                line++;
                                currentNode = currentNode.getNeighbour(Direction.NORTH);
60                      }

                        currentNode = node.getNeighbour(Direction.WEST);

                        while (currentNode != null) {
65                              column++;
                                currentNode = currentNode.getNeighbour(Direction.WEST);
                        }

                        image.setRGB(column, line, 0);
70              }

                g2d.dispose();
                nodes = null;

75              try {
                        File file = new File(imagePath);
                    ImageIO.write(image, "png", file);

                } catch (IOException e) {
80                      logger.error("Error rendering population.");
                        logger.error(e.getMessage());
                }

                return null;
85      }

}
```

B.2.7 *Simlations*

Listing 45: PathSimulation.java

```
package com.luizabrahao.msc.simulations;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.CancellationException;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import org.junit.Test;

import com.luizabrahao.msc.ants.agent.AntAgent;
import com.luizabrahao.msc.ants.agent.AntAgentFactory;
import com.luizabrahao.msc.ants.agent.AntNestAgent;
import com.luizabrahao.msc.ants.agent.WorkerAntType;
import com.luizabrahao.msc.ants.env.AntEnvironmentFactory;
import com.luizabrahao.msc.ants.env.ForageStimulusType;
import com.luizabrahao.msc.ants.env.PheromoneNode;
import com.luizabrahao.msc.ants.render.ExploredSpaceRenderer;
import com.luizabrahao.msc.ants.render.PheromoneRenderer;
import com.luizabrahao.msc.ants.test.TestUtil;

public class PathSimulation {
	private final int nLines = 500;
	private final int nColumns = 500;
	private final int maximumNumberOfThreads = 60;
	private final long secondsToRun = 10;
	private final long secondsToRender = 10;
	private final double initialConcentration = 0.001;

	@Test
	public void cancelTest() throws InterruptedException {
		final ScheduledExecutorService executor = Executors.newScheduledThreadPool(maximumNumberOfThreads);
		List<Callable<Void>> renderers = new ArrayList<Callable<Void>>();

		final PheromoneNode[][] grid = AntEnvironmentFactory.createPheromoneNodeGrid(nLines, nColumns);
		TestUtil.setIntensity(0, nLines, 0, nColumns, initialConcentration, grid);

		final AntNestAgent nest = new AntNestAgent("nest", grid[0][Integer.valueOf(nColumns / 2)]);
		final List<AntAgent> agents = AntAgentFactory.produceBunchOfWorkers(50, "worker", grid[0][Integer.
			valueOf(nColumns / 2)]);

		List<Future<Void>> agentsFutures = executor.invokeAll(agents, secondsToRun, TimeUnit.SECONDS);

		for (Future<Void> future : agentsFutures) {
			try {
				future.get();
			} catch (ExecutionException e) {
				e.printStackTrace();
			} catch (CancellationException e) {
				// nothing to do...
			}

			future.cancel(true);
		}

		renderers.add(new ExploredSpaceRenderer(grid, "target/path - space explored - ic=" +
			initialConcentration + ", as=" + WorkerAntType.TYPE.getStimulusIncrement(ForageStimulusType.TYPE.
			getName()) + ".png", nColumns, nLines));
```

```
                renderers.add(new PheromoneRenderer(grid, "target/initial — ic=" + initialConcentration + ", as=" +
                        WorkerAntType.TYPE.getStimulusIncrement(ForageStimulusType.TYPE.getName()) + ".png", nColumns,
                        nLines, ForageStimulusType.TYPE));

                List<Future<Void>> renderersFutures = executor.invokeAll(renderers, secondsToRender, TimeUnit.
                        SECONDS);

                for (Future<Void> future : renderersFutures) {
                        try {
                                future.get();
                        } catch (ExecutionException e) {
                                e.printStackTrace();
                        }

                        future.cancel(true);
                }

        }
}
```

Listing 46: StudyOnRadius.java

```
package com.luizabrahao.msc.simulations;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.CancellationException;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.luizabrahao.msc.ants.agent.AntAgent;
import com.luizabrahao.msc.ants.agent.AntAgentFactory;
import com.luizabrahao.msc.ants.agent.AntNestAgent;
import com.luizabrahao.msc.ants.agent.StaticPheromoneUpdaterAgent;
import com.luizabrahao.msc.ants.env.AntEnvironmentFactory;
import com.luizabrahao.msc.ants.env.FoodSourceAgent;
import com.luizabrahao.msc.ants.env.ForageStimulusType;
import com.luizabrahao.msc.ants.env.PheromoneNode;
import com.luizabrahao.msc.ants.render.PheromoneRenderer;
import com.luizabrahao.msc.ants.test.TestUtil;
import com.luizabrahao.msc.sim.util.CallableAdapter;

public class StudyOnRadius {
        private static final Logger logger = LoggerFactory.getLogger(StudyOnRadius.class);

        private final int nLines = 500;
        private final int nColumns = 500;
        private final int maximumNumberOfThreads = 60;
        private final long secondsToRun = 30;
        private final long secondsToRender = 10;
        private final double initialConcentration = 0.01;
        private final long secondsToRunDecayAgent = 3;

        @Test
        public void run() throws InterruptedException {
                for (int i = 0; i < 25; i++) {
                        this.executeExperiment(i);
                }
```

```
	}


	@SuppressWarnings("unused")
	public void executeExperiment(int executionNumber) throws InterruptedException {
		final ScheduledExecutorService executor = Executors.newScheduledThreadPool(maximumNumberOfThreads);
		List<Callable<Void>> renderers = new ArrayList<Callable<Void>>();

		final PheromoneNode[][] grid = AntEnvironmentFactory.createPheromoneNodeGrid(nLines, nColumns);
		TestUtil.setIntensity(0, nLines, 0, nColumns, initialConcentration, grid);

		final AntNestAgent nest = new AntNestAgent("nest", grid[0][Integer.valueOf(nColumns / 2)]);
		final List<FoodSourceAgent> foodSources = AntEnvironmentFactory.placeRowOfFoodSources(grid[nLines -
			100][Integer.valueOf(nColumns / 2) - 100], 200, 30);
		final List<AntAgent> agents = AntAgentFactory.produceBunchOfWorkers(50, "worker", grid[0][Integer.
			valueOf(nColumns / 2)]);

		final StaticPheromoneUpdaterAgent pheromoneUpdater1 = new StaticPheromoneUpdaterAgent("pheromone-
			updater-1", grid[0][0], nLines / 2);
		final StaticPheromoneUpdaterAgent pheromoneUpdater2 = new StaticPheromoneUpdaterAgent("pheromone-
			updater-2", grid[250][0], nLines / 2);

		ScheduledFuture<?> rf1 = executor.scheduleWithFixedDelay(CallableAdapter.runnable(pheromoneUpdater1)
			, secondsToRunDecayAgent, secondsToRunDecayAgent, TimeUnit.SECONDS);
		ScheduledFuture<?> rf2 = executor.scheduleWithFixedDelay(CallableAdapter.runnable(pheromoneUpdater2)
			, secondsToRunDecayAgent, secondsToRunDecayAgent, TimeUnit.SECONDS);

		List<Future<Void>> agentsFutures = executor.invokeAll(agents, secondsToRun, TimeUnit.SECONDS);

		rf1.cancel(true);
		rf2.cancel(true);

		for (Future<Void> future : agentsFutures) {
			try {
				future.get();
			} catch (ExecutionException e) {
				e.printStackTrace();
			} catch (CancellationException e) {
				// nothing to do...
			}

			future.cancel(true);
		}

//  renderers.add(new ExploredSpaceRenderer(grid, "target/studyOnRadius - space - radius = " + ForageStimulusType.
	TYPE.getRadius() + ".png", nColumns, nLines));
//  renderers.add(new PheromoneRenderer(grid, "target/studyradius - " + ForageStimulusType.TYPE.getRadius() + " - " +
	executionNumber + ".png", nColumns, nLines, ForageStimulusType.TYPE));
//
//  List<Future<Void>> renderersFutures = executor.invokeAll(renderers, secondsToRender, TimeUnit.SECONDS);
//
//  for (Future<Void> future : renderersFutures) {
//   try {
//    future.get();
//   } catch (ExecutionException e) {
//    e.printStackTrace();
//   }
//
//   future.cancel(true);
// }

		logger.info("Amount of food collected; {}", nest.getAmountOfFoodHeld());
	}
}
```

BIBLIOGRAPHY

[1] M.D. Allen. The "shaking" of worker honeybees by other workers. *Anim. Behav.*, 7:233–240, 1959.

[2] James Andreoni and John H. Miller. Auctions with adaptive artificial agents. *Games and Economic Behavior*, 10(1):39–64, 1995.

[3] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic. Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study. *Proceedings of the National Academy of Sciences*, 105(4):1232, 2008. URL http://www.pnas.org/content/105/4/1232.abstract.

[4] J. Bloch. *Effective Java*. Addison-Wesley Java series. Addison-Wesley, 2008. ISBN 9780321356680. URL http://books.google.co.uk/books?id=ka2VUBqHiWkC.

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, USA, 1999. ISBN 9780195131581. URL http://books.google.co.uk/books?id=fcTcHvSsRMYC.

[6] M.E. Bratman. *Intention, Plans and Practical Reason*. The David Hume series of philosophy and cognitive sciences reissues. Center for the Study of Language and Inf, 1999. ISBN 9781575861920. URL http://books.google.co.uk/books?id=SvyJQgAACAAJ.

[7] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14 – 23, mar 1986. ISSN 0882-4967. doi: 10.1109/JRA.1986.1087032.

[8] Rodney Allen Brooks and Jonathan H. Connell. Asynchronous distributed control system for a mobile robot. 1986.

[9] D.R. Butenhof. *Programming With Posix Threads*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997. ISBN 9780201633924. URL http://books.google.co.uk/books?id=TXiJDj9kbiAC.

[10] Paul Caplat, Madhur Anand, and Chris Bauch. Symmetric competition causes population oscillations in an individual-based model of forest dynamics. *Ecological Modelling*, 211:491 – 500, 2008. ISSN 0304-3800. doi: 10.1016/j.ecolmodel.2007.10.002. URL http://www.sciencedirect.com/science/article/pii/S0304380007005303.

[11] Gianni Di Caro and Marco Dorigo. Antnet: A mobile agents approach to adaptive routing. Technical report, 1997.

[12] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 1998.

[13] Gregory S. Chirikjian. Kinematics of a metamorphic robotic system. In *ICRA*, pages 449–455, 1994.

[14] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed Optimization by Ant Colonies. In *European Conference on Artificial Life*, pages 134–142, 1991.

[15] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. An investigation of some properties of an "Ant algorithm". In *Proceedings to the Parallel Problem Solving from Nature Conference (PPSN)*, pages 509–520, 1992.

[16] R. Dawkins. *The Selfish Gene*. Popular Science. Oxford University Press, USA, 1990. ISBN 9780192860927. URL http://books.google.co.uk/books?id=WkH09HI7koEC.

[17] J. L. Deneubourg, S. Goss, N. Franks, A. Sendova Franks, C. Detrain, and L. Chrétien. The dynamics of collective sorting robot-like ants and ant-like robots. In *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, pages 356–363, Cambridge, MA, USA, 1990. MIT Press. ISBN 0-262-63138-5. URL http://portal.acm.org/citation.cfm?id=116517.116557.

[18] Gianni Di Caro and Marco Dorigo. Antnet: distributed stigmergetic control for communications networks. *J. Artif. Int. Res.*, 9(1):317–365, December 1998. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622797.1622806.

[19] M Dorigo. *Ottimizzazione, Apprendimento Automatico, ed Algoritmi Basati su Metafora Naturale.* Phd thesis, Politecnico di Milano, 1992.

[20] M. Dorigo, A. Colorni, and V. Maniezzo. Positive feedback as a search strategy. Technical Report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1991. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.6342.

[21] Marco Dorigo. Swarm-bot: An experiment in swarm robotics. In *In Proc. of the 2005 IEEE Swarm Intelligence Symp*, pages 192–200. IEEE Computer Society Press, 2005.

[22] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the traveling salesman problem, 1997.

[23] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 1997.

[24] Birgit Ehmer. Orientation in the ant paraponera clavata. *Journal of Insect Behavior*, 12:711–722, 1999. ISSN 0892-7553. URL http://dx.doi.org/10.1023/A:1020987922344. 10.1023/A:1020987922344.

[25] T. L. Erwin. Canopy arthopod biodiversity: a chronology of sampling techniques and results. *Revista Peruana de Etomologia*, 32:71–77, 1989.

[26] E. J. Fittkau and H. Klinge. On biomass and trophic structure of the central amazonian rain forest ecosystem. *Biotropica*, 5(1):2–14, 1973.

[27] M. Fowler. *Uml Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Object Technology Series. Addison-Wesley, 2004. ISBN 9780321193681. URL http://books.google.co.uk/books?id=nHZslSr1gJAC.

[28] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss. Structure decision method for self organising robots based on cell structures-cebot. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 695 –700 vol.2, may 1989. doi: 10.1109/ROBOT.1989.100066.

[29] B. Goetz and T. Peierls. *Java concurrency in practice*. Addison-Wesley, 2006. ISBN 9780321349606. URL http://books.google.co.uk/books?id=6LpQAAAAMAAJ.

[30] Paul Graham and Ken Cheng. Which portion of the natural panorama is used for view-based navigation in the australian desert ant? *Journal of Comparative Physiology A: Neuroethology, Sensory, Neural, and Behavioral Physiology*, 195:681–689, 2009. ISSN 0340-7594. URL http://dx.doi.org/10.1007/s00359-009-0443-6. 10.1007/s00359-009-0443-6.

[31] Martin Heusse, Dominique Snyers, Sylvain Guérin, and Pascale Kuntz. Adaptive agent-driven routing and load balancing in communication networks. In *Advances in Complex Systems*, pages 15–16, 1998.

[32] B. Holldobler. Multimodal signals in ant communication. *Journal of Comparative Physiology*, 1999.

[33] B. Holldobler and E.O. Wilson. *The Ants*. Belknap Press of Harvard University Press, http://books.google.co.uk/books?id=ljxV4h61vhUC, 1990.

[34] Bert Hölldobler and Edward O. Wilson. *The superorganism : the beauty, elegance, and strangeness of insect societies*. W.W. Norton, 2009. URL http://www.worldcat.org/oclc/227016678.

[35] Stuart A. Kauffman. Approaches to the origin of life on earth. *Life*, 1(1):34–48, 2011. ISSN 2075-1729. doi: 10.3390/life1010034. URL http://www.mdpi.com/2075-1729/1/1/34.

[36] J.F. Kennedy, J. Kennedy, R.C. Eberhart, and Y. Shi. *Swarm Intelligence*. The Morgan Kaufmann Series in Evolutionary Computation. Morgan Kaufmann Publishers, 2001. ISBN 9781558605954. URL http://books.google.co.uk/books?id=vOx-QV3sRQsC.

[37] J. H. Klotz and B. L. Reid. Nocturnal orientation in the black carpenter ant (camponotus pennsylvanicus (degeer) (hymenoptera: Formicidae). *Insectes Sociaux*, 40:95–106, 1993. ISSN 0020-1812. URL http://dx.doi.org/10.1007/BF01338835. 10.1007/BF01338835.

[38] R.B. Laughlin. *A Different Universe: Reinventing Physics From the Bottom Down*. Basic Books, 2008. ISBN 9780786722181. URL http://books.google.co.uk/books?id=djQKg_XBsLEC.

[39] M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.

[40] E. Lumer and B. Faieta. Diversity and adaptation in populations of clustering ants. In *Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, 1994.

[41] Nithin Mathews, Anders Lyhne Christensen, Rehan O'Grady, Philippe Retornaz, Michael Bonani, Francesco Mondada, and Marco Dorigo. Enhanced directional self-assembly based on active recruitment and guidance. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4762 –4769, sept. 2011. doi: 10.1109/IROS.2011.6094854.

[42] ROBERT M. MAY. How many species are there on earth? *Science*, 241(4872):1441–1449, 1988. doi: 10.1126/science.241.4872.1441. URL http://www.sciencemag.org/content/241/4872/1441.abstract.

[43] Robert M. May. Why worry about how many species and their loss? *PLoS Biol*, 9(8):e1001130, 08 2011. doi: 10.1371/journal.pbio.1001130. URL http://dx.doi.org/10.1371%2Fjournal.pbio.1001130.

[44] J.H. Miller and S.E. Page. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton Studies in Complexity. Princeton University Press, 2007. ISBN 9780691127026. URL http://books.google.co.uk/books?id=XQUHZC8wcdMC.

[45] Camilo Mora, Derek P. Tittensor, Sina Adl, Alastair G. B. Simpson, and Boris Worm. How many species are there on earth and in the ocean? *PLoS Biol*, 9(8): e1001127, 08 2011. doi: 10.1371/journal.pbio.1001127. URL http://dx.doi.org/10.1371%2Fjournal.pbio.1001127.

[46] G.F. Oster and E.O. Wilson. *Caste and Ecology in the Social Insects. (MPB-12)*. Monographs in Population Biology. Princeton University Press, 1979. ISBN 9780691023618. URL http://books.google.co.uk/books?id=RGE0MwY_NWIC.

[47] Ruud Schoonderwoerd, Owen Holland, Janet Bruten, and Leon Rothkrantz. Ant-based load balancing in telecommunications networks, 1996.

[48] T.D. Seeley. *The Wisdom of the Hive: The Social Physiology of Honey Bee Colonies*. Harvard University Press, 1995. ISBN 9780674953765. URL http://books.google.co.uk/books?id=zhzNJjI2MqAC.

[49] Devika Subramanian, Peter Druschel, and Johnny Chen. Ants and reinforcement learning: A case study in routing in dynamic networks. In *In IJCAI (2*, pages 832–838. Morgan Kaufmann, 1998.

[50] H.C. Tijms. *Understanding Probability: Chance Rules in Everyday Life*. Cambridge University Press, 2007. ISBN 9780521701723. URL http://books.google.co.uk/books?id=Ua-_5Ga4QF8C.

[51] W.R. Tschinkel. *The Fire Ants*. Belknap Press of Harvard University Press, 2006. ISBN 9780674022072. URL http://books.google.co.uk/books?id=vxt5BqOKEAIC.

[52] Karl von Frisch. *The Dance Language and Orientation of Bees*. The Belknap Press of Harvard University Press, 1967.

[53] William Morton Wheeler. The ant-colony as an organism. *Journal of Morphology*, 22(2):307–325, 1911. ISSN 1097-4687. doi: 10.1002/jmor.1050220206. URL http://dx.doi.org/10.1002/jmor.1050220206.

[54] E.O. Wilson. *The Ergonomics of Caste in the Social Insects*. American Naturalist, 1968. URL http://books.google.co.uk/books?id=9wwzygAACAAJ.

[55] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2009. ISBN 9780470519462. URL http://books.google.co.uk/books?id=X3ZQ7yeDn2IC.