AGENTS: INVESTIGATING ENVIRONMENT PERCEPTION

LUIZ FILIPE POLIMENO ABRAHAO

MSc Project Report

Engineering Department

King's College London

University of London

September 2012 – Draft 1

ABSTRACT

We are surrounded by cases 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. 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.

Three experiments are proposed and run. The first experiment investigates the relationship between agents and pheromone concentration present in the environment. After a study on velocity of reaction against predator warning stimulus is carried. The last experiment investigates the affect of changing the radius of action of chemical stimuli on the colony's foraging capability.

This paper also proposes possible improvements to the proposed model and and its implementation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ABM	Agent-based Model
API	Application Public Interface
BDI	Beliefs Desires Intentions
CLT	Central Limit Theorem
CPU	Central Processing Unit
OOP	Object Oriented Programming
UML	Unified Modelling Language

Part I

SETTING THE CONTEXT

In this first part an introduction to the field is given, also some background information and an overview of the proposed computational model are presented.

1

INTRODUCTION

Recent studies have estimated that there are 8.7 million eukaryote species on the planet.[25] [23] Most of them are still to be discovered. Dawkins argues that the utmost 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 its the 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. There are numerous ways to define intelligence though. Kennedy et al. 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 before. [19] One shares this same point of view.

We are surrounded by examples of animals with a wide variety of complexity, from the very intellectually limited termites to us, humans beings, solving the problem of surviving in an ever changing environment in the most different and ingenious ways. It could be argued that humans are the most successful creatures wandering in this planet, it is very easy to blindly believe this statement, for the vast majority of us grow up being told that is the case. As beauty is in the eyes of the beholder, there are different ways to think about success though. If survival is all that matters, the number of individuals of certain species could be used as a good benchmark for success, and in this case humans are losing emphatically to insects. [22]

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 environ-

ment around the nest can be completely different from what it was last time they went out. Leafs might have fallen, thus changing how the nest can access food in the rain forest, the wind can blow the sand in the dessert, covering food sources and despite this unpredictable factors, they face the necessity to see off in search for food, and indeed they do. As a complex decentralised system, the nest is able to absolve the disruptions introduced by the environment and keep going on, doing its business. These questions are very hard to answer, and indeed this project has no intention to do so.

Social insects are difficult to study for the very nature of the structures they create. These structures are decentralised and behaviours emerge from interactions between the system's agents. Accordingly, understanding of the agents' behaviour in isolation does not guarantee the understanding of the overall picture. Also, small changes in the way agents interact and react to external stimulus normally introduce great 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. Generalising, 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 spread through space faster than Pheromone A. How does this spread area affects the colony's forage capability? The experiment in section 4.3 investigates that.

But pheromones are not the only means of ant communication, one of the other ways is by antennation. Naturally ants have limited vision when compared with other animals, and their antennas serve as an effective way to 'see' each other. As it will be discussed later on, different types of ants have different chemical coats, and their

fellow ants are able to recognise these chemicals, therefore the type of their fellows. This is also used as a way to communicate. For example, if a worker is following a pheromone trail that supposedly leads to a source of food, but suddenly it starts to come across many workers moving in the direction of the nest, and most of these workers are not caring food. This could be interpreted as a sign of danger. If workers that are supposed to collect food, are running back to the nest without any, something ahead must be wrong. How does this second way of communication affects the velocity of reaction of the colony to danger? How does the variation of the number of workers going back to the nest without food as a threshold of danger situation affects the colony's reaction to danger? The experiment in section 4.2 investigates that.

In order to carry 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 agents are defined by their type and a list of tasks associated to them. The model is inspired in the subsumption architecture proposed by Brooks. [6] [7]. In the same way Brooks's 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 tasks, each task is completely independent from one another as far their execution goes. There is nothing to prevent a task to use another task to achieve a desired goal though. Differently from the subsumption architecture the agents from 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 BDI [5] [33] agents can be implemented by extending the abstract infrastructure with minimum effort.

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 agents are autonomous offers the ideal toolset to be used to create decentralised complex systems.

The model is presented in UML as class diagrams and activity diagramsand it is implemented using the computer language Java. UML has become the standard modelling language within the OOP world as well as it has become a popular popular technique outside the OOP circle. [13], 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 it is 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 researches in a variety of areas.

Most important for this project is the fact that Java offers a very 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 completely autonomous from each other, and only share the same information about the environment around them, which means that, there is not access to global information, but only the local context is available decision making and task 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 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 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 gets to be done by a large number of individuals that in fact can easily switch from one task to another. Large amount of individuals can forage a relatively large amount of food when compared to solitaire insects. Additionally, the genetic loss in the case of a worker 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 labor 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"* [31]. 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 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 cast. Each casts have different thresholds for different stimuli associated with tasks, what results in specialised work because individuals of different casts will respond differently to the environment around them. For example, a worker picks up food as soon as encounters it, on the other hand an ant belonging to the *Soldier* cast 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 [32] [26]. This is because the challenges the environment imposes to the colony requires 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. [27]. Honeybees are known for their extraordinary communication method - they use a vocabulary in which messages are expressed in form of dances. [30] But, in more than 90 percent of cases, social insects use some form of chemical signals when communicating. [17] 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], or 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. For example, workers of some types of ants can interpret the encounter of ants of another casts as a signal of danger. (((need citation))) There is no direct transmission of information in this case, but it does happen, indirectly.

2.2.1 *Ants*

In order to efficiently forage on the ground, 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. [15] There are at least 12 functional categories of communication deployed by social insects [17]:

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 are presented by Holldobler and Wilson in *The Ants*.

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. [17] [29]

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 hight importance because it affects the size of the *active space*, which is the zone that the intensity of a chemical signal is above the threshold concentration necessary to trigger action from the agents. [17]

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 are within the signal trails of known food sources.

2.3 EMERGENCE

Emergence is one of the type of concepts that are 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 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.[24] As an illustration to that, one could use any of most of the multi-cellular animals, including ourselves. The theoretical biologist Kauffman 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. [28]. 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.[20] 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 to 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. In 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 positive 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. Sock market is one example of such system. Although there are regulatory instruments in place to avoid abuse, the large number of dealers regulate the market as long as the value of shares are concerned. In 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 much attractive anymore and agents involved in trading will go after other options. With 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 regulate its price themselves. They are autonomous in regards 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. 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 in-

formation 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 [9] 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 [33] :

1. Reactive: In most of the applications that 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 to 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 not just react to stimuli from the environment, but 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 interacting 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. [21]

All in all, agent-oriented modelling offers the best methodology 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 Object Oriented Programming (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 site 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 we can controlled as needed.

At a first moment it is hard to look at computational models as a science tool of the same level of mathematical methods that have been used for centuries to build great part of 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 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. 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 coordinated. This is necessary to avoid reading invalid or inconsistent states of the shared variable. This is 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 very use to implement very complex concurrent systems.

A very common problem in concurrent systems is the so-called *race conditions*. Goetz and Peierls 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. *Intrinsic 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 [8], Java locks are granted on a per-thread basis. This greatly facilitates the development of object-oriented concurrent systems. [14]

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, what minimises the cost of creating and stopping threads, 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.[14] 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.

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 CURRENT RESEARCH

Research on agents have been carried on a wide variety of areas. Zhong et al. Zhong et al. 34 have

3

MODEL OVERVIEW

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 wish, 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 formalises 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 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 and how many neighbours each node has. This gives a great degree of freedom to users,

for example it is ease 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 though, by implementing the *Node* interface with 6 neighbours, one in each direction - north, south, east, west, above and bellow, 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 section XXX.

Each node has an unique string that is used to identify them. 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), this is 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 few conditions before allowing the agent into the node. Agents that are starting their lifecycle could fail to pass this conditions. 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 indirectly communicate to each other 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 stimulus present in the node
- void addCommunicationStimulus(CommunicationStimulus): add a new communication stimulus to the node.
- CommunicationStimulus getCommunicationStimulus (CommunicationStimulusType): 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 makes no sense in having two stimulus of the same tape 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 where 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 specifies 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 synchronisation mechanisms that would guarantee thread-safety. The reason for that is that this 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 subsection B.1.1 for more details.

Each basic node can have a maximum of four neighbours, one in each direction defined by the Direction enumeration (see B.1.2): 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 of everything present in the super class, it declares two new methods that come handy when dealing with chemical communication stimuli (see subsection 3.1.4.1):

- ChemicalCommStimulus getCommunicationStimulus(ChemicalCommStimulusType): 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 created a new chemical stimulus of that type, adds it to the stimuli list and returns the new 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 buy 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 illustrates 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. (((need to find another example in nature that are not ants or bees))).

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 o 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 (see section XX) in order to save computational resources.

3.1.4.1 *Chemical Communication Stimulus*

As seen in section 2.2, ants use pheromone as a way to communicate indirectly to fellow ants from the same nest. The interface *ChemicalCommStimulusType* extends *CommunicationStimulsType* 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 intensity decay. It uses the following:

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

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.

3.1.4.2 *Forage Communication Stimulus*

This communications stimulus type is used by ants when foraging. Is is deposited whether when ants are searching for food or going back to the nest caring food. The amount deposited should be higher if ants are caring food and going back to their nest. This is a form of positive feedback (see 2.3.2).

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. As well as 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*, it represents a whole different type of information though. It is an analogy to the pheromone laid by soldier ants to warn their fellows of any danger around. Different type 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 soldier of 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*

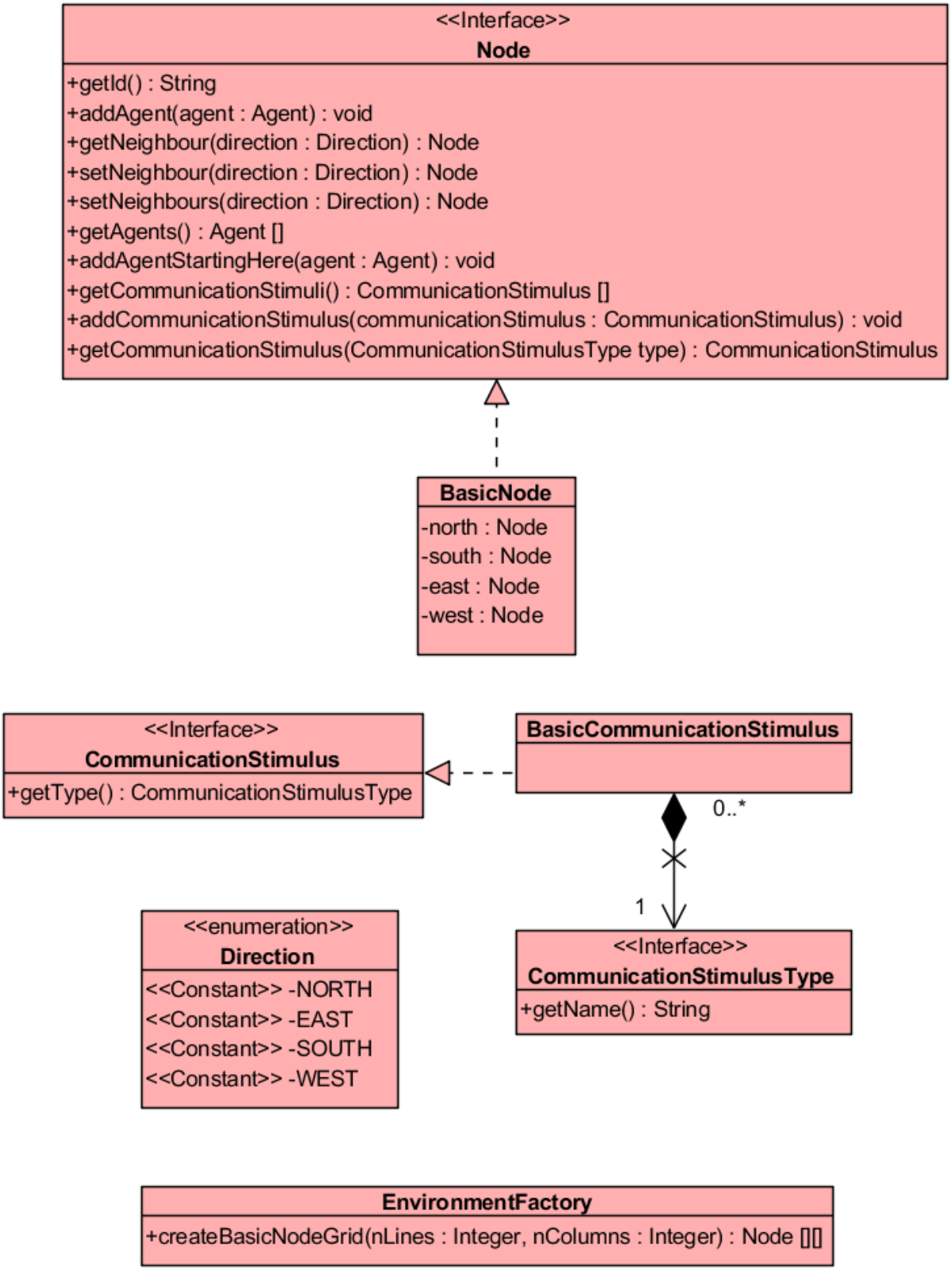


Figure 1: Generic model of environment package

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, and 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 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 with 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 de 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.

3.2.3 *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 isCaringFood(): Returns true if the agent is caring 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 agentâs moving direction will be West. See section XXX for more details.
- void depositFood(AntNestAgent): Deposit the food the agent is caring into the nest passed as parameter.

For the full API and more detail explanations of all the methods, please see section XXX.

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 on 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 as the distance from the node where the agent is at increases according to the following:

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

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 is defined by the agent type, see section 3.2.2 for a detailed explanation.

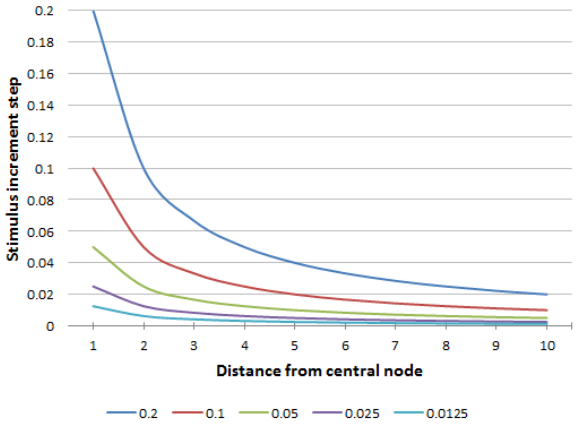


Figure 2: Variation of increment according to distance to central node

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.

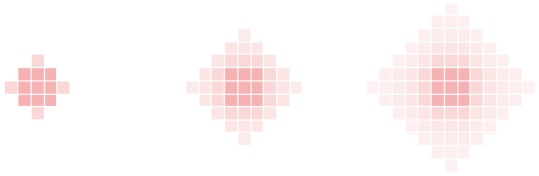


Figure 3: Pheromone deposit for stimulus with different radius sizes

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

(((need to discuss the way the memory is implemented)))

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 getStimulusIncrement(String)* is used to query these amounts.

3.2.5.1 *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*.

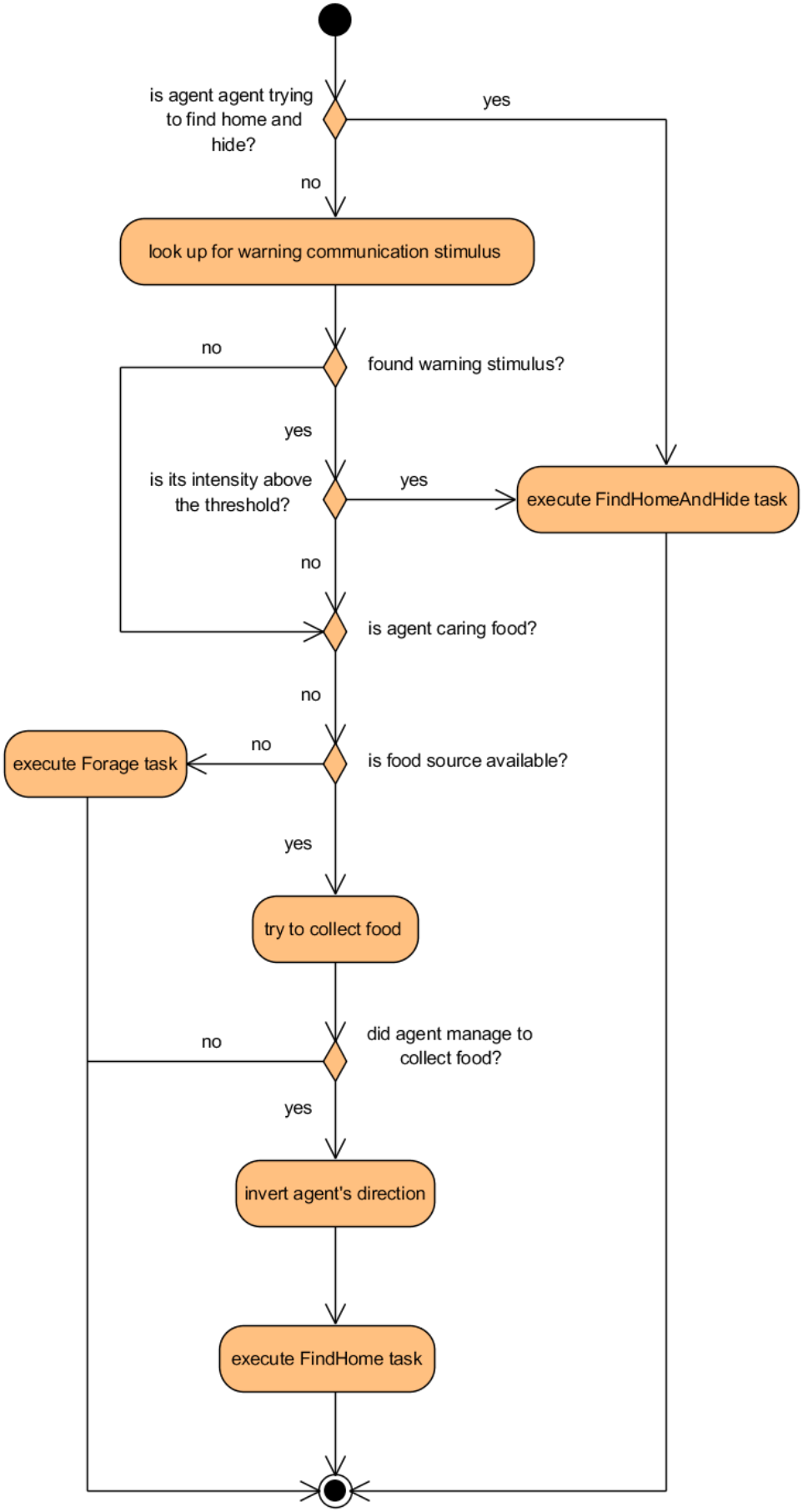


Figure 4: Worker Type ant algorithm

3.2.5.2 *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 types 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 *Agents Package*

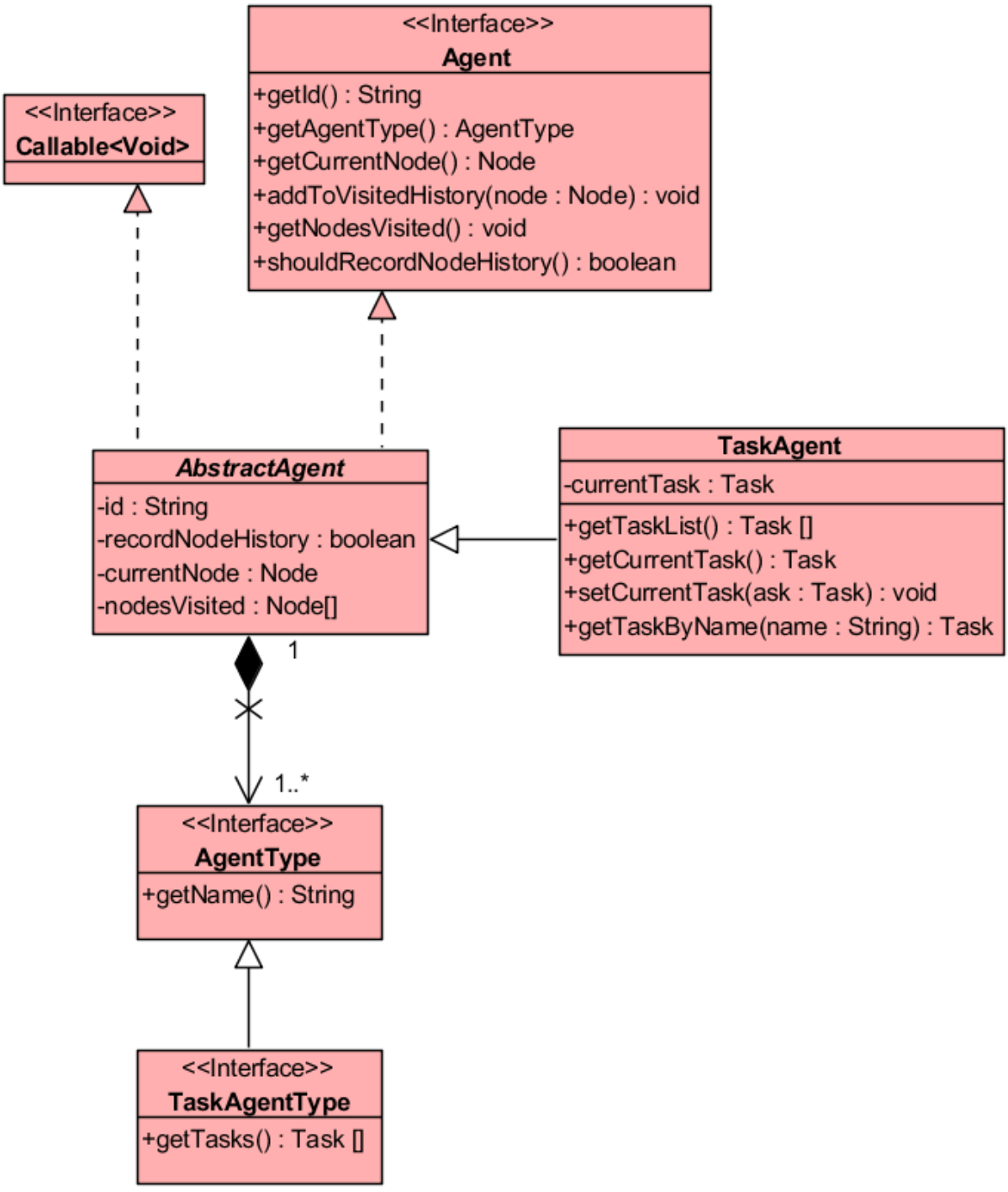


Figure 5: Generic model of the agent package

3.3 TASKS

Tasks are a unit of specialised work. They are the means to agents to 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 XXX.

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. Tt virtually impossible to model all this cases though, more, it would be wrong to try to model all the possible interferences in the colony.

So, the agents present a stochastic behaviour as far as moving through the space is concerned. So 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 (((cite))) like selection on the neighbour nodes is executed using these rates.

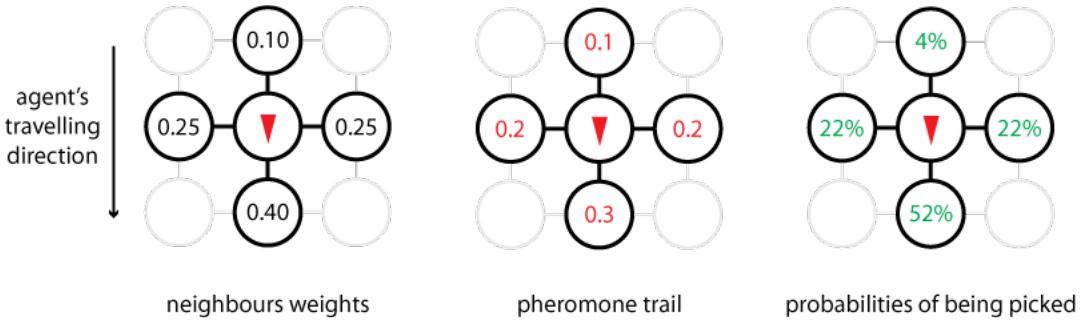


Figure 6: Elements used to select next node to move to

Figure 6 illustrates these steps. It is important to understand that when applying the weights to the neighbours the task uses the 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.

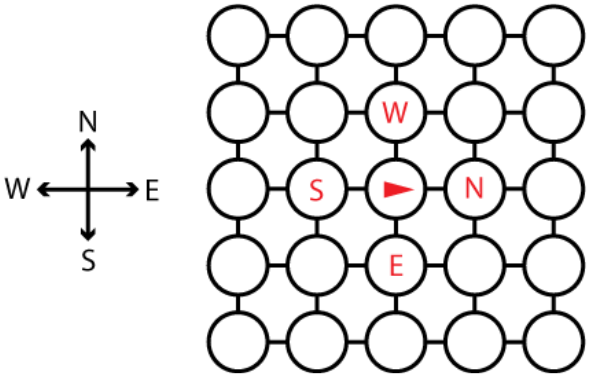


Figure 7: Reference of directions

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*.

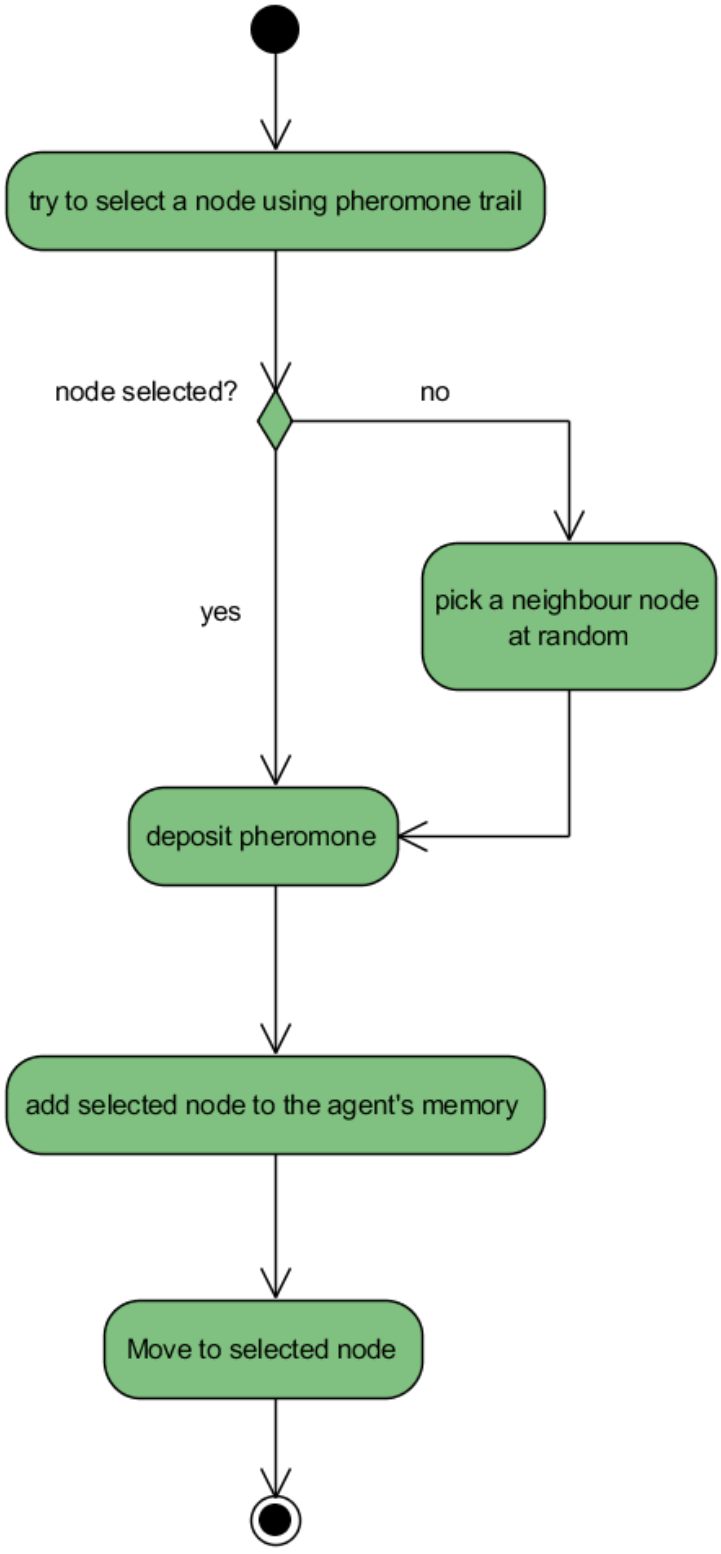


Figure 8: Forage task execution activity diagram

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.

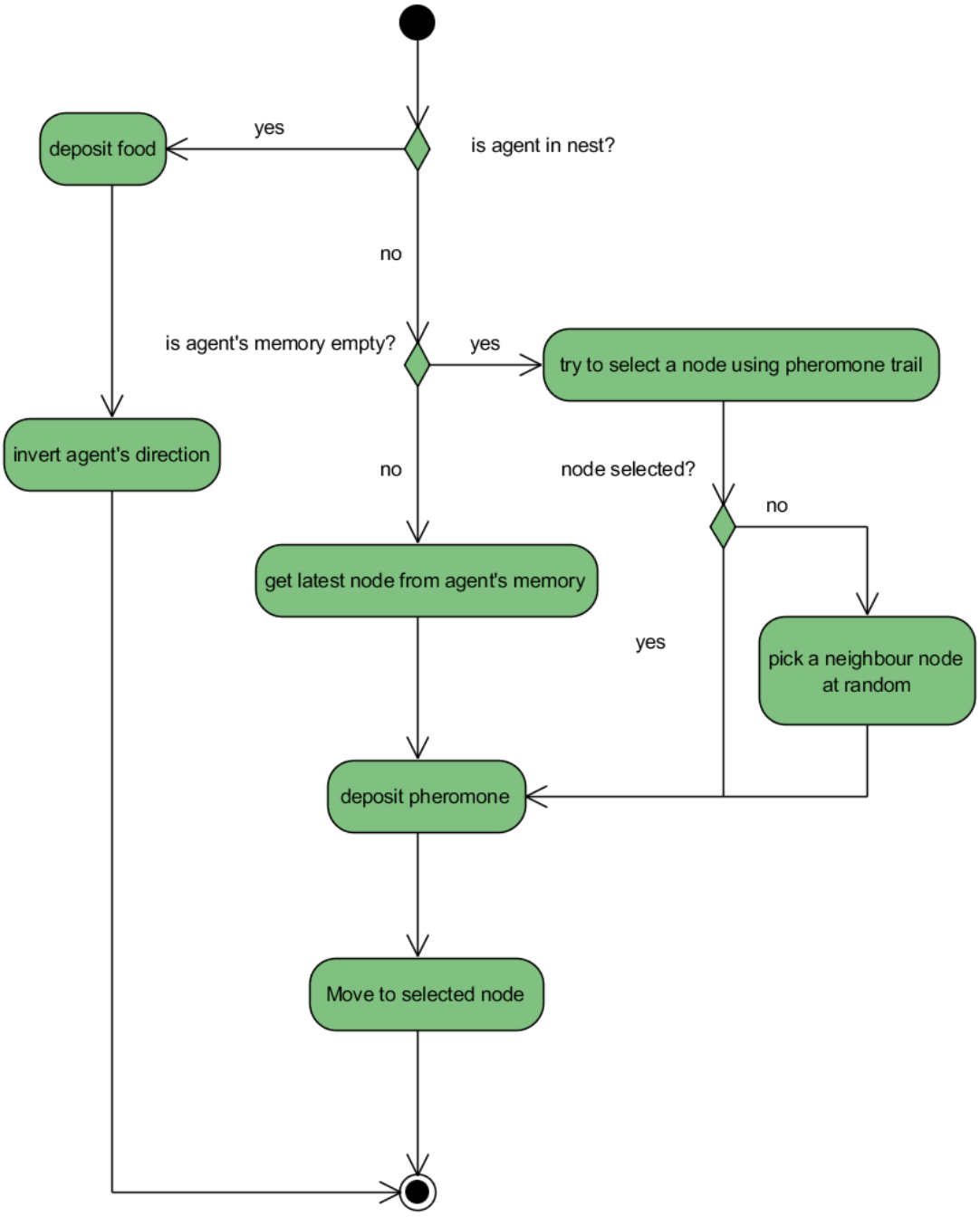


Figure 9: Find home task execution activity diagram

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.

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.

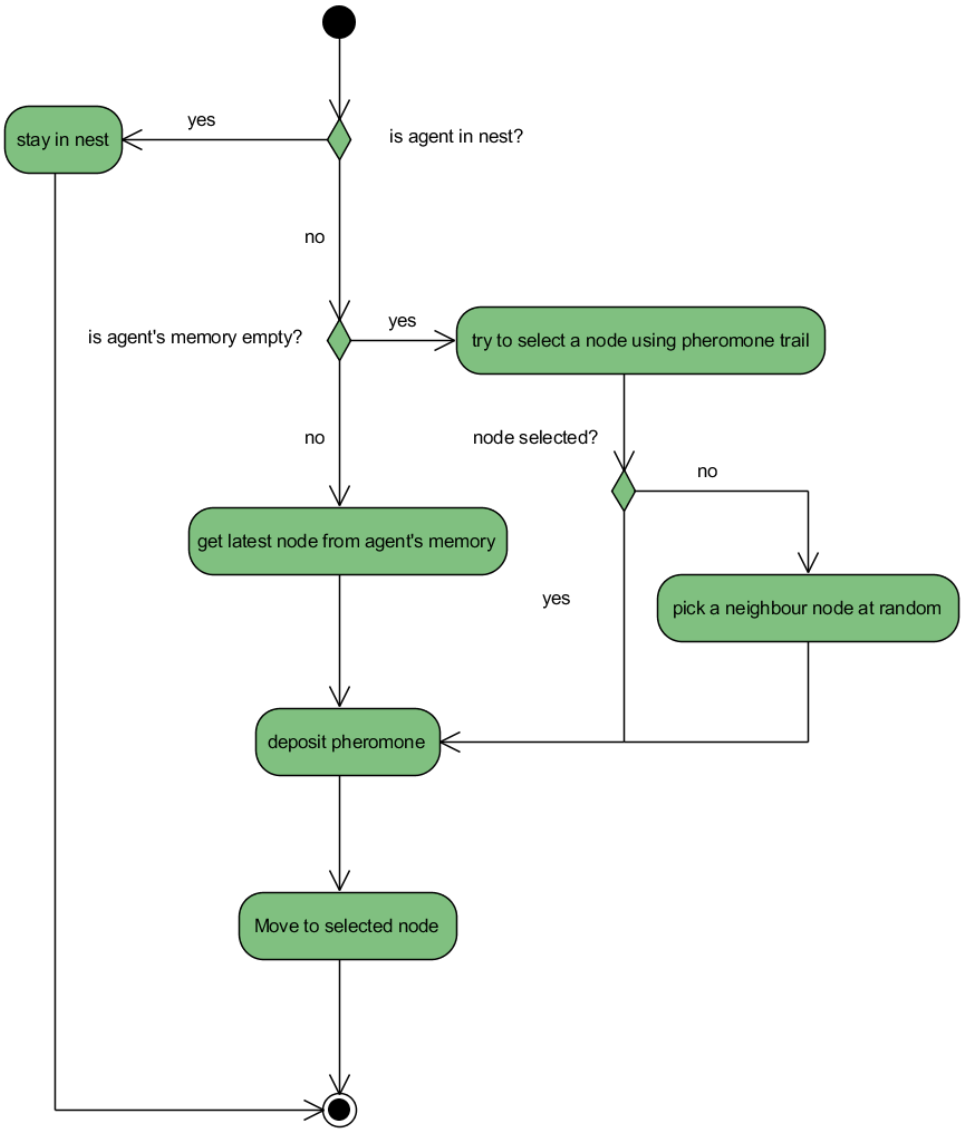


Figure 10: Find home and hide task execution activity diagram

3.4 GENERIC COMPUTATIONAL MODEL DIAGRAM

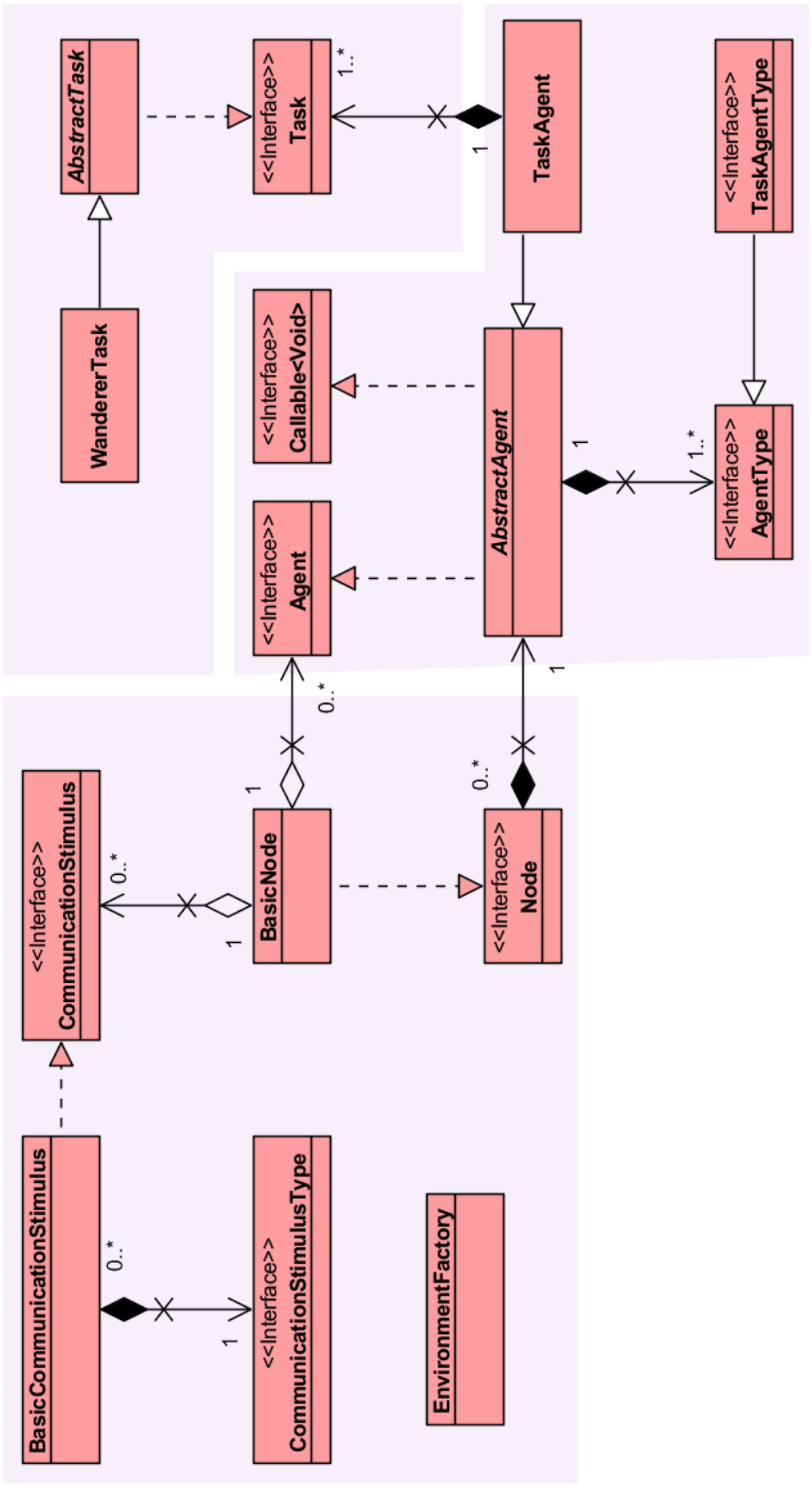


Figure 11: Generic Computational Model

Part II

EXPERIMENTS AND OBSERVATIONS

In this part of the report three experiments are proposed and executed. The results observed are discussed. At the end some possible improvements to the model and possible experiments are proposed.

4

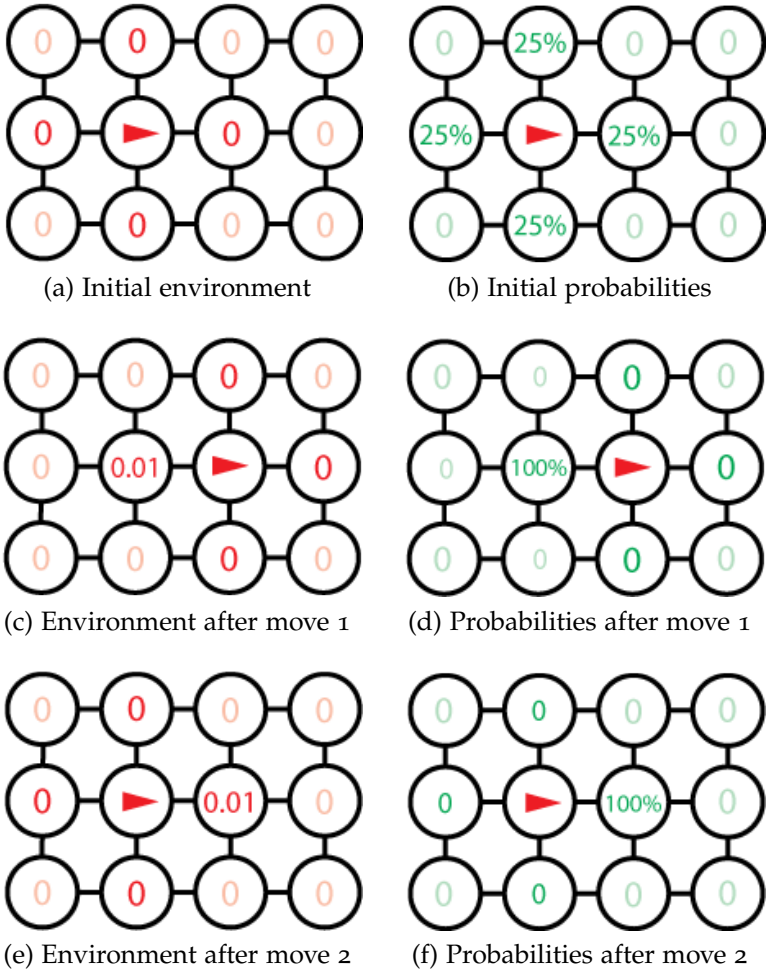
EXPERIMENTS AND OBSERVATIONS

4.1 PHEROMONE CONCENTRATION SENSIBILITY

Due to the model of node selection agents are sensitive to the pheromone concentration presented by the environment around them. This happens because the probability of choosing a neighbour node swings form 0 to a 100 percent 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 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: Affect 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 2 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.

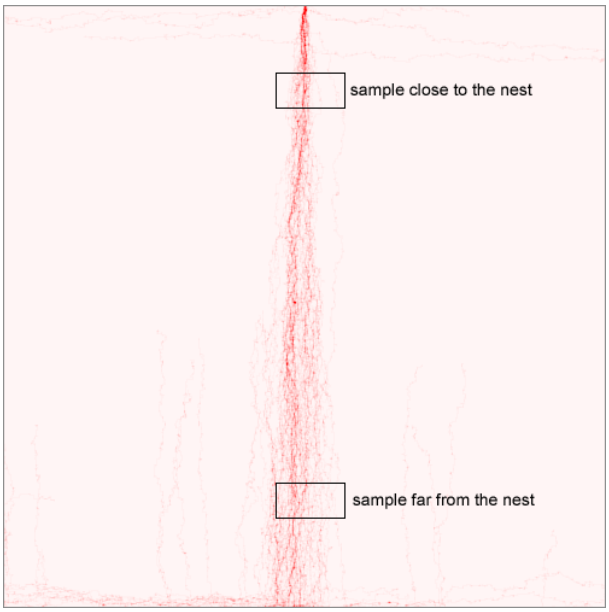


Figure 13: How the two samples of the environment is made

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.

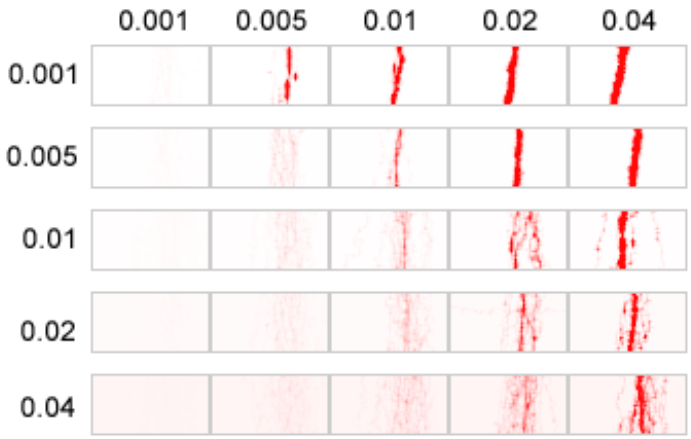


Figure 14: Resulting pheromone trail close to the nest

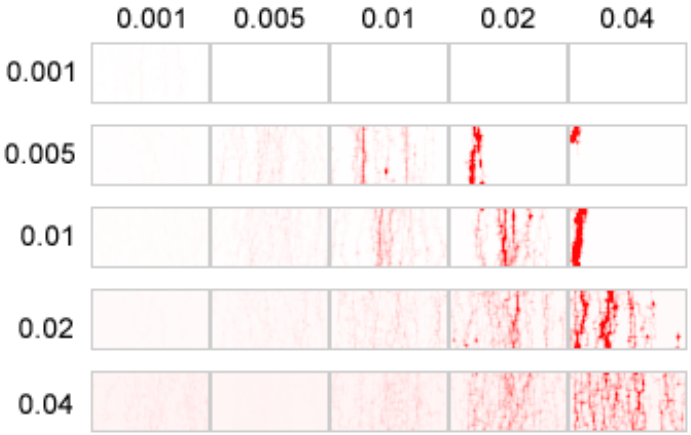


Figure 15: Resulting pheromone trail far from the nest

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.

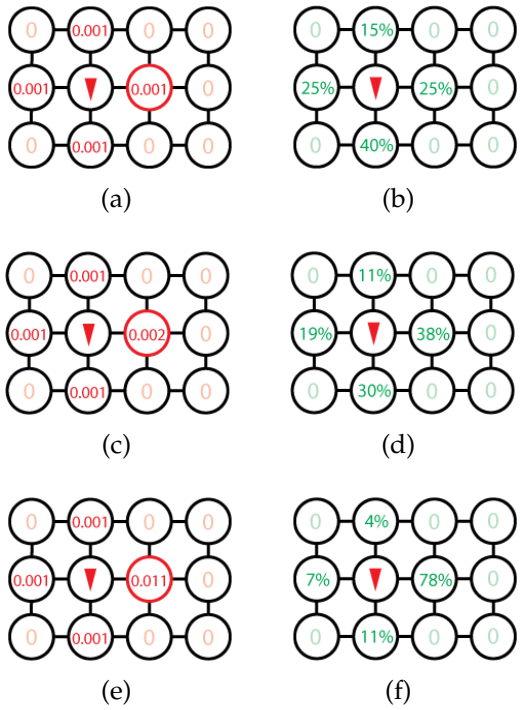


Figure 16: Shift of probability depending on agent update

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-like curve. 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.

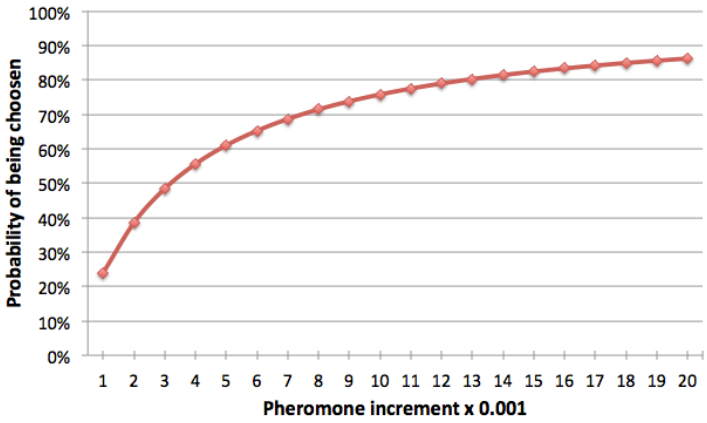


Figure 17: Increase of probability selection according to the increase of the deposit increment

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.

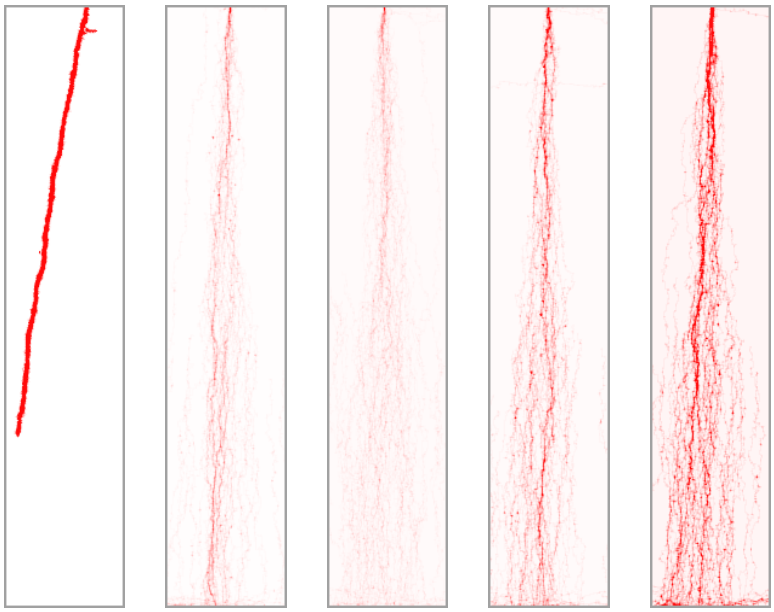


Figure 18: Complete trail pheromone trails left by simulation using 0.001,0.04 (a), 0.01,0.01 (b), 0.02,0.01 (c), 0.02,0.02 (d) and 0.4,04 (e) for initial pheromone concentration and update step respectively

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.

4.2 WARNING PHEROMONE RESPONSE

In this experiment the response of the agents to a warning communication stimulus is tested.

Describe the experiment setup.

Explain that the experiment is done in two phases, the first one the agents react only to amount of warning pheromone and the second one the agents react also to the number of other agents they meet that are traveling in the opposite direction and are not caring food.

Add block diagrams that explain the algorithms the agents use to decide on task selection.

Discuss the results. [experiment not done yet] Results should vary with the use of different parameters, such as the number of agents traveling in the opposite direction before the agent abort the current task and change to findNestAndHide task.

4.3 FORAGE RADIUS INVESTIGATION

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.

Explain how the variation of the radius actually impact the pheromone deposition. Add images to explain how the pheromone is actually deposited depending on the radius. Explain that the deposition follow 1/x.

Describe the experiment setup

Radius values used in the experiment were 0, 1 and 2.

Add simulation images.

Add the simulation data results

Discussion: Agents depend on the pheromone trail that they create to forage food, when the radius is 0 the agents are able to forage well, because the trail is well defined, but this indirectly limit the amount of agents recruited as the width of the main pheromone trail is very narrow.

Using radius equals to one the main pheromone trail gets wider allowing more agents to be pointed to the right direction.

Now when using radius equals to two, the pheromone trail gets too wide, this gets on the way of the agents as they do not have a clear direction to follow when they are within the trail.

Add image of an agent history, showing how much steps it spend inside the trail going up and down, going nowhere.

5

FUTURE WORK

5.1 MODEL IMPROVEMENTS

5.2 IMPLEMENTATION ISSUES

5.2.1 *Four way connected grid*

5.2.2 *Simulation handler*

Part III

APPENDIX

A

EXTRA EXPERIMENTAL RESULTS

Add extra images from the experiments described in the report.

B

MODEL AND SIMULATION SOURCE CODE

B.1 MODEL IMPLEMENTATION DETAILS

A more technical talk on the implementation of the model, with general UML diagrams.

B.1.1 *Documentation Annotations*

B.1.2 *Direction Enumeration*

B.2 SOURCE CODE

Add source code.

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] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] R. Dawkins. *The Selfish Gene*. Popular Science. Oxford University Press, USA, 1990. ISBN 9780192860927. URL http://books.google.co.uk/books?id=WkHO9HI7koEC.

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

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] W.R. Tschinkel. *The Fire Ants*. Belknap Press of Harvard University Press, 2006. ISBN 9780674022072. URL http://books.google.co.uk/books?id=vxt5BqOKEAIC.

[30] Karl von Frisch. *The Dance Language and Orientation of Bees*. The Belknap Press of Harvard University Press, 1967.

[31] 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.

[32] E.O. Wilson. *The Ergonomics of Caste in the Social Insects*. American Naturalist, 1968. URL http://books.google.co.uk/books?id=9wwzygAACAAJ.

[33] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2009. ISBN 9780470519462. URL http://books.google.co.uk/books?id=X3ZQ7yeDn2IC.

[34] F. Zhong, S.O. Kimbrough, and D.J. Wu. Cooperative agent systems: artificial agents play the ultimatum game. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 2207 – 2215, jan. 2002. doi: 10.1109/HICSS.2002.994150.