VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MULTI-AGENT STRATEGY GAME OVER ANTS

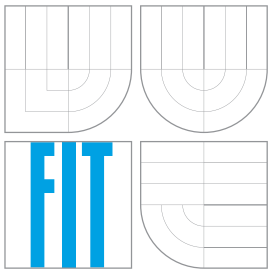
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH ŠIMETKA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MULTI-AGENTNÍ STRATEGICKÁ HRA S MRAVENCI

MULTI-AGENT STRATEGY GAME OVER ANTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH ŠIMETKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ KRÁL

BRNO 2013

Abstrakt

Tato práce se zabývá problematikou koncepce a vývoje multi-agentní strategické hry hrané v reálném čase. Práce analyzuje teorii těchto her a agentních systémů a výsledky této analýzy následně zohledňuje v koncepci samotné hry. Ta kromě herních prvků a ovládání běžně dostupných v současných RTS hrách implementuje několik úrovní umělé inteligence, kde každá jednotka je agent. Navíc nabízí jedinečný mód kooperace - společné ovládání jednotek umělou inteligencí a hráčem. Zároveň je hra koncipovaná tak, aby byla snadno rozšířitelná o nové umělé inteligence a mohla být využita pro výuku agentních systémů. Závěrečná část práce se soustředí na srovnání jednotlivých umělých inteligencí a zhodnocení efektivity implementace hry. Za tímto účelem byla provedena série automatizovaných testů.

Abstract

This thesis describes challenges in design and development of a multi-agent real-time strategy game. It discusses necessary theoretical background and its consequences for the game design. The resulting game implements, apart from features and game control which can be found in nowadays RTS games, three different levels of artificial intelligence in which each unit is an agent. Moreover, there is a unique cooperation mode, where units can be controlled by user and artificial intelligence at the same time. In addition, the game is designed in such way that it can be easily extended with new artificial intelligences, therefore, used for teaching agent systems. A number of experiments was performed in order to evaluate both game design and capabilities of artificial intelligence.

Klíčová slova

Multi-agentní systémy, RTS, Java, Jason, agent, strategická hra, rámec, umělá inteligence, BDI agent, reaktivní systémy, rozhodování, A*, ray-casting.

Keywords

Multi-agent systems, RTS, Java, Jason, Agent, strategy game, framework, artificial intelligence, BDI agent, reactive systems, reasoning, A*, ray-casting.

Citace

Vojtěch Šimetka: Multi-Agent Strategy Game over Ants, bakalářská práce, Brno, FIT VUT v Brně, 2013

Multi-Agent Strategy Game over Ants

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Krále a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Šimetka
May 15, 2013

Poděkování

Chtěl bych poděkovat vedoucímu bakalářské práce panu Ing. Jiřímu Králi za jeho ochotu, iniciativu, nápady a pomoc při vývoji hry.

Contents

Chapter 1

Introduction

Games are an integral part of our life. From early childhood we play games in order to develop various abilities that may be useful some-when in the future. Computer games are not an exception. In fact, it has been proven that some computer games may be more beneficial for modern human in their everyday life then classic games like hide-and-seek [16]. We can divide computer games into a number of genres which associate similar games. In this thesis we will be particularly interested in so called real time strategy games (RTS).

A real time strategy is a computer game where you take control of armies, direct their development and fight battles. The game is played in real time so there are no turns and therefore players with lower reaction times have an advantage. Game objectives usually are elimination of all enemy's buildings and/or units. However, we will be more interested in artificial intelligence of such games then in game-play itself. From the genre we can already determine some requirements for properties of the artificial intelligence. The AI has to consider a large number of inputs, process them and act accordingly in a timely fashion. In most RTS games the intelligence acts as a human - it manages all the units centrally and delegates actions they should take. We have decided to try another, admittedly more computer resource demanding, approach.

Every unit will have its very own intelligence and will be responsible for its actions. We hope that this way the unit's reaction will be much faster. All units will have a limited knowledge of the game environment so the decisions will not be fully informed. While this makes decision making simpler, it will be more difficult to coordinate units and exchange their knowledge of the environment. Probably the best approach would be a combination of centralized and decentralized decision making.

From the requirements on every unit we can deduce that it will act as an agent. The agent model introduces several qualities that will be useful in dynamic and fast-paced games like RTS. The agent model is an event-driven execution model providing proactive and reactive behaviour [6]. Every agent is in some environment and reacts on stimuli from the environment. The agents can also share the environment so that the actions of one will affect others. So the game itself will actually be a multi-agent system. The main feature of agent systems is the level of abstraction which is very close to human experience. We will also heavily use agents ability to communicate and delegate actions and information about the environment. The code of agents will be written in and interpreted by Jason [8] programming language and interpreter.

1.1 Goals

The purpose of this thesis is to introduce a development real time strategy game with a focus on the multi-agent artificial intelligence. The game we have developed is entitled the Anthill Strategy Game (TASG) and will be released as an open source program therefore, anyone can modify it. We will implement three levels of artificial intelligence to test different possible approaches (as discussed earlier) and how they compete against each other. Below you can see the list of goals of this thesis.

- Create a multi-agent game and test whether the Jason can handle powering the RTS game on an average computer.
- Document the program so somebody else can modify it if they are keen.
- Evaluate results for implemented AIs and compare them with one to another.

1.2 Overall Structure

The thesis is structured as follows: The first three chapters of this thesis provide essential theory background for understanding the game implementation. As the main focus of the game is the artificial intelligence, especially multi-agent systems, a brief theory can be found in chapter 2. The Anthill Strategy Game has all the distinctive features of the real time strategy game. We will discuss them in chapter 3 along with a brief history of some A class games from the RTS genre. Chapter 4 introduces the implementation of AgentSpeak language in the Jason programming language. We use the Jason to program artificial intelligence in the game.

The specific game mechanics for The Anthill Strategy Game along with AI designs can be found in chapter 5. More in depth explanation of how the game works describes chapter 6. There we will focus on the implementation details and how the game can be modified to fulfil players needs. The last but one chapter 7 puts the game through the series of tests. We will compare performance of different levels of the artificial intelligence with one to another.

In the final chapter 8 we will summarize the development, achieved results and implemented game features. Because we see potential in this project, we will also discuss improvements we plan to implement in the future. More in depth implementation details along with some customization tutorials and game analysis can be found in annexes.

Chapter 2

Theoretical Background

In this chapter you will be introduced into multi-agent systems, how do they work and what are their capabilities. We will briefly talk about reactive system, more precisely agents, and theirs properties. To approach the topic of reactive system we will demonstrate how does the development of (multi-)agent systems differ from functional programming. We will take a closer look on main aspects of agent oriented programming which is the ability to store information about environment and use them for agent's benefit. This characteristic will be closely described on BDI agents, which extended version is implemented in the Jason [8] programming language. The main focus in this chapter will be on decision making process for BDI agents.

2.1 Reactive Systems

To help you better understand reactive systems problematic, we can compare it to functional programming. In functional programming the whole program is a mathematical function of some sort:

$$f : I \rightarrow O \tag{2.1}$$

where I is some domain of possible inputs and O is a range of possible outputs. This approach to software development is well established, and from the standpoint of software development, straightforward to engineer. However, many programs don't meet reasonably simple input-compute-output structure. We call them reactive systems.[2]

Every reactive system is situated in an environment which provides inputs for the system and which is affected by the system's actions. Reasons why we can't use functional programming for reactive systems are both the dynamic character of theirs environment and the necessity to maintain a long-term ongoing interaction with their environment. Upon that, the number of possible inputs from environment and states in which the system can be found is too high, or even infinite, so using standard functional approach is proven to be difficult or even impossible. The world *reactive* describes the nature of how are handled any environmental changes by the system. The reactive system chooses action (reaction), based on state of its environment, to influence the environment. Examples of a reactive system would be operation systems, web servers, anti-virus programs or plane's autopilot.

2.2 Agent

An agent is a subset of the reactive systems. More precisely the agent is anything that can be viewed as perceiving its environment through sensors *(e.g. eyes, cameras)* and acting upon that environment through actuators *(eg. wheels, arms)*. You can see schematic example of the agent in the figure 2.1. The term percept refers to the agent's perceptual inputs at any given instant. The complete history of everything that agent has ever perceived is called percept sequence. In general, an agent's choice of action at any given instant can depend on the entire percept sequence.[5]

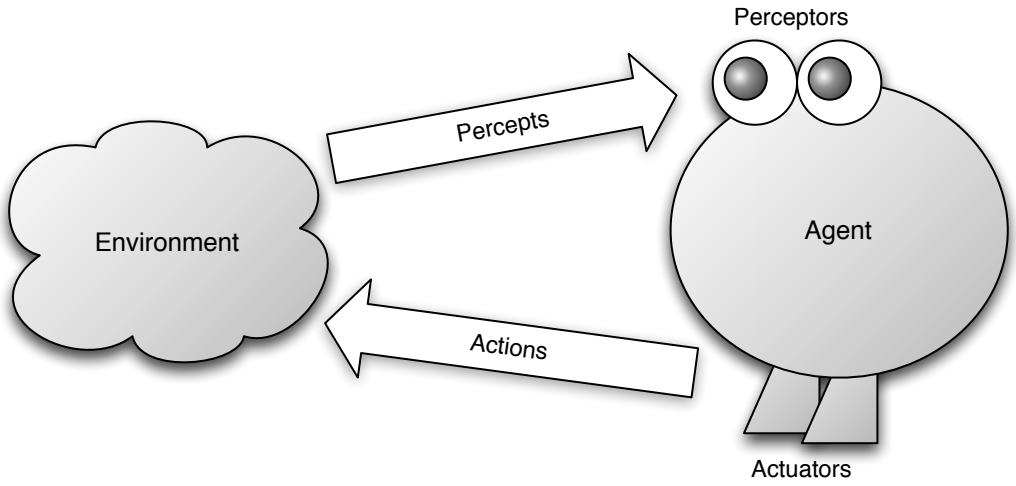


Figure 2.1: Scheme relation between agent and environment.

Mathematically the agent's behaviour can be described by an agent function which maps any given percept sequence to an action. This function is implemented by agent program. We can write:

$$f : PS \rightarrow A \tag{2.2}$$

where PS is the percept sequence and A is a list of possible actions. Because the percept sequence can be infinitely long, we usually limit it to reasonably long subset which is represented by agent's internal state. There is a number of architectures for representing agent and its inner state. We will talk more about this in section 2.4.

Characteristics of Agent

We can distinguish four main properties which each agent should have (as suggested in [2] and [7]):

Autonomy

The agent decides how best to act in order to achieve its goals without direct intervention of humans or others. The actions which the agent can take to achieve its goals are bounded by the plans that has been given to him by humans or others. Also the agent has to have some kind of control over its actions and internal state.

Social ability

The agent has an ability to interact with others using some kind of agent-communication language. By communicating it is able cooperate with other agents in order to accomplish goals. This can be achieved by introducing the ability to delegate the agent's beliefs, goals and plans to it's peers.

Reactivity

Reactivity is the ability of the agent to respond to environmental changes in timely fashion with an effective balance between *goal-directed* and *reactive* behaviour. *Goal-directed* behaviour is an execution of a plan in order to achieve goal while *reactive* behaviour is a series of actions designed to react upon environmental change. For example moving towards a desired location on a map is the goal-directed behaviour and attacking on an enemy, who has been discovered in the process, is the reactive behaviour.

Pro-activeness

The agent is able to exhibit the *goal-directed* behaviour by taking initiative. By initiative we mean active, and if necessary repeatable, behaviour represented by choosing a course of actions in order to achieve a goal.

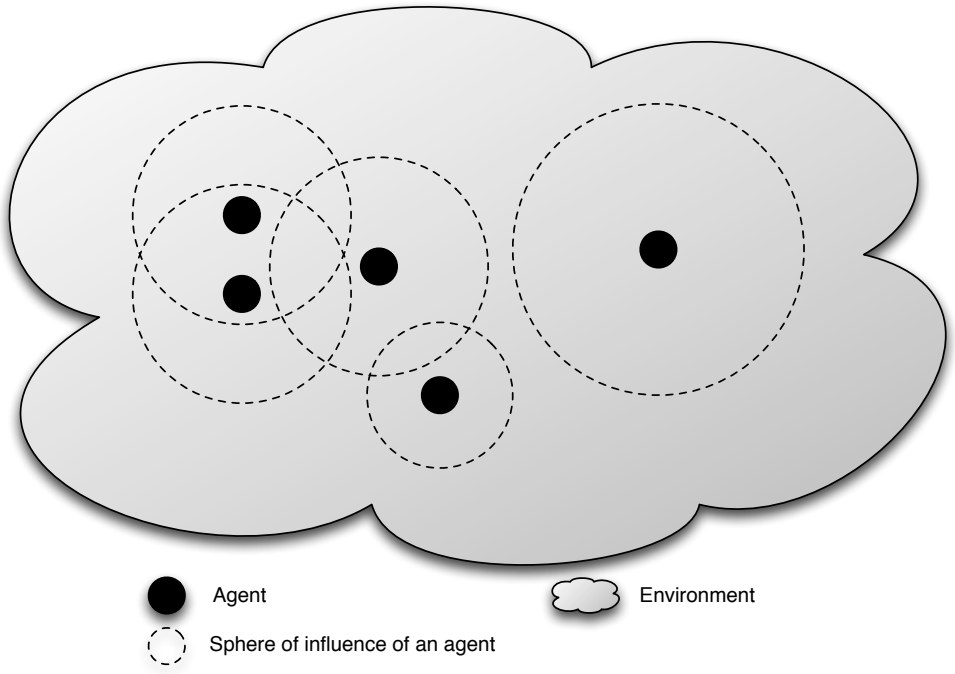


Figure 2.2: Scheme of multi-agent system demonstrating influence of agents.

2.3 Multi-agent Systems

Most systems have more then one agent sharing the same environment. We call them multi-agent systems. In such system each agent can influence its part of shared environment, so called sphere of influence. Parts of environment can be controlled by one agent (spheres of influence don't intersect) or by a group of agents (spheres of influence overlaps). This makes everything more complex, because each agent has to take into account how other agents, with some control over the environment, are likely to act. We can identify a need for the agent to interact with one another. The agent has to have a social ability; a way to delegate goals, believes and plans and the ability to pass its percepts to others. You can see an example of multi-agent system in figure 2.2.

2.4 The BDI Agent Architecture

Belief-desire-intension (BDI) model is based on a model of human behaviour that was developed by philosophers. BDI model introduces some sort of *mental state* into computer programs. Main parts of BDI model can be described as follows [2, 4, 6]:

Beliefs

Beliefs are information the agent has about its environment and are stored in so called *belief base*. Because the environment is dynamic, and the agent can't perceive the whole environment, beliefs may be out of date or inaccurate. In the Jason, a belief is stored as a predicate, for example information about temperature in Brno would be stored like `(temperature(brno,28°))`. Because the belief base has to be consistent, it is advisable to use some sort of time stamp that may help preventing data conflict.

Desires

Desires represents all the possible states of affairs that the agent might like to accomplish. However, having a desire, does not imply that an agent acts upon it. Desires are possible influencers of the agents actions and may be incompatible with one another. We can think of an desire as being an option for the agent. Let us try to make it more clear using following example. An agent has two desires: to bake bread and cake. For both of them it needs to have eggs, however it has enough for either bread or cake. By baking cake, agent makes it impossible to achieve second desire and vice versa.

Intentions

Intentions are the states of affairs that the agent has decided to work towards. By other words, the intention is the option which the agent chooses and is committed to fulfil. The agent is bound to follow execution of intention until it is achieved while believing such intention is achievable. Agent may drop an intention under these conditions: Either the intention has been fulfilled or it is believed not being achievable any more.

From these definitions we can see that beliefs, desires and intentions are the key data structures for holding informations about agent's internal state.

2.5 The Reasoning System

Now that you know how are the information about environment and agent's desires stored, we can describe how they can be used. After obtaining some new desire the agent has to decide what to do and which one to follow, if there is more then one. The reasoning system, based on BDI agents, is responsible for agents behaviour. It chooses an intention or a goal to follow and course of actions that may lead to achieving it.

Practical Reasoning

The decision making for BDI agents is called practical reasoning. It consist of two distinct activities: deliberation and means-ends reasoning. Practical reasoning is a process of selecting appropriate action based on agent's state. Practical reasoning is a matter of weighing conflicting consideration for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what agent believes in [6].

Deliberation

The process of selecting intention to follow is called deliberation. It vastly depends on implementation therefore we will discuss it more in chapter 4.

Means-Ends Reasoning

Means-End reasoning is responsible for planing course of actions in order to achieve intention. Its input is:

- An intention or a goal that the agent wants to achieve.
- The agent's current beliefs about the state of the environment.
- List of possible actions available to the agent.

The output of Means-Ends reasoning (planner) is a course of actions (plan) which execution will result in achieving goal/intention. Please note that the selection of actions by agent itself is computationally very costly. Therefore nowadays its up to the programmer to create a collection of plans and the task of the agent is to choose from these plans.

Computational Model for BDI Agents

In figure 2.3 you can see the implementation of reasoning system for BDI architecture. It is an application of principles mentioned before. Based on agents internal state (context + goals), the interpreter chooses plan to follow from a plan library and executes it's actions (plans body). [6]. A variation of BDI computational model is implemented in Jason. You can find more information in chapter 4.

AgentSpeak Language

Some interpreters, including Jason, use so called AgentSpeak language for writing Agent's behaviour through plans. The AgentSpeak language is a formal abstract agent programming language. A plan in AgentSpeak consists of following components:

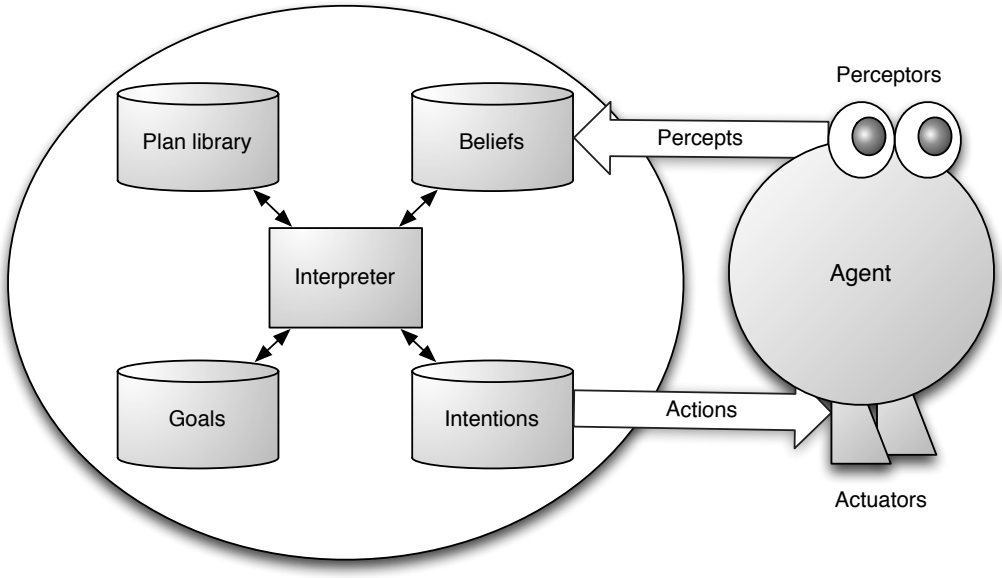


Figure 2.3: Schematics of BDI agent computation model.

- `A goal`: The post-condition of the plan. What will believed to be achieved after fulfilling all actions of the plan.
- `A context`: The pre-condition of the plan. What has to be believed for the plan to be applicable.
- `A body`: The course of action to execute. What has to be done in order to fulfil goal.

Interpreter stores plans in Plan library and for each goal, based on it's reasoning system, it chooses adequate plan. In this thesis we will be using Jason [8] which uses slightly modified version of the AgentSpeak language. For more details please see chapter 4.

Chapter 3

Real-Time Strategic games

In this chapter you will be introduced into game mechanics of a real-time strategy games. We begin with definition of the RTS game and a brief introduction into their history. Some important games will be mentioned along with their game features and consequences for nowadays RTS games. Then we will focus on the game-play of nowadays RTS genre. We will also explain the concept of micro and macro-management and how they affect the game.

Figure 3.1: A screenshot from the StarCraft (1999) game. Courtesy of Jakub Dušek.

3.1 Evolution of RTS

The Wikipedia [14] defines Real-Time strategy (RTS) as a sub-genre of strategy video games which does not progress incrementally in turns. First RTS games didn't had much in common with nowadays RTS. However, it is worth mentioning, that the RTS sub-genre initially evolved separately in the United Kingdom, Japan and North America. Officially the very first RTS game ever created is *Utopia*[12] released in 1981. However, it didn't had much in common with nowadays real time strategy games and can be categorized as a simulation game. First RTS game that unified shattered game scene and introduced the game mechanics was *Warcraft: Orgs & Humans* (1994) [9]. Even though it was quite easy to fool the AI, the game became an instant blast thanks to the multi-player mode. The successor, *Warcraft II: Tides of Darkness* [9], improved game control and introduced new game mechanics. One of the most significant improvement was the fog of war, which covers the area until a unit is nearby. The genre was shaped ever since then by a number of games, namely *Command & Conquer: Red Alert*, *Total Annihilation*, *Age of Empires*, but it was Blizzard with the game *StarCraft* [10] to set the trend for next years. [1, 3]

Figure 3.2: A screenshot from the game StarCraft II: Hearth of the Swarm (2013).

The StarCraft was the first game to be played on a professional level. Especially in South Korea the game was, and still is, extremely popular. In fact it still is played competitively. Even though several RTS game titles are released every year, only some of them are played at competitive level: *Warcraft 3*, *Command & Conquer 4*, *Age of Empires 2*. But nowadays all the attention goes to *StarCraft II: Wings of Liberty* [11]. The original game was released in July 27 2010 with an expansion *StarCraft II:Hearth of the Swarm* released few months

later. Starcraft II pushed the eSports to the new level. There is a number of major tournaments with prize pool over $10,000 held all over the world. Probably one of the best known are MLG, GSL and IPL [13]. StarCraft II has become a business where professional players are trained in teams just like in any other sport. The eSport is becoming as big as the most successful traditional sports like basketball, hockey or even soccer.

3.2 Gameplay

As you can see RTS games went through a long evolution. In this section we will explain how does the game mechanics work. The goal of the real time strategy game is elimination of all enemies' buildings and/or units, using strategic thinking and planing. Real-time strategy game is usually controlled by a mouse and a keyboard short-cuts to issue commands and scroll over the game area (map). The game is divided into two distinct sections: a map and an interface for commanding units. The map area is displaying the game world; terrain, units, and buildings, and is used for selecting units and issuing commands. The interface usually contains command and production controls and often a minimap which shows a smaller version of entire map and can be used for navigation. Most real time strategy games are generally fast-paced and requires very quick reflexes. How fast you can issue commands can be measured in so called actions per minute (APM) value which is usually updated every few seconds.

The strategy used for winning the game generally involves a positioning of units in a favourable position. In most RTS games there is more than one type of units. Generally certain type of units is better against one type but terrible when facing another type. By choosing the right unit composition, you increase your chance of winning battles and minimize the casualties. However, not all types of units are available in the beginning of the game. A certain buildings (according to technological tree) has to be build in order to unlock/create some types of units. Most of the RTS games has some sort of upgrades for units which can give them new abilities or improve theirs properties. By using building to produce new units the player has to build an army and use it to either defend themselves from enemies' units or to eliminate enemies who possess bases with unit production capacities of their own.

New units and upgrades are usually funded by gathering a set of resources. There exist a special type of unit(s) which can mine resource from resource fields and collect them into specific building(s). In addition to gathering resources these units, also known as workers, construct new buildings. Workers are usually very fragile and their use in combat is inefficient. Some titles limit the maximum amount of units existing in the game to a certain value. [14]

Micro-management

In the context of real-time strategy games a micro-management refers to a situation when a player's attention is directed more toward the management and maintenance of their individual units. Micro-management frequently involves the use of combat tactics. It requires a constant interaction with the game and is usually quite action demanding. In games like StarCraft II, micro-management is used more at professional level of game-play.

Macro-management

Macro-management refers to situation when a player's focus is directed towards gathering resources and building new structures/units. It allows players to think and consider possible solutions and adapt their unit composition to new situations. Macro-management tends to look to the future of the game whereas Micro-management tends to the present. Usually the macro-management is nowadays more efficient because it allows user to overcome micro-management disadvantage especially at lower level play.

Chapter 4

Introduction into Jason

In this chapter you will be introduced into the Jason programming language and it's interpreter. A basic knowledge of logical languages is expected in order to fully understand. In first part of this chapter we will talk about basic language constructions and the overall idea. The second part briefly introduces how is the decision system implemented and how does it differ from principles described in chapter 2.

4.1 Language Constructions

The Jason programming language uses the AgentSpeak language in a Prolog-like fashion. It uses predicates and terms to store information and execute actions.

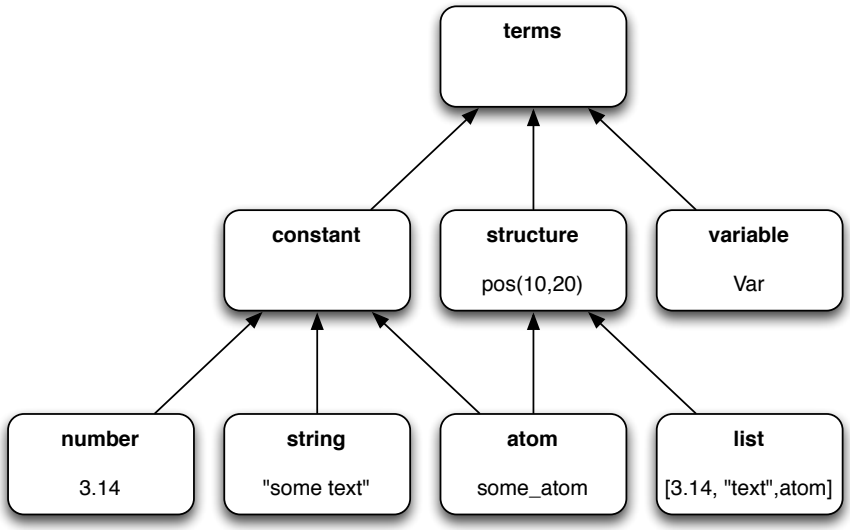


Figure 4.1: Term hierarchy in Jason.

Terms

As in Prolog, the basic data type in Jason is a term. It is an abstract type that encapsulates lower level types and unifies work with them. Another type inherited from Prolog is an

atom. Any alphabetic symbol, or sequence of symbols, starting with a lower case letter is called an atom, which represents particular object and is equivalent to constant. Terms can also carry any *number* value or a *string*. All these basic components can be combined to create *lists* or *structures*. Any term can also be represented as a variable. In Jason, a *variable* is atom starting with upper-case letter. What will the variable represent is determined by process called unification which will be explained in section 4.2. There is a special variable _ , so called anonymous. The anonymous variable is used each time we don't care about the value of the term. In figure 4.1 you can find a graphical representation of terms with examples of all types we've talked about.

Beliefs

Beliefs are used for storing informations available to an agent. It can be stored in positive meaning or as a negative counterpart.

```
enemy(10, 100).
~enemy(10, 100).
```

The literal enemy(10, 100) means that there is an enemy at location 10, 100. When ~ is added before the belief, the meaning changes - the agent believes there is no enemy in location (10,100). The ~ sign is called strong negation. New beliefs can be added to an agent from number of sources: environment, another agent or agent itself. The source of origin is saved in annotations. For example the following belief was obtained from both environment and agent called *eva*.

```
enemy(10, 100)[source(percept), source(eva)].
```

Goals

Goals are another basic language construction. Goals play a role of desires - what the agent wants to achieve. There are three types of goals in Jason as you can see in table 4.1.

!g	achievement goal	The current Agent goal is paused until the goal *g* is fulfilled.
!!g	achievement goal	Add new goal to achieve but don't wait for it's execution. This is used in recursive goals.
?g	test goal	Tests if goal is achievable and returns true if so.

Table 4.1: Type of goals in Jason.

Plans

In section 2.5 we have introduced a concept of plans. Plans in Jason are based on the AgentSpeak language and have three distinct parts:

```
triggering_event : context  <- body.
```

Triggering event

The triggering event is the implementation of two key agents characteristics - pro-activeness and re-activeness. There are two types of changes in an agent's mental attitudes: changes in beliefs and changes in agent's goals. These changes can be of two types - addition and deletion. The table 4.1 shows available types of triggering events.

+l	Belief addition.
-l	Belief deletion.
+!l	Achievement goal addition.
-!l	Achievement goal deletion (fail goal event).
+?l	Test goal addition.
+?l	Test goal deletion (fail goal event).

Table 4.2: Triggering events in Jason

Each time a belief or a goal is added, a triggering event is created and an applicable plan selected. While it is OK for beliefs to not have any selectable plans, if a goal addition (`+!g`) is triggered and there is no selectable plan a fail goal (`-!g`) is generated. If no fail goal is provided or applicable, the interpreter writes warning. The plan selection in Jason is driven by so called practical reasoning system explained in section 4.2.

Plan context

The plan context can contain a condition that determines whether or not the plan is applicable. Literals in condition can be either in conjunction (denoted by &) or in disjunction (denoted by |). As a literal we can use a whole belief. If such belief is in the belief base, plan is applicable. However, sometimes we want to select plan only when a certain belief is not in the belief base which can be done by operator `not`. However, the biggest strength of the plan context is the ability to use internal actions and relation operators.

Plan body

The body of a plan defines a course of action for the agent to do when the plan is selected. Each action is separated by ';' and the plan body ends with dot. There are six different types of formulae that can appear in a plan body:

- **Environment actions:** Environment actions are analogy to agents actuators. By calling an environment action, the agent should change environment in some way. After each environment action, all agent's percepts should be updated updated.
- **Achievement goals** In plan body we can have a number of achievement goals. This approach is very convenient because difficult problems can be disassembled into small goals. If a goal appears in the plan body, it is added to agent's goals stack. Execution of this plan is up to reasoning system of the agent and does not have to be immediate. Because each goal returns true or false value, the plan's execution is halted until the new goal's execution is finished. If newly created goal fails it returns false and the plan is aborted with `-!g` event triggered. If we don't need to wait for an execution and return value of newly created achievement goal, a `!!g` can be used.

- **Test goals** The use of test goals in plan body allows us postponing context part of the plan. Note that behaviour of test goal is similar to behaviour of achievement goal.
- **Mental notes** Sometimes the agent needs to store some information into the belief base or withdraw them. Operation `+b` adds new belief to agent's belief base whilst `-b` removes belief. Any belief added to belief base will have `[source(self)]` annotation and only such beliefs can be removed by the agent.
- **Internal actions** Sometimes it is useful, or even required, to execute part of the plan in Java. Internal actions can do just that. You can use them to retrieve information or calculate some values that just are not available in Jason or the implementation would be too slow. There are some predefined internal actions (like .print(..), .send(...)) which can be used.
- **Expressions** In plan context and body you can also use several expressions. There are both arithmetic and relation operators available in Jason.

Below you can see example of a plan that is triggered when goal `g` is added, applicable if there is belief `b(X)` and `X` is greater then 1. The plan body increases value `X` by one, stores it to variable `Y` and calls the internal action .print() to write the value into Jason's MAS console.

```
+!g
: b(X)
& X > 1
<-
Y = X + 1;
.print(Y).
```

4.2 Program Evaluation

The Jason programming language is logical languages. In these languages programmer specifies facts and rules but does not write the evaluation itself. The flow of the program is controlled by the interpreted and rules are recursively used to solve satisfiability problem. In Jason this technique is applied to both rules and plan selection. How precisely the computation works is described below.

The Reasoning Cycle

The term reasoning cycle refers to an evaluation of sequence of conditions in order to determine which action (or plan) will be executed within next agent's cycle. Each reasoning cycle checks perceptible part of the environment and determines best action to perform. This being said we can see that re-activeness of the unit depends on frequency of reasoning cycle. From now on we will use a term reasoning cycle as a synonym to the process of making decision. For more information on how reasoning cycle in Jason's interpreter works consult the literature [6].

Jason's Practical Reasoning System

The Jason programming language uses practical reasoning system (PRS) for selection of a plan to be executed after each reasoning cycle. When the agent starts up, the goal to be achieved is pushed onto a stack, called the intention stack. This stack contains all the goals that are pending achievement. The agent then searches through its plan library to see what plans have the goal on the top of the intention stack as their triggering event. Of these, only some will have their context satisfied, according to the agent's current beliefs. These plans become the possible options for the agent and are called applicable plans.

The process of selecting between different possible plans is deliberation. The simplest method how to choose from applicable plans is to use utilities for plans. These are simple numerical values; the agent simply chooses the plan that has the highest utility. The chosen plan is then executed in its turn. Execution may involve pushing further goals onto the intention stack, which may then in turn involve finding more plans to achieve these goals, and so on. The process bottoms-out with individual actions that can be directly computed.

Please note that the order of plans matters. Jason's reasoning system selects from applicable plans by their position within source code. This means that the first applicable plan is always selected.

Unification

The Prolog-like unification is used in Jason for evaluation of expressions and bounding free variables. Two terms equals when one of the following rules applies:

1. Terms are identical.
2. Variables in both terms can be substituted in the way that rule 1 applies.

The decision if term S is unifiable with term T is directed by following algorithm:

1. If S and T are identical constants, then they are unifiable.
2. If S is variable and T is term, they are unifiable and S is substituted for T.
3. If S and T are structures, they are unifiable under these conditions:
   (a) They have same functor and parameters count.
   (b) Pairs of parameters are unifiable.

Chapter 5

Design of the Game

In this chapter you will be introduced into the design of the Anthill Strategy Game. In first few sections we will briefly talk how is the game controlled, available game modes and user interface design. After reading these sections you should be able to play the game yourself. The last section contains designs of available artificial intelligences. Although this is the crucial part of this thesis, it is not mandatory knowledge for the player.

5.1 Game Modes

The Anthill Strategy Game offers two game-play modes. It can be either played as a real time strategic game for a single player versus computer opponent(s) or it can simulate battles between computer controlled sides.

Single Player Mode

In the single player mode the player takes command over an anthill and all its units. The player is responsible for creation of new units and managing upgrades. Units obey commands and do whatever he or she commands them to do just like in any other RTS game. However, unless unit is put on hold or has any action to perform, it auto-attacks any enemy unit within its perception range. Also collection of resources is repeatable action which requires only initial command and then is performed automatically until the resource field is depleted.

Nevertheless, managing all units can be quite action demanding and stressful, because the game is played in real time. To help the player succeed in the game we're introducing assistance mode. When assistance mode is chosen, units are driven by artificial intelligence of users choice, so the player can fully concentrate on managing upgrades and creation of new units. Of course he or she can interfere with unit control and for selected units override behaviour, but once all player's actions are fulfilled, unit returns under command of the chosen AI (see section 5.5).

Simulation Mode

The simulation mode is primarily for testing capabilities of artificial intelligences available in game. In this mode you can witness battles between computer controlled units. You can still select any unit to review its status, upgrades and actions to perform but you can't

influence the game in any way. Information about every team of ants are shown in the bottom panel.

Experiment Mode

The experiment mode is basically simulation mode which is repeated. We use this mode to repeatedly test agents behaviour so we can compare their effectiveness. Data from each experiment are stored to folder experiments.

5.2 Game Mechanics

The Anthill Strategy Game mechanics is based on other RTS games. There are units under the command of player and/or AI which can freely move across map and gather resources or attack enemies' units. Gathering resources allows creation of new units and purchasing upgrades. Game ends when there are units of only one alliance remaining in the game.

Units

There is only one type of units in the Anthill Strategy Game. These units starts with 10 attack and 0 armour, can collect resources and attack at me-lee range. Nevertheless, with a little modification of source code, you are able to create uneven types of units because every attribute is stored for every unit separately (see chapter 6). Some attributes can be upgraded from anthill control panel and are shared across all units from one team. The list of attributes can be found in appendix C.1.

Anthill

The anthill stores information about collected resources and upgrades. For every player there can be only one anthill which is the starting location for every unit. The anthill is also a place, where a unit has to bring a resource in order to collect it. For AI driven team there is a special intelligence managing anthill logic whereas in single player mode, the anthill role is taken by player. He or she can see amount of available resources, army size and upgrades status. Based on collected food and water the player can purchase upgrades and create new units in order to win the game.

Resources

There are two types of resources in the Anthill Strategy Game - food and water. Both of these resources are necessary to create new units and upgrade them. However, upgrades are more water demanding while creating new unit requires more food. In order to collect new resource unit has to discover it and carry to anthill. New resources appear on map randomly (the rate and and chance for new resource to appear can be set in settings). Also when a unit is killed, it drops any resource it carries and its body turns into resources as well.

Upgrades

As we suggested before, some attributes of units can be upgraded. All upgrades are purchased by anthill (or player) and are shared across all units under anthills command. The

table C.1 in appendix shows prices of upgrades available from game. Please note that the price of an upgrade increases with its level.

5.3 Game Control

This section describes how is the game controlled by the player. Control wise the Anthill Strategy Game doesn't differ from other RTS games. Because we have just one type of units which are produced at one building, management of units and upgrades can be simplified.

Unit Control

You can select any unit or resource by clicking or dragging the left mouse button. If multiple game elements lies within selection, what will be selected is determined based on priority. Elements has following descending priorities: player controlled units, other units, resources. If at least one user controlled unit is selected, it can take 5 basic commands: *move, attack, gather, stop* and *hold*. The further description is in appendix C.2. Please note that every action can be triggered by keyboard short-cut. To review all available short-cuts see table C.2.

Units and Upgrades Management

There is handful of buttons for managing anthill. From anthill control panel (see section 5.4) you can upgrade or create new units by clicking on adequate button. Please note that every anthill command can be triggered by keyboard short-cut. To review all available short-cuts see table C.3.

Keyboard and Mouse Short-cuts

For more convenient and faster game control we have added number of keyboard short-cuts. Table C.2 describes useful short-cuts for controlling units. Table C.3 shows keyboard short-cuts for managing upgrades and creation of new units and table C.4 describes overall game short-cuts. Some short-cuts are not mention because they are used for navigation in menus and therefore are not relevant for game-play itself. However, short-cut that triggers some button in menu is written right after button name in brackets. Please note that some control can not be achieve differently then by using keyboard short-cuts.

5.4 Game Interface

Even the game interface of the Anthill Strategy Game is heavily inspired by other RTS games. The basic layout divides the whole game into 5 basic blocks as seen in figure 5.1 and is explained below. Depending on presence of player some panels changes theirs content but the overall layout remains.

1. **Map** shows part of the environment with all its units and resources. Map is used for selecting and commanding units.

2. **Minimap** displays complete map with simplified game elements. You can use minimap to easily navigate across map.

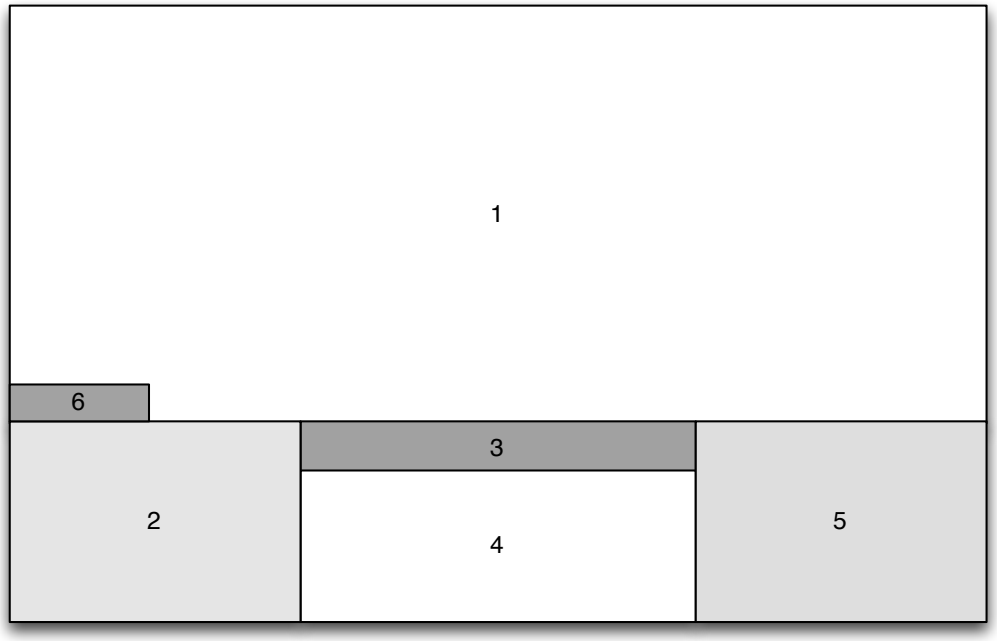


Figure 5.1: Basic game interface layout.

3. **The game overview** panel displays information about current game like number of resources and units. Information shown differs for single player mode and simulation mode.

4. Using **the actions** panel player can manage upgrades and units (in single player mode) or review detailed information about each competing side (simulation mode).

5. **The selection** panel displays relevant information for selected game elements. If a single unit is selected, it displays all known information about it. For a group of units, it shows them in a grid so player can easily review their statistics and select individual units. If resources are selected, the selection panel displays number of remaining resources.

6. **Clock** displays game time from the start of the game. Please note that game time depends on game speed and is not actual game duration in real time. Game time describes number of rounds for all units to move or do some action. At game speed 100, one game second should be one real time second but it depends vastly on CPU load and threads switching.

Single player mode

User interface for single player offers a way to manage upgrades and control units. Both upgrades and available actions for selected units are situated in the actions panel.

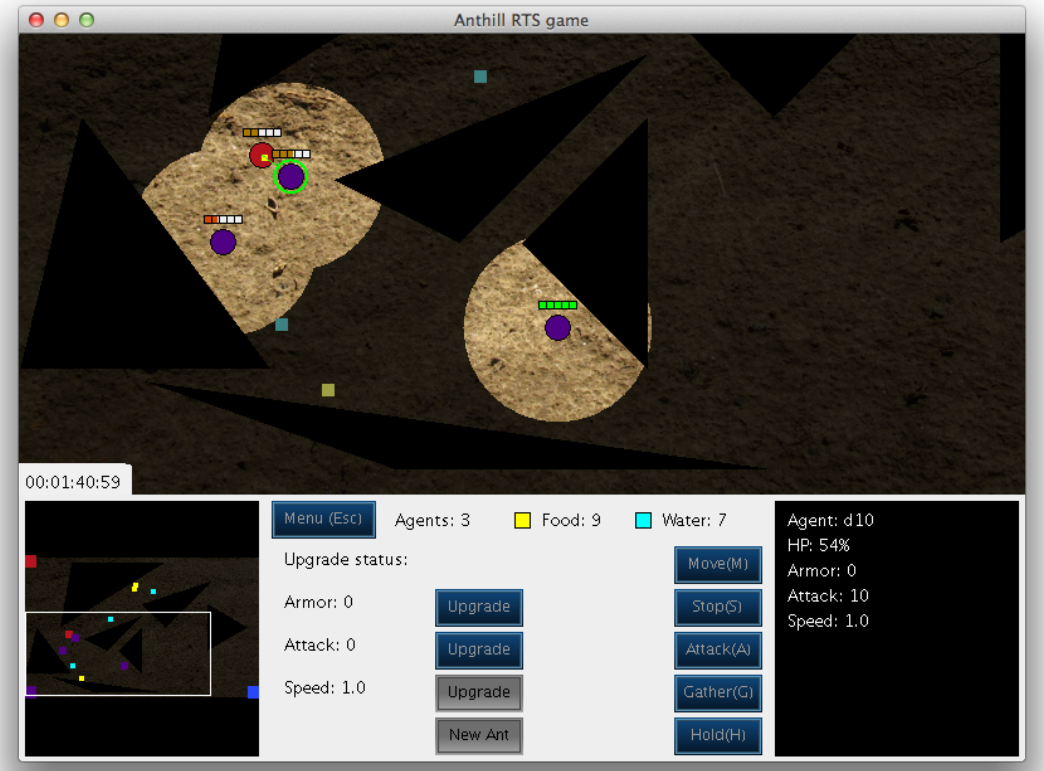


Figure 5.2: Single player user interface.

Simulation mode

In the simulation mode, the action panel is missing and is replaced by information about all factions. Also the game overview panel contains total number of ants and amount of uncollected resources in the environment.

5.5 Draft of Artificial Intelligence

We've implemented three different levels of artificial intelligence in the Anthill Strategy Game. The basic level of intelligence doesn't communicate and therefore every unit works for itself best benefit. The second intelligence, called cooperative, is fully informed about perceptions of other friendly units and can commit it's beliefs to achieve greater benefits across community of friendly units. Last level of AI goes even further. It doesn't just blindly perform best possible actions that are available now but it plans ahead by prioritizing collection of one resources and allocating them to ants.

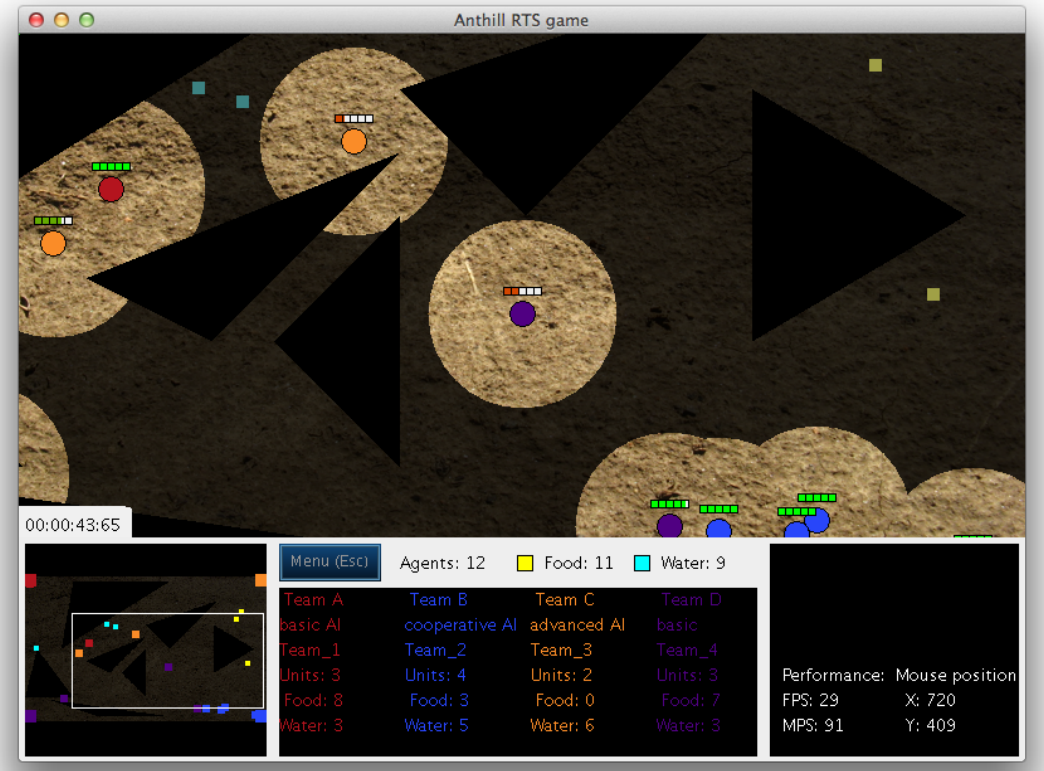


Figure 5.3: Simulation mode user interface.

Basic Artificial Intelligence

In basic artificial intelligence we don't have to worry about units' cooperation. Bearing this in mind, we can design a simple solution using just a few „conditions", as you may see in the reasoning cycle in figure 5.4. All we need to do is to keep somehow an information about unit's state. Unit moves around randomly until a resource or an enemy is discovered. If some resource is discovered it is pushed from the environment as a belief. However, once unit walks away, such belief would be destroyed. Therefore we use mental notes to keep information about resources in belief base and achieve persistence. This mental note is then updated each time the resource is perceived. Also when a resource is successfully brought to anthill, the ant decides which resource is the closest one to pursue. When unit encounters enemy, it checks it's strength and decides whether it is advisable to attack or to flee. To avoid heavy computing, when there is more then one enemy, the decision is influenced just by the ratio between number of enemy units and friendly units in perception range. The only struggle we have to deal with is a scenario, where the unit carries resources and enemy unit appears within its perception range. Solution that is described here prioritizes enemy unit over resource collection.

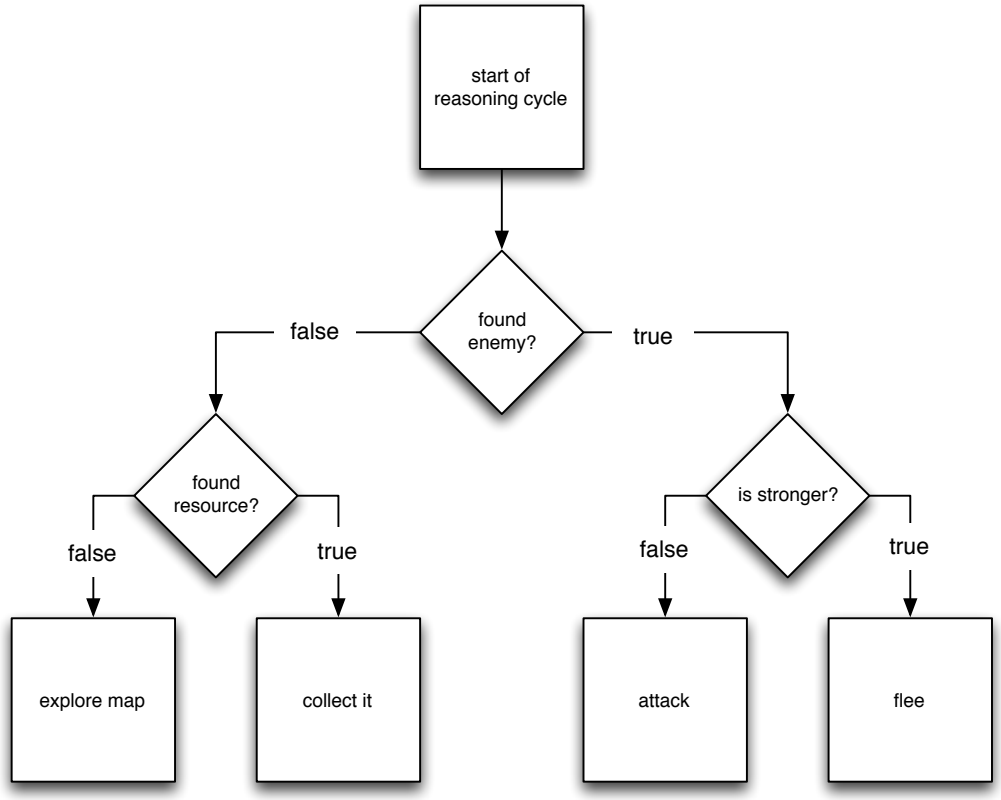


Figure 5.4: Basic artificial intelligence reasoning cycle.

Cooperative Artificial Intelligence

Another step towards human intelligence is to introduce cooperation across units (outlined in figure 5.5). This is achieved by alternating reasoning cycle to allow communication between units. Each time a resource is discovered, adequate number of units (based on number of resources in resource field) is notified. These units have to respond with their bid, which describes how suitable they are for collecting the resource. The evaluation function for bid equals to squared distance from unit to resource field plus constant. The constant is non zero if unit is already carrying resource and/or enemy is within its range. This makes resource collection faster but slows discovery of new resources. If some rich resource field is found quite far from the anthill, all units commit to collect the resource instead of searching for closer ones. Even though we tried to implement this „unit allocation“ as efficiently as we could, there are also other problems. Let there be group of units which are heading towards resource with intention to collect it. However, they are intercepted by a group of enemy's units and decides to attack them. Meanwhile some other unit discovers the resource where all those units were heading to and broadcasts offer with this resource. Since these original units are fighting with enemies, they bid high and are not selected by offering agent. Ultimately much more units heads towards the resource then is necessary. However, at least once a resource field is depleted, the last ant to collect resources notifies others so that they can remove the resource from theirs belief base.

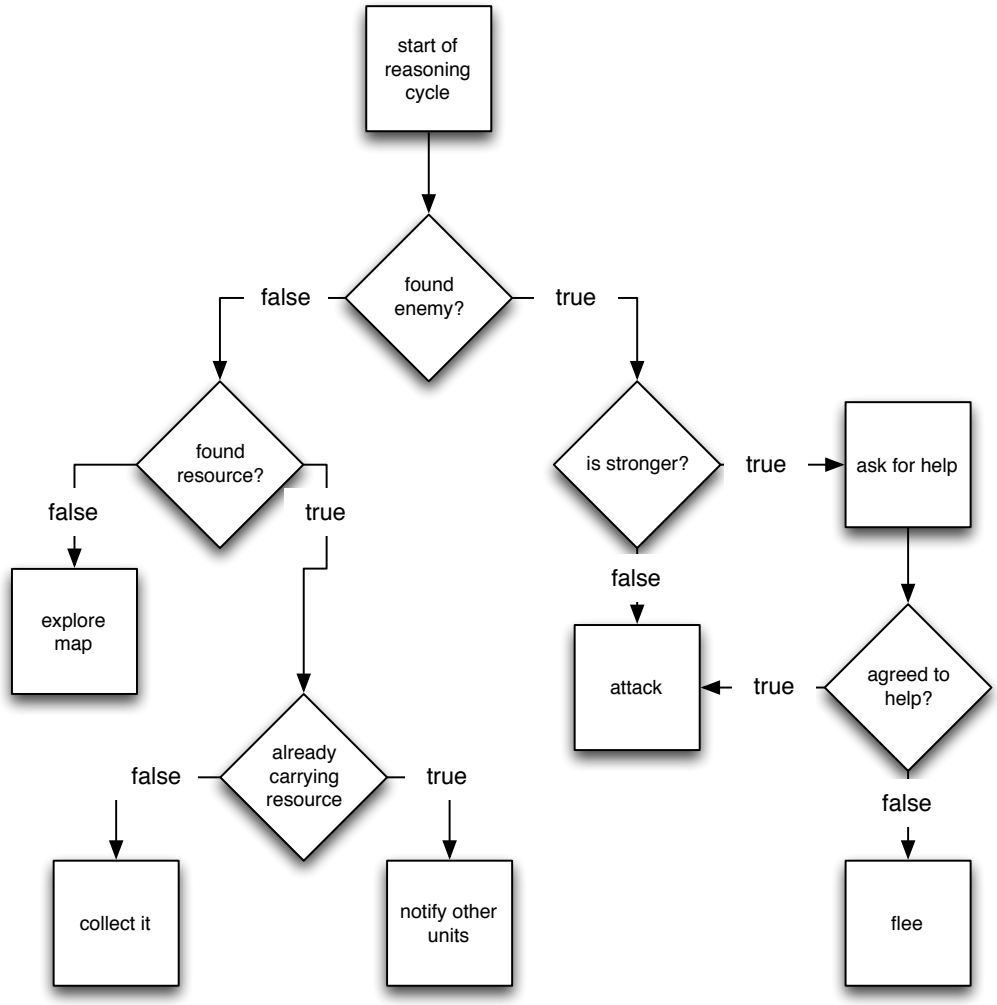


Figure 5.5: Cooperative artificial intelligence reasoning cycle.

We've decided to implement this communication feature even when units are attacked. In cooperative AI, if a group of units is overwhelmed by enemies, they are able to ask friendly units for assistance. These units are once again bidding and ones with lowest bid are selected to help. This also brings quite a big inefficiency because units can be allocated to help friends with enemies' units instead of collecting resources. However, we hope that despite these problems, the cooperative AI will be superior to the basic AI. As you may suspect the biggest technical problem with such approach is speed of communication between units, which vary on implementation and CPU load.

Advanced Artificial Intelligence

The last AI we've chosen to implement extends cooperative capabilities of the cooperative AI. The advanced AI (see reasoning cycle in figure 5.6) combines multi-agent approach with centralized AI. Units don't choose what to do on their own but they are driven by dedicated meta-level intelligence - the anthill. When a unit finishes its task, it requests new

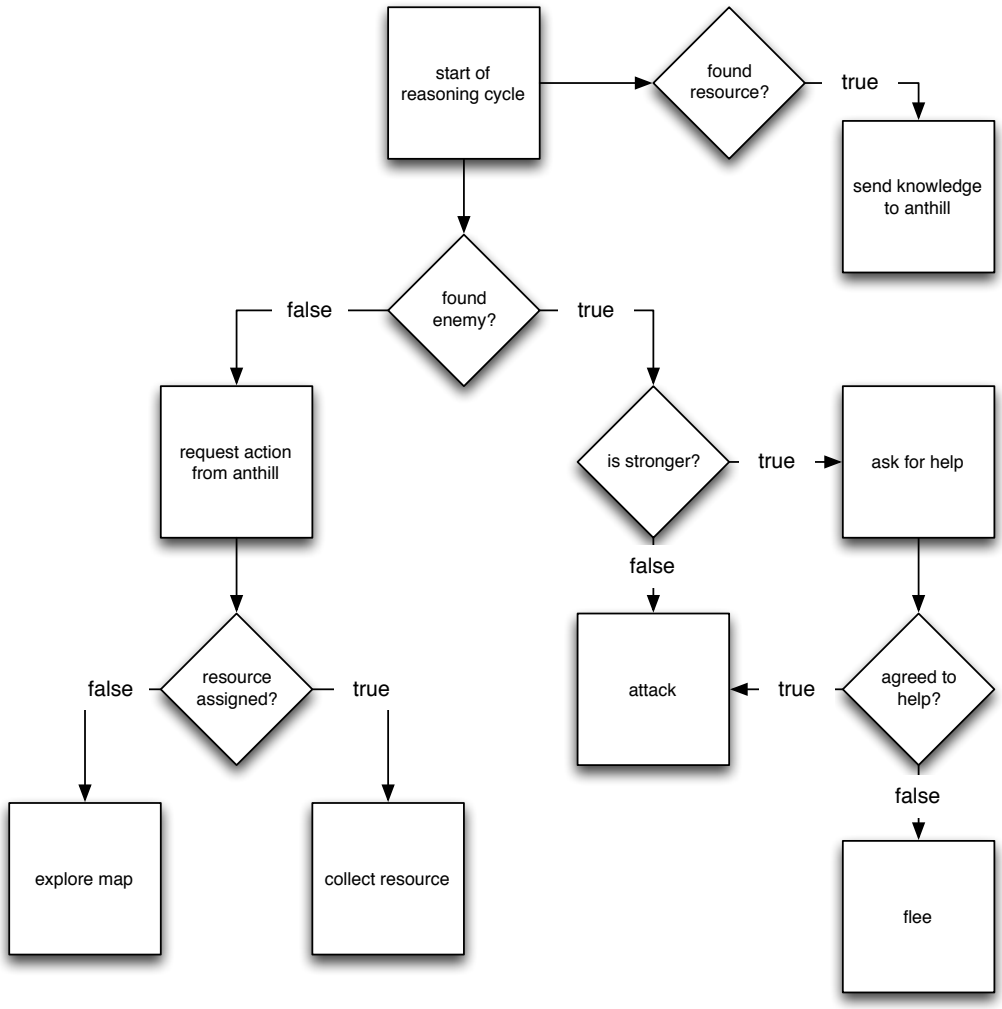


Figure 5.6: Advanced artificial intelligence reasoning cycle.

one from the anthill. Based on current knowledge of environment and status of resources, the anthill delegates either a goal to collect some resource or a position on map to discover. This allows us optimization of both resource collection and environment discovery. Because resources are managed centrally, the anthill AI can allocate just the right number of units for collection of resource from some resources field. The downside of this approach is excessive communication between ants and the anthill and the delay caused by such communication. Another problem is delegation of informations about environment to the anthill. Each unit has to send its perceptual information to anthill but at a reasonable rate so that the anthill is not flooded by messages, like with a D-DOS attack. We've decided it would be best to store information about discovered resources as a mental note in both ant and the anthill. Now when a resource appears in perceptual range of an ant, the ant tries to unify it with his previously discovered resources. Percept is sent to the anthill only when such unification wasn't possible either because the number of resources in the resource field changed or because the resource field wasn't discovered by this ant before. Surely there still is problem

with multiple agents discovering same resources, but this communication is done once per unit. Once resource is depleted, the ant sends this knowledge to the anthill and all friendly units, so that they can remove such knowledge from their belief base.

If there is an enemy unit within percept range, the ant automatically attacks it. However, unlike in the cooperative AI, the ant has to notify anthill about such action so that the resource it is heading to can be assigned to somebody else. When it comes to attacking enemy, the behaviour is basically same as in the cooperative AI - if enemy unit can be defeated then the ant attack, if not, it requests help from peers. The biggest problem we have here is the resource allocation. Imagine situation where the ant is killed before it can notify the anthill that it is being attacked. The allocated resource for this ant will remain inaccessible for others, because it is still allocated to now dead ant. The solution for this problem is some sort of resource belief management for the anthill belief base. For each resource allocation there is a time stamp with last modification. Once in a while a management goal is triggered with a purpose of deletion of all allocation beliefs with lower modification time then certain threshold. Even though this can't detect allocation mishmash on frequently modified rich resources, it can detect such errors on near to depletion resource fields. But this is not really an issue because this problem doesn't affect resource fields with more resources then ants. It will, however, appear once the resource field is near depletion.

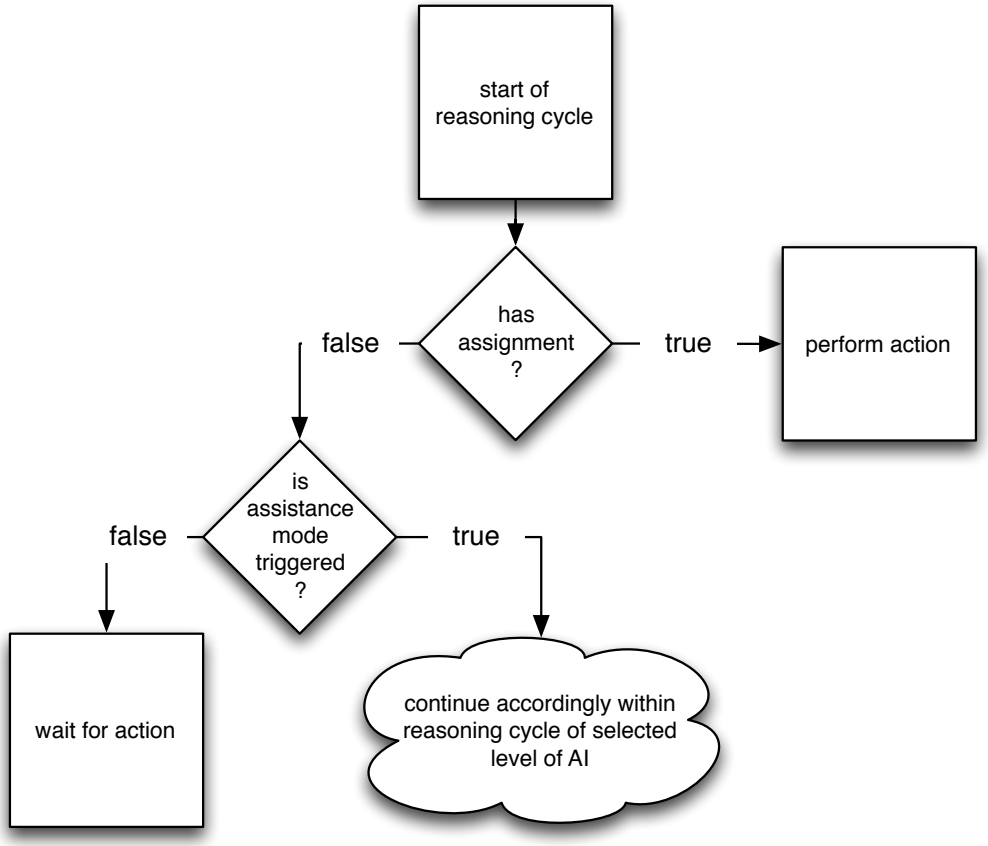


Figure 5.7: User controlled artificial intelligence reasoning cycle.

Artificial Intelligence of Player Controlled Units

Player's units are nothing else but special kind of intelligence. For a purely user mode this intelligence is completely handled within Java code. From a standpoint of our thesis is not really interesting and was already described in section 5.3. What, however, is quite interesting is the assistance mode. In this mode units are manage jointly by both player and some artificial intelligence. This intelligence can be either of previously mentioned ones. The top one priority are now actions from the player and only if such actions are not issued, the unit control is up to the AI. As you may imagine there is quite a lot of problems caused by interruption of the AI cycle from the user. Especially advanced AI with its resource allocation suffers quite heavily. Nevertheless there are no special modification to these AIs in the `asl` source code, which is quite convenient for further game customization. Even though the player should be responsible for managing the anthill (upgrades and creation of new units), for the advanced AI we had to preserve the anthill AI with disabled actions. An outline of player's AI can be found in figure 5.7.

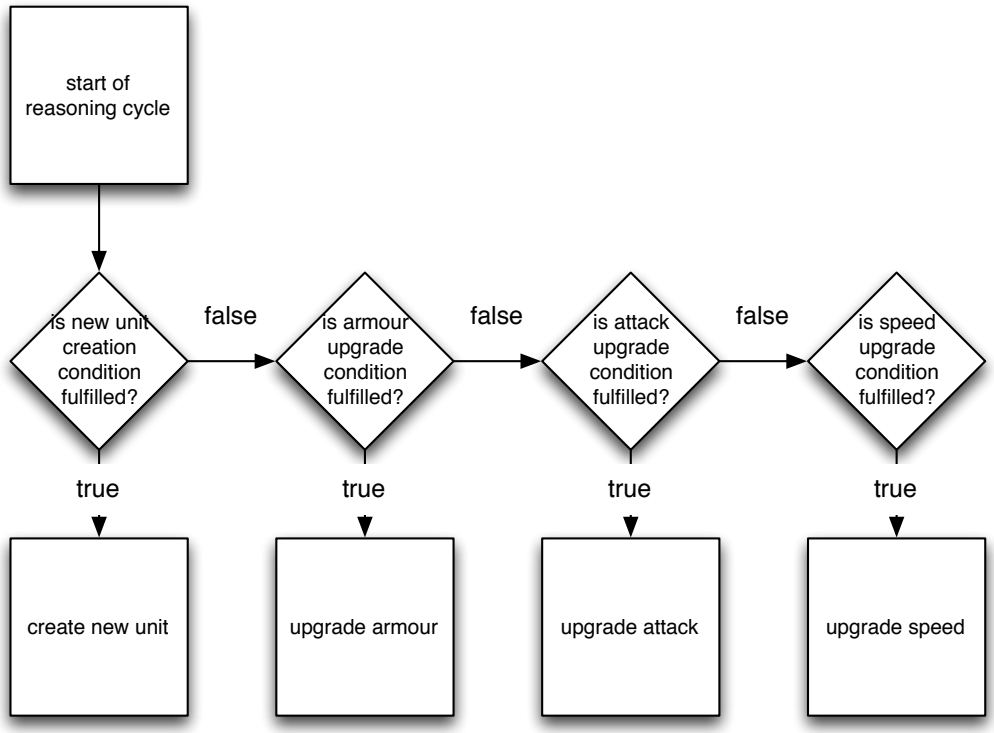


Figure 5.8: The basic and cooperative anthill reasoning cycle.

The Anthill Artificial Intelligence

We have already mentioned presence of some sort of centralised anthill AI when describing advanced AI, but the truth is all three AIs has such element. The anthill AI is responsible for managing bank of resources - spending them on new ants and upgrades. The reasoning cycle (figure 5.8) for basic and cooperative anthill AI is quite simple and could be implemented

in Java with an ease, but the advanced AI requires presence of this element in Jason so we've decided to preserve it for these intelligences as well. The advanced anthill AI, apart from upgrades and army, also manages resources and is responsible for choosing actions of ants. It's reasoning cycle can be found in figure 5.9.

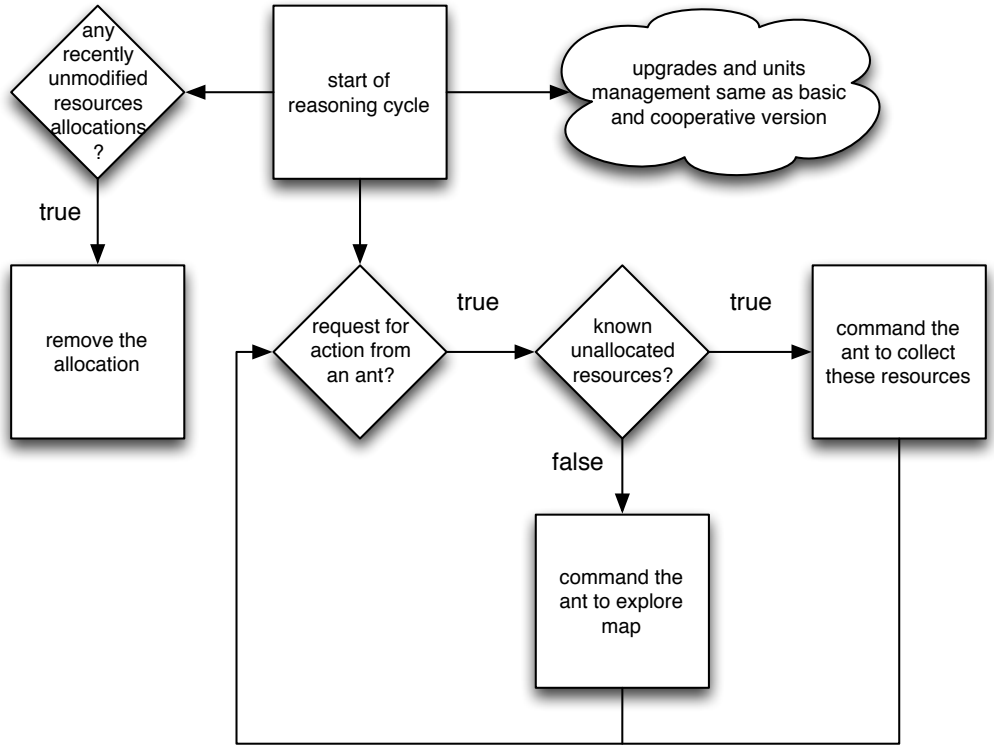


Figure 5.9: The advanced version of anthill intelligence's reasoning cycle.

Chapter 6

Implementation

In this chapter, we will briefly talk about selected implementation problems of the Anthill Strategy Game. More then on implementation details, we will focus on overall game design and customization capabilities. Because this game will be released as an open-source project, the aim of this chapter is to introduce to you the interface between Java and Jason. We will also talk about improvements that may be done in future and some know bugs. Please note that the aim of this chapter is not detailed description of the whole game. Such information can be easily found in Doxygen [15] documentation, source code and annexes.

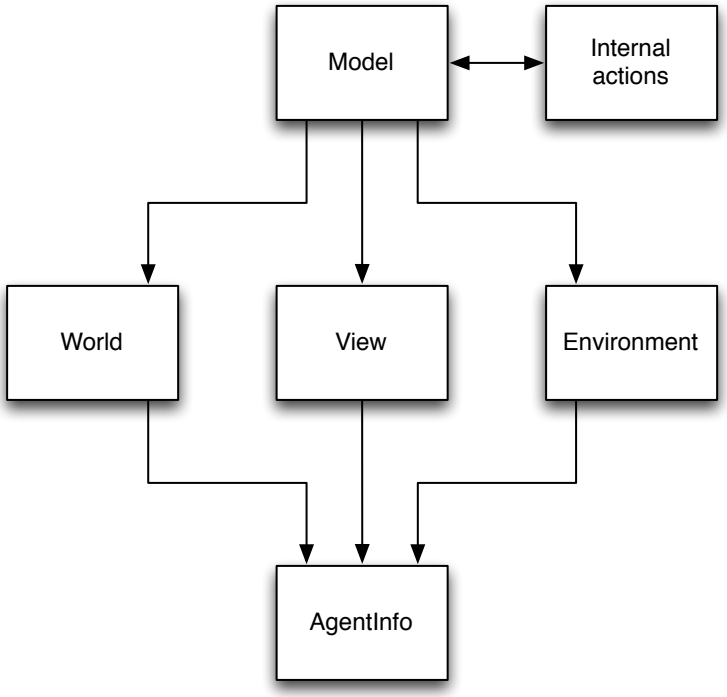


Figure 6.1: Main classes and interfaces.

6.1 Overall Game Implementation Pattern

The game consists of five main blocks implemented in Java (shown in figure 6.1):

The `Model` is a static class containing all information about current game. It is responsible for creation of new game and management of the game's data. The only reason why we had to make it static are internal actions. To access game data, internal actions would have to keep some reference to game class. Unfortunately, as far as we know, Jason does not provide any way to store reference to such data. So the Model uses singleton pattern and acts as an interface for internal actions and other game blocks.

Instance of the `World` class is responsible for storing representation of agent's environment. It manages obstacles, resources, anthills and units. In combination with movement thread, the World instance is responsible for granting same number of steps for all units. In fact it initiates these steps and its up to the `AgentInfo` instance what action will be performed. If you want to implement your own world with your own design please see map customization section D.4 in annexes.

The `Environment` class extends Jason's environment. It is the basic block of the game created directly from Jason and the game starts from the `init()` function. The Environment class is responsible for reacting on actions of agents and updating their percepts. All percepts are pushed to the agent after each action. There also are functions for creating and killing agents, anthills and upgrading various properties of units.

The `View` class is responsible for creating all screens of the game and reacting on inputs from the graphic user interface. Each screen of the game has its own class. The probably most important one is the `ScreenGame` which initializes and displays the game. Each panel from the game layout, described before and shown in figure 5.1, is also a separate class. To separate model of the game world (stored in instance of class `World`) from their visual representation and to fasten the game within critical sections, all displayed elements are also stored within `PanelMap`. This panel is responsible for displaying the map with all the game elements.

Information about every unit is stored in the `AgentInfo` class. This class is crucial component of the game because it implements behaviour of both user's and AIs' units. The most important method is `doStep()` which is called from `World` periodically and does one step of the agent. These steps can either be movement towards certain destination, collecting resources or attacking an enemy unit. Unlike units, every anthill is an instance of class `Team`. This class stores all information for the faction: collected food, collected water and upgrades status.

All **internal actions** available from Jason to the agent, are implemented in package Actions. The access point for all internal actions is the `Model` class. The full list of internal actions with a brief description can be found in section D.1.

6.2 Interesting Solutions

Because the game is so extensive, we will focus on just a several algorithms which are important and cleverly used. However, if you are interest in deep understanding of the game functionality, please consult programmers documentation, source files and annexes.

Obstacles

Even though, in comparison to nowadays RTS games, this game is quite simple, we've decided to implement obstacles into the game. Obstacles are triangles defined by three points, lying within the *worlds* area, which can't be passed by any unit. The problem we are facing when implementing obstacles, is the necessity for a unit to get around the obstacle using the shortest path and without any heavy calculations. We've decided, that even though all units have limited percept range, obstacles will be known prior their exploration. This allows us to calculate best possible path between every two points on the game world. All we now would need to do, is to cast some sort of ray from current unit's position to the destination and detect any collision with any obstacle. Then cast ray from every vertex of the obstacle to unit's position and destination, and if necessary repeat this action for any obstacle these rays intersects with. We assemble list of all paths by connecting vertexes of obstacles that are directly visible from each other and select the shortest one.

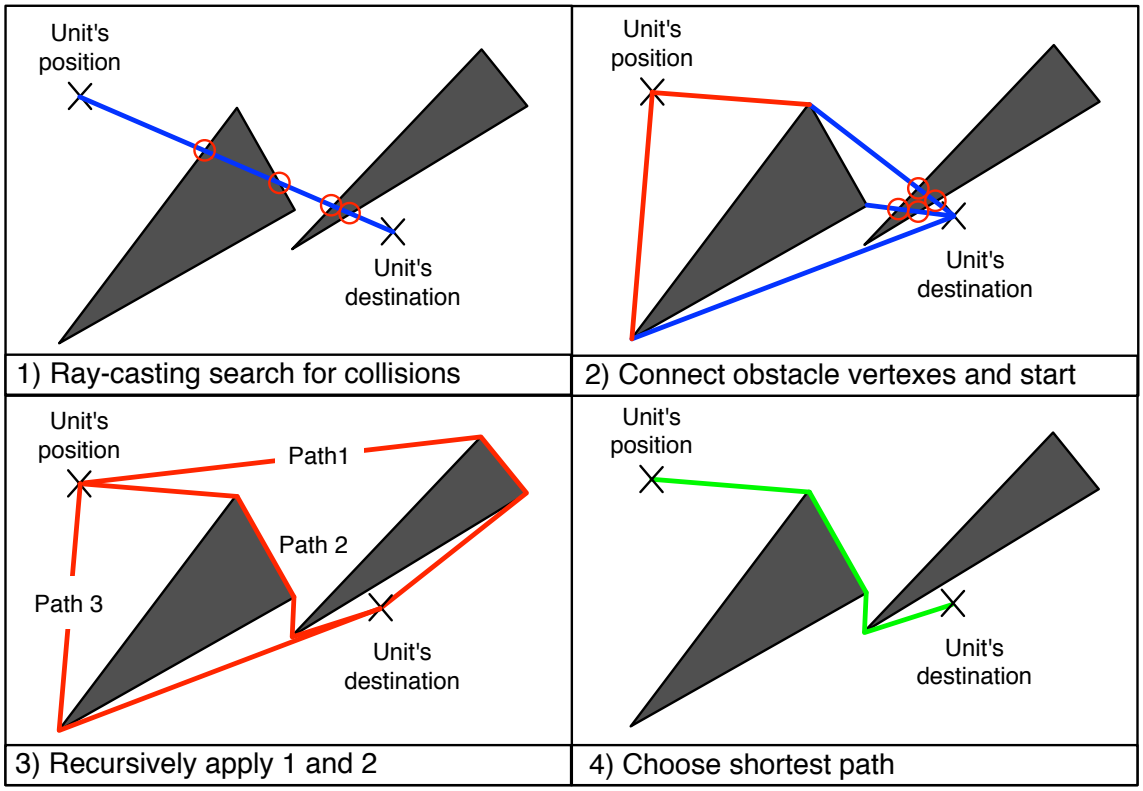


Figure 6.2: Principle of path searching algorithm.

However, this approach has extremely high computational complexity and is not suitable for calculating path in real time. Especially when, in the worst case scenario, we need to calculate around 100 unique paths per fraction of second. There is number of improvements we've implemented in order to improve computational complexity:

- **Storage:** We need fast determination if some point contain an obstacle and which obstacle it is. We've decided to store for each point in the world area information if there is an obstacle and if so which one it is. This is done by matrix `obstacles[world height][world width]` in which each cell contains either reference to obstacle that

lies there or `null` value if the cell (and corresponding location in map area) is free of obstacles.

- **Ray-casting:** Now when we know which point contains an obstacle, we can use ray-casting to determine if there is any obstacle between unit's location and its destination. For maximal efficiency we use Bresenham's line algorithm, that uses only integer arithmetic. Because the matrix of obstacles has the same coordinate system as the game world, we can quite easily detect any obstacle in the way and even get it's vertexes.

- **Pre-calculation:** The most important part of our algorithm, is the ability to calculate paths between each pair of obstacle vertexes before the start of the game. We can connect all visible vertexes of all obstacles and using modified version of A* algorithm, create list of shortest paths between every pair of vertexes. These paths are stored in a customized `HashMap` with first key being the obstacle vertex visible from unit's current position and second key the obstacle vertex visible from unit's destination.

Now, each time new destination for any unit is assigned, we can easily calculate best possible path using this algorithm (shown in figure 6.2):

1. If there is no obstacle between agent's position and destination, then move directly towards destination and end the algorithm.

2. Otherwise create list P containing all directly visible obstacle vertexes from unit's position.

3. Create list D containing all directly visible obstacle vertexes from unit's destination.

4. Retrieve minimal path from hashmap for each pair of points $a \in P$ and $b \in D$, add distance from unit's position to point a and from unit's destination to b.

5. From all possible paths select one which is the shortest.

Agents' Movement and Reasoning

Another problem we had to solve, was granting same conditions for each unit. This results in separation of action execution and unit's reasoning cycle. We've implemented a special thread, defined in class `ThreadAgentMovement`, which performs one step for all units by triggering method `doStep()` on `AgentInfo` instance. One step means either movement towards certain destination, collecting resources or attacking an enemy unit. Which step will be executed depends on state of the unit set by artificial intelligence or the player.

The unit's reasoning is also quite fair. All agents have unlimited time for selecting action. How fast they can command an action is just matter of computational complexity of theirs decision making and context switching done by OS. However, all perceptual updates are done periodically from AI reasoning cycle in Jason with same period for every AI (at least currently implemented AIs).

Configuration Capabilities

The Anthill Strategy Game is implemented with an emphasis on easy modification and customization. Almost everything can be centrally adjusted from *Settings* menu. The setting is stored within user's preference file in Java, so it persist after the game is closed

and loaded next time the game starts. The `Configuration` class stores most of the game settings, but you can also change specific behaviour for every unit in the `AgentInfo` class. With a little effort, you could easily achieve different properties for each unit. Please note that not all settings are currently implemented.

6.3 Artificial Intelligence Implementation

In this section we will describe how are different levels of AI implemented. For better understanding, please consult list of implemented actions which can be found in appendix D.1. The `asl` source files for all currently available can be found in `./src/asl/`.

Basic AI

The basic AI is driven by an infinite goal loop `!check_percepts`, which is used for updating perceptual information from the environment. The frequency of the loop can be set in configuration. Based on percepts, the loop decides which action will be performed. If there is an enemy, the `!enemy` goal is triggered if not, and agent finished all actions, the goal `!finished` is triggered. The reasoning cycle for basic AI is described in figure 5.4.

Resources Collection

Resources perception and collection is implemented as belief base addition plans `+resource (X, Y, Amount, Type)[source(percept)]`. If the agent already knows about this resources and the amount did not changed, nothing happens. If amount changes (resource rediscovery), the ant modifies the mental note to reflect the new amount. When there is no mental note for a resource (new resource discovery), it is created and the agent decides if he wants to collect it. The system is quite straightforward, each time agent finished action, it decides what to do. If it knows about any resource, it chooses the closest one and collects it, otherwise the agent just randomly searches map for new resources.

Combat

The goal `!enemy` implements all possible scenarios for encountering enemy. When there is any percept `enemy(X,Y,Name)` this goal is always triggered. The agent has just two choices - attack or run away. If there is only one `enemy(X,Y,Name)` percept, the agent determines it's winning chances using internal action `isWeaker(Enemy,Agent)`, and if the action returns true, the agent attacks the enemy unit. If there are more then one enemies and/or there are some friendly agents perceived, decision is made by equation:

$$enemies < friends + 1 \tag{6.1}$$

If the equation applies, the agent attacks random enemy, if not it runs away. It's worth mentioning that in scenario where there is only one enemy unit and no friend, the decision is affected by upgrades, but when there are more agents, the decision is based purely on the friend:enemy ratio.

Cooperative AI

The cooperative AI introduces communication which is basically just delegation of percepts. The process of perception updates and action selection is more or less same as in the basic

AI, but both resource collection and reaction on enemy presence differs. The reasoning cycle for the cooperative AI can be seen in figure 5.5.

Resources Collection

When new resource field is discovered, the agent broadcasts message that there is a resource field at that location. Other agents have to respond with their bid - the squared distance from their position to the resource + some constant if they are already carrying a resource. The agent then chooses to whom he will send the resource percept. The maximal amount of agents to which the agent can send the percept is the amount of the resources in the resource field. However, the problem is, that each time an agent with a mental note that differs from actual resource field amount rediscovers it, new message is broadcasted and possibly some other agents allocated. Therefore every agent, who is already heading towards the resource, responds with bid of value 0. Agents also notifies others that some resource field was depleted, which saves some game cycles for units heading towards already depleted resources.

Combat

The cooperation also helps defeating enemy. Each time an agent assesses combat situation as unfavourable, it broadcasts request for assistance. Then, again based on the bid, it chooses who will help and continues evading enemy until help arrives. However, the agent who promised to help (provider) may encounter enemy on his own. If such scenario happens, the provider is bound to notify the agent who requested assistance (applicant), that it can't help. After such event a new assistance request is issued and new group of ants allocated. As in resources collection, agent who already agreed to help bids with 0. The applicant is also bound to notify providers that it is no longer being pursued and no assistance is needed.

Advanced AI

The last level of AI implements centralized resource management. Each time an agent finishes action, it notifies anthill which centrally stores all discovered resources. The anthill chooses what action will be most useful and delegates it to the ant. In terms of combat, no changes were issued compared to the cooperative AI. The reasoning cycles for the advanced unit's and anthill's AIs can be seen in figures 5.6 5.9.

Resources Collection

Every time some resource is discovered and the ant does not have the exact same percept, the perceptual information is send to anthill. The anthill then decides if the amount of resources changed and should be updated, if the exactly same percept already exists and the message can be ignored or if an entirely new resource field was discovered. Only when new resource field is discovered, anthill notifies all ants and offers this resource field to them. Based on bids that returns, the anthill chooses who should take the newly discovered resource. For every resources field in the anthill belief base, there is at most one `allocated(X, Y, Resource, NAgents, GameTime)` belief which stores how many agents are heading towards this resource field. Only resource field with more resources than there is allocated agents can be assign to an agent.

When agent finishes an action, it requests new action from anthill. If there is a resource field in the anthill belief base and it is not fully allocated, the anthill may assign it to the agent. However, if there is no such belief, a goal to explore map is delegated to the agent. Under certain circumstances the exploration can also be assigned even though the anthill has some unallocated resource fields. This can happen, when the anthill has excess of one type of resource and has knowledge only about this type of resource field. This mechanism grants faster development of the anthill because both resources are kept at about same value.

We also had to solve situation, where the ant is heading towards resource field but is offered with closer one or is interrupted by enemy's unit. Such ant is bound to inform the anthill about dropping intention to collect the resources from the field. However, sometimes the ant can be killed before sending such message. We've implemented a mechanism, which manages all allocations and deletes ones that hadn't been modified for more than 30 game seconds.

6.4 Known Bugs

Even though we did our best, the Anthill Strategy Game, as any other program, has number of bugs. However, all of them occur in very rare conditions and should not affect game-play much.

- **Unit Teleportation:** Under extremely rare conditions units teleport to first point in their path. This is probably some synchronization error in the AgentInfo class, but we couldn't the issue and fix it. Chances are that you will never see this bug occur.
- **Passing through Obstacle:** If the unit is close to the edge of some obstacle, it can crawl through it. This bug is caused by combination of rounding error in Bresenham's line algorithm used for searching visible obstacles vertexes and truncation error for agent's position. Unit is simply so close to the obstacle and its surroundings is so indented that none of any obstacles vertex is seen as directly visible. This bug is also quite rare and we have seen it occurred less then ten times.
- **Resources Allocation:** The advanced AI allocates every piece of resource in the resource fields to some agent and this way optimizes resources collection. However, when unit dies, the anthill doesn't know that the resource should be unallocated and for some time can not be allocated to any other agent. We've implemented mechanism that after some time deletes recently unmodified allocations, but we see this as a temporary solution.
- **Unit Selection:** Sometimes the selection rectangle does not disappear after releasing left mouse button and no game element is selected. This is caused by not receiving mouse release event but we weren't able to fix it. Nevertheless this bug is not frequent and you can easily reselect units by re-dragging.

Chapter 7

Experiments

In this chapter we will perform number of experiments which purpose is to compare available artificial intelligences between each other. All maps for each experiment are fair for all competing sides resource and obstacle wise. However, in some experiments there is a random element of generating new resources on random locations on the map.

7.1 Experiments Conditions

If not stated otherwise, all experiments were done separately without any user intervention on machine with following specification:

Model:	MacBookPro6,1 (2010)
OS:	OS X 10.8.3
CPU:	Intel Core i5 2.53 GHz
RAM:	8 GB 1067 MHz DDR3
GPU:	NVIDIA GeForce GT 330M and Intel HD Graphics
HDD:	OCZ-AGILITY4 512 GB
Java:	JavaVM 14.7.0
Jason:	Jason 1.9.0

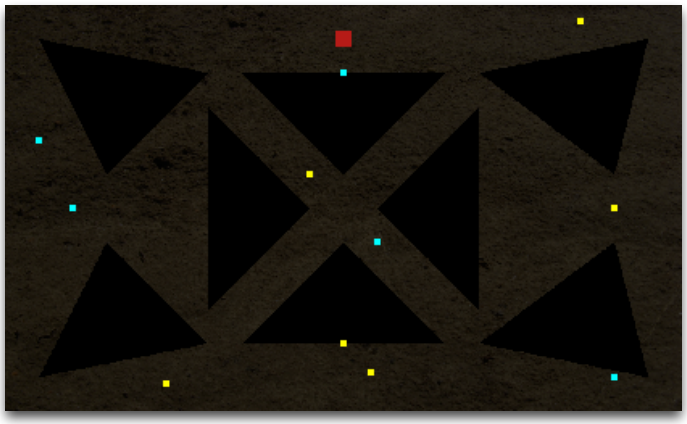
Experiments were launched with game speed = 200, which sets how many environment steps are performed per one real time second (actual number is less due to context switching). The number of perceptual updates per second for each ant was set to 5. For most experiments we did not display the game GUI - only the game overview and action panels with a refresh rate equals to one frames per second. All experiments can be repeated with same settings using the *Experiments* bookmark in game's main menu. The output of these experiments is saved to a folder `experiments` containing various information about the course of the game.

7.2 Resource Gathering Experiment

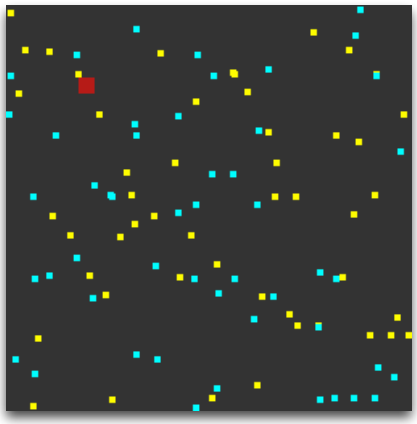
In this experiment we focus on AIs ability to gather resources and use them effectively. We will examine five main aspects which sums up Macro-management:

- Total time necessary to collect all resources.

- Average ants army value for the experiment.
- Average unspent resources.
- Average unspent resources food:water ratio.
- Effective actions per minute for one ant.



Map: ResourceExperiment1
Size: 1000 x 600
Food: 5 fields 50 each
Water: 5 fields 50 each



Map: ResourceExperiment2
Size: 2000 x 2000
Food: 50 fields 10 each
Water: 50 fields 10 each

Figure 7.1: Maps used in resource gathering experiment.

Experiment Set-up

At any point in this experiment there is only one team collecting resources. All available AIs compete on two different maps with number of resource fields. You can see map designs with description in figure 7.1. Every game in the experiment ends when all resources are collected and is repeated ten times for each configuration. The average values are then displayed in graphs. The random resource generator is switched off.

Results Evaluation

The figure A.1 shows, how long it took for every AI to deplete all resource fields on the map. Both communicating AIs are faster then the basic AI. The cooperative AI is about 5% and advanced AI at about 15% faster compared to the basic AI. The difference is apparent for smaller number of ants where, especially for experiment map 1, the advanced AI absolutely dominates resource collection.

The advanced AI is also much better at overall macro-management. Figure A.2 compares average army values throughout the whole experiment. For small amount of ants, the advanced AI is much more effective because of its ability to prioritize certain resources. However, in map #2 there is not much difference between all three AIs. Resource fields are reasonably low so that all AIs are able to mine both resources at about same rate.

This is shown in figure A.3 displaying average unspent resources for the anthill throughout the game. Once again, the Advanced AI is superior, being twice as effective as the basic

and the cooperative AI. The figure A.4 describes ratio between unspent food and water. You can see there is an excess of food in the game. Reason for such phenomenon is lack of fights between factions - food is used primarily for creation of new ants. Also the intelligence of anthills in terms of upgrades and new ant creation, is far from being perfect.

What is quite interesting is figure A.5 which shows effective actions per minute (EAPM) per one ant. This number describes how many effective actions, on average, were performed by an ant every minute. Effective action is an action which changes game environment. Examples of such actions are: collecting resources, attacking enemy or choosing a destination where to go. The anthill actions are included as well. We've expected that the advanced AI would have most EAPM because it finishes resource collection much faster. However, it seems like, if the game length is taken into account, the worst AI in terms of EAPM is the cooperative. One of explanation could be inefficient resource delegation.

7.3 Death match experiment

Purpose of the death match experiment is to determine combat capabilities of all featured levels of artificial intelligence. This can be characterized as a micro-management but the already described macro-management analysed in resource experiment 7.2 has a big influence as well.

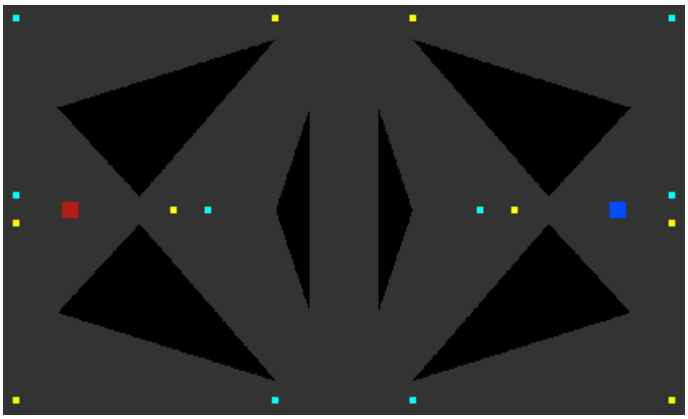


Figure 7.2: Map used in the death match experiment.

Experiment Set-up

In the death match experiment two factions with different level of AIs competes against each other. Experiment ends when all units from one faction are defeated and is repeated with same settings 50 times. The design of the map used for this experiment is pictured in figure 7.2. Starting resource fields are compromise between both maps used in resources experiment. The map is rather small to force ants to attack each other and limit resource collection influence to minimum. The random food and water generator was switched on

for this experiment with 50% chance for both resources and period of 5 seconds. For each competing side the experiment is run with 1 to 9 starting units and the overall win-loss ratio determined.

Results Evaluation

The combat between the basic and the cooperative AIs is not very surprising. Figure A.6 shows win-loss ratio for this duel from a standpoint of the cooperative AI and figure A.7 from a standpoint of the basic AI. Chances for cooperative AI to win against basic AI are in sum 59.38%.

However, the duel between basic and advanced AIs does not turned out as well as expected. Figures A.8 and A.9 shows how poorly the advanced AI competes against the basic AI despite much better macro-management. In terms of combat, the plans for cooperative and advanced AI are identical so we've expected from advanced AI to be even more successful then cooperative. Nevertheless, chances of advanced AI to win are in sum 53.31% which is much worse then expected. Results of the duel between cooperative and advanced AIs can be found in figures A.10 and A.11. The win chance for the advanced AI is 50,07% so both AIs are more or less equal. However, quite surprising is, that the advanced AI competes much worse then cooperative AI with a large amount of units.

The effective actions per second per one unit, displayed in figure A.12, shows how action demanding is attacking enemy. EAPM for all AIs grows linearly with slopes as follows:

basic AI:	34,12
cooperative AI:	46,83
advanced AI:	32,5

These values are three times higher for one agent than EAPM values for similar map in resource collection experiment. With increasing number of agents the difference is even more striking. Surprisingly the cooperative AI EAPM is much higher then the EAPM of basic and advanced AIs.

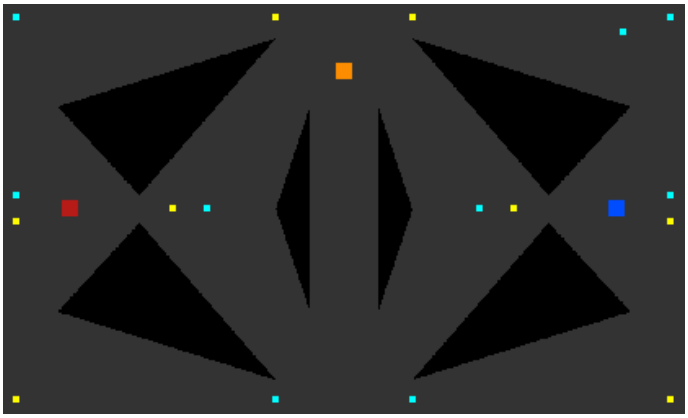
The hypotheses that average unspent resource ratio will be closer to value 1, in comparison to resource experiment map 1, was fulfilled. Once again the advanced AI was much better at keeping resources in balance as you can see in figure A.13.

7.4 Time Complexity Experiment

The time complexity experiment's purpose is to determine reasoning performance. We will measure so called reasoning per minute (RAPM) attribute and compare it with EAPM values.

Experiment setup

To measure RAPM a special internal action is added to the end of every goal plan. This action counts how many plans were triggered for each agent which allows us to determine average frequency and estimate average period for each plan. The map used in this experiment is shown in figure 7.3. Each experiment was performed 30 times with rotating anthills position in clockwise fashion to ensure every AI has same conditions. The random resource generator is switched on with 50% chance for both resources and period of 5 seconds.



Map: DeathMatch
Size: 1000 x 600
Food: 8 fields 25 each
Water: 8 fields 25 each

Figure 7.3: Map used in the time complexity experiment.

Results Evaluation

The graph A.14 shows how many reasoning actions per minute were performed by one ant. As expected, the cooperative AI leads with almost twice as many reasoning actions as other AIs. Surprisingly the advanced AI has at about the same amount of RAPM as the basic AI. The number of RAPM per one ant for the cooperative AI grows exponentially while for the advanced and the basic AI the growth is linear.

We've also measured the EAPM which grows logarithmically (see figure A.15).

7.5 Performance Experiment

This experiment determines how the GUI affects game performance. The CPU load was measured using UNIX utility `top`. Please note that this experiment is not fully automated as previous three are.

Experiment Set-up

The performance experiment does not differ much from the time complexity experiment. We used same map design and same set-up. All factions were driven by the basic AI with a constant number of units. The number of coexisting units was influenced by buttons and such unit count kept for some time. The experiment was performed without GUI, with GUI and with GUI and the fog of war switched on. The FPS was set to 1 when not displaying map and to 30 in mode with GUI and GUI + fog. The random generator was switched on with same settings as in the time complexity experiment.

Results Evaluation

The first graph A.16 compares how does the number of coexisting units influence the game. When performed without GUI, the growth was linear as well as with GUI turned on. In

fact the difference between these two remained same which is quite surprising. However, when GUI and fog of war was turned on, the CPU gap was more apparent and growing with much higher slope but still linearly.

We have also measured how does the map size influence the performance. The map was increased by 200 units in both x and y direction and the number of ants also increased by one ant. While performed without GUI, the average CPU load grew linearly and only because the number of ants differed, when the GUI and fog of war was switched on, the CPU load grew exponentially. In fact some performance issues could be observed with map higher then 2000 x 2000 pixels. The result is shown in figure A.17.

7.6 Observed problems

During testing and experiments, we have discovered certain unpleasant patterns in the behaviour of implemented AIs. Even though the cooperation in combat is one of the strengths for the cooperative and the advanced AIs, they also suffers from this cooperation. Especially when facing each other, ants of these AIs are one by one called into fight until all units from one faction are involved in combat slowly waiting for defeat by more numerous opponent. This is one of the reason why the basic AI done so well in free-for-all scenarios. The cooperative and advanced AIs almost killed each other after first encounter allowing the basic AI to focus on macro-management. What is probably worth mentioning was poor design of the time complexity experiment map. The map was quite small, and with more then 5 units in map, the experiment lead to a bloodbath. The problem is constant reinforcement of units by the environment to ensure that at any given instance there will be same amount of units for each side. In extreme scenario units stopped gathering resources and focused purely on combat which lead to major performance issues and excess of resources as displayed in figure 7.4.

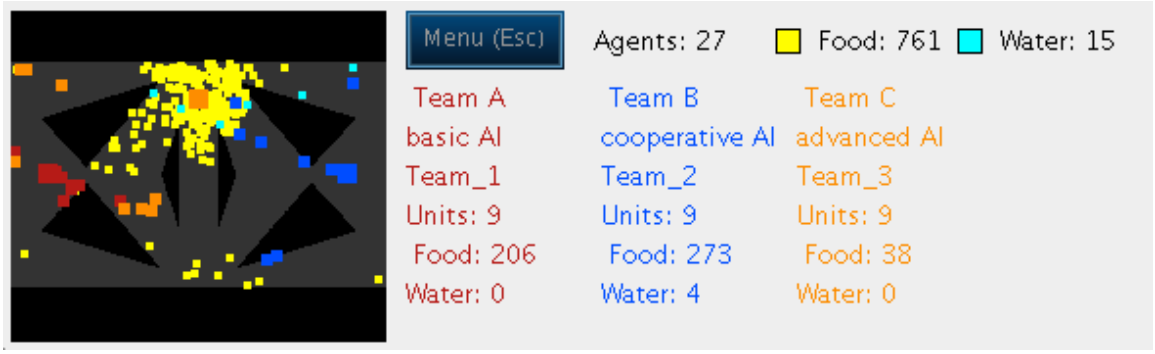


Figure 7.4: Problem that occasionally occurred during time complexity experiment.

Chapter 8

Conclusion

The main goal of this thesis was a creation of a fully functional real-time strategy game featuring three different levels of the artificial intelligence and the single player mode. The resulting game implements most of nowadays standard RTS game features and provides rich game-play experience. We have also added a unique mode which allows the player to play the game with a help of the AI so that they can focus on interesting aspects of the game without any worries of falling behind. The post-game analysis helps the player in better understanding of the game mechanics along with detailed overview of the just finished game. Such data can be found in the `logs` folder and we used a similar mechanism for analysing experiments.

The game itself was implemented using Java programming language which allows the game to be run on all major operation systems. We have also used the Jason programming language for creation and interpretation of the AIs' behaviour. The game is designed in such fashion that it can be easily extended and used as a framework for your own AI. One of our goals was a creation of an interface which can be used by other students to easily add a new artificial intelligence. All necessary steps for game customization can be found in annexes.

One of key parts of this thesis is evaluation of the game performance. We have discovered how big the impact of graphic user interface is. In current implementation the GUI takes around 70% of total game resources. Implemented AIs were also tested quite heavily. We have not verified our hypothesis that the advanced AI, combining decentralized and centralized decision making, would be superior. However, all implemented AIs have their strengths and weaknesses and when competing one to another the win-loss ratio is quite balanced.

From experiments we can conclude that the advanced AI excels at macro-management. It collects resources at about 15% faster compared to the basic AI and 10% when compared to the cooperative AI. However, the cooperative AI is superior in terms of combat followed by the advanced AI. The strength of the basic AI is apparent when having a large amount of units. It is also the bast match for the assistance mode where the player and AI manages same anthill.

During the game development we have used several useful algorithms and design patterns like Bresenham's line algorithm and A* path finding. We see the Anthill Strategy Game as an early beta project with a lot of aspects which need to be improved. We did our best but creating a complete RTS game with all its features is work for a team of programmers with much longer development period. Below you can see some improvements we have planned to implement but we have not manage to do so yet.

- **Collision Model:** One of the biggest game-play problem of the Anthill Strategy Game is absence of the collision model. While this is not an issue for AI controlled units, players have difficulties with selection of individual units and their management as they like to clump up. These clusters of units are even bigger problem when encountering enemy. Player never knows how many enemy units are there until he selects the cluster.

- **Rendering Performance:** All parts of the game are rendered within `JPanels` by overloading `paintComponent()` method. Even though this is quite optimized and partially processed by graphic card, we could achieve much better performance using OpenGL. The game was planned to be played in a full-screen mode but because we use internal frames for menu dialogues, which can not be displayed in full-screened canvases, the game supports only windowed mode.

- **Multi-player:** Despite the game heavily focuses on the artificial intelligence, it would be more fun to play against another player or play against the AI in team.

- **Game Control:** In a current state the game lacks some industry standards which could improve the game-play experience.

- **Game Configuration:** We have listed the game configuration capabilities as one of the game features. However, such deep level of configuration can be done only within source code in class `Configuration`. The original plan was a creation of persistent game settings which could be set by the user from the graphic user interface.

To sum up, in this thesis we have proved that it is possible to create playable RTS game in Java and Jason using multi-agent approach. All artificial intelligences undergone series of tests and the results were closely examined in the experiments chapter. The game will be released under some open source licence to allow others to participate on the game development.

Bibliography

[1] Dan Adams. The State of the RTS [online]. http://www.ign.com/articles/2006/04/08/the-state-of-the-rts, 2006-04-07 [cit. 2013-04-15].

[2] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems Using Jason*. John Wiley & Sons Ltd., 2007. ISBN 978-0-470-02900-8.

[3] Bruce Geryk. A History of Real-Time Strategy [online]. http://www.gamespot.com/gamespot/features/all/real_time/, 2008-04-27 [cit. 2013-04-15].

[4] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *IN PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS-95)*, pages 312–319, 1995.

[5] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach*. Pearson Education Inc., 2003. ISBN 0-13-080302-2.

[6] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd., 2002. ISBN 0-471-49691-X.

[7] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[8] WWW pages. Jason | a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net/, [cit. 2013-01-20].

[9] WWW pages. Blizzard Entertainment:Legacy Games [online]. http://us.blizzard.com/en-us/games/legacy/, [cit. 2013-04-15].

[10] WWW pages. Blizzard Entertainment:StarCraft [online]. http://us.blizzard.com/en-us/games/sc/, [cit. 2013-04-15].

[11] WWW pages. Blizzard StarCraft II [online]. http://us.battle.net/sc2/en/, [cit. 2013-04-15].

[12] WWW pages. GameSpy: Utopia [online]. http://uk.gamespy.com/articles/495/495918p1.html, [cit. 2013-04-15].

[13] WWW pages. Major Tournaments - Liquipedia - The StarCraft II Encyclopedia [online]. http://wiki.teamliquid.net/starcraft2/Major_Tournaments, [cit. 2013-04-15].

[14] WWW pages. Real-time strategy -Wikipedia, the free encyclopedia [online]. http://en.wikipedia.org/wiki/Real-time_strategy, [cit. 2013-04-15].

[15] WWW pages. Doxygen: Main Page [online]. http://www.stack.nl/~dimitri/doxygen/, [cit. 2013-05-10].

[16] WWW pages. Why SEO Is Like An RTS Game (and why you should care) |SEOmoz [online]. http://www.seomoz.org/blog/why-seo-is-like-an-rts-game, [cit. 2013-05-10].

Appendix A

Experiment Results

Explanation of Death Match Experiment Graphs

Each column represents chance to win (green colour) of the one AI with certain number of starting ants (x axis) against another AI with 1 to 9 starting ants. Black lines show threshold value where the team with more starting ants always wins and with same number of ants has a 50% chance to win.

For example the first column in graph A.6 shows win chances of the cooperative AI with one ant at the start of the game in duels with the basic AI starting with 1 to 9 ants. Duels are as follows (each repeated 50 times): 1 cooperative ant vs 1 basic ant, 1 cooperative versus 2 basic ants, 1 vs 3, 1 vs 4, 1 vs 5, 1 vs 6, 1 vs 7, 1 vs 8 and 1 vs 9.

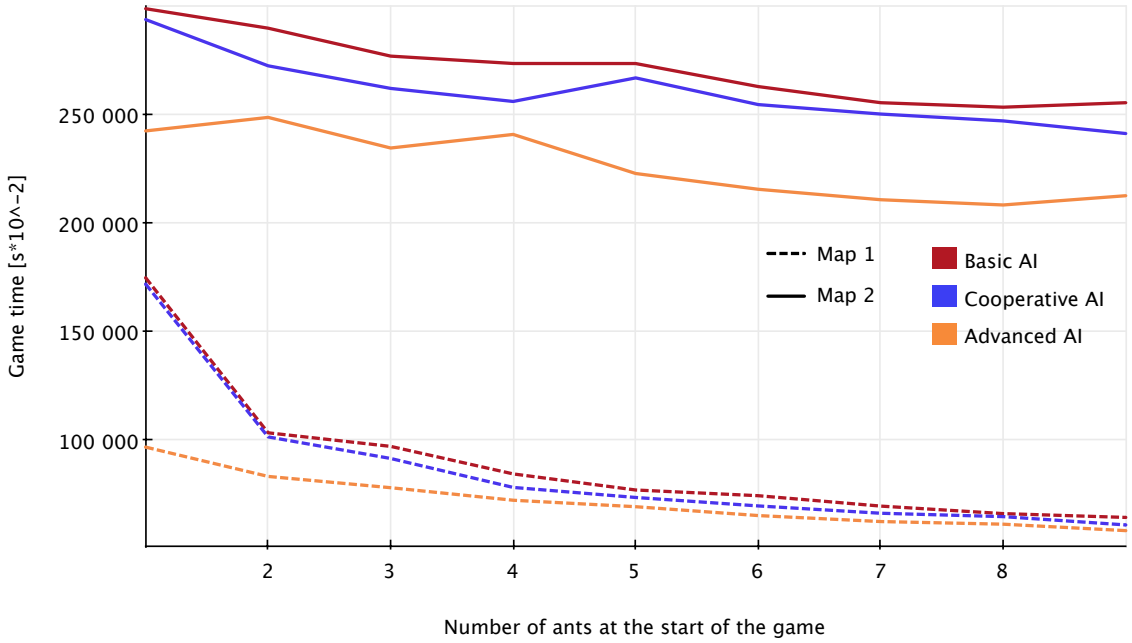


Figure A.1: Resource experiment: Game time required for collecting all resources on both experiment maps.

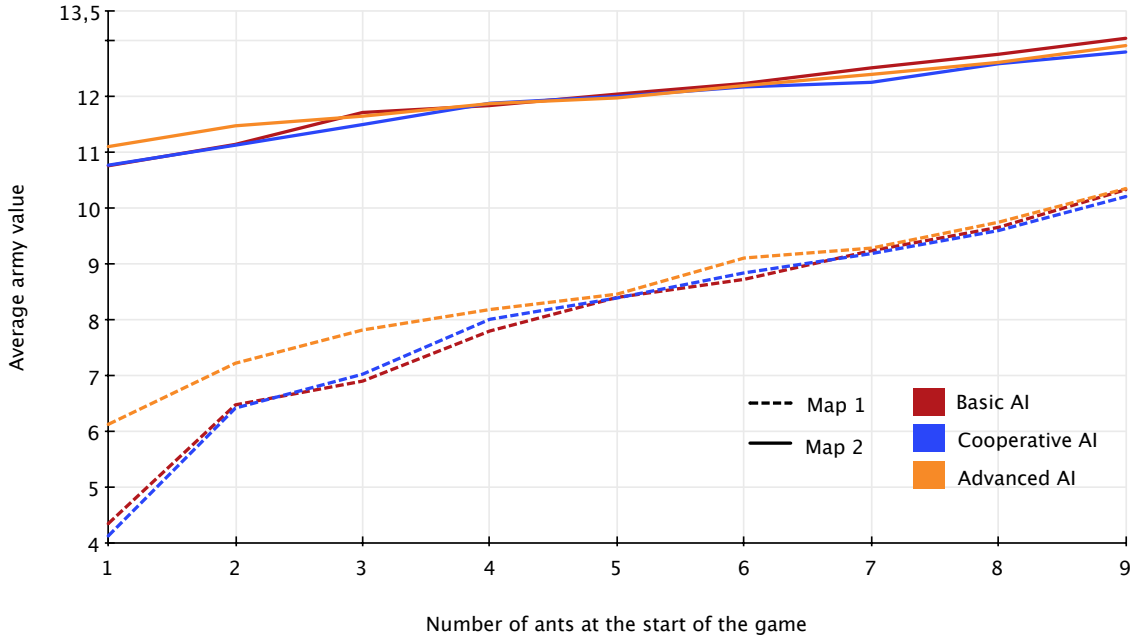


Figure A.2: Resource experiment: Average ants army value on both experiment maps.

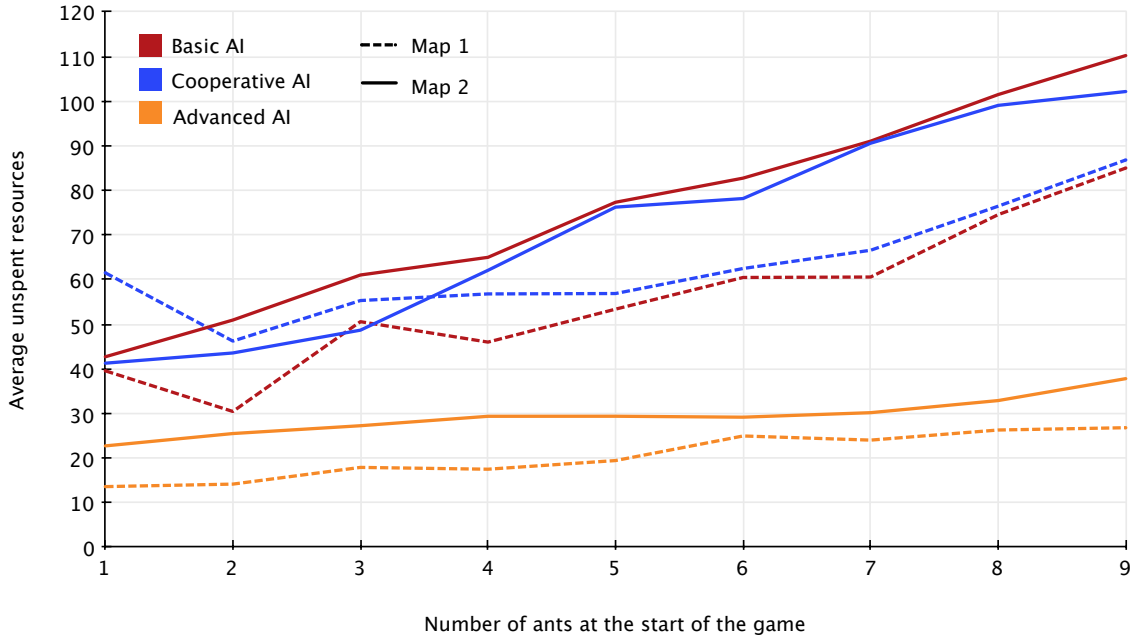


Figure A.3: Resource experiment: Average unspent resources on both experiment maps.

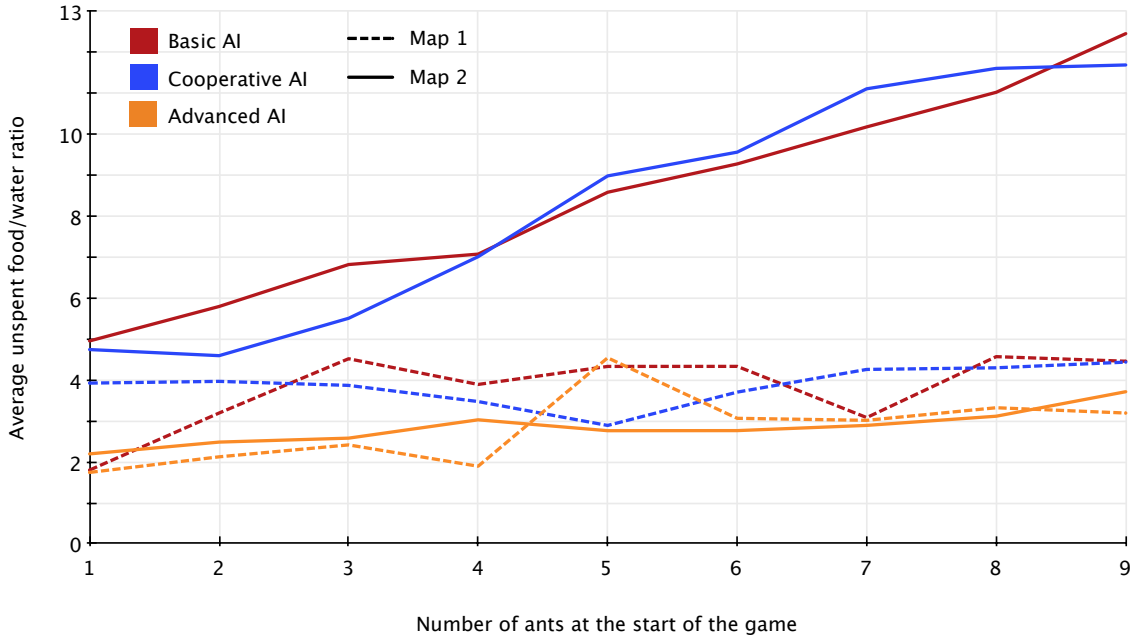


Figure A.4: Resource experiment: Average unspent food:water ratio on both experiment maps.

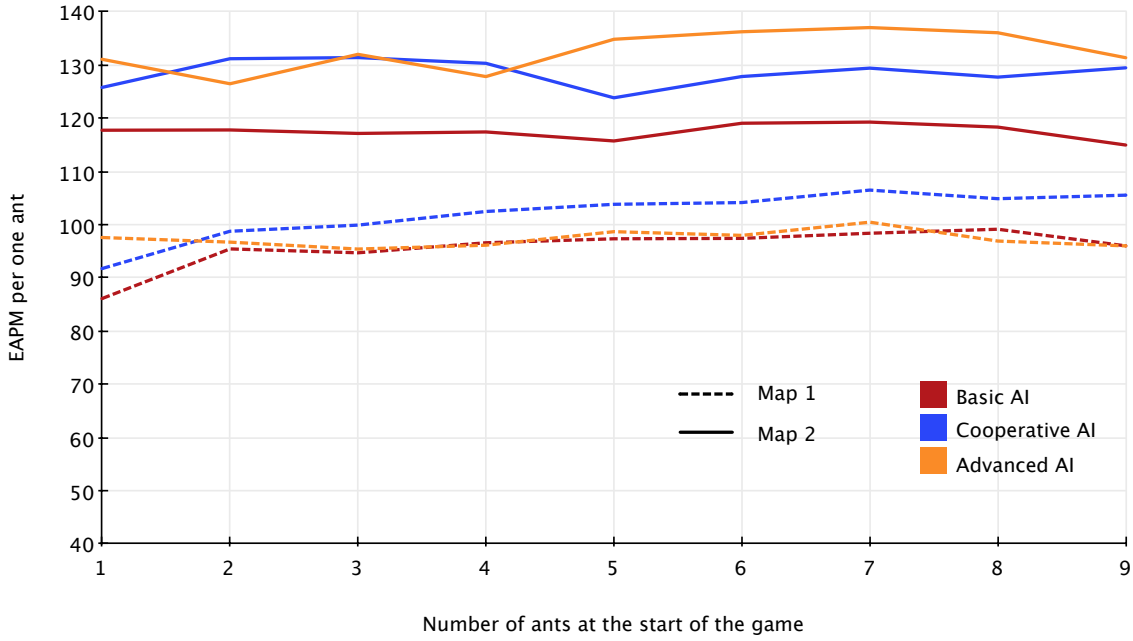


Figure A.5: Resource experiment: Average effective actions per minute per one ant on both experiment maps.

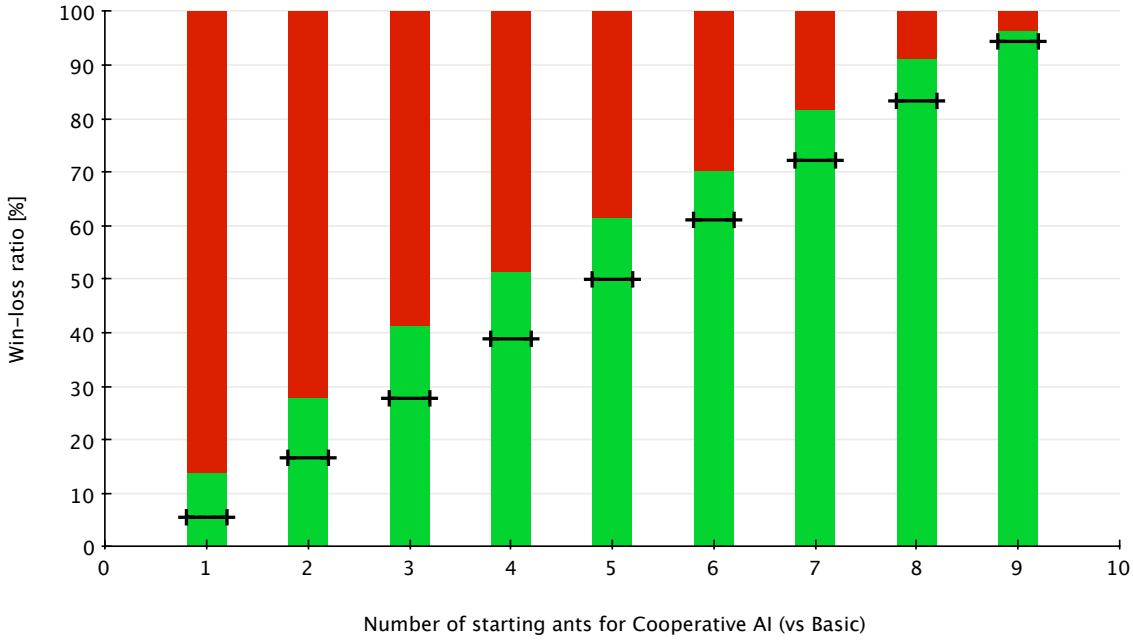


Figure A.6: Death match experiment: Chances of the cooperative AI to win against the basic AI as explained at the beginning of this appendix A.

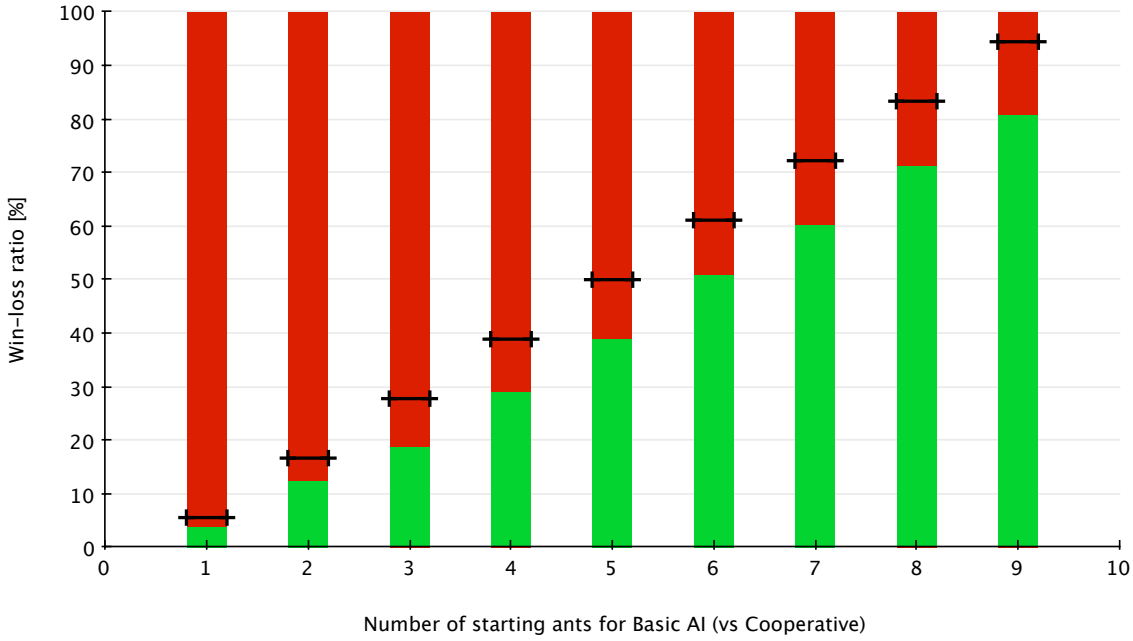


Figure A.7: Death match experiment: Chances of the basic AI to win against the cooperative AI as explained at the beginning of this appendix A.

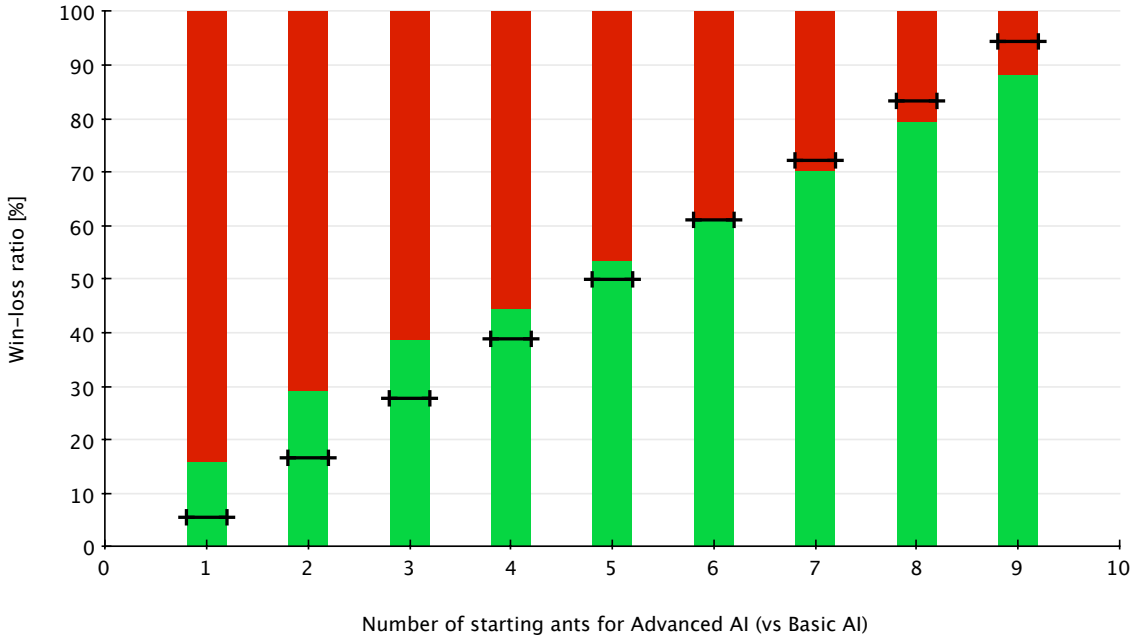


Figure A.8: Death match experiment: Chances of the advanced AI to win against the basic AI as explained at the beginning of this appendix A.

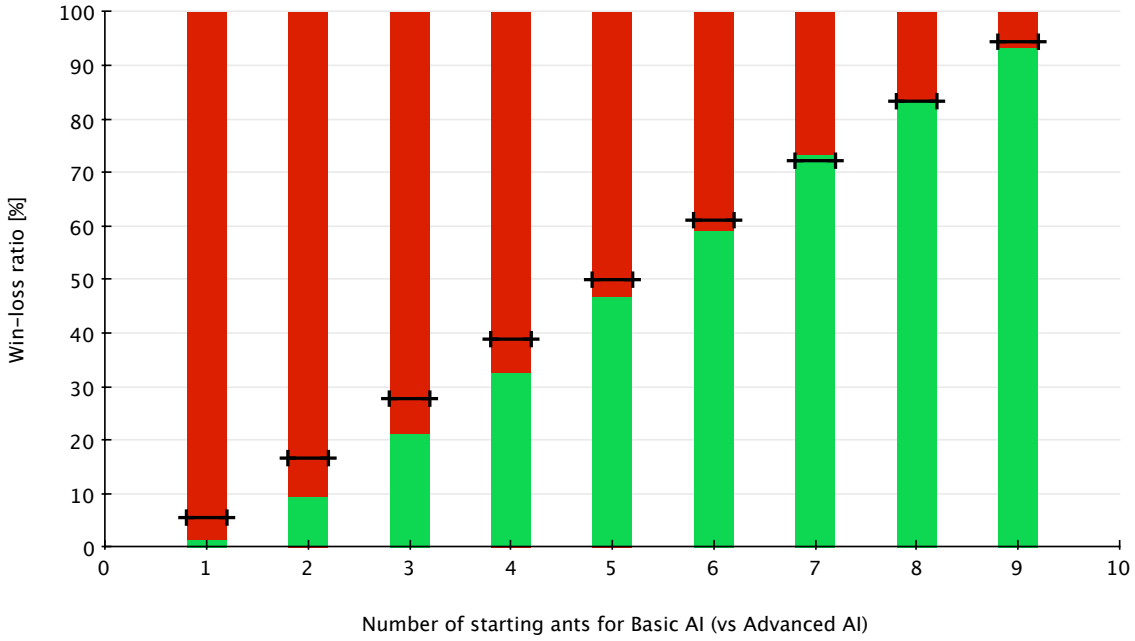


Figure A.9: Death match experiment: Chances of the basic AI to win against the advanced AI as explained at the beginning of this appendix A.

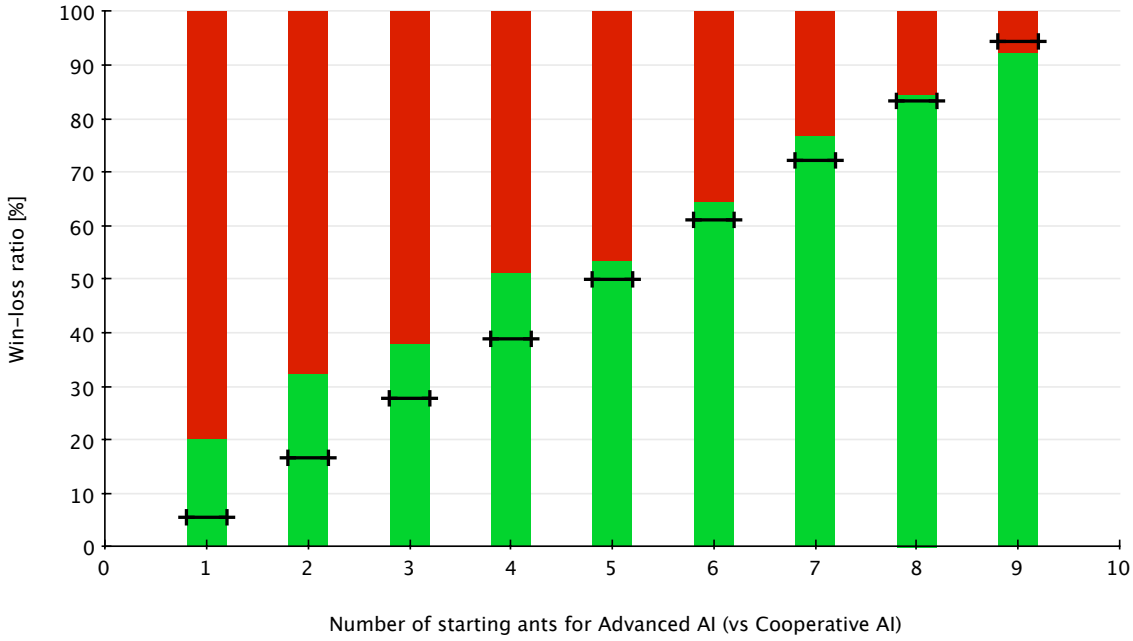


Figure A.10: Death match experiment: Chances of the advanced AI to win against the cooperative AI as explained at the beginning of this appendix A.

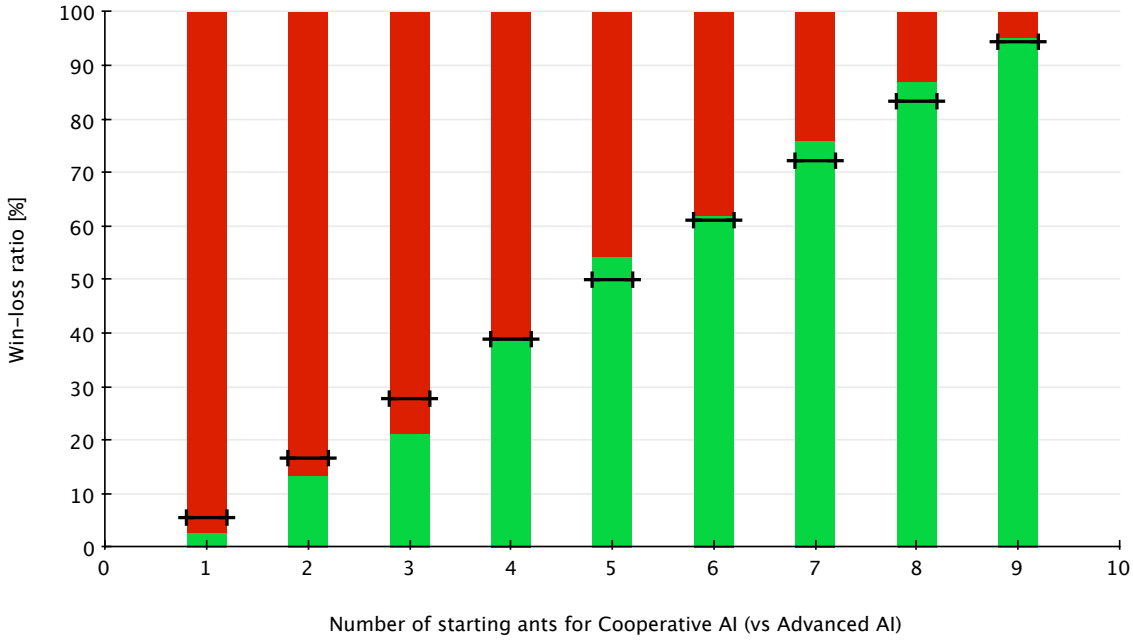


Figure A.11: Death match experiment: Chances of the cooperative AI to win against the advanced AI as explained at the beginning of this appendix A.

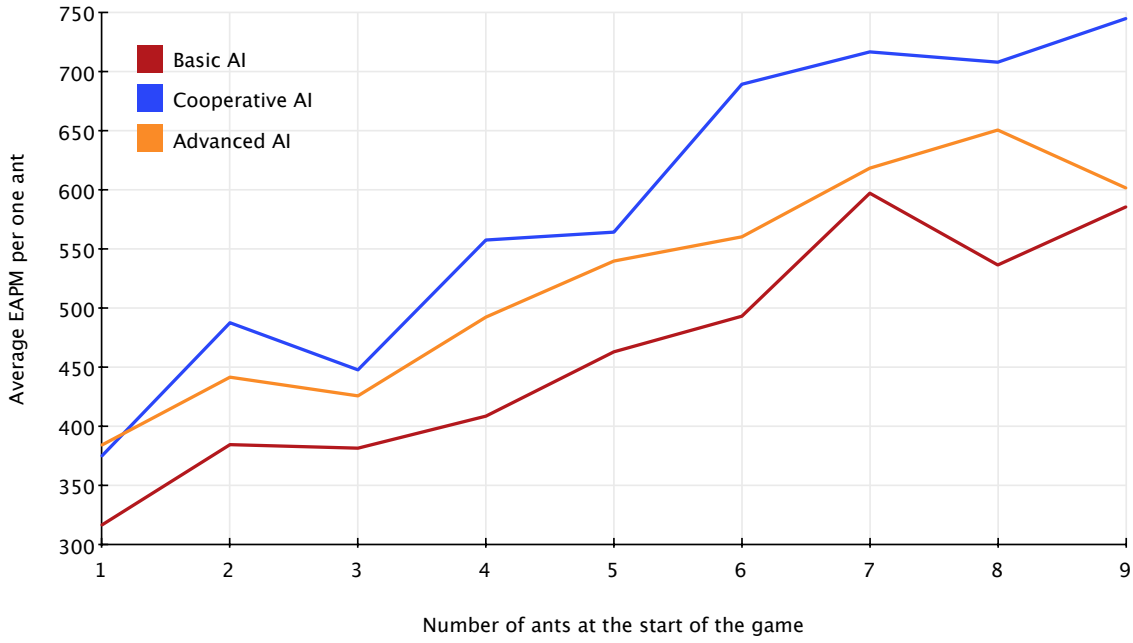


Figure A.12: Death match experiment: Average effective actions per minute per one ant with various number of ants in the beginning (only mirror match-ups are shown: 1 vs 1, 2 vs 2...).

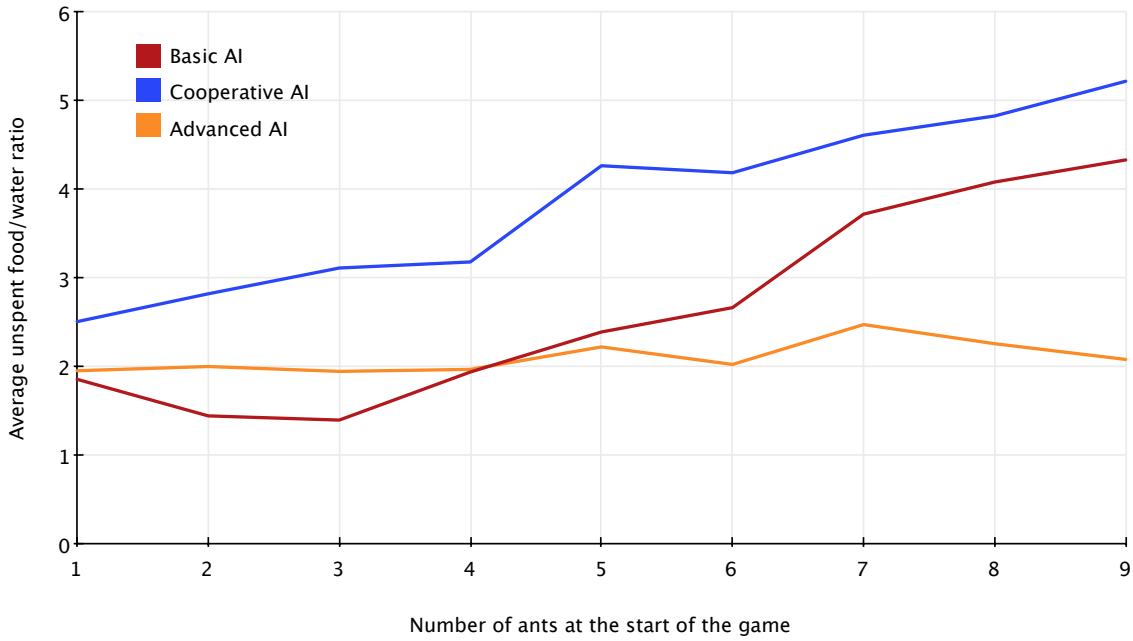


Figure A.13: Death match experiment: Average unspent food water ratio per mirror match-up with same number of ants in the beginning of the game.

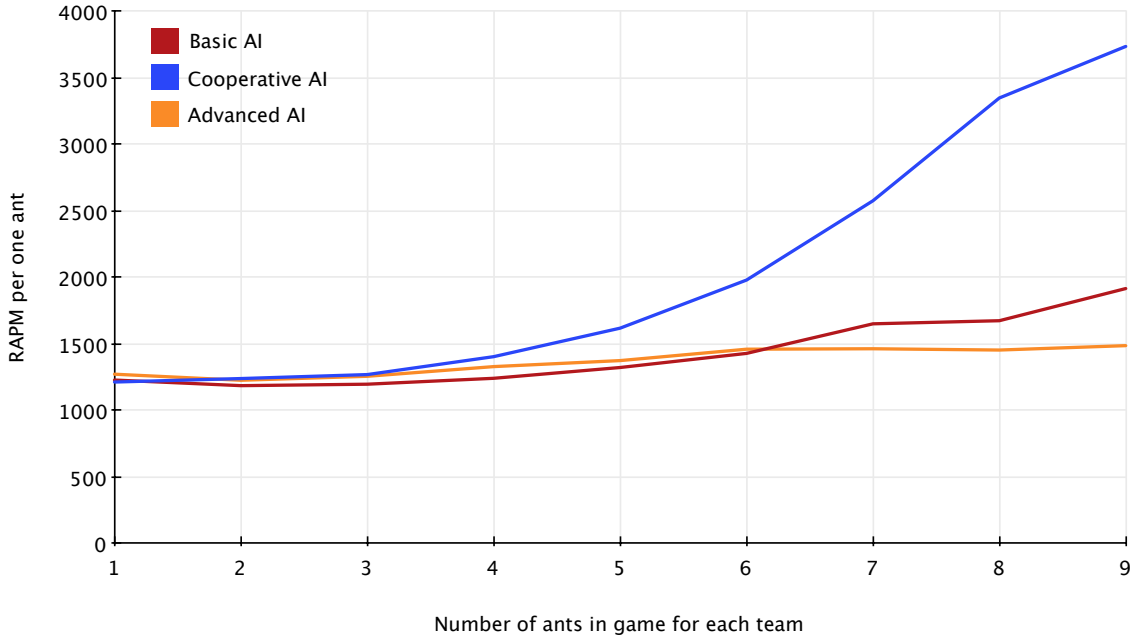


Figure A.14: Time complexity experiment: Average reasoning actions per minute per one ant and the dependency on number of coexisting units per each AI.

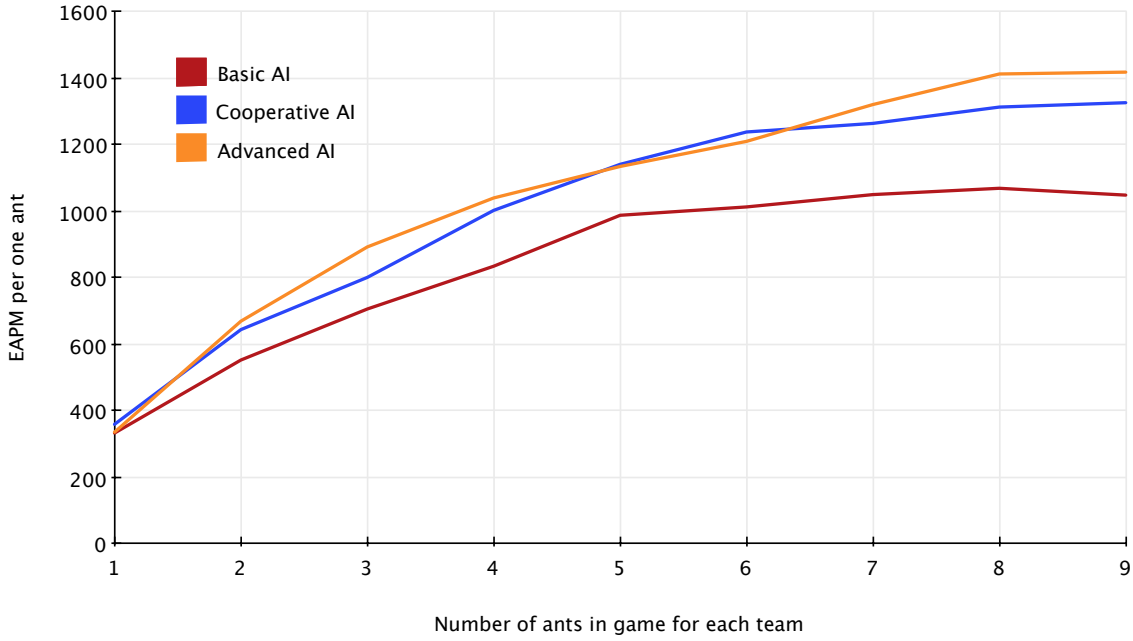


Figure A.15: Time complexity experiment: Average effective actions per minute per one ant and the dependency on number of coexisting units per each AI.

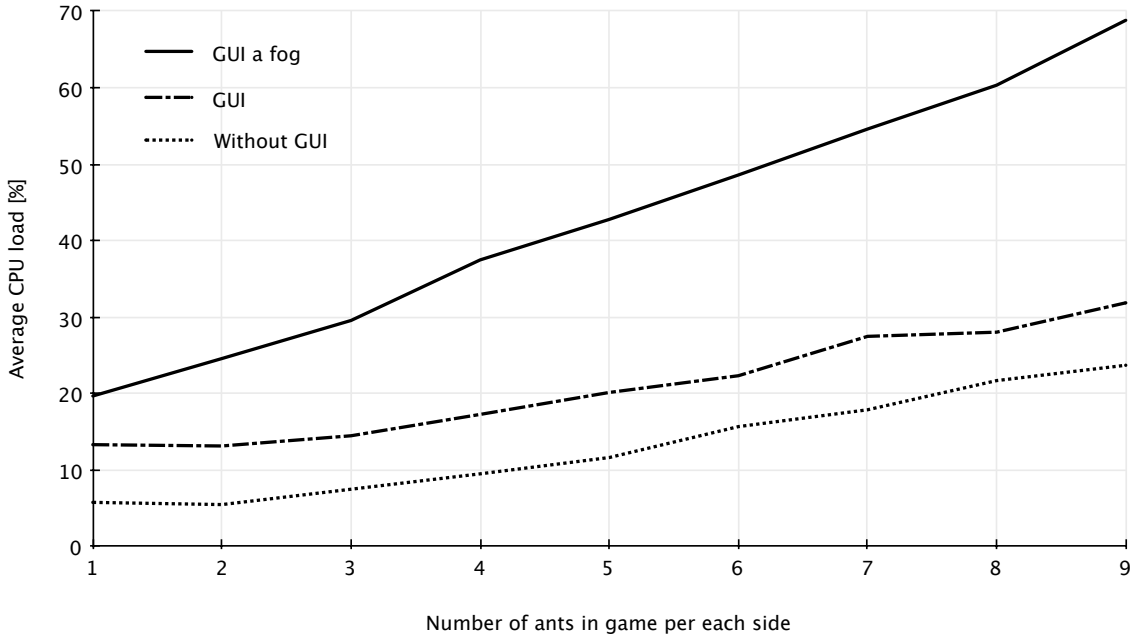


Figure A.16: Performance experiment: Impact of GUI and fog of war on the game performance for various number of ants (four factions of basic AI with 1 to 9 ants).

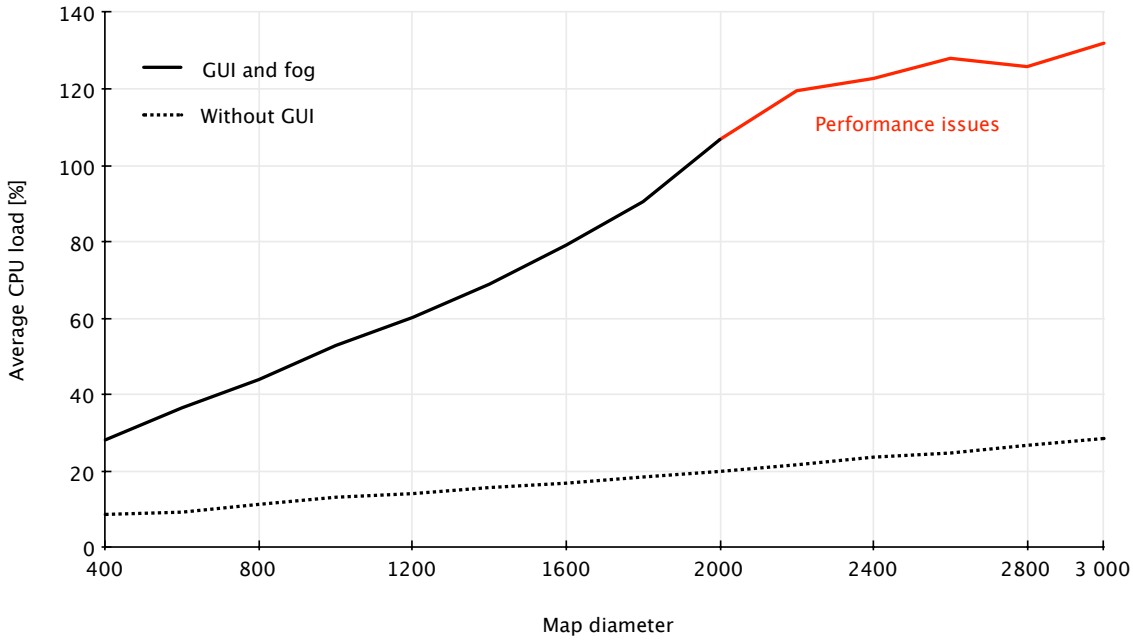


Figure A.17: Performance experiment: Impact of map size on the game performance.

Appendix B

Game Analysis Using Game's Logging Capabilities

This appendix shows logging capabilities of the Anthill Strategy game. All graphs in this chapter can be made from data saved after each game to the `logs` folder. In the analysed games three factions competed against each other driven by basic, cooperative and advanced AIs. They all started with 3 ants on the map shown in figure B.1. Please note that the map is unfair for the advanced AI.

Game length:	275457
Maximal army:	22
Total food created:	1650
Total water created:	1733
Winner:	faction A - basic AI

Figure B.1: Map used in game analysis.

Description	Basic AI	Cooperative AI	Advanced AI
Score:	85.2902	47.9523	27.8888
Ants created:	33	39	15
Average army:	12.7858	5.6905	1.4317
Armour upgrades:	18	7	6
Attack upgrades:	19	7	7
Speed upgrades:	20	11	5
Average upgrades:	31.9302	20.622	17.4927
Effective actions:	174999	148520	40711
EAPM:	3811.8254	3235.0603	886.7664
Average food collection rate:	20.1796	11.0239	4.7868
Average water collection rate:	20.3947	10.6159	4.1776
Average unspent food:	11.2146	8.744	6.0884
Average unspent water:	66.205	74.1894	8.61898
Unspent food/water ratio:	0.1694	0.1179	0.7064

Table B.1: Game overview.

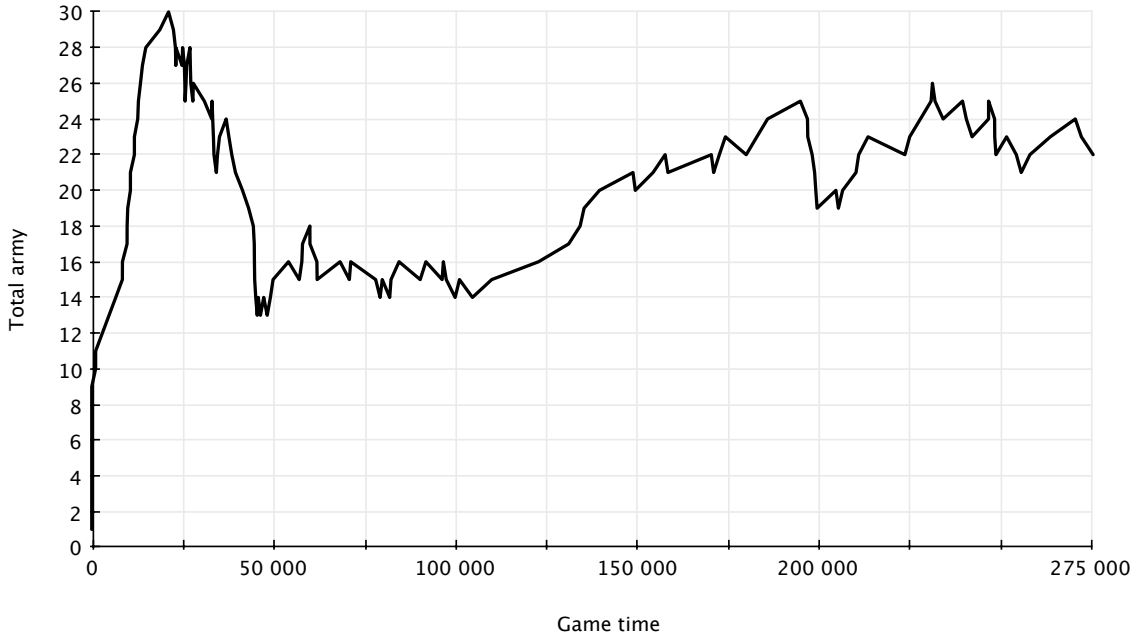


Figure B.2: Overall army size throughout the game.

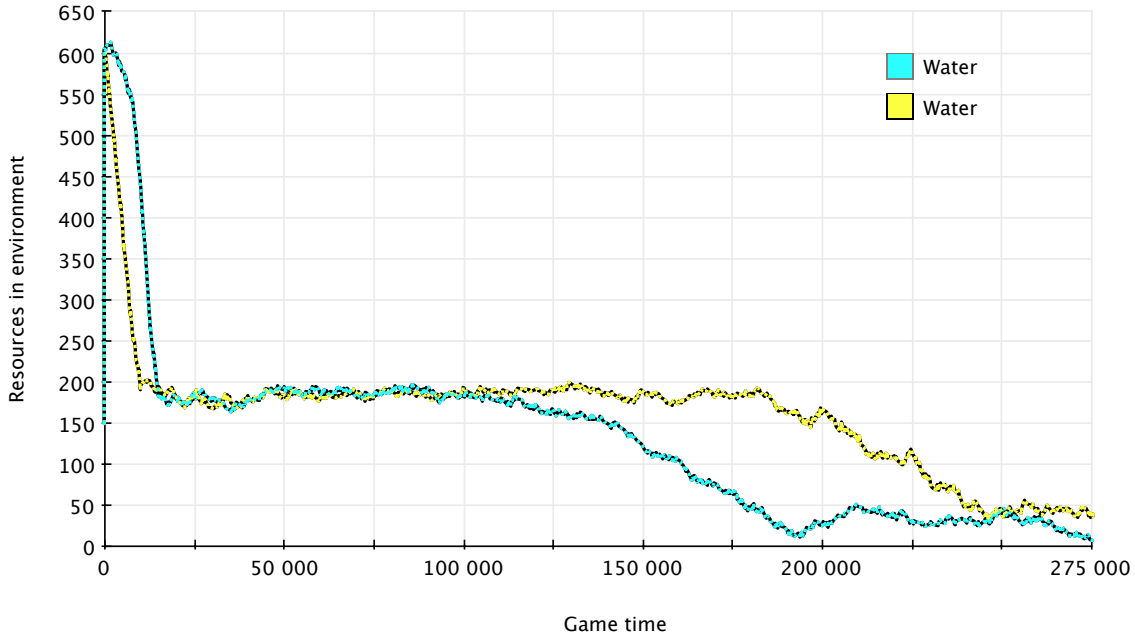


Figure B.3: Uncollected resources in the game environment.

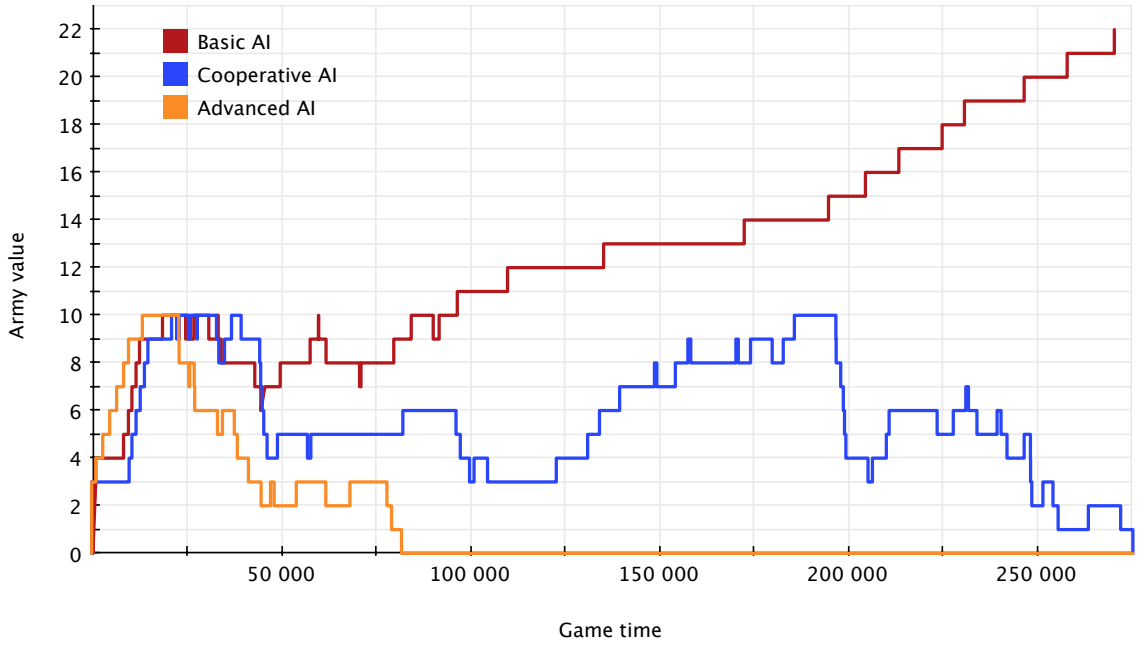


Figure B.4: Number of living ants for each faction.

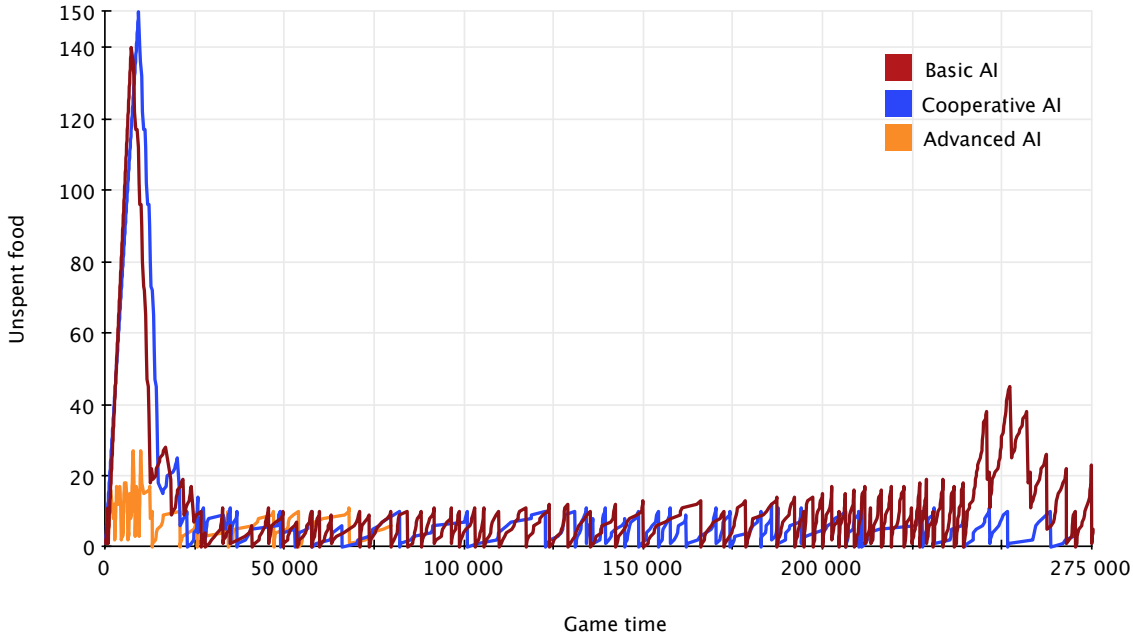


Figure B.5: Unspent food for each faction.

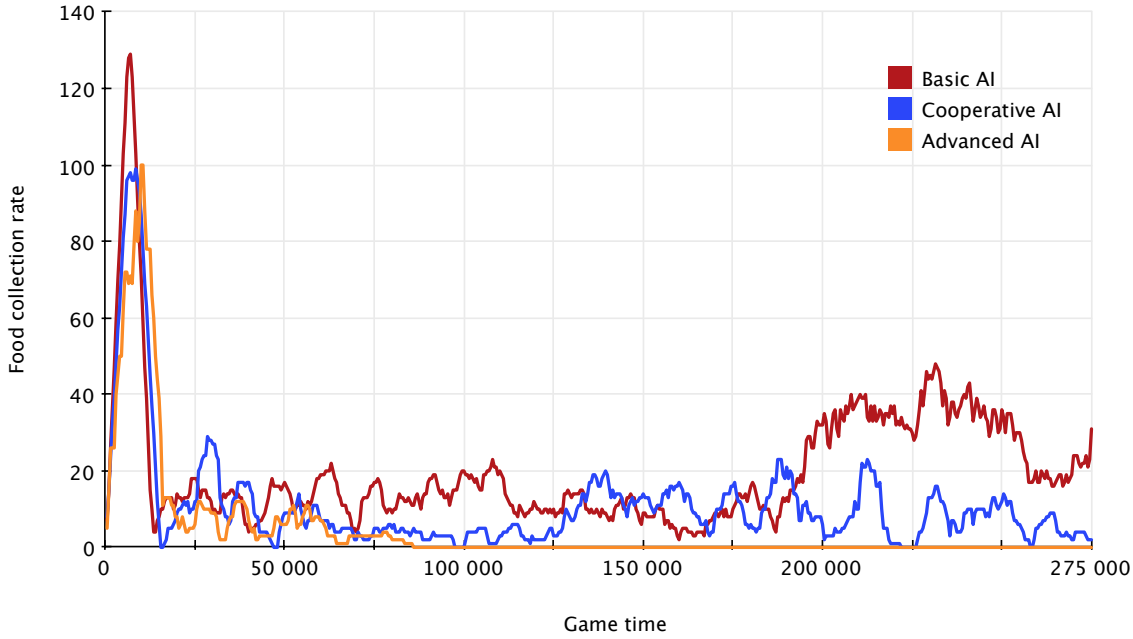


Figure B.6: Food collection rate per last game minute for each faction.

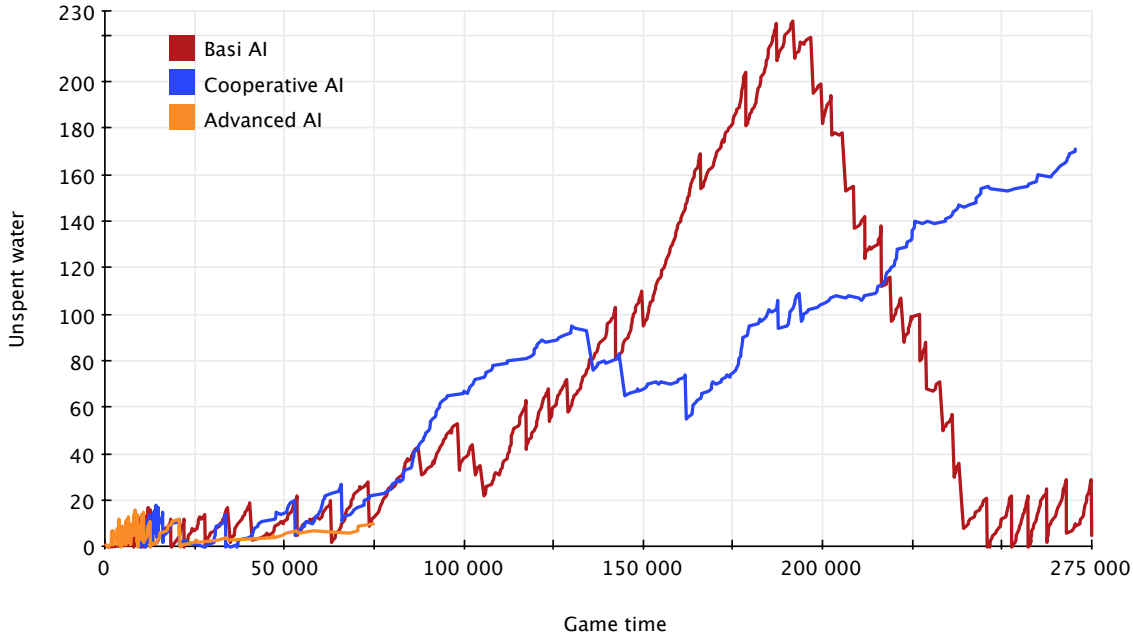


Figure B.7: Unspent water for each faction.

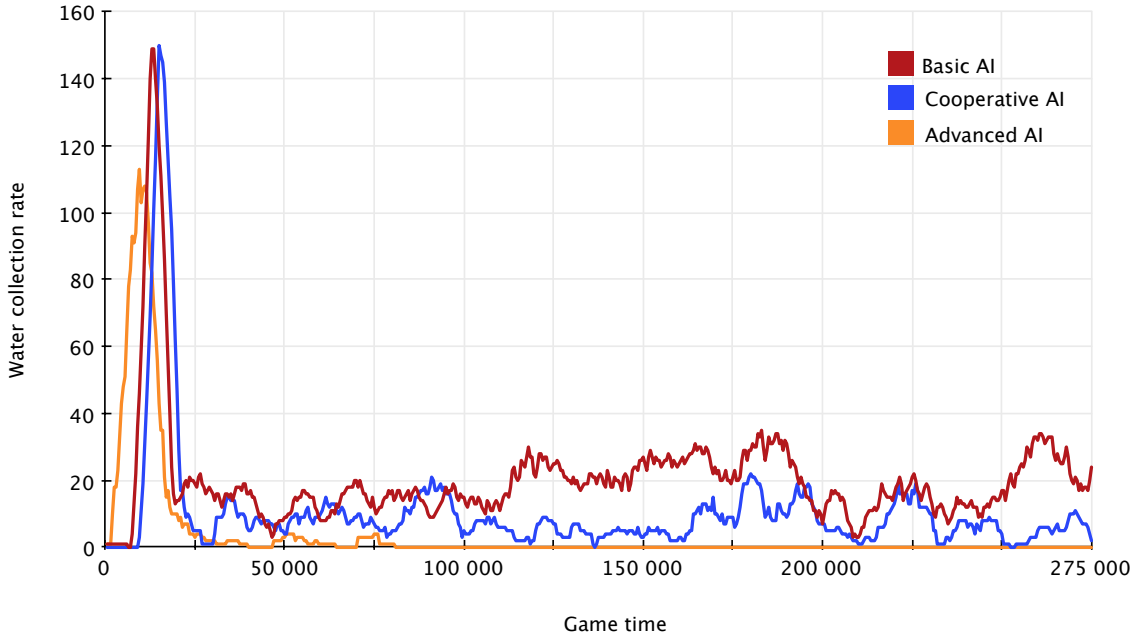


Figure B.8: Water collection rate per last game minute for each faction.

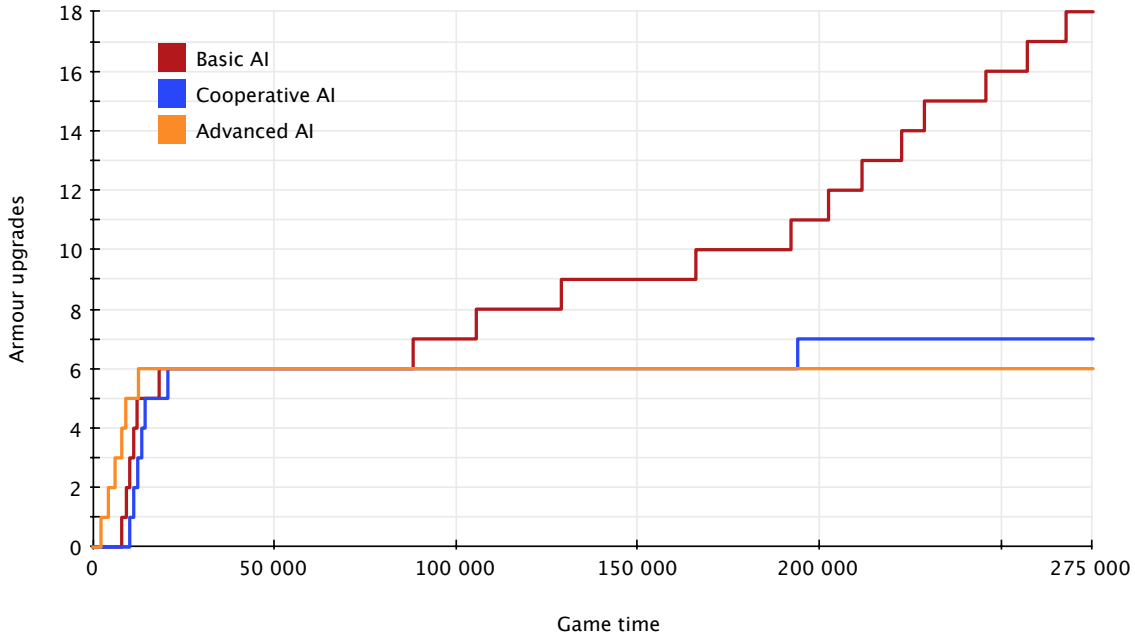


Figure B.9: Armour upgrades for each faction.

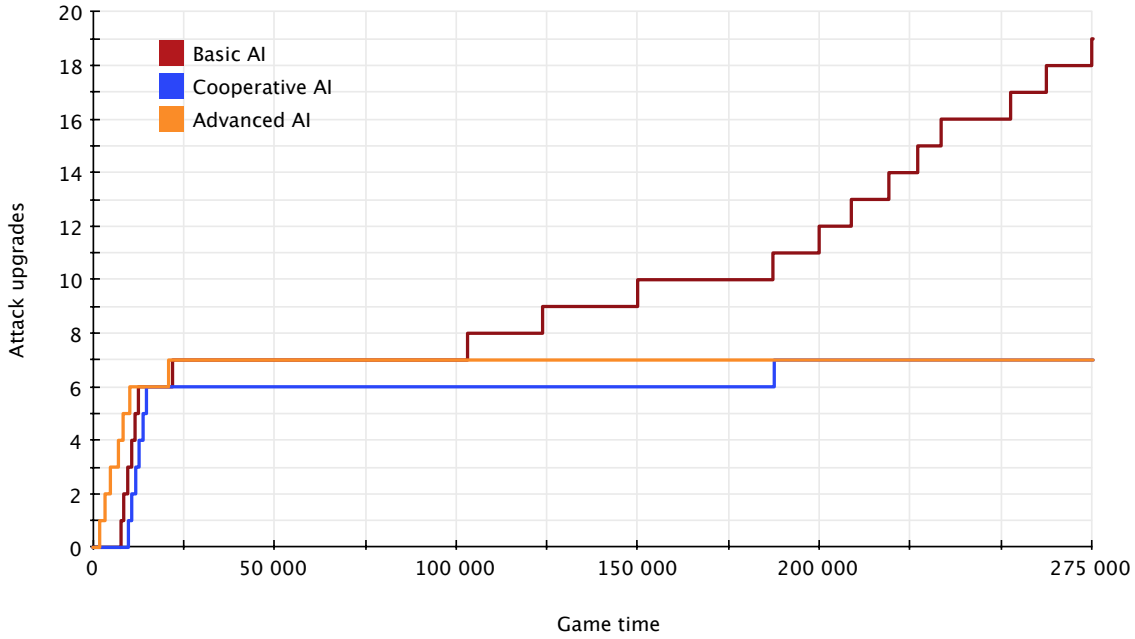


Figure B.10: Attack upgrades for each faction.

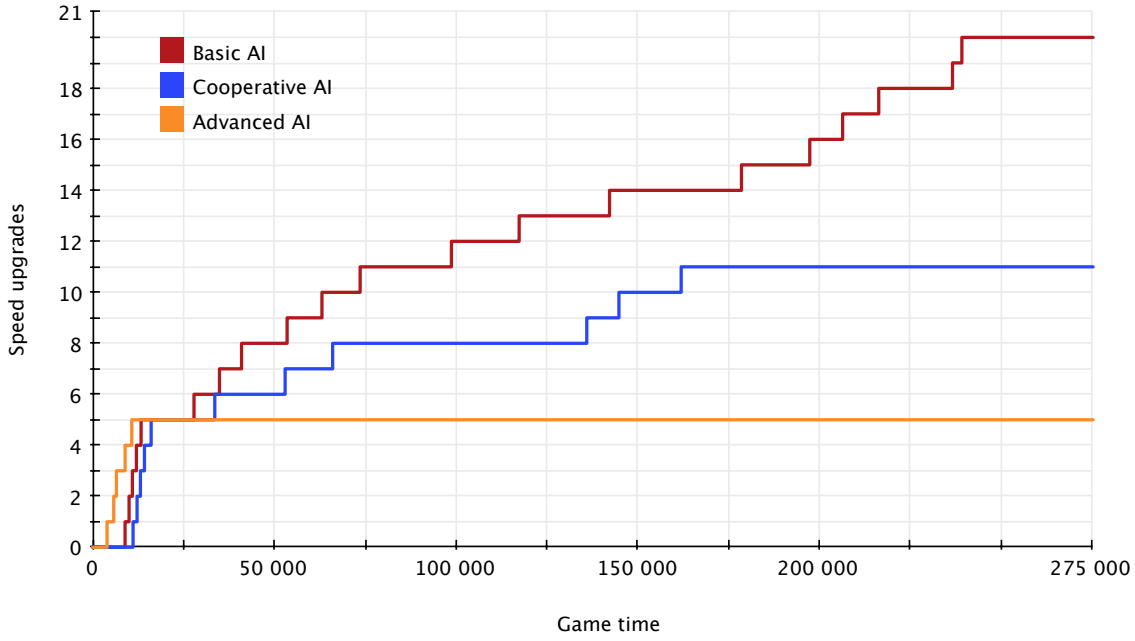


Figure B.11: Speed upgrades for each faction.

Appendix C

Game Design

This appendix offers more in depth game design overview extending the design chapter 5 of the thesis.

C.1 Unit's Attributes

- **Armour:** Armour reduces damage dealt to unit by enemy's attack. If enemy has attack 10 and unit has armour 3, the resulting damage will be 10 - 3 = 7.
- **Attack:** The value of attack describes damage dealt to enemy unit with armour of 0. The damage dealt to an enemy is reduced by its armour. Starting attack value is 10.
- **Speed:** Speed describes how fast can the unit move. Starting value is 100 pixels per game second.
- **Health:** Every unit starts with 100 health points. Any damage dealt to unit decreases number of health points and when health drops below zero unit dies. However health are replenished over time.
- **Perceptual range:** Unit can only see limited part of the environment. Only units and resources within perceptual range are perceived by unit. However resources that had been once perceived are remembered with values at the time when they were perceived lastly.
- **Collection rate:** Describes for how long will the unit fell asleep after collecting or unloading resource.
- **Attack rate:** Describes for how long will the unit fell asleep after attacking an enemy.

C.2 Unit Control

- **Move:** To order unit to move you can select it, choose action *move* from the action panel and click with left-mouse button anywhere on the map. Alternatively just right click with mouse.

Upgrade	Food	Water	Starting value	Change	Value cap
Armour	6	4	0	+1	-
Attack	6	4	0	+1	-
Speed	0	10	100 pixels/game sec	+1	150

Table C.1: Available upgrades for units.

- **Attack:** To attack with a unit please choose action *attack* from the action panel and left click on an enemy. If the chosen location is free of enemy units, unit moves to location where you clicked while attacking on every enemy unit it encounters. Alternatively you can attack by right clicking on an enemy unit. Please note that when unit has no action to perform, it automatically attacks on any enemy unit within perception range and whilst the enemy unit remains within perception range it will pursue it.
- **Gather:** To gather resource please choose action *gather* from the action panel and left click on any resource. Alternatively just right click on the resource.
- **Stop:** By selecting action *stop* unit will immediately abort any action it does. Please note that if a unit is controlled jointly by AI and user a new action (that can be same as aborted one) can be immediately chosen by AI. To prevent any action use hold instead.
- **Hold:** When action *hold* is selected, unit will ignore all non user actions. This means that if a unit is controlled jointly by AI and user holds prevents AI to perform any action. On top of this hold prevents auto attack on units within perception range.

C.3 Game Short-cuts

Key	Description
M	Selects action move
A	Selects action attack
G	Selects action gather
S	Selects action stop
H	Selects action hold
Shift + selection	Adds new units to already selected group (not yet implemented).
Shift + action	Adds new action to units queue of actions (not yet implemented).
Shift + number	Adds selected units to control group 0 - 9 (not yet implemented).
Ctrl + number	Replaces control group 0 - 9 with selected units (not yet implemented).

Table C.2: Short-cuts useful for commanding units

Key	Description
N	Creates new agent
T	Upgrades attack
R	Upgrades armour
E	Upgrades speed

Table C.3: Anthill management short-cuts

Key	Description
Escape	Pauses game and displays menu and if pressed again returns to game.
Mouse left click	Selects game elements under mouse cursor, or within rectangle created by dragging mouse, based on their priority. The priority of selection is as follows: players units > other units > resources. So for example when user drags over area where are enemy's units and resources, the game will select those enemy's units.
Mouse right click	Commands selected group of a player controlled units to either attack enemy unit, collect resource or move to location where player clicked.

Table C.4: Other game short-cuts and inputs.

Appendix D

Game Customization

In this section we will briefly talk about easy to use customizations of the game. You can quite easily create your very own map as well as add new AI.

D.1 Java-Jason Interface

Because we need to access game's properties from the Jason, we had to create a number of internal and environment actions. Internal actions allows us to get information from the game into the Jason. They are implemented in package actions and can be triggered from Jason by `actions.name_of_action`. You can see list of them along with brief description below.

Custom Internal Actions

- `calculateDistance(X1,Y1,X2,Y2,D)` - Unifies D with square distance between two points with coordinates X1, Y1 and X2, Y2.
- `dead(Agent)` - Returns true if agent with name unified with variable Agent is dead.
- `finishedMovement(Agent)` - Returns true if agent with name unified with variable Agent has finished all it's actions.
- `getEscapeDirection(Agent, Enemy, X, Y)` - Bounds X and Y with coordinates suitable for running away from Enemy.
- `getGameTime(Time)` - Unifies Time with current game time.
- `getHome(Agent,X,Y)` - Unifies X an Y with coordinates of anthill for Agent's faction.
- `getPosition(Agent,X,Y)` - Unifies X and Y with position of Agent.
- `getRandomDirection(X,Y)` - Unifies X and Y with random coordinates within game area that are free of obstacles.
- `getUpdateRate(Rate)` - Unifies Rate with period of perceptual update for current game settings.
- `isCloser(X1,Y1,X2,Y2,X3,Y3)` - Returns true if distance between points with coordinates X1, Y1 and X3, Y3 is lower then distance between points with coordinates X2, Y2 and X3, Y3.

- `isWeaker(Enemy,Agent)` - Returns true if Enemy is stronger then Agent based on current updates.
- `teamBroadcast(Action, Message)` - Sends the Message to all agents of the triggering agent's team. How the message is interpreted depends on Action which can be one of following: achieve, askOne, tell or untell. The semantics is same as in Jason's build-in internal action .send.

Internal actions can also be used for influencing environment, but better approach is use of environment's `executeAction()` method. Following actions are used for any actions done by agents in Java.

Ant Environment Actions

- `update_percepts` - Updates percepts about ant surrounding.
- `crawl(X,Y)` - Set new destination with coordinates X and Y for the ant.
- `collect(water)` - Collects water on ant's current location. If there isn't any returns false.
- `collect(food)` - Collects food on ant's current location. If there isn't any returns false.
- `unload` - Unloads any food or water from ant to the anthill. Returns false if ant is not at Anthill location.
- `attack(Enemy)` - Sets enemy to be attacked by ant.
- `stop_attack` - Stops any attack.

Anthill Environment Actions

- `update_percepts` - Updates percepts about anthill resources, units status and upgrades price.
- `new_agent` - Creates new ant for the team of anthill. Removes adequate amount of resources.
- `upgrade_attack` - Upgrades attack for all ants under anthill command. Removes adequate amount of resources.
- `upgrade_armor` - Upgrades armour for all ants under anthill command. Removes adequate amount of resources.
- `upgrade_speed` - Upgrades speed for all ants under anthill command. Removes adequate amount of resources.

D.2 Percepts

Ant Percepts

- `pos(X,Y)` - Agent's current position.
- `hp(HP)` - Agent's current health. When the HP drops to 0 the ant dies.
- `collected(Resource)` - Describes which resource, if any, the ant is carrying.
- `resource(X, Y, Amount, Type)` - Percept about resource field at location with coordinates X and Y, some amount and a certain type.
- `friend(X, Y, Name)` - Percept about perceived friendly ant.
- `enemy(X, Y, Name)` - Percept about perceived enemy ant.

Anthill Percepts

- `food(Amount)` - Amount of available food for the anthill.
- `water(Amount)` - Amount of available water for the anthill.
- `armour(Level)` - Level of current armour upgrade.
- `attack(Level)` - Level of current attack upgrade.
- `speed(Level)` - Level of current speed upgrade.
- `army(Number)` - Number of ants under anthill command.
- `armour_upgrade_price(Food,Water)` - Price of the next armour upgrade.
- `attack_upgrade_price(Food,Water)` - Price of the next attack upgrade.
- `speed_upgrade_price(Food,Water)` - Price of the next speed upgrade.

D.3 Initial Beliefs

Ant Beliefs

- `home(X,Y)` - Coordinates of the anthill for the faction.
- `update_rate(Rate)` - Time between perceptual upgrades.
- `anthill(Anthill)` - Name of the anthill for the faction.

Anthill Beliefs

- `update_rate(Rate)` - Time between perceptual upgrades.
- `speed_cap(Cap)` - The speed upgrade cap.
- `new_ant_price(Food, Water)` - Price of creation of the new ant.

D.4 Custom Maps

You can create your own map by modification of the `WorldsDatabase` class. First step is to add the name of your map to the `Worlds` enumerator. Then you need to add case statement for the map name to switch in `createWorld()` function. The last step is creation of a function which defines your new map. There are several methods you can use described below. Please note that all of them are invoked on reference of `World` instance.

Mandatory Settings

- `setHeight(int)` - Sets height of your map.
- `setWidth(int)` - Sets width of your map.
- `getTeam(EnumTeams.t).setAnthill(int x,int y)` - Sets position of anthill for faction t (substitute with a, b, c or d) to location x, y.

Optional Settings

- `addFood(int x, int y, int amount)` - Adds food with certain amount to location with coordinates x,y.
- `addWater(int x, int y, int amount)` - Adds water with certain amount to location with coordinates x,y.
- `addObstacle(Point p1, Point p2, Point p3)` - Creates an obstacle covering area within points p1, p2 and p3.
- `addTexture(String path)` - Adds reference to a texture which has to have same size as the newly created map and will be displayed on the background of the map.

D.5 Custom Artificial Intelligence

Apart from your own map, you can easily create your own AI for both units and anthills. Firstly add name of your AI to enumerator `EnumAI` in the `AIDatabase` class. Then add path to your `asl` file, implementing behaviour of the anthill for your AI, to functions `getAnthillAI()` in the same class. Last step is adding the path for unit's `asl` file to function `getAnthillAI()` in the `AIDatabase` class. Rebuild your project and your newly implemented AI should be ready to go. You can use the `blankAI.asl` and `blank_anthill.asl` source files and extend them at your will.

D.6 Compilation from Source Codes

To recompile the project, run the following command while in the game folder:

```
ant -f bin/c-build.xml jar
```

This command should compile all source files and pack them to executable jar file in the game folder. It also adds any `asl` files from folder `./src/asl` and any images from folder `./img`.