uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Alejandro Luján

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Generation of Rule-based Adaptive Strategies for Games

TITRE DE LA THÈSE / TITLE OF THESIS

Azzedine Boukerche

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Emil Petriu

Dorina Petriu

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Generation of Rule-based Adaptive Strategies for Games

by

Alejandro Luján

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.Sc. degree in
Master of Computer Science

School of Information Technology and Engineering
Department of Computer Science
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-51847-2
Our file Notre référence
ISBN: 978-0-494-51847-2

Abstract

Real Time Strategy Games (RTSG) are a strong test bed for AI research, particularly on the subject of Unsupervised Learning. They offer a challenging, dynamic environment with complex problems that often have no perfect solutions. Learning Classifier Systems are rule-based machine learning techniques that may rely on Genetic Algorithms to discover a knowledge map used to classify an input space into a set of actions. This work focuses on the use of Accuracy-based Learning Classifier System (XCS) as the learning mechanism for generating adaptive strategies in a Strategy Game. The performance and adaptability of the strategies and tactics developed with the XCS is analyzed by facing these against scripted opponents on an open source game called Wargus. Our results indicate that Genetic Algorithms can be used effectively to enhance Real Time Strategy Games.

List of Publications

The following publications by the author are relevant to the work in this thesis, with the collaboration of the supervisor Dr. Azzedine Boukerche.

- Journals
  - Lujan, A., Boukerche, A., and Pazzi, R.: Using Accuracy-based Learning Classifier Systems for Adaptable Strategies in Games and Interactive Virtual Simulations. To be submitted.
- Conferences
  - Lujan, A., Werner, R., and Boukerche, A.: Generation of Rule-based Adaptive Strategies for a Collaborative Virtual Simulation Environment. In *HAVE 2008 7th IEEE International Workshop on Haptic Audio Visual Environments and Games*, 2008.

Acknowledgements

With regard to the debt of gratitude owed to our parents, our father may be likened to heaven and our mother to the earth, and it would be difficult to say to which parent we are the more indebted

—The Writings of Nichiren Daishonin, page 930 (The Sutra Of True Requital)

To my parents, who have forged and cultivated everything I am today. I will forever hold a profound debt of gratitude, that can only be repaid by being the best I can be.

To Dani, my life partner, who has the unique ability to extract the best of me. Thank you for always helping me raise my life condition even when obstacles seem too hard to overcome.

To my examples and guides in faith: Michael, Ken, Sue, Koichi, Dougerma, and of course, Dani, my most consistent example and inspiration. A special mention to Mig, whose personal guidance helped me create a strong determination to finish the last phase of this work and achieve victory.

To my fellow buddhists, to my personal friends, to my couchsurfing pals... thank you for being an inspiration at one point or another in my life.

Finally, a special mention to my supervisor, Dr. Azzedine Boukerche, who provided the support needed to make this work what it is, and to Dr. Richard Pazzi who collaborated in reviewing this thesis and provided valuable suggestions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Games and Artificial Intelligence

The relationship between Artificial Intelligence and multimedia systems is twofold. On one hand, the creation of a realistic experience in multimedia systems is often an objective where AI can be of aid. On the other hand, AI researchers can find in these systems — particularly large simulations and games — a robust test-bed for their work [59] [66] [88] [15] [16]. Many researchers have been working in the area of Game AI where the needs and constraints of both fields intersect, resulting in many opportunities and open questions. It is the belief of many, and mine in particular, that the advancements of Artificial Intelligence will shape the use of technology in decades to come. Games are no exception.

The main motivation for game players is entertainment. Millions of players are willing to invest money in their personal recreation. This is one of the reasons for the continuous growth the game industry has experienced in the last three decades. Nonetheless, deciphering *what* pleases the crowd (and what will please them *tomorrow*) is one complex and dynamic art. We, as consumers, seem to be drawn to a rich mix of realism and fantasy. So, we have witnessed the success of games ranging from cartoonish-looking such as the Zelda series [89] to titles with very realistic graphics and physics such as FIFA 08 [50]. Nonetheless, we can easily point to one aspect of reality that seems to spark the interest of many users: the human-like behavior of virtual agents. Regardless of the graphical realism of the game, players tend to be drawn toward an environment that shows signs of intelligence. A character will be more believable if his dialog seems rational, the animation of a pet will seem more real if signs of animal-like intelligence

are shown. A player will create a stronger bond with an ally character if he senses some humanity in it. Facing an opponent that shows intelligent characteristics, such as planning or adaptation, can make the challenge more difficult and entertaining. The user experience is, then, heavily influenced by the behavior of artificial agents.

Historically, game agents controlled by the computer are usually driven mostly by scripts written by designers and developers. A script is typically a sequence of game actions, such as:

1) Build a town centre

2) Train some soldiers

3) Attack enemy town

One disadvantage with scripted opponents is that they can be deciphered and countered by an intelligent player. As soon as the player discovers the pattern behind the opponent's behavior, he has the opportunity to find weaknesses and act accordingly. For example, if the sequece of actions of the artificial agent depends on a particular type of building, the human player could destroy that building, preventing the agent to continue with his plan. While some degree of randomization can mask the script nature of the agent, it is also possible to develop solutions that integrate Artificial Intelligence techniques.

1.2 Some Artificial Intelligence Techniques Used in Games

The portfolio of AI techniques that can be used in games is as wide as the array of problems to be solved. Table 1.1 shows some well known techniques that have been used in games to solve particular problems. Techniques such as A*[41] and Finite State Machines[83] are widely used in commercial games, while others have been tested in research projects but have yet to see the public light. Chapter 2 will expand on this topic.

In this thesis, we are proposing the use of an unsupervised machine learning technique called Accuracy-based Learning Classifier Systems (XCS) for adaptable strategy generation[122]. XCS uses a Genetic Algorithm to evolve a knowledge base in the form

Technique	Example of use
A*	Navigation, Planning
Finite State Machines	Reactive agents
Genetic Algorithms	Tactical planning
Influence maps	Spatial decision making
Natural language processing	Human-computer interaction
Neural Networks	Pattern matching

Table 1.1: Sample of AI techniques

of rules. These rules classify the input space into an action space, and its particular representation allows for powerful generalizations.

XCS have been proved to be successful on a number of classification and learning tasks, including data mining [1], function discovery [48] and economical models [20]. In this thesis, XCS will be used to evolve sets of rules that represent game tactics for the Wargus game[40], essentially by combining different game actions while receiving information from the game state as inputs.

1.3 Motivation

The motivation of this thesis is to explore the implementation of an Artificial Intelligence technique in games. An environment that seems static or scripted is much less interesting and immersing than one that reacts and adapts to the actions of the player. Working on the adaptability of agents in simulations and games seems to be an important task toward improving the role of AI in these systems.

Some techniques have been used for modeling reactive agents[107], such as Finite State Machines[83], but their limitations tend to generate behaviors too basic for the human opponent. Accuracy-based Classifier Systems have shown very positive results in previous works [1][40][127][47][129], and their ability to generalize allows for representation of high-level concepts. Our review of the literature showed little evidence of works using XCS for strategy generation in complex games. This thesis aims to seize such opportunity.

We can state the research question as follows: *Can Learning Classifier Systems be used to generate adaptive game strategies with performance comparative with usual scripted solutions?*

1.4 Contributions

The main contribution of this thesis to the areas of Classifier Systems and Games can be summarized as follows:

We have applied Accuracy-based Classifier Systems to a Strategy Game. We report on its performance. We study the behavior of the system in two variations of the problem: *immediate* rewards and *delayed* rewards, and then we report on the implications and challenges faced in each case.

1.5 Thesis Structure

This thesis is structured as follows: Chapter 2 presents background information and a review of related work on Game AI and Classifier Systems. Chapter 3 describes the architecture and implementation of our system and highlights challenges and limitations that were faced during the design phase. Chapter 4 reports on the evaluation of our system using different game scenarios. Finally, Chapter 5 presents conclusions on the overall work and future research directions.

Chapter 2

Background Information and Related Work

> *It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not understand something until he can teach it to a computer.*
>
> —*Donald E. Knuth*

This chapter reviews background information on the topics of Game Artificial Intelligence and Learning Classifier Systems, and examines related work on these areas.

2.1 Games

2.1.1 Brief History

Artificial Intelligence in Games has gone a long way since the era of Atari and Space Invaders in the 70's. Back then, industry produced single player games with very simple AI or multiplayer games without any AI at all, nonetheless resulting in users quite amazed by the capabilities of these new 'toys'. The AI routines were limited to very simple deterministic scripts, with the addition of some randomization to escape monotony. The two following decades saw a fast rise on graphical power in the consumer end hardware[69], while AI was relegated to a much lower priority. In the mid 90's, the field of AI took important steps, among them the well known victory against the Chess world champion, with IBM's program Deep Blue [49]. By that moment, players were starting to appreciate

more the importance of a game being *intelligent.* AI techniques, long ago proved useful in academia, started being implemented in commercial products, such as Creatures[29]. As time passed, some games went even further, making AI part of the main game experience, such as Black and White[115] and NERO[112]. With the advent of network-able multiplayer games, artificial agents continued to be pressured further into exhibiting strong intelligent traits such as collaboration and communication[32]. Figure 2.1 shows a timeline with some of the most important games from the point of view of Artificial Intelligence.

The future of games surely holds a good few surprises for us, but we expect to see sooner than later more solid attempts at stronger behaviors such as adaptability, planning, reasoning and natural language processing.

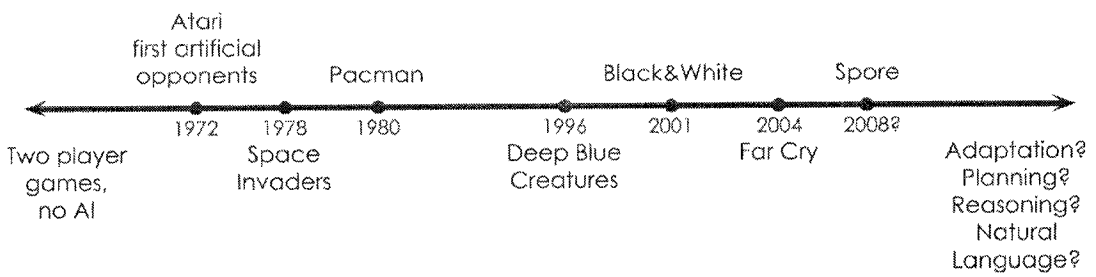


Figure 2.1: An AI timeline

2.1.2 Strategy Games

Among the many types of games, there is one characterized for its complexity and difficulty from the point of view of Artificial Learning: Strategy Games. Strategy Games are very complex environments, that typically develop in a setting that involves growing small civilizations, in scenarios that include some form of technological advances, diplomacy and war. Players execute actions of a varied nature, that usually include gathering resources, trading with other players, creating buildings, training units, engaging troops in battle and developing technology. Usually, the final objective is to eliminate enemy troops or conquer areas of the maps, and intermediate goals can include controlling resources, forming armies and equipping them with the proper technology.

Each player has partial knowledge of the game state; he will have detailed information of his situation, and he might know how many opponents he faces and their overall characteristics, but he will rarely know their specific status such as resources gathered or amount of units trained. This incomplete knowledge makes strategies hard to design, but it is also an important part of the entertainment value of the game. Strategy Games games are very popular in the market today, and examples of successful commercial products are Civilization[34], Age of Mythology[116] and Warcraft[31].

The level of complexity of a Strategy Game is increased by having multiple players making decisions that will have effects in a future state of the game. The construction of a particular building might seem unnecessary at a particular moment in the game, but could have strategic sense later. Development of a particular technology provides advantages that are often less visible to players. Interactions in the game are very complex, but players tend to adapt quickly to these interactions.

From the perspective of learning, Strategy Games represent a challenge: environments with multiple agents, imperfect knowledge, where actions have a delayed effect [1]. Developing strategies in such a setting using machine learning techniques implies exploring the space of game states and actions. Such space is too large for traditional search methods, and other techniques, that work well in similar problems, must be considered. These types of games pose a challenge for AI research in a number of areas [15][17].

2.1.3 Wargus: a Practical Example

A good example of a Strategy Game is Wargus[117], the game selected for this thesis. In Wargus, players have access to a set of maps, each with different amounts of resources, opponents and terrains. The actions that a player can perform on a particular moment fall within the following categories:

1) **Gathering Resources** Players can collect gold, wood and oil, which are the basic resources needed to perform other actions in the game.

2) **Building Structures** Such as Town Halls, Barracks and Towers. Some buildings allow training of particular units while others have defensive capabilities, supply food or improve resource gathering.

[1] In a delayed reward environment, the response to an action is received N time steps after it was executed. See 2.3.4 for more details.

3) **Training Units** Such as peasants, footmen or archers. Some units are able to build structures, others have different levels of strength in attacking and defending against enemies.

4) **Technology Research** Some buildings are able to develop technologies, which influence diverse parameters in the game, such as attacking statistics or vision range of units.

5) **Attacking** A player may send some (or all) his military units to attack opponent units or structures at any given time.

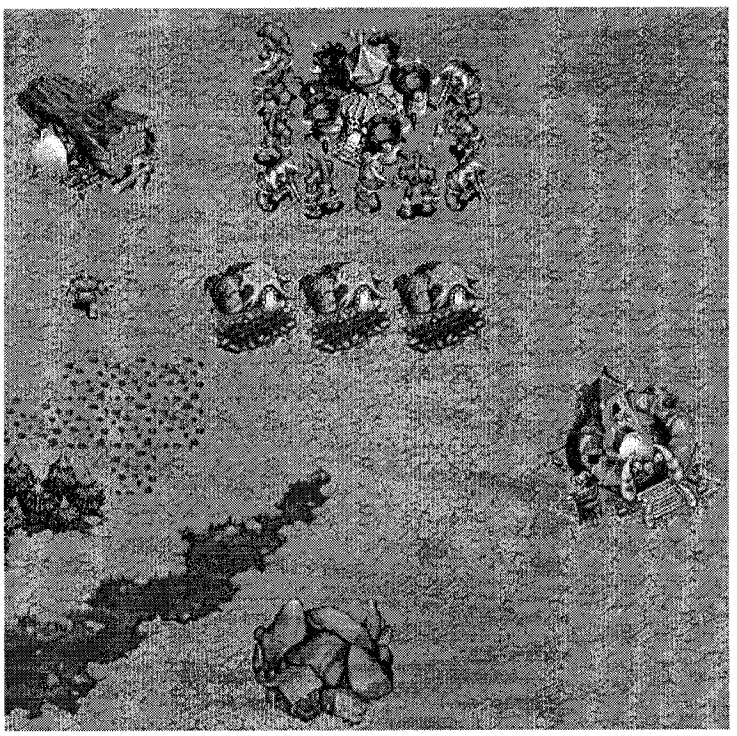


Figure 2.2: Screenshot of Wargus

Figure 2.2 presents a screenshot of Wargus, and shows some of the elements of the game environment. Two resources are present: Wood (center left) and a Gold Mine (bottom). Four types of structure are displayed in the image: a Town Hall at the right, Farms on the center, a Barrack at the top, and a Lumber Mill on the top left corner. Military units surround the barrack, and a peasant is moving towards the Lumber Mill.

As the game advances, and more units and structures are built, the amount of actions available to the player increases.

Selecting what actions to perform, and in what order, is what constitutes a specific strategy. We can use the terms static and dynamic, where a static strategy does not depend on particular game parameters, and can be expressed as a script. A dynamic strategy, on the contrary, has a number of possible variations depending on the course of the game. A particular goal might be achieved in many different ways, and selecting the most convenient one by observing the game state constitutes an effective dynamic strategy. Our goal with this thesis is to develop effective dynamic strategies, that are developed and represented using Classifier Systems.

Chapter 3 will explain the implementation of the system in more detail, including the interaction between the Classifier System and Wargus.

2.2 Game Artificial Intelligence

One of the roles of Artificial Intelligence in Games is to describe the behavior of artificial agents [2]. For this purpose, it often attempts to imitate intelligent behavior to efficiently solve problems such as pathfinding, planning, and spatial decisions[59]. In the next sections we describe two different approaches to behavior modeling: scripts and AI oriented systems.

2.2.1 The Common Technique: Scripts

In commercial games, the predominant form of behavior modeling is scripting, which consists of defining the behavior of agents through a particular set of high level functions, implemented in a scripting language. For example, a simple script might instruct an artificial agent to perform the following actions:

1) Build barracks

2) Gather 1000 units of gold

3) Train 10 knights

4) Attack enemy town

[2] Agents controlled by the computer

In a real game the scripts are very complex, but they tend to be static and show little variation. Scripts will most of the times fulfill their purpose of describing simple behaviors, but lack the dynamism needed to express more complex ones. Even with current efforts on generating scripts automatically[27], it seems there is very little room to implement adaptive behavior with this modeling scheme. If adaptation could somehow be implemented in the agent behavior, artificial players might be able to achieve game objectives in a variety of scenarios. Also, they could be able to exploit weaknesses in their opponents. These possibilities would make artificial agents pose a difficult challenge to human players, as well as an engaging experience. Our research focuses precisely in a technique that can exhibit adaptivity when used for behavior modeling.

2.2.2 Other Techniques

Despite the divergence in objectives between games and academy, some games have used AI techniques to solve some of the problems they face. Section 2.4 details some of the main issues related with AI, the techniques used to tackle those problems, and a few of the commercial games that have used them. We encounter games that have been praised for their advanced Artificial Intelligence, such as *The Sims*[76] and *Black and White*[115], as well as less known games that have explored different techniques with some degree of success.

We can categorize most of the issues related to decision making and intelligent behavior of an agent, as shown below. Learning Classifier Systems have been included in the list of techniques even though no commercial games are known to use it, because they are of particular interest in this work. LCS will be described in detail in Section 2.3.

1) **Pathfinding** Although present in a vast amount of games, it has been proven very hard to do efficient pathfinding with the limited computing resources typically available for AI. Consider dozens, possibly hundreds of units needing to do pathfinding at the same time, with complicated and changing terrains. By far, the most common technique used for pathfinding is A^*[41].

2) **Planning** Based on a representation of the game states and knowledge of its actions, agents should be able to define plans that lead them to the desired goals. Some of the techniques used to adress planning are *Finite State Machines*, *Fuzzy State Machines* and *Genetic Algorithms*.

3) **Language interaction** Used when communicating with users through natural language to allow the agents to understand the user's messages. *Natural Language Processing*, a very important research area today, is occupied with these issues.

4) **Spatial decisions** In many games, part of the dynamics require selecting locations for particular events, such as attacking or building. These spatial decisions can make a big difference in the effect of the event, and ultimately in the outcome of the game. Techniques such as *Influence Maps* are used to determine good locations for certain actions, based on heuristic or learned parameters.

5) **General decision making** Other problems fall within a wider category, where decisions must be made given a set of game parameters. Generalized techniques, such as rule-based frameworks, are suitable for a wide array of problems. *Decision Trees*, *Neural Networks*, *Classifier Systems* and *Expert Systems* are examples of such techniques.

Artificial learning can be used across the whole spectrum of problems, as shown by Furnkranz in [36]. One important distinction to make when talking about learning is the difference between what is often called *online* and *offline* learning. Offline learning is restricted to the preliminary phases of the system, before the final users interact with it. The result is typically a set of data or structures that remain static through the actual game[93].

Online learning, on the other hand, continues while the user is playing. Thus, a game can continue to improve based on the interactions with the user. Online learning demands much more resources than its offline counterpart, and so far has been scarcely applied to games[1][110]. The applications of online learning are typically environments with low processing requirements.

In this thesis we concentrate on applying an offline general decision making technique, called Learning Classifier Systems, to a Strategy Game, that represents a very difficult challenge to game designers. In principle, the design of LCS could be applied as online learning, but the analysis of such application falls outside of the scope of this work.

2.2.3 Artificial Intelligence Layers

The Artificial Intelligence engine of a complex game such as a Strategy Game is usually divided in a few parts. Although many different designs are possible, this is one common subdivision:

1) **Task Layer** This includes the most basic commands, such as moving a unit to a particular location, building a structure, or attacking a unit.

2) **Tactical Layer** The tactical layer takes care of higher level decisions, such as what resources to gather, moving a group of units, choosing the best location for a building, or determining unit formations in battle.

3) **Strategy Layer** The strategy layer shapes the behavior of the agent at the highest level, making decisions such as what buildings to create, what type of units to train, and when to attack opponents.

The representation of these layers on a particular implementation might vary. In the case of Wargus, the game selected for this thesis, the two first layers are implemented as part of the engine core. The strategy layer is implemented as Lua scripts[58], using a set of functions that reflect the game logic. Table 2.1 shows some of the functions available to the strategic layer for the Wargus game.

Function	Description
`AiNeed(unit)`	Request training for one unit
`AiResearch(upgrade)`	Request research of an upgrade
`AiDefineForce(units)`	Define a group with a given set of units
`AiAttackWithForce(force)`	Attack an opponent with a group of units

Table 2.1: Scripting functions

The operations in the first two layers can be solved without the need of artificial learning, by using a set of heuristics. Problems such as navigation or unit formations have been solved with widely known space search algorithms such as A*[41]. The scripts that conform the third layer are written by developers, and reflect a deep knowledge of the game, as well as transferred knowledge from similar games and simulations.

In this thesis, we concentrate on the strategy layer, and we use the functions defined in the other two layers for lower level decision making and game actions. Developing strategic-level behaviors would eliminate the need to write and optimize these scripts, and would reduce the need to use game specific knowledge to define the agent behavior.

2.3 Learning Classifier Systems

Learning Classifier Systems are generalized frameworks that belong to the area of Machine Learning, and propose a set of mechanisms general enough for a wide variety of problems. This section explains the structure of an LCS and its main mechanisms, as well as the difference between LCS and XCS. We also highlight some strengths and challenges of XCS as discovered by researchers.

We can describe a Learning Classifier System as a system that interacts with the environment in two ways: by sensing information through its *sensors*, and by taking actions through its *effectors*. The system keeps a population of *classifiers* [P], which represent its knowledge base. This population, also called *ruleset*, is adjusted to model the environment using the feedback received after actions are executed. Figure 2.3 illustrates the main components of an LCS.

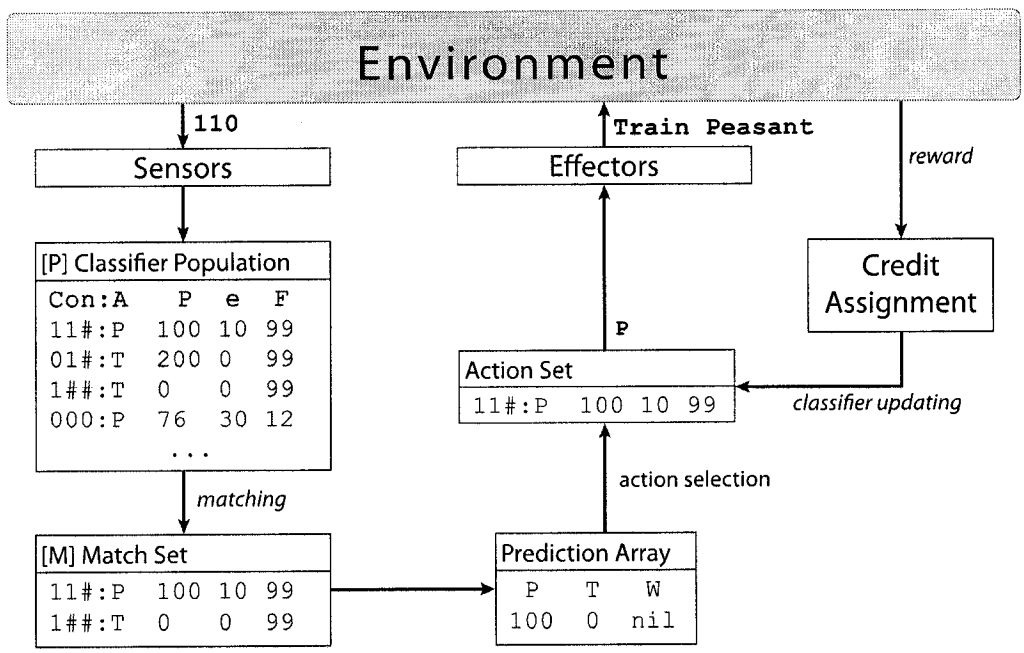


Figure 2.3: Overview of XCS

2.3.1 Structure of a Learning Classifier System

To study LCS in detail, we must look at its parts: its sensors and effectors, the ruleset, the rule selection mechanism, credit assignment and rule discovery. Figure 2.3 shows an overview of the flow of information within LCS.

Sensors Receive signals from the environment, and code information such as image data or game state in different formats. Binary is the most common representation, although various works have used integer and real values[125].

Ruleset Also called a Classifier Population, the ruleset is essentially a collection of Classifiers. They constitute the heart of the decision making engine.

Classifier A classifier expresses a knowledge unit, and it can be read as "when this *condition* is met, performing this *action* is expected to have the following effect (*prediction*)". A classifier has the following components:

1) *Condition*: Typically represented with a ternary alphabet {0,1,#} to describe the values that this classifier will match. The symbol # is a don't-care character that matches both 0 and 1. As a concrete example, the first classifier in Figure 2.3 has a condition string of 11#, which matches 110 and 111, and recommends the action P (Train Peasant).
2) *Action*: One of the set of possible environmental actions this agent can perform.
3) *Prediction*: Feedback expected from the environment after performing the proposed action.
4) *Error*: A measure of how accurate this classifier's prediction has been when activated in the past.
5) *Fitness*: A strength associated with this classifier, which can be calculated in various forms.
6) *Experience*: A record of how many times this rule has been activated.

Match Set When an input is sensed from the environment, the set of values are compared to all the classifiers in the population. Those classifiers whose rules match the input are selected, and will compete for activation. These rules form what is called the **Match Set**.

Prediction Array From the Match Set, a weighted sum is calculated, to obtain the predicted value of all the actions proposed by the classifiers. This weighted sum is reflected in the **Prediction Array**, where each possible action (P, T and W in Figure 2.3) has an expected feedback.

Action Set From the Prediction Array, an action is chosen. If the system is on *exploitation* mode, typically the action with the highest prediction is selected. Otherwise, if the system is in *exploration* mode, a random action might be chosen, or one with low experience, which stimulates the exploration of the search space. Once an action has been selected, the effectors are signaled to act on the environment.

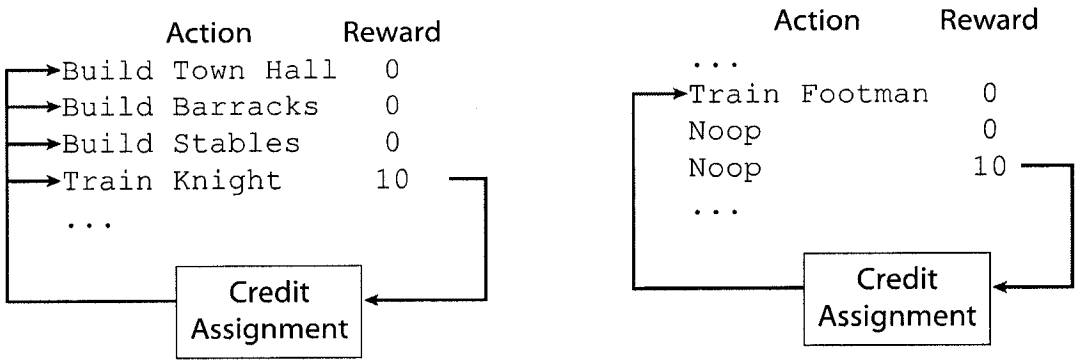
Effectors Generate actions in the environment, such as moving a robotic arm, training a unit, or moving the artificial agent toward a given direction. Actions will typically change the environment, and their effect will be sent to the Credit Assignment mechanism.

Credit Assignment Signals will be received from the environment, which will provide measures of the performance of the agent. These might come every timestep (immediate rewards) or every few timesteps (delayed rewards). This feedback must be used to update the previously activated classifiers, so their prediction, error and fitness are adjusted to reflect the real effect the action had on the environment.

Rule Discovery The system starts with a randomized or empty set of rules. The discovery of new rules is delegated to a Genetic Algorithm, that acts by combining the population of classifiers to create new and choose the best ones.

Two situations are challenging for the credit assignment system: *action chains* and *delayed rewards*. An action chain is a sequence of actions that need to be executed in order to receive a particular feedback. Consider the situation depicted in Figure 2.4(a), where the agent executes three actions that are not rewarded by the environment, but that are necessary to set the conditions for the fourth action. Only after executing *Train Knight* the environment sends a positive reward to the agent, who somehow needs to distribute this reward between the last action and those who worked to set the proper conditions for it.

The second situation where credit assignment becomes non-trivial is with delayed rewards. Figure 2.4(b) shows an example of this case, where the agent executes an action that generates a reward, but such reward is received a few timesteps later. Since



(a) Action chains

(b) Delayed rewards

Figure 2.4: Reward scenarios

it is difficult (if not impossible) to know what actions are responsible for this feedback, the typical solution is to reward a number of past actions, possibly with some discount policy. As a consequence, some actions will be rewarded without deserving it, which can result in the presence of *parasite actions*. The reason for the delay varies with the problem, but this case is very common in real world environments, such as games and simulations.

Well known algorithms such as Bucket Brigade[43] address these problems with a large degree of success in Markovian problems. Non Markovian problems pose a more difficult challenge, and solid solutions to credit assignment in this class of problems are still needed.

2.3.2 Accuracy Based Learning Classifier Systems

Accuracy based Learning Classifier Systems (XCS) were proposed by Wilson[122], and constitute a particular class of LCS. Two main aspects characterize XCS: the calculation of the strength of each classifier and the use of a Niche Genetic Algorithm.

In the original LCS, the strength of a classifier is proportional to the *value* of its prediction. This resulted in populations of classifiers densely concentrated on the high-paying areas of the action set. XCS introduces a new way to calculate the strength, in proportion to the *accuracy* of the prediction. Thus, a classifier will have a high strength, or fitness[3], if it is able to predict accurately the result of the given action, regardless

[3]The terms *strength* and *fitness* are used interchangeably.

of the actual value of the reward received. Results show that XCS is able to create a population of classifiers that occupy the search space more uniformly.

The second characteristic of XCS is the application of the Genetic Algorithm in niches instead of the whole population. This means that when the GA is activated, it does not consider the whole classifier population as his list of individuals, but instead takes only a subset of the population. This subset is determined by the match set [M], and selecting from this set has an important effect: rules that participate in the match set more often (i.e.. are more general) have more opportunity to reproduce. A pressure toward generalization is then introduced, that was not present in LCS. This, in conjunction with the accuracy pressure, generates classifiers that are as general as possible while at the same time as accurate as possible.

In the paper that proposes XCS, Wilson points out that "the results [...] demonstrate that accuracy-based fitness and a niche GA can evolve [...] complete payoff maps containing accurate maximally general classifiers."[122].

2.3.3 XCS Strengths

After studying the basics of XCS, it is important to examine what is their strength in terms of learning, what kind of problems can they be applied to, and what characteristics make a problem hard for XCS. Numerous works in the literature have used XCS to develop knowledge maps that comply with three important measures of success:

1) *Completeness*: a ruleset is said to be complete if the condition-action space is explored in its entirety.

2) *Accuracy*: having the rules describe the condition-action-reward space with a very low error.

3) *Generality*: the proper use of generic characters to compactly represent condition clusters[4]. A maximally general ruleset is one where it is not possible to replace two or more rules with a more general one without sacrificing accuracy.

The ability to generalize is crucial to XCS, because it allows the system to compress its ruleset, and to act more quickly on adjusting the prediction and error values of those rules. Obtaining a maximally general ruleset allows for the maximum compression of the list of rules. Finding complete mappings ensures that the system is able to predict the

[4]A group of conditions that share an action and a reward value.

outcome of an action despite the actual value of the reward, not only concentrate on the high paying portions of the search space.

2.3.4 XCS Challenges

Kovacs and Kerber have studied complexity in XCS[54][55], and have highlighted a few characteristics that make problems hard for XCS. Their work focuses on single step problems, rather than the more general case of multi-step, also known as sequential. Multi-step problems are those that require more than one action from the agent in order to end. For example, maze navigation is typically a sequential problem, because the agent cannot know the result until he reaches the exit (or an artificial stop condition is met, such a maximum number of steps). Sequential problems introduce a few complexities into the learning process that are worth studying. Typically, simulations and games will fall within this class.

Formally studying complexity in multistep problems falls outside of the scope of this work, but we will highlight the results presented in previous works with regards to difficulty in XCS. Following is a list of elements that determine the difficulty of a problem when applying XCS.

1) **Input space** The size of the input space [5] affects directly the learning process. A larger space means a larger amount of rules to discover and adjust.

2) **Action space** In combination with the input space size, the action space determines the amount of rules that will be learned by the system.

3) **Optimal population** Although not obvious in some problems, the optimal population size takes into account the generalizations and compression that can be achieved within the ruleset. If the optimal ruleset contains rules too general or too specific, the system might have difficulty finding the right expressions. There are parameters of the system that have an impact on this issue and can be tuned to improve performance (P#, for example). System parameters are discussed in Section 4.4.

4) **Delayed rewards** An action might have an immediate effect on the environment, or a delayed effect. If the effects are delayed, and the length of the delay is unknown, a problem of credit assignment arises that affects the efficiency of the learning

[5] The space defined by all the input variables

process. Assigning the rewards to the responsible actions is a studied problem that is far from solved, particularly in environments with delayed rewards[11].

5) **Repetitive feedback** An action might generate one effect, or multiple ones. For example, assigning a unit to gather a particular type of resource can result in multiple increases in the amounts of that resource over time. Having repetitive effects can insert noise in the credit assignment system and negatively affect the learning process.

6) **Noise** The environment might contain other forms of noise, which can affect the system in varying degrees. If noise is too high, the real signals are indistinguishable and learning is impossible.

2.4 Related Work

Substantial research efforts have been made on the area of Game Artificial Intelligence, particularly in Machine Learning. Although board games dominated the experimental domain for many years, the attention of the academic community is shifting toward dynamic video games[87]. In this section we present a summary of the research progress in the areas of Game AI, Machine Learning and Adaptivity, with a focus on modern video games.

2.4.1 Artificial Intelligence in Games

Researchers have found in modern games a promising field of experimental platforms, particularly on Strategy Games[15][16]. Games exhibit many open problems that serve as testing ground for different techniques and algorithms[59].

Even with the considerable amount of work that has been done, there are many problems that remain open for practical solutions. Adaptivity is one of them. The challenge of generating behavior that is highly adaptive to different scenarios and opponent strategies is yet to be solved.

Techniques Present in Commercial Games

The majority of commercial games avoid using advanced AI techniques, fearing that they will jeopardize the reliability of their systems. Nonetheless, some games have been able to include AI in their products, as shown in Table 2.2.

Problem	Technique	Examples of usage
Pathfinding	A*	Most games use some variation of A* for pathfinding.
Planning	Fuzzy State Machines	The Sims, Close Combat 2.
	Genetic Algorithms	Creatures series.
Language Interaction	Natural Language Processing	The Chronicles of Jaruu Tenk.
Spatial decisions	Influence maps	Age of Empires.
General decision making	Decision Trees	Chessmaster, Deep Blue.
	Neural Networks	Fields of Battle, Battlecruiser 3000 AD, Black and White, NERO[112].
	Learning Classifier Systems	No commercial games known.
	Expert Systems	Age of Empires.

Table 2.2: AI problems and techniques in games

We can classify the main issues found in Artificial Intelligence applied to games and simulations in five groups: pathfinding, planning, language interaction, spatial decisions and general decision making. Pathfinding is, on the vast majority of the cases, solved using A*[41], although recent proposals offer slightly better performance[14]. The problem of planning is possibly one of the most addressed one, and the most used technique is Finite State Machines[104][38][98], and its related variant Fuzzy State Machines[39]. The area of Language Interaction is largely dominated by Natural Language Processing, and very few games have successfully implemented it to some degree[37][25]. Spatial decisions are critical to a large number of games[35], and influence maps are a common alternative[28].

Some techniques, called General Decision Making for simplicity, can have different applications. Decision trees, Neural Networks and Expert Systems are examples, as well as Learning Classifier Systems. Decision trees[102] are used in a large number of games, including the famous Deep Blue[49].

Many more examples of AI techniques in games, and particularly Machine Learning, are present in research works available for the academic community. Let us review the most important ones.

2.4.2 Machine Learning

A considerable amount of research has been carried out in applying Machine Learning techniques to commercial-level games, as illustrated in Figure 2.5. In this topic, Fumkranz [36] studies problems related to games and connects them with Machine Learning techniques.

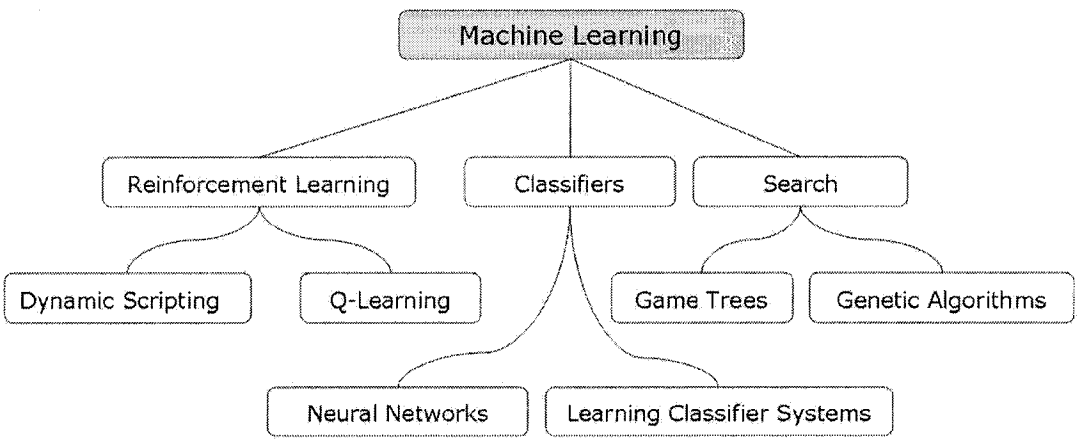


Figure 2.5: Machine Learning Techniques

Dynamic Scripting

On the subject of planning and strategic decisions, Spronck et al. propose a technique called Dynamic Scripting[110], which uses domain knowledge to reduce the search space and improve the performance of the system. In [109], DS is analyzed in light of four functional requirements (clarity, variety, consistency and scalability) and four computational requirements (speed, effectiveness, robustness and efficiency). Ponsen used Dynamic Scripting in [96] to develop game tactics with AKADS (an evolutionary algorithm), achieving positive results. DS was also used in [93] and [94] to develop behaviors for a RTS game by applying other evolutionary learning techniques. Pieter Spronck also

studies the topic of difficulty scaling with DS in [111]. Difficulty scaling refers to the ability to vary the level of hardness of the game AI according to the skills of the player.

Later, Aha et al. applied Dynamic Scripting to larger objectives: winning against non-static opponents[3]. The mentioned work by Aha also uses TIELT[84], a framework developed to integrate simulations with Artificial Intelligence modules.

Q-Learning

Q-Learning has also been explored as a technique for decision making. Madeira et al. have worked extensively on complex games using a Q-Learning technique called SARSA [26] [71] [72][70]. Their approach was tested on a war game system with positive results. Ishibuchi et al. used fuzzy Q-Learning on a multi-player repeated game with a market interaction model, and later applied a similar system to a soccer playing agent[86].

Game Trees

Game Trees are a commonly used technique for game planning and reactive agents. Laursen and Nielsen study small-grain behaviors and its combination with rule-based techniques in [65].

Genetic Algorithms

Genetic Algorithms have been used in [24], [100] and [12] for behavior generation and spatial decisions, whereas [81] and [110] use Influence Maps for a similar purpose.

In [100], Revello et al. have generated strategies for an RTS using a Genetic Algorithm. It is worth noting that their schema was tested on a simplified game, not on a fully sized system.

Louis and Miles have also worked on generating strategies for RTS games using CIGARs (Case Injected Genetic Algorithms) in [82] and [68]. CIGARs[30] are a variation of Genetic Algorithms, where the system is deviated from the *optimal* solutions toward others favored by particular examples or heuristics. The use of human knowledge in combination with the search capabilities of GAs is an interesting line of research.

Miles et al. worked in generating influence maps in [80] and [81], using Genetic Algorithms in combination with other techniques.

Neural Networks

Neural Networks have been used by Miikkulainen et al. in [79] and [112] as the decision-making elements for action games. Karpov et al.[52] use Evolving Neural Networks to develop an agent that navigates spaces in Unreal Tournament[33], a known commercial game.

Fuzzy Systems

Fuzzy systems, initially proposed by Lofti[128], are based in the well known set of theories called Fuzzy Logic[53]. These techniques have been used in different applications, including control systems[119], simulations[77] and many others[2]. There has also been a significant effort in applying fuzzy systems to the area of Games, from behavior design[113] to state evaluation[57] to robot path planning[78].

Learning Classifier Systems

Learning Classifier Systems were proposed in 1986 by Holland [42][44], and quickly became an important area of research. Initially proposed as a general framework within Machine Learning, LCS were composed of a set of rules that represented a knowledge map of a particular problem.

LCS were further refined by other researchers, who created concrete instances and variations of the initial framework. Two important and opposing proposals competed for some time: the Michigan approach[46] and the Pittsburgh approach[106][8][4]. The main difference between the two is that in the Pittsburgh approach, the GA considers the whole ruleset as an individual and keeps a population of rulesets, whereas in the Michigan-style each rule is an individual and the population contains a set of rules. Shortly, the Michigan approach became more widely used, although the Pittsburgh approach continues to be studied[5][6][7][67].

The most important variation of LCS is the Accuracy-based Classifier System (XCS) proposed by Stewart Wilson [122], which became the standard in the research community. The main characteristic of XCS is that the strength of the rules are associated not with the expected feedback, but with the accuracy of the prediction of the feedback that each rule keeps. This modification allowed for a more complete and accurate mapping of the search space. Wilson further refined XCS in [123] and compared it to Genetic Programming. Other authors have studied XCS from a theoretical point of view, such

as Butz[22][20], Lanzi and Wilson[61] and Bernado[73]. Also, Kovacs has studied the domain of problems where XCS has difficulties developing accurate rulesets[54] and [55].

Lanzi has also studied the performance of XCS in [64], proposing an internal memory mechanism. The issue of accuracy was revised by Butz in [21], who details the mechanisms that allow XCS to develop accurate classifiers for particular problems. XCS has also been compared with other Machine Learning techniques with quite favorable results[13].

Many other variations of LCS have been proposed, including:

- **ZCS** Zeroth level Classifier System, a simplification of LCS[121].
- **UCS** sUpervised Classifier System, a supervised version of XCS[91].
- **XCSI** an extension of XCS for Integer valued parameters[125][126].
- **FCS** a classifier system that uses fuzzy-valued parameters[118].
- **ACS** Anticipatory Classifier Systems[114][75].

Anticipatory Classifier Systems were proposed by Stolzmann, and are essentially a variation of LCS that attempt to predict the future state of the system based on the actual state and the action selected. Sigaud and Wilson worked in comparing ZCS, LCS and XCS in [105].

We will concentrate on XCS, because this was the particular implementation selected for this thesis, given its success among different domains and the readily available libraries.

Many problems arose and have been studied within the framework of XCS. One of them is the timing of the rewards that are received from the environment. Having immediate rewards is the simplest case, whereas having feedback that is not present until a few timesteps later (delayed rewards) inserts noise and introduces more complexity in the rewarding schema. The problem of delayed rewards has been found to be quite challenging. Research has found that the credit assignment technique can be inadequate for some problems[11], although positive results have also been reported [62].

2.4.3 Applications of XCS

Accuracy based Classifier Systems have been extensively researched, and applied to a variety of different domains. XCS have been studied in the domain of Data Mining[1], economic simulations[40][127], medical data analysis[47] and personal software agents[129].

There are a few works that have applied XCS to games: Pérez applied FXCS in a co-operative game with positive results[92]. Sato and Inoue used LCS for artificial learning in an online soccer game [51][103]. Also, Robert et al. developed an action selector for a MMORPG[101], with a very limited number of sensors and effectors. Ravichandran et al. applied XCS for Automatic Target Recognition, a task that is of interest for both military simulations and games[99].

2.4.4 Further Information on XCS

If the reader is interested in resources for XCS research, it is recommended to visit the following works. Wilson gives a thorough review of XCS in [124]. Butz offers a step by step description of XCS in [23]. Also, [63] has very complete surveys of Classifier Systems. For a view of LCS by some of the most important researchers, see [45]. In terms of concrete implementations, Butz provides a Java implementation of XCS in [19], while Lanzi developed an XCS library in C++[60]. Finally, [56] offers a number of resources on LCS, most of them still relevant today.

2.4.5 Adaptivity

Adaptivity has been studied in different areas, and two works are remarkably important: Spronck[108] and Manslow[74]. The first dissects the subject of adaptation from a practical point of view, and the second studies adaptive game AI extensively and applies his approach to two different games.

Chapter 3

Proposed Approach: Classifier Systems in Strategy Games

This chapter will define the research problem, describe the proposed solution and some implementation details, such as data mapping and system constraints. The contributions of this work are also emphasized.

3.1 Problem Statement

An important part of defining a game environment is the modeling of agent behaviors. A common solution to this problem is to invest time in writing scripts tailored to each scenario, map or level. Another alternative is to use artificial learning to let the agent develop effective behaviors by itself, allowing it to interact with the environment. In this thesis, we concentrated on the problem of automatically developing *strategies* by making use of Unsupervised Learning on a Strategy Game. The strategic level of the agent takes care of gathering information from the game state and making high level decisions that contribute to achieving his main goals in a given scenario. We applied an Accuracy based Classifier System (XCS) as the strategic AI module of the game, and we explored two instances of the problem: immediate rewards and delayed rewards. The XCS was implemented as an offline learning technique[1].

[1]See 2.2.2 for more details on online and offline learning, and 2.3 for information on Classifier Systems.

3.1.1 Objectives

The main objectives in this thesis are the following:

1) Apply XCS as the decision-making module of an agent in an RTS game.

2) Observe and analyze the performance of the XCS in the immediate and delayed rewards cases.

3) Compare the performance of the XCS with the script based solutions.

3.2 Proposed Solution

The implementation phase of this thesis consisted on modifying the Artificial Intelligence module of the Wargus game, to include artificial learning as part of the agent modeling. In Wargus, the high level behaviour of the agents (strategies) is expressed using scripts, with a set of functions defined in the scripting language Lua[58]. These scripts were replaced by an XCS module, which was trained in a series of maps allowing it to learn different strategies, depending on the objective of the training scenario.

The implementation work focused on the following tasks:

1) **Customize XCSLib:** Implement parameter handling on the XCS system according to the characteristics of the problem.

2) **Integrate XCSLib and Wargus:** Define inputs and outputs of XCS, develop communication details and implement parameter mapping.

3) **Define training goals:** Determine a group of different objectives, maps and environment variables.

4) **Learning:** Train the XCS-based agents in the different objectives, monitor and log results.

5) **Evaluate:** Analyze and interpret the results of the training sessions.

In the following subsections we will describe the overall architecture of the system and the input and output mapping of the XCS.

3.2.1 Architecture

The two main components of the system are Wargus and XCSLib, as described on 2.1.3 and later on this section. The classes implemented to connect them, called *middleware* for simplicity, take care of the flow and formatting of information between them. Figure 3.1 shows a high level view of the components and their interaction.

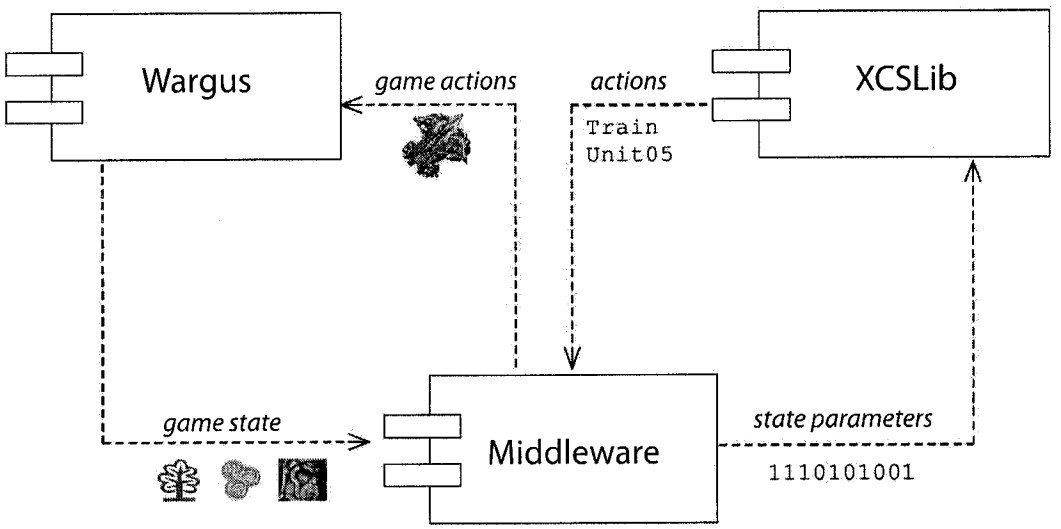


Figure 3.1: High level architecture of the proposed system

The middleware receives information about the game state and converts it into parameters usable for the XCS. This inputs are converted to a binary representation and passed to the XCS module. Once the XCS is ready to request a game action, it does so through the middleware again, who invoques the corresponding function in the game. This architecture allows the XCSLib to work independently of the implementation of the environment classes, which are customized to each problem.

Other responsibilities of the middleware are: saving and recovering the environment state, keeping track of the problem state through a number of steps, and starting an instance of the Wargus game when needed. The middleware also keeps a table of all the possible actions availabe on the environment.

3.2.2 Implementation

The library XCSLib includes two environments (woods and parity), which serve as testing examples. Implementing new environments is relatively easy, and requires the creation

of two classes. In this work they are called wargus-env and wargus-action. Wargus-env defines the global environment parameters, reads the environment configuration from an text file, and manages other problem-level tasks such as setting up a problem and cleaning up after a problem is finished. Wargus-action defines the specific actions present in the game, as well as methods for sending these actions to the wargus game.

Implementing the communication between XCSLib and Wargus required the implementation of a proxy agent within the Wargus package [2]. An artificial agent in Wargus is implemented by writing a LUA script[58], which has access to a number of game-specific functions. Our implementation includes an artificial agent called xcs-agent, that calls back on the XCSLib when activated, to request the XCS system for the next action to activate in the environment.

The flow of the system is as follows. First of all, the Wargus main class is executed with a special parameter that indicates that the game will run in learning mode. Then, a modified version of the initialization phase of the game is called, which takes care of setting up the game and XCSLib. Once the XCS classes and the game classes are initialized and ready to run, the control is passed to the XCS main branch. XCS then initiates the learning process, starting a new game everytime there is the need to evaluate a ruleset.

A special Lua script was implemented, called `xcs.lua`, which is used for the artificial agent. Figure 3.2 shows the pseudocode of the main method of `xcs.lua`. Following is a brief explanation of this code.

First, the possible stop conditions are checked: victory and timeout. Scenarios have particular goals, and once those goals are reached, the victory flag is set to true. The timeout was set for practical purposes, so that the agent stops after some time even if the goal is not met. If any of these two conditions is true, the game will be stopped and the system will continue with the next evaluation. If the game is not stopped, the current score of the agent is calculated according to the particular scoring function of the scenario. The reward is set based on the difference of the current score with the score calculated in the previous step.

Next, the environment parameters are collected and sent to the XCS using the `xcs_step` call. The middleware formats these parameters and concatenates them to form an input string. After the XCS module returns, the resulting action is executed and the game continues.

[2] An agent that contains no game logic, only the necessary code to communicate to the XCSLib.

```
Agent AI step()
{
    // verify stop conditions
    check_for_victory()
    check_for_timeout()

    // Calculate reward for last step
    calculateScore()
    setReward()

    // Get game state and run xcs
    env = getEnvironmentParameters(this_player)
    local action = xcs_step(this_player, env)

    // Execute the action provided by the xcs module
    execute_action(action)
}
```

Figure 3.2: Agent AI Step

Figure 3.3 shows the pseudocode of the main branch of the XCS module. This is invoked by the agent, as shown in Figure 3.2. A brief review of this code follows.

The XCS step begins by obtaining the inputs from the environment (see the code block marked as (1) in Figure 3.3). This information has already been formatted into an input string by the middleware, and it is ready to use. Then, a matching process begins, which compares every classifier in the population to the input string. All the classifiers whose condition match the input string are selected and added to a *prediction array.* These are the candidate classifiers that might be used to select an action.

The prediction array is then scanned to select an action, according to the current selection policy. The selection policy can be either *exploration* or *exploitation.* In exploration mode, the actions are selected randomly, providing all rulesets equal opportunities to participate. In exploitation mode, actions are selected based on their prediction — the action with the highest prediction value on the prediction array is selected. Tipically,

```
XCS:step()
{
    // (1) Get current input from environment
    // and obtain matching classifiers
    current_input = Environment->state();
    match(current_input);
    build_prediction_array();

    // (2) Select action according to selection
    // policy: exploitation or exploration
    action = select_action();
    build_action_set(action);

    // (3) Send the action to the environment for execution
    Environment->perform(action);

    // (4) Get the reward from the environment
    reward = Environment->reward();

    // (5) update the fitness and prediction of the action set
    update_previous_action_set(reward);

    // (6) If it is time to run the GA, do it
    if( need_ga() )
        run_genetic_algorithm();
}
```

Figure 3.3: XCS Step

the system will alternate between these two modes, allowing the ruleset to learn when *exploring* and evaluating its performance when it is in *exploitation* mode. This makes it possible to have a trace of the system's overall performance as it advances in the learning process.

Once an action `A` is selected, all the classifiers from the *prediction array* that proposed `A` are added to the *action set* (2). This action set will later be updated with the environment feedback. The action is then sent to the environment for execution (3).

After the action has been executed, a reward is obtained (4). This reward is used to adjust the classifiers that were present in the previous *action set*, with the objective that their prediction reflects the result of this activation (5). Their error, experience and fitness fields are also updated.

Every few steps, the Genetic Algorithm is invoked (6), which creates new rules as the result of combining existing ones. These new rules enter the population to replace the less ones with the lowest fitness, and will participate in the learning process in future steps.

The performance of the agent is monitored by the XCS system continuously. To do so, evaluations alternate between exploration mode and exploitation mode. The environment feedback on each of these exploitation mode evaluations is logged for later analysis. Chapter 4 explains the experiments and results in detail.

3.2.3 Updating Classifiers

The process of updating the classifiers follows the method described by Wilson in[122]. We will describe this process briefly, which is central to developing accurate classifiers.

The reinforcement component is responsible for adjusting the Fitness, Prediction and Error of classifiers according to their performance in the environment. For that purpose, at each step of the XCS, a reward is obtained and compared to the prediction of the last action set active. The classifiers are then updated, according to the following steps:

1) An auxiliary value **P** is calculated as follows:

$$P = previous\_reward + discount\_factor * max\_prediction \tag{3.1}$$

where previous_reward refers to the reward obtained from the environment, the discount_factor is a parameter of the system with values between 0 and 1, and max_prediction refers to the maximum value of prediction present in the classifiers

from the last action set. **P** is used in the calculation of the prediction, error and fitness (see following steps).

2) The **Prediction** of each classifier is then updated using the Widrow-Hoff delta rule[120]:

$$prediction+ = learning\_rate * (P - prediction) \tag{3.2}$$

where learning_rate is a parameter of the system with values between 0 and 1.

3) The **Error** is updated in a similar way:

$$error+ = learning\_rate * (|P - prediction| - error) \tag{3.3}$$

4) The **Fitness** is updated in proportion to the accuracy of the classifiers, and relative to the accuracies of other classifiers in the action set. For that purpose, the relative accuracy **ra** is calculated first:

$$ra = alpha * (\frac{error}{epsilon\_zero})^{-vi} \tag{3.4}$$

and then the fitness is adjusted as:

$$fitness+ = learning\_rate * (\frac{ra}{accuracy\_sum} - fitness) \tag{3.5}$$

where alpha and epsilon_zero are also system parameters.

The most important parameters of the system are shown in Table 3.1. These values were selected following previous research works, as described in [23] and [122]. Appendix A contains a full list of the system parameters.

Parameter	Value
alpha (α)	0.1
learning_rate (β)	0.2
epsilon_zero(ϵ_0)	2
discount_factor (γ)	0.71

Table 3.1: XCS Parameters

3.3 Wargus

For this thesis, we selected the game Wargus[117] as the test platform, because it meets our requirements: an open source Strategy Game, with commercial-level complexity. An open source project allows us to examine and modify the totality of the source code, giving us the flexibility we need. Wargus is in fact a clone of the popular commercial game Warcraft II[31], and is developed over a game engine called Stratagus[97][95]. Wargus uses the same graphic and audio resources than Warcraft II, which requires that the user have access to a copy of the original game, but since it runs in the Stratagus engine, the entire game is accessible for developers to examine and modify.

Wargus has been used previously in research projects, such as [85], [93] and [94]. The game was described in detail in 2.1.3.

3.4 XCS Library

There are a few XCS implementations available, and we had to evaluate the options to determine which one to use in this thesis. The first criteria was the programming language, and C++ was chosen for two reasons: speed, and interoperability with the game of Wargus. Thus, a few libraries were ruled out, such as XCSJava[19], XCSFJava[18] and JXCS[9], all implemented in Java. The latter also has the strong limitation of working only with single step problems.

The two main candidates left were XCSLlib[60] and XCSC[10]. Of these two, the first one is more easily used as a library, instead of a standalone application. XCSLib, developed and maintained by Pier Luca Lanzi, has been used in past research works, and it has shown positive results on the well known *Multiplexer* and *Woods* environments. Both of these environments are described in detail in [122] and represent common test problems for classifier systems. XCSLib proved to be sufficient to the needs of this work, although some effort was needed to completely integrate it with the game.

3.4.1 Parameter Mapping

One crucial design issue when implementing the system was the parameter mapping between the game and XCS. The middleware had to be designed with the flexibility of modifying this mapping without the need to rebuild the whole system. Moreover, the definition of the inputs and actions visible to XCS determine the search space. It is

necessary to reduce the search space as much as possible without losing much accuracy, i.e., assuring that the learned strategies are suitable for a complex environment. The following two subsections will describe the input and action mapping of the system. These definitions could be expanded in future works, to further examine the capabilities of the system in more complex instances of the problem.

3.4.2 Inputs

The XCS system receives a number of inputs from the environment, as shown in Table 3.2. These are in fact a subset of all the inputs that can be extracted from a Wargus environment. They were selected to be representative of the game state, while at the same time reducing the search space.

Input	Range	Step	Bits
Gold	0-4000	500	3
Wood	0-4000	500	3
Archers	0-24	3	3
Ballistas	0-24	3	3
Footmen	0-24	3	3
Knights	0-24	3	3
Peasants	0-24	3	3
Towers	0-24	3	3
Barracks	0-1	1	1
Blacksmith	0-1	1	1
Lumber Mill	0-1	1	1
Stables	0-1	1	1
Town Hall	0-1	1	1

Table 3.2: System Inputs

There are three types of inputs: resources, troops and buildings. To further allow the agent to abstract from the large search space, some inputs were normalized to a set of values depending on the logical ranges of each one. For example, the resource *gold* is expected to be on the range [0,4000], and the values are normalized in steps of 500. Thus, the agent will sense this parameter as having values from 0 to 7. Other inputs have binary values, so normalizing is not necessary.

Furthermore, the agent receives information about these inputs from itself and the opponent. The resulting input strings have a length of 29 bits for each player. Through randomization of the initial conditions of each problem, the space was explored thoroughly.

Inputs are obtained from the game using a set of functions available in the scripting language provided by Wargus. Table 3.3 shows some examples of the functions used to obtain data from the game[3].

Function	**Returns**
GetPlayerData(p, 'Resources', 'gold')	Amount of gold currently owned by player p
GetPlayerData(p, 'UnitTypesCount', 'footman')	Number of footmen trained by player p
GetPlayerData(p, 'Upgrade', 'upgrade-arrow1')	Whether or not player p has researched the arrow technology

Table 3.3: Examples of Wargus Functions

3.4.3 Actions

At each step, the XCS senses the environment and makes a decision on what action to take. Table 3.4 shows the actions available to the system. In the same way that the inputs are limited, the system actions are also a subset of all of the available in the game of Wargus. An agent in this environment can *train* 5 types of units, *build* 6 types of structures and *attack* in 15 different ways. The attack modes are a simplification of the attack possibilities in the game. An agent can choose to attack with groups of a specific type (archer, ballista, etc), in which case all the units will be of the same type. It can also request an attack with a *combined* army, composed of units of all the types available to that player. In addition, it can choose three different policies regarding the amount of units: *light*, *medium* or *heavy* army. This parameter will determine the number of units used to attack. A light attack will send 25% of the units, a medium one 50%, and a heavy one will send all the available units of the type requested. This abstraction should provide enough flexibility for the agent, while drastically reducing the search space. The resulting action set contains 26 possible actions.

[3]A complete reference of the available functions is included with the Wargus package[117]

Train	Build	Attack	
Peasant	Barracks	Archer	Light
Archer	Blacksmith	Ballista	Medium
Ballista	Lumber Mill	Footmen	Heavy
Footman	Stables	Knight	
Knight	Tower	Combined	
	Town Hall		

Table 3.4: Actions

We could compare this search space with the one present in a multiplexer problem[122]. The 37 multiplexer has an input of size 37, and a binary action (a set of size 2). The search space size is, then, $2^{37} \times 2 = 2^{38}$. The next larger size of multiplexer is 70, and its search space has size 2^{71}. The Wargus learning problem has input strings of size 58 and 26 different actions. Thus, the search space is roughly $2^{58} \times 26 \approx 2^{63}$. It seems to be located, in terms of search space, between the two sizes of multiplexer problems.

	Multiplexer37	Wargus	Multiplexer70
Search space size	2^{38}	$\approx 2^{63}$	2^{71}

Table 3.5: Comparison of Wargus problem with Multiplexer

The multiplexer is a widely known problem, and is typically used as a benchmark for Machine Learning[122]. In this case, our problem seems to present a search space size comparable to instances of the problem that are solvable by XCSLib within a reasonable time. In our preliminary tests, XCSLib solves multiplexer 70 in approximately 20 minutes in a single processor linux machine with 1GB RAM. In our implementation, learning times were within this range, and confirmed this estimation, although running the game posed an important overhead in the overall system.

3.5 Constraints and Limitations

The components selected for this work imposed a few constraints on the design of the final system. First of all, both Wargus and XCSLib are implemented in C++, so the obvious choice for the middleware was to use the same programming language. Also,

the middleware classes were designed to work with the available APIs offered by the components.

From the point of view of the problem to solve, there was an important consideration that affected the design of the experiments. When evaluating a ruleset, it is selected to play a game of Wargus. So, how is a ruleset evaluated quantitatively? The simplest way would be to look at the result of a game, and the victory or defeat (and possibly a score calculation at the end of the game) would give a fair value to the actions of the agent. This would result in a problem with long delayed rewards. Since part of the objective of this work was to compare the performance of the system in both immediate and delayed rewards environments, the scoring scheme had to be designed accordingly. Instead of waiting for the end of the game, a scoring function was defined for each map, that could be invoked at every step. This function defines the objective of the map from the point of view of the individuals. Each scenario, then, has its own evaluation function that allows the system to receive feedback every timestep. Whether an action has an immediate effect or a delayed one was something to manipulate, with the guarantee that the scoring scheme would not pose a problem. This also facilitated the definition of maps with simple objectives, rather than the more general and difficult goal of defeating an opponent in a full sized map.

There was an important limitation found when working with the Wargus application. Completely disabling graphics in the game proved to be a challenge, and because of time constraints it could not be done. This was an essential condition for parallelling the execution of the games, which was a possibility given the platform available at Paradise Laboratory, where this work was developed[90]. Thus, the experiments were ran on a single CPU, which was not a major problem given the experiments as designed. More complex learning objectives could have required more computation power, and paralelization would be a possible direction for future work. A brief proposal for paralellization is presented in Section 5.2.

3.6 Contributions

The current work contributed to the areas of Artificial Intelligence and Computer Games in several ways. In this work we explored a very difficult learning game environment: a Strategy Game, with a level of complexity similar to commercial games. We reported on the results of this application, which indicate that is it possible to apply this technique or similar ones to future game projects.

The performance of the system was studied in two variations of the problem: immediate and delayed rewards. Each case poses different challenges, and the result of the experiments help identify what aspects of the problem are the most difficult.

The most specific aspect observed on this research was the adaptability of the developed rulesets. The results confirm there is a degree of adaptability that can be reached by using the adequate techniques. Adaptability is one of the important facets of intelligent behaviour, and games can benefit from incorporating more advanced adaptive capabilities in their products.

Finally, we report on the challenges faced by Machine Learning when applied to a problem such as a Strategy Game. We believe this study is of great value for both the research and industry communities.

Chapter 4

Experiments and Results

This chapter describes the experiments performed with the system, the different reward policies (immediate and delayed rewards) and the parameters used for the runs. We will also show the results of the learning experiments, and analyze them in light of the objectives of this work. This will include a quantitative analysis that will examine the performance of the agent in the training environment, as well as a qualitative examination that will present some of the rules obtained in the evolved rulesets. Finally, the results of overhead measures are presented.

4.1 Design of Experiments

For the experimentation phase we designed four training scenarios with different levels of difficulty. Each one corresponded to a particular goal, a map, and a scoring function that reflected the learning goal. Each game had a maximum running time, large enough to allow the agent to execute a considerable amount of actions on the environment. For each scenario, a random player was also implemented, allowing us to compare its performance with the trained agents.

4.1.1 Learning Scenarios

The following are the four scenarios designed for this phase:

1) **Peasants** The agent was rewarded for training peasants. This is the easiest ruleset to learn, and only two actions are relevant: building a town hall and training peasant units.

2) **Footmen** The agent was expected to train an army of footmen, for which he needed to build a town hall and barracks (if these were not present in the initial map). The set of rules needed to accomplish this objective is slightly more complicated than the first scenario.

3) **Archers** The agent was rewarded for training an army of archers, and three different buildings were relevant for this task, which makes this objective harder than the previous. There is also an extra reward for a building that is not necessary but useful (Lumber Mill).

4) **Cbuilding** The agent was rewarded for different units, all of which are effective against enemy buildings.

The scenarios were selected to have different levels of complexity. As Table 4.1 shows, the optimal rulesets have an increasing number of relevant rules. Moreover, the condition part of each set of rules is more complicated, with different levels of generalization. The Genetic Algorithm has to discover more rules, with varying levels of generality, which makes the learning process more difficult.

Scenario	Number of Relevant Rules
Peasants	2
Footmen	3
Archers	4
Cbuilding	10

Table 4.1: Relevant rules in each scenario

4.1.2 Scoring Functions

Table 4.2 shows the score calculation functions used for each training scenario. The score functions define the objective that the agent will pursue; the rulesets are adjusted to reflect the rewards defined there. The design of these scoring functions is such that if the actions are immediate (meaning their effect is visible on the environment on the next step), the game becomes an *immediate reward* problem. This is a consequence of rewarding every necessary building block, such as creating the needed structures and training the units.

Scenario	Score
Peasants	has.town.hall + count.peasants
Footmen	has.town.hall + has.barracks + count.footmen
Archers	has.town.hall + has.barracks + has.lumber.mill + 2*archer.count
Cbuilding	has.town.hall + has.barracks + has.lumber.mill + has.blacksmith + has.stables + 3*count.ballistas + count.footmen + count.knights

Table 4.2: Score calculation

Once the scoring functions have been defined, determining the reward in time t is merely a matter of calculating the $\Delta score$:

$$reward = score_t - score_{t-1} \tag{4.1}$$

4.1.3 Maps

The game map selected was the default Wargus map (`dragon.smp`), which is of medium size according to the game standards. It is big enough to allow enough space for the agents to develop without being too limited. A much bigger map would mean slower runs.

The resources, buildings and units given to the agent at the beginning of each game are randomized. This forces the XCS to explore the whole search space, so that the rulesets do not over-learn a particular subset of the problem conditions.

4.2 Reward Policies

As explained in 2.3.4, two important issues with credit assignment in multistep problems are *delayed rewards* and *action chains*. In this work, the experiments were designed to test the capabilities of XCS in both situations. Two sets of experiments were executed:

1) **Immediate rewards:** the game had to be modified, such that the game state reflected the changes caused by the agent actions as soon as these were executed. For example, when a player requests the construction of a particular building, the following steps are executed:

   - Check that the preconditions are met (resources available, selected location adequate, etc.).
   - Move a peasant to the chosen location.
   - Start building, show animation. This step lasts for a few timesteps.
   - Building is ready. Only then is the game state updated to reflect the new building.

   These steps needed to be modified, so that the game state was updated as soon as the action was requested. Thus, the preconditions are checked and the game state is updated, and then the rest of the steps are followed. This allowed the agent to sense the result of his actions immediately, changing the game into an immediate reward problem.

   It is important to note that not all actions are so easily modified. In the case of attacks, it is more difficult to know the outcome of an encounter before the actual battle takes place. For these experiments, an estimation function was used, that predicted the result of a battle based on the amount and strength of the units (See 4.2.1). Based on a number of tests, the function was adjusted to be conservative and relatively accurate.

2) **Delayed rewards:** Since the game is inherently a problem of delayed rewards, no special modifications needed to be done for this phase. Delays typically varied between 1 and 15 steps, according to preliminary experiments.

To study the impact of action chains, the experiments were done adding rewards to all the actions on the chain. This facilitated the learning process, and did not represent a problem with the goals set in the learning objectives.

4.2.1 Battle Prediction

Estimating the outcome of a battle with high accuracy is a difficult task, since many variables and random events affect the process. Nonetheless, it was necessary for the experiments to predict the result of battles based on the information available. Figure 4.1 shows the pseudocode of the resulting function, which is given in Appendix C in full detail. Essentially, each player's army is assessed, and a total strength is calculated based on values per unit, plus modifiers for particular game upgrades. For example, a footman unit has a strength of 200, and the weapon upgrade adds to it, to a maximum of 240. The MIN_ARMY_FACTOR parameter indicates how much stronger must the player army be to be safe when attacking; a higher value means a more conservative prediction. The values of strength per unit and the MIN_ARMY_FACTOR were developed empirically.

4.3 Overhead Study

A series of experiments were carried out to measure the overhead of the XCS module on the game performance. The measuring consisted on running each of the four learning scenarios already described, and log two values: the duration of a game AI cycle and the time spent in an XCS call. An AI cycle is considered to start and end at the end of the script entry point. An XCS call is measured including the parameter gathering, the actual XCS step, and the execution of the returned action. Appendix D shows a segment of the code used to log these values.

4.4 Parameters

Table 4.3 shows the most important parameters used in the experiments. Some training sessions used slightly modified values for some of the parameters (such as population size and number of generations) but most were used as indicated. All the parameters were based on values found in the literature, mainly guided by [23]. Some had to be adjusted experimentally to improve the performance of the system. For a detailed parameter sheet, the reader is invited to examine Appendix A.

```
local MIN_ARMY_FACTOR = 1.1

unit_strength["footman"]      = 200
unit_strength["archer"]       = 100
unit_strength["ballista"]     = 450
unit_strength["knight"]       = 400
unit_strength["guard-tower"]  = 500
unit_strength["cannon-tower"] = 500

// Returns true if this player is expected to win the battle,
// false otherwise
function predict_battle(army)
  player_army_strength = 0

  // Calculate the strength of this player's attacking army
  // extra damage is calculated based on the technology upgrades
  // achieved by the player
  for each unit in army
      player_army_strength += ( unit.count * unit_strength[unit]
                              + extra_damage(unit) )
  end

  // Calculate the strength of the opponent's army
  for each unit in opponent.army
      op_army_strength += ( unit.count * unit_strength[unit]
                          + extra_damage(unit) )
  end

  // True if strength is at least MIN_ARMY_FACTOR times
  // the opponent's strength
  return (player_army_strength / op_army_strength) > MIN_ARMY_FACTOR
end
```

Figure 4.1: Battle Prediction

Parameter	Value
Population size	300
Problems	50-200
Crossover method	2-point crossover
Crossover probability	0.8
Mutation probability	0.04
GA Selection policy	Tournament
Don't care probability (P#)	0.75
Learning rate	0.2
θ GA	5

Table 4.3: XCS Parameters

The amount of problems varied for the different training scenarios as shown in Table 4.4. Each problem corresponded to one Wargus game. Each game is a multistep problem, and required an average of 50 steps to solve. Thus, training sessions consisted of runs of 2500 to 10000 steps.

Scenario	Problems	Steps
Peasants	50	2500
Footmen	100	5000
Archers	100	5000
Cbuilding	200	10000

Table 4.4: Number of Problems and Steps per Scenario

It is relevant to point out the reason behind the high value of the *don't care probability* parameter. Other works have used lower values, but the selection was made to circumvent the potential *covering challenge*, as presented by Butz et al.[21]. In short, the covering challenge states that a low value of generalization in the initial population can generate a cycle of covering and deletion of overspecific classifiers [1]. The size of the input strings heavily affects the possibility of entering this cycle, and choosing a P# high enough prevents the system from degrading in such way.

[1]Classifiers where the condition part contains too many *dont-care* characters. These classifiers are rarely activated in the systems and thus have little opportunity to increase their fitness.

The parameter θ GA indicates how many steps must pass before the Genetic Algorithm acts on the population. Each θ GA steps, the Genetic Algorithm takes one iteration. This parameter was set experimentally.

4.5 Results

This section presents the results of the experiments, in both the immediate and delayed rewards cases.

4.5.1 Immediate vs. Delayed Rewards

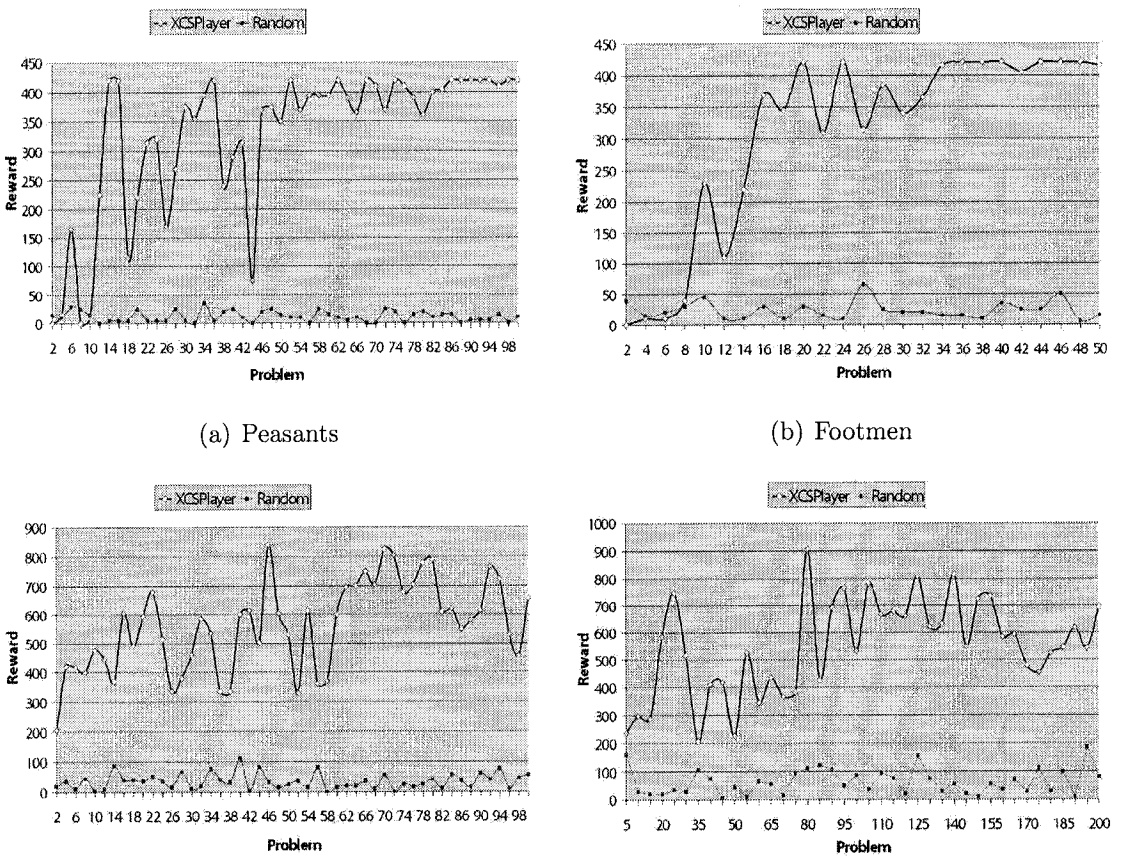
As explained previously, the experiments were ran in two modes: immediate rewards and delayed rewards. The delayed rewards mode poses an important challenge for the type of problem being addressed in this work.

The experiments carried out with delayed rewards were not successful in exhibiting learning. Adjusting the two main parameters of the credit assignment mechanism (window size and discount rate) did not have any positive impact on the results. For time constraints, it was decided to leave this opportunity open, and concentrate on the immediate reward results, which were showing success in learning the scenarios presented. Thus, the rest of the results presented in this chapter will refer only to the immediate reward mode.

4.5.2 Learning Results

As presented in 4.1.1, the experiments corresponded to four learning scenarios, which we will call **peasants**, **footmen**, **archers** and **cbuilding**. To measure the performance of the agents in each scenario, the XCS runs a number of problems (between 50 and 200), where each step alternates between learning mode and testing mode. In the learning mode (also called exploration mode), an action is usually selected randomly between the matching rulesets. In testing mode (also known as exploitation), the matching rulesets are examined, and the classifier with the highest prediction is selected. Thus, the testing mode gives a measure of the progress of the agent. As the rulesets evolve, if learning is indeed taking place, the results of these testing-mode actions should improve.

Figures 4.2(a) to 4.2(d) show the results of the testing mode evaluations through time. The x axis represents the problems ran, where each problem is one Wargus game. Each



(a) Peasants

(b) Footmen

(c) Archers

(d) Counter-building

Figure 4.2: Learning results

game is a multistep problem, consisting of 50 steps each on average. Thus, 100 problems will represent an average of 5000 learning steps. On the y coordinate, the graph shows the reward in learning mode for each problem. These are shown as a moving average of the last 2 problems (5 in the case of **cbuilding**) The graphics also include a second curve, representing the performance of a random player. This is an agent that performs random actions at every step, and is shown here just as a means of comparison with the XCS-based agent.

4.5.3 Quantitative Analysis: Performance

All resulting charts show a consistent tendency to improve the agent's rewards through time. Also, it is quite clear that their performance rapidly differentiates from the random

player, which is a first indicative of some degree of learning. The performance of random agents consistently oscillate between 0 and 50 in figures 4.2(a) and 4.2(b), and between 0 and 100 in 4.2(c) and 4.2(d), while the XCS agents reach rewards of 5 to 10 times those values.

All charts are drawn on a scale that reflects the optimal performance of each scenario. In the **peasants** and **footmen** scenarios, optimal scripts with ideal initial conditions will achieve average rewards of 42. Since initial conditions are randomized, the optimal performance in each problem will oscillate, with a variation of 10%, according to preliminary experiments. In **archers** and **cbuilding**, the optimal rewards will vary between 800 and 1000, depending on the initial conditions. We can see in the charts that the XCS based agents obtain rewards very close to optimal values, which indicates that the learning goals have been clearly achieved.

We can also observe in the charts that the rewards stabilize to a great deal in **peasants** and **footmen**, while the other two still exhibit some degree of variation even in the last problems of the learning phase. This is mostly due to the randomization of the initial conditions. Resources, available buildings and units, are all set randomly at the beginning of each problem. In some cases the agents will need to gather different amounts of resources and create buildings in order to start executing actions that provide some reward. Thus, the total score obtained at the end of each game will vary, even for optimal scripts. This largely explains the observed variation in the rewards. It is also likely that the rulesets reach a near-optimal behavior, and thus achieve a performance slightly lower than an optimal script. This in itself is not a problem, because the objective of the learning is not to perform better than the scripts, but rather be able to perform consistently in a wide variety of conditions. The results do show a very good degree of consistency in achieving high rewards on the goals set in each scenario.

4.5.4 Qualitative Analysis: Rulesets

Studying the rulesets obtained on a Learning Classifier System is usually a difficult task, specially for large rulesets with many parameters. In our case, the resulting populations had 170 to 250 classifiers. Interpreting these sets of rules is not a trivial task. Here, we present a small subset of the rules evolved in two scenarios: **footmen** and **archers**. The rules shown represent the fittest rules, which are also the most used in the training experiments.

Figure 4.3 shows the two most important rules in the **footmen** scenario[2]. The first one takes care of the most important task of the problem: training the footmen. This is the action that is more largely rewarded by the scoring function. We can read this classifier as saying: "if there is a certain amount of gold, and barracks are available, train a footman". This is the simplest rationale that can be developed for this goal, and the amount of gold checked for corresponds with the resources needed to execute the training action.

The second rule is more difficult to interpret. Given the nature of the first rule, and others present in the population, this rule will only be selected if the gold parameter has a value of 001, which is not enough to execute the action of training a footman. Thus, the XCS chooses, through this classifier, to carry a No-Operation action, which in practice will do nothing but wait until the next step. The effect of this combination of rules, is that when the agent has too little gold, it will wait until its peasants gather enough of it. By then, the Gold parameter will change and the first rule will be selected. Other rules in this ruleset take care of building the necessary structures (Town Hall and Barracks) if they are not present.

```
Gol Woo Arc Bal Ftm Kni Psn Tow Br Bsm LuMi St TwnH Action
#1# ### ### ### ### 0## ### ### 1   #   #   #   #   TRAIN_FOOTMAN
##1 ##1 #1# 0## ### #0# ### 1## #   #   #   #   #   NO_OPERATION
```

Figure 4.3: Fittest and most used rules in the footmen scenario

In the case of the **archers** scenario, the top rules developed show a very similar pattern. As shown in Figure 4.4, the main action of this scenario is to train archers, which is taken care of by a set of 9 rules that complement each other to cover most of the cases where that action is feasible. Other rules in this scenario take care of building the needed structures, such as the Lumber Mill, as shown in the second group of rules in Figure 4.4.

The **peasants** and **cbuilding** scenarios show behaviors very similar to the ones just presented in terms of rule-action pairs. The agents quickly discover the actions that generate rewards, generalized to an almost optimal point, in some cases relying on the

[2]The parameters shown in the figures have been abbreviated for presentation purposes, and correspond to: Gold, Wood, Archers, Ballistas, Footmen, Knights, Peasants, Towers, Barracks, Blacksmith, Lumber Mill, Stables and Town Hall respectively. The rest of the parameters are omitted for brevity, but are all fully generalized in the resulting rulesets, which is optimal given the learning goals.

```
Gol Woo Arc Bal Ftm Kni Psn Tow Br Bsm LuMi St TwnH Action
##1 ##1 ### 0## ### 0## ### ### 1  #    #   #   #   TRAIN_ARCHER
1#1 ### ### ### ### ### ### ### 1  #    #   #   #   TRAIN_ARCHER
111 ### ### ### ### ### ### ### 1  #    #   #   #   TRAIN_ARCHER
111 ### ### ### ### ### ### ### 1  #    #   #   #   TRAIN_ARCHER
111 ### ### #0# ### ### ### ### 1  #    #   #   #   TRAIN_ARCHER
111 ### ### ### ### ### 0## ### 1  #    1   0   #   TRAIN_ARCHER
111 ### ### ### ### ### 0## ### 1  #    1   0   #   TRAIN_ARCHER
#11 ##1 ### ### ### 0## ### ### #  1    #   #   #   TRAIN_ARCHER
##1 ### #1# ### ### 0## 0## ### #  1    #   #   #   TRAIN_ARCHER

#11 ##1 ##1 ### ### 0## ### ### #  #    #   #   #   BUILD_LUMBER_MILL
#11 ##1 ### ### ### 0## ### ### #  #    #   #   #   BUILD_LUMBER_MILL
#11 ##1 ### ### ### 0## ### ### #  #    #   #   #   BUILD_LUMBER_MILL
#11 ##1 ### ### ### 0## ### ### #  #    #   #   #   BUILD_LUMBER_MILL
#11 ### 0#1 ### ### 0#0 ### ### #  #    #   #   #   BUILD_LUMBER_MILL
```

Figure 4.4: Fittest and most used rules in the archers scenario

combination of a few rules to properly classify the space. In all scenarios, we can see a consistent success in obtaining rewards, and this ability improves with the number of problems used for training.

4.5.5 Overhead Study

The measured times for the game AI cycle and XCS calls are shown in Table 4.5. These times reflect the overhead that the XCS module is imposing on the overall system. The results show that the XCS calls occupy only 1.6% of the cycle time, which is a very reasonable overhead. For a commercial application, it would be possible to consider further optimizing the library, to achieve even better performance.

Average cycle duration	114175.86 msec
Average XCS calls duration	1882.40 msec
XCS/Cycle proportion	0.016

Table 4.5: Overhead measures

Chapter 5

Conclusions and Future Work

This chapter will draw conclusions based on the results observed in light of the original research question, and propose future research directions and applications derived from the work done through this thesis.

5.1 Conclusions

In this thesis, we have successfully applied an Accuracy-based Classifier System as the decision making module on a Strategy Game. Four scenarios were used for the experiments, with different levels of difficulty. In the immediate reward mode, the agents developed rulesets that exhibit clear signs of learning. A quantitative analysis shows improvement throughout the training process, reaching levels that closely approach those of an optimal script. An examination of the rulesets seems to confirm that the rationale represented by the rule-action pairs makes sense in the context of the learning goals.

The agents trained in the delayed reward mode did not show clear signs of learning. This could be attributed in principle to the various sources of noise in the rewards, an ineffective credit assignment technique, and the arising of parasite classifiers that feed on the rewards of other classifiers [1]. All these conditions were present in the environment used, and can contribute to degrade the effect of learning in the population.

Given that the rulesets were trained in randomized initial conditions, they are forced to explore the search space rather than concentrating on the high paying areas. For this

[1] Parasite Classifiers are a common problem in delayed rewards and action chains environments. These are individuals that do not contribute to the objective of the agent, but that receive positive feedback because they were executed just before valid classifiers

reason, the rulesets are able to adapt to a wide array of conditions on the environment, including the behaviors of its opponents. A linear game script, the most common behavior representation for artificial agents, is limited in its ability to adapt to its environment. It is our belief that a decision making schema such as the one applied in this thesis is of greater value than the existing script solutions.

An implementation such as the one presented here can be expanded for commercial-level games. This would provide a framework to automatically generate strategies without the need to invest developing time in writing and tuning scripts tailored to each scenario and map. Furthermore, these strategies would be stronger than scripts in adapting to a wide array of conditions and user actions. Adaptivity is clearly one of the important traits that characterizes the human player, and offering this as part of a gaming experience through artificial agents will certainly make it more engaging and challenging.

5.2 Future Work

This thesis has revealed a number of challenges that could represent future directions of work. The following items highlight these challenges.

- The credit assignment technique needs to be revised, and possibly replaced by a different proposal. The goal is to achieve a credit assignment approach that performs well with different levels of noise and delayed rewards.
- It is important to identify the sources of noise, and make efforts to minimize its effects in the learning process.
- Having XCS successfully work with a delayed reward game will also open possibilities for more complex training and testing, where the evaluation functions do not need to assign rewards to intermediate states in the action chains.
- Other variations of Classifier Systems can be examined in the context of game environments, and its performance compared with XCS. Since parameters in the game are mostly integer by nature, it seems natural to explore XCSI. Seeing the success of FCS in some domains, it represents an alternative to examine as well.
- This work used XCS as an offline learning technique, but the overhead analysis suggests that online learning would be feasible. This possibility could be explored,

and the focus should be on the performance and stability of the XCS rulesets once the agent starts interacting with the user.

- If research is carried out with larger learning objectives that demand more computation power, parallelization could be a path to explore. Learning is a processor-intensive phase, and running in a parallelized platform would offer considerable advantages. One possibility is to run evaluations in parallel: each node could be running one game, which is the evaluation mechanism for classifiers. This would allow the system to run an increased amount of evaluations per time unit. Evidently, there would be a need to redesign the architecture to allow for the coordination of these simultaneous evaluations, and to ensure the consistency of the system.

Appendix A

XCS Parameters

```
<condition::ternary>
    condition size = 58
    dont care probability = 0.75
    mutate with dontcare = on
    crossover method = 2
    restricted mutation = on
</condition::ternary>

<action::wargus>
    actions = TRAIN_PEASANT:train(AiWorker());
              TRAIN_ARCHER:train(AiShooter());
              TRAIN_BALLISTA:train(AiCatapult());
              TRAIN_FOOTMAN:train(AiSoldier());
              TRAIN_KNIGHT:train(AiCavalry());
              BUILD_BARRACKS:AiNeed(AiBarracks());
              BUILD_BLACKSMITH:AiNeed(AiBlacksmith());
              BUILD_LUMBER_MILL:AiNeed(AiLumberMill());
              BUILD_STABLES:AiNeed(AiStables());
              BUILD_TOWER:AiNeed(AiTower());
              BUILD_TOWN_HALL:AiNeed(AiCityCenter());
              RESEARCH_ARROWS:research('arrow');
              RESEARCH_ARMOR:research('armor');
              RESEARCH_WEAPONS:research('weapon');
```

```
                RESEARCH_CATAPULT:research('catapult');
                UPGRADE_TO_GUARD_TOWER:upgrade(AiGuardTower());
                UPGRADE_TO_CANNON_TOWER:upgrade(AiCannonTower());
                UPGRADE_TO_KEEP:upgrade(AiBetterCityCenter());
                ATTACK_ARCHER_LIGHT:attack(AiShooter(),'light');
                ATTACK_BALLISTA_LIGHT:attack(AiCatapult(),'light');
                ATTACK_FOOTMAN_LIGHT:attack(AiSoldier(),'light');
                ATTACK_KNIGHT_LIGHT:attack(AiCavalry(),'light');
                ATTACK_COMBINED_LIGHT:attack('combined','light');
                ATTACK_ARCHER_MEDIUM:attack(AiShooter(),'medium');
                ATTACK_BALLISTA_MEDIUM:attack(AiCatapult(),'medium');
                ATTACK_FOOTMAN_MEDIUM:attack(AiSoldier(),'medium');
                ATTACK_KNIGHT_MEDIUM:attack(AiCavalry(),'medium');
                ATTACK_COMBINED_MEDIUM:attack('combined','medium');
                ATTACK_ARCHER_FULL:attack(AiShooter(),'full');
                ATTACK_BALLISTA_FULL:attack(AiCatapult(),'full');
                ATTACK_FOOTMAN_FULL:attack(AiSoldier(),'full');
                ATTACK_KNIGHT_FULL:attack(AiCavalry(),'full');
                ATTACK_COMBINED_FULL:attack('combined','full');
                NO_OPERATION:noop()
</action::wargus>

<environment::wargus>
    maps_prefix = maps/xcsproj/
    maps_suffix = .smp
    maps = archers
    binary sensors = on
    centers = 0.5,0.75,1,1.5,2
    inputs = s:gold:500:7,s:wood:500:7,s:archers:3:7,s:ballistas:3:7,
             s:footman:3:7,s:knights:3:7,s:peasants:3:7,s:towers:3:7,
             s:barracks:1:1,s:blacksmith:1:1,s:lumbermill:1:1,
             s:stables:1:1,s:townhall:1:1,s:op_barracks:1:1,
             s:op_blacksmith:1:1,s:op_lumbermill:1:1,s:op_stables:1:1,
             s:op_townhall:1:1,r:relative_gold,r:relative_wood,
             r:relative_archers,r:relative_ballistas,r:relative_footman,
```

```
            r:relative_knights,r:relative_peasants,r:relative_towers,
            f:upgrade_arrows_1,f:upgrade_arrows_2,f:upgrade_armor_1,
            f:upgrade_armor_2,f:upgrade_weapons_1,f:upgrade_weapons_2,
            f:upgrade_rangers,f:upgrade_longbow,f:upgrade_markmanship,
            f:upgrade_guard_tower,f:upgrade_cannon_tower,f:upgrade_keep,
            f:op_upgrade_arrows_1,f:op_upgrade_arrows_2,
            f:op_upgrade_armor_1,f:op_upgrade_armor_2,
            f:op_upgrade_weapons_1,f:op_upgrade_weapons_2,
            f:op_upgrade_rangers,f:op_upgrade_longbow,
            f:op_upgrade_markmanship,f:op_upgrade_guard_tower,
            f:op_upgrade_cannon_tower,f:op_upgrade_keep
</environment::wargus>

<classifier_system>
      population size = 300
        learning rate = 0.2
      discount factor = 0.7
    covering strategy = action_based 4
  discovery component = on
            theta GA = 5
crossover probability = 0.8
 mutation probability = 0.04
        epsilon zero = 2
                  vi = 5
               alpha = 0.1
      prediction init = 0.0
           error init = 5.0
         fitness init = 0.01
        set size init = 1
      population init = empty
 exploration strategy = SEMIUNIFORM:1.0
    deletion strategy = ACCURACY-BASED
         theta delete = 20
         theta GA sub = 20
         theta AS sub = 100
```

```
       GA subsumption = on
       AS subsumption = on
   update error first = on
              use MAM = on
GA tournament selection = on
      tournament size = 0.4
</classifier_system>

<experiments>
           first experiment = 0
      number of experiments = 1
              first problem = 100
         number of problems = 100
number of condensation problems = 0
    number of test problems = 0
           test environment = off
                 save every = 1
          trace experiments = on
      save population state = on
     save population report = on
                 save state = on
                 trace time = on
               compact mode = off
      save statistics every = 1
     save learning problems = on
    maximum number of steps = 0
</experiments>
```

Appendix B

Lua Script Sample

```
----------------------------------------
-- Load some common functions
----------------------------------------
Load("scripts/ai/common.lua")

----------------------------------------
-- Var declarations
----------------------------------------
local player = 0

----------------------------------------
-- Init function indexes
----------------------------------------
InitFuncs:add(function()
  ai_sample_func = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
  ai_sample_end_loop_func = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
end)

----------------------------------------
-- End loop functions
----------------------------------------
local sample_end_loop_funcs = {
  function() return AiForce(1, {AiSoldier(), 10) end,
```

```
  function() return AiWaitForce(1) end,
  function() return AiAttackWithForce(1) end,
  function() return AiSleep(500) end,
  function() ai_sample_end_loop_func[player] = 0; return false end,
}

----------------------------------------
-- End loop
----------------------------------------
function AiSampleEndloop()
  local ret

  while (true) do
    ret = sample_end_loop_funcs[ai_sample_end_loop_func[player]]()
    if (ret) then
      break
    end
    ai_sample_end_loop_func[player] = ai_sample_end_loop_func[player] + 1
  end
  return true
end

----------------------------------------
-- Main loop
----------------------------------------
local sample_funcs = {

  -- Get a city center and a worker
  function() return AiSleep(AiGetSleepCycles()) end,
  function() return AiNeed(AiCityCenter()) end,
  function() return AiSet(AiWorker(), 3) end,
  function() return AiWait(AiCityCenter()) end,
  function() return AiWait(AiWorker()) end,

  -- Get lumber mill and barracks
```

```
  function() return AiNeed(AiLumberMill()) end,
  function() return AiNeed(AiBarracks()) end,
  function() return AiForce(0, {AiSoldier(), 2}) end,
  function() return AiForce(1, {AiSoldier(), 1}) end,
  function() return AiWaitForce(1) end,
  function() return AiAttackWithForce(1) end,

  -- Go into an attacking loop
  function() return AiSampleEndloop() end,
}

---------------------------------------
-- Script entry point
---------------------------------------
function AiSample()

  local ret
  player = AiPlayer() + 1

  checkVictory(AiPlayer())
  checkTimeout(AiPlayer())

  while(true)do
    ret = sample_funcs[ai_sample_func[player]]()

    if (ret) then break
    end
    ai_sample_func[player] = ai_sample_func[player] + 1
  end
end

---------------------------------------
-- Define this AI
---------------------------------------
DefineAi("sample", "*", "sample", AiSample)
```

Appendix C

Estimating Battle Outcome

The following is the code used for estimating the outcome of a battle. Although it is not entirely accurate, it can be used as a rough estimation with a degree of uncertainty.

```
local MIN_ARMY_FACTOR = 1.1
local sword_upgrade = 20
local arrow_upgrade = 10

local unit_strength = {}
unit_strength["unit-footman"] = 200
unit_strength["unit-archer"] = 100
unit_strength["unit-ballista"] = 450
unit_strength["unit-knight"] = 400
unit_strength["unit-human-guard-tower"] = 500
unit_strength["unit-human-cannon-tower"] = 500

function predict_battle(army)
  local player_army_strength = 0
  local i=1

  -- Calculate the strength of this player's attacking army
  while(i<#army) do
    if(army[i+1]>0) then
      player_army_strength = player_army_strength +
```

```
                                (army[i+1]*
                                  get_unit_strength(army[i], AiPlayer()))
    end
    i = i+2
  end

  -- Calculate the strength of the opponent's army
  local op_army = get_player_army(opponent())
  local op_army_strength  = 0
  i=1

  -- Calculate the strength of each unit
  for unit, amount in pairs(op_army) do
    if(amount>0) then
      op_army_strength = op_army_strength +
                         (amount*get_unit_strength(unit, opponent()))
    end
    i = i+1
  end

  return (player_army_strength / op_army_strength) > MIN_ARMY_FACTOR
end

-- Returns a table of the following form:
-- table["unit-name"] = unit-amount
function get_player_army(player)
  local army = {}

  army["unit-footman"]
        = GetPlayerData(player, "UnitTypesCount", AiSoldier())
  army["unit-archer"]
        = GetPlayerData(player, "UnitTypesCount", AiShooter())
      + GetPlayerData(player, "UnitTypesCount", AiEliteShooter())
  army["unit-ballista"]
        = GetPlayerData(player, "UnitTypesCount", AiCatapult())
```

```
  army["unit-knight"]
        = GetPlayerData(player, "UnitTypesCount", AiCavalry())
  army["unit-human-guard-tower"]
        = GetPlayerData(player, "UnitTypesCount", AiGuardTower())
  army["unit-human-cannon-tower"]
        = GetPlayerData(player, "UnitTypesCount", AiCannonTower())

  return army

end

function get_unit_strength(unit, player)

  local extra_damage = 0

  if(unit==AiSoldier()) then
    extra_damage = sword_upgrade *
                   GetPlayerData(player, "Upgrade", AiUpgradeWeapon1())
                   + sword_upgrade *
                   GetPlayerData(player, "Upgrade", AiUpgradeWeapon2())
  elseif(unit==AiShooter()) then
    extra_damage = arrow_upgrade *
                   GetPlayerData(player, "Upgrade", AiUpgradeMissile1())
                   + arrow_upgrade *
                   GetPlayerData(player, "Upgrade", AiUpgradeMissile2())
  elseif(unit==AiCavalry()) then
    extra_damage = sword_upgrade *
                   GetPlayerData(player, "Upgrade", AiUpgradeWeapon1())
                   + sword_upgrade *
                   GetPlayerData(player, "Upgrade", AiUpgradeWeapon2())
  end

  return unit_strength[unit] + extra_damage
end
```

Appendix D

Overhead Study Sample Code

```
----------------------------------------
-- Script entry point
----------------------------------------
function AiXcsAction()

    ...

    if(OVERHEAD_LOG) then
        xcs_time_t1 = getTime()
    end

    -- collect the parameters from the environment
    env = getEnvironmentParameters(AiPlayer())

    -- get action (function name and parameters)
    local action = xcsStep(AiPlayer(), env)

    -- execute action
    assert(loadstring(action))()

    if(OVERHEAD_LOG) then
        xcs_time_t2 = getTime()
        print("xcs.time;" .. os.difftime(xcs_time_t2,xcs_time_t1) )
```

```
        frame_time_t2 = getTime()
        print("frame.time;"..os.difftime(frame_time_t2,frame_time_t1) )
        frame_time_t1 = getTime()
    end
end
```

Appendix E

List of Acronyms

ACS Anticipatory Classifier System.

AI Artificial Intelligence.

AKADS Automatic Knowledge Acquisition for Dynamic Scripting.

CIGAR Case Injected Genetic Algorithm.

DS Dynamic Scripting.

FCS Fuzzy Classifier System. An extension of LCS that uses fuzzy-valued parameters.

FXCS Accuracy-based Fuzzy Classifier System. An extension of XCS that uses fuzzy-valued parameters.

GA Genetic Algorithm.

LCS Learning Classifier System.

MMORPG Massive Multiplayer Online Role Playing Games.

RTS Real Time Strategy.

RTSG Real Time Strategy Game.

SARSA State-Action-Reward-State-Action. A Q-Learning technique used for Markov Processes.

TIELT Testbed for Integrating and Evaluating Learning Techniques.

UCS Supervised Classifier System, a supervised version of XCS.

XCS Accuracy-based learning Classifier Systems.

XCSI XCS Modified for Integer Inputs. An extension of XCS for integer valued input parameters.

ZCS Zeroth level Classifier System.

References

[1] Hussein A. Abbass, Jaume Bacardit, Martin V. Butz, and Xavier Llora. Online adaption in learning classifier systems: Stream data mining, 2004.

[2] M.F. Abbod, D.G. von Keyserlingk, D.A. Linkens, and M. Mahfouf. Survey of utilisation of fuzzy technology in medicine and healthcare. *Fuzzy Sets and Systems*, 120(2):331–349, 2001.

[3] David W. Aha, Matthew Molineaux, and Marc J. V. Ponsen. Learning to win: case-based plan selection in a real-time strategy game. In Héctor Muñoz-Avila and Francesco Ricci, editors, *ICCBR*, volume 3620 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.

[4] Jaume Bacardit. *Pittsburgh Genetics-Based Machine Learning in the Data Mining era: Representations, generalization, and run-time.* PhD thesis, University, Barcelona, Catalonia, Spain, 2004.

[5] Jaume Bacardit. Analysis of the initialization stage of a pittsburgh approach learning classifier system. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1843–1850, New York, NY, USA, 2005. ACM.

[6] Jaume Bacardit, David E. Goldberg, and Martin V. Butz. Improving the performance of a pittsburgh learning classifier system using a default rule, 2004.

[7] Jaume Bacardit, David E. Goldberg, Martin V. Butz, Xavier Llora, and Josep Maria Garrell. Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy, 2004.

[8] Jaume Bacardit and Natalio Krasnogor. Empirical evaluation of ensemble techniques for a pittsburgh learning classifier system.

[9] Alwyn Barry. Jxcs. URL: http://www.cs.bath.ac.uk/ amb/code/jxcsawt.zip.

[10] Alwyn Barry. Xcsc. URL: http://www.cs.bath.ac.uk/ amb/LCSWEB/xcsc.zip.

[11] Alwyn Barry. Limits in long path learning with xcs, 2003.

[12] Jeffrey K. Bassett and Kenneth A. De Jong. Evolving behaviors for cooperating agents. In *International Syposium on Methodologies for Intelligent Systems*, pages 157–165, 2000.

[13] Ester Bernado, Xavier Llorà, and Josep M. Garrell. XCS and GALE: a comparative study of two learning classifier systems with six other learning algorithms on classification tasks. *Lecture notes in computer science*, 2321:115–132, 2001.

[14] Yngvi Bjrnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer. Fringe search: beating A* at pathfinding on game maps. In *CIG*. IEEE, 2005.

[15] Michael Buro. Real-time strategy games: a new AI research challenge. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1534–1535. Morgan Kaufmann, 2003.

[16] Michael Buro. Call for AI research in RTS games. *Proceedings of the AAAI Workshop on AI in Games*, pages 139–141, 2004.

[17] Michael Buro and Timothy Furtak. RTS games as test-bed for real-time AI research. In *Proceedings of the 7th Joint Conference on Information Science, JCIS 2003*, 2003.

[18] Martin V. Butz. Xcsfjava. URL: http://medal.cs.umsl.edu/files/XCSFJava1.1.zip.

[19] Martin V Butz. XCSJava 1.0: An implementation of the XCS classifier system in Java. Technical Report 2000027, Illinois Genetic Algorithms Laboratory, 2000.

[20] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. PAC learning in XCS. Technical Report 2004011, Illinois Genetic Algorithms Laboratory, 2004.

[21] Martin V. Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W. Wilson. How XCS evolves accurate classifiers. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 927–934, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[22] Martin V. Butz, Pier Luca Lanzi, Xavier Llorà, and David E. Goldberg. Knowledge extraction and problem structure identification in XCS. *Lecture notes in computer science*, pages 1051–1060, 2004.

[23] Martin V. Butz and Stewart W. Wilson. An algorithmic description of XCS. *Lecture Notes in Computer Science*, 1996:253–??, 2001.

[24] Nicholas Cole. Using a genetic algorithm to tune first-person shooter bots. *Congress on Evolutionary Computation, 2004. CEC2004.*, 1:139–145, 2004.

[25] Andrea Corradini, Adrian Bak, and Thomas Hanneforth. *A Natural Language Interface for a 2D Networked Game*, pages 225–234. Springer Berlin / Heidelberg, 2007.

[26] Vincent Corruble, Charles A. G. Madeira, and Geber Ramalho. Steps toward building of a good ai for complex wargame-type simulation games. In Quasim H. Mehdi and Norman E. Gough, editors, *GAME-ON*. ?, 2002.

[27] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, Harvey Duff, and Stephanie Gillis. Scriptease: A generative/adaptive programming paradigm for game scripting. *Sci. Comput. Program.*, 67(1):32–58, 2007.

[28] Marc A. DeLoura, editor. *Game Programming Gems 2*, page 287. Charles River Media, 2001.

[29] Gameware Development. Creatures, 1996. URL: http://www.gamewaredevelopment.co.uk/creatures_index.php.

[30] R. Drewes, S.J. Louis, C. Miles, J. McDonnell, and N. Gizzi. Use of case injection to bias genetic algorithm solutions of similar problems. *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, 2:1170–1177 Vol.2, 8-12 Dec. 2003.

[31] Blizzard Entertainment. Warcraft, 1999. URL: http://www.blizzard.com/us/war2bne.

[32] Ubisoft Entertainment. Far cry, 2004. URL: http://www.ubi.com/ENCA/Games/Info.aspx?pId=1328.

[33] Digital Extremes Epic Games. Unreal tournament, 1999. URL: http://www.unrealtournament.com.

[34] Firaxis. Civilization revolution, 2008. URL: http://www.civ3.com.

[35] Kenneth D. Forbus, James V. Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game ais. *IEEE Intelligent Systems*, 17(4):25–30, 2002.

[36] Johannes Furnkranz. Machine learning in games: a survey. *Machines that learn to play games*, pages 11–59, 2001.

[37] Malte Gabsdil, Alexander Koller, and Kristina Striegnitz. *Natural language and inference in a computer game*, pages 1–7. Association for Computational Linguistics, Morristown, NJ, USA, 2002.

[38] Arthur Gill. *Introduction to the theory of finite state machines.* McGraw-Hill, 1962.

[39] J.L. Grantner and M.J. Patyra. Fuzzy logic finite state machine models for real time systems. In *NAFIPS/IFIS/NASA 94. Proceedings of the First International Joint Conference of the North American Fuzzy Information Processing Society Biannual Conference*, pages 296–300, 1994.

[40] Zahia Guessoum, Lilia Rejeb, and Olivier Sigaud. Using XCS to build adaptive agents. In *Proceedings AAMAS Symposium 2004*, 2004.

[41] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume 4, pages 100–107, 1968.

[42] John Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[43] John H. Holland. Properties of the bucket brigade. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 1–7, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[44] John H. Holland. Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. *Computation & intelligence: collected readings*, pages 275–304, 1995.

[45] John H. Holland, Lashon B. Booker, Marco Colombetti, Marco Dorigo, David E. Goldberg, Stephanie Forrest, Rick L. Riolo, Robert E. Smith, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. What is a learning classifier system? In *Learning Classifier Systems, From Foundations to Applications*, pages 3–32, London, UK, 2000. Springer-Verlag.

[46] John H. Holland and Judith S. Reitman. Cognitive systems based on adaptive algorithms. *SIGART Bull.*, (63):49–49, 1977.

[47] John H. Holmes. *Evolution-assisted discovery of sentinel features in epidemiologic surveillance.* PhD thesis, Drexel University, Philadelphia, PA, USA, 1996.

[48] Albert Orriols i Puig and Ester Bernadó i Mansilla. Analysis of reduction algorithms for XCS classifier system. *Recent Advances in Artificial Intelligence Research and Development*, pages 383–390, 2004.

[49] IBM. Deep blue. URL: http://www.research.ibm.com/deepblue.

[50] Electronic Arts Inc. FIFA 08, 2008. URL: http://www.fifa08.ea.com.

[51] Hiroyasu Inoue, Keiki Takadama, and Katsunori Shimohara. Exploring xcs in multiagent environments. In *GECCO '05: Proceedings of the 2005 workshops on genetic and evolutionary computation*, pages 109–111, New York, NY, USA, 2005. ACM.

[52] Igor Karpov, Thomas D'Silva, Craig Varrichio, Kenneth O. Stanley, and Risto Miikkulainen. Integration and evaluation of exploration-based learning in games. In Sushil J. Louis and Graham Kendall, editors, *CIG*, pages 39–44. IEEE, 2006.

[53] George J. Klir and Bo Yuan. *Fuzzy sets and fuzzy logic: theory and applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[54] Tim Kovacs and Manfred Kerber. Some dimensions of problem complexity for XCS. In Annie S. Wu, editor, *Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop Program*, 2000.

[55] Tim Kovacs and Manfred Kerber. What makes a problem hard for XCS? *Lecture Notes in Computer Science*, 1996:80–??, 2001.

[56] Timothy Kovacs. Learning classifier systems resources. Technical Report CSRP-00-19, 2000.

[57] Tani Kyuichiro and Kamei Katsuari. State evaluation system for chess game by intuitive fuzzy reasonings. *Faji Shisutemu Shinpojiumu Koen Ronbunshu*, 15:249–250, 1999.

[58] Pontifical Catholic University of Rio de Janeiro Lablua. Lua scripting language. URL: http://www.lua.org.

[59] J. Laird and M. van Lent. Human-level ai's killer application: Interactive computer games, 2000.

[60] Pier Luca Lanzi. The XCS library. URL: http://xcslib.sourceforge.net.

[61] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Extending xcsf beyond linear approximation. In *GECCO '05: Proceedings of the 2005 conference on genetic and evolutionary computation*, pages 1827–1834, New York, NY, USA, 2005. ACM.

[62] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Xcs with computed prediction in multistep environments. In *GECCO '05: Proceedings of the 2005 conference on genetic and evolutionary computation*, pages 1859–1866, New York, NY, USA, 2005. ACM.

[63] Pier Luca Lanzi and Rick L. Riolo. A roadmap to the last decade of learning classifier system research. In *Learning Classifier Systems, From Foundations to Applications*, pages 33–62, London, UK, 2000. Springer-Verlag.

[64] Pier Luca Lanzi and Stewart W. Wilson. Toward optimal classifier system performance in non-markov environments. *Evolutionary Computation*, 8(4):393–418, 2000.

[65] Ronni Laursen and Daniel Nielsen. Investigating small scale combat situations in real time strategy computer games. Master's thesis, University of Aarhus, Department of computer science, 2005.

[66] Michael Lewis and Jeffrey Jacobson. Game engines in scientific research - introduction. *Communications of the ACM*, 45(1):27–31, 2002.

[67] Xavier Llora, Kumara Sastry, and David E. Goldberg. The compact classifier system - scalability analysis and first results 2005. 2005.

[68] S. J. Louis and C. Miles. Playing to learn: case-injected genetic algorithms for learning to play computer games. *IEEE Trans. Evolutionary Computation*, 9(6):669–681, 2005.

[69] Carl Machover. Four decades of computer graphics. *IEEE Comput. Graph. Appl.*, 14(6):14–19, 1994.

[70] Charles Madeira. Adaptive agents for modern strategy games: an approach based on reinforcement learning, 2007.

[71] Charles Madeira, Vincent Corruble, Geber Ramalho, and Bohdana Ratitch. Bootstrapping the learning process for the semi-automated design of a challenging game AI. In D. Fu, S. Henke, and J. Orkin, editors, *Proceedings of the AAAI-04 workshop on challenges in game artificial intelligence*, pages 72–76. AAAI Press, 2004.

[72] Charles A. G. Madeira, Vincent Corruble, and Geber Ramalho. Designing a reinforcement learning-based adaptive ai for large-scale strategy games. In John E. Laird and Jonathan Schaeffer, editors, *AIIDE*, pages 121–123. The AAAI Press, 2006.

[73] Ester Bernado I Mansilla, Xavier Llorà, and Ivan Traus. Multiobjective learning classifier systems, 2005.

[74] John Manslow. *Learning and adaptation*, pages 557–566. Charles River Media, 2002.

[75] Wolfgang Stolzmann Martin V. Butz, David E. Goldberg. Introducing a genetic generalization pressure to the anticipatory classifier systems - part 2: performance analysis. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, 2000.

[76] Maxis. The sims, 2000. URL: http://thesims.ea.com/.

[77] M. Luisa McAllister, Sergei A. Ovchinnikov, John T. Dockery, and Klaus-Peter Adlassnig. Tutorial on fuzzy logic in simulation. In *WSC '85: Proceedings of the 17th conference on Winter simulation*, pages 40–44, New York, NY, USA, 1985. ACM.

[78] Qingchun Meng, Xiaodong Zhuang, Changjin Zhon, Jianshe Xiong, Yulin Wang, Tao Wang, and Bo Yin. Game strategy based on fuzzy logic for soccer robots. *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, 5:3758–3763 vol.5, 2000.

[79] Risto Miikkulainen, Bobby D. Bryant, Ryan Cornelius, Igor V. Karpov, Kenneth O. Stanley, and Chern Han Yong. Computational intelligence in games. *Computational Intelligence: Principles and Practice*, 2006.

[80] Chris Miles, J. Quiroz R. Leigh, and Sushil J. Louis. Co-evolving influence map tree based strategy game players. *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, (1):88–95, 2007.

[81] Chris Miles and Sushil J. Louis. Towards the co-evolution of influence map tree based strategy game players. *2006 IEEE Symposium on Computational Intelligence and Games*, pages 75–82, 2006.

[82] Chris Miles, Sushil J. Louis, Nicholas Cole, and John McDonnell. Learning to play like a human: case injected genetic algorithms for strategic computer gaming. *Evolutionary Computation, 2004. CEC2004. Congress on*, 2:1441–1448 Vol.2, 19-23 June 2004.

[83] Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[84] M. Molineaux and D. W. Aha. Tielt: A testbed for gaming environments. In *Proceedings of the national conference on Artificial Intelligence*, pages 1690–1691, 2005.

[85] Matthew Molineaux, David W. Aha, and Marc Ponsen. Defeating novel opponents in a real-time strategy game. *Reasoning, Representation, and Learning in Computer Games: Papers from the IJCAI Workshop (Technical Report AIC-05-127)*, 2005.

[86] T. Nakashima, M. Udo, and H. Ishibuchi. Acquiring the positioning skill in a soccer game using a fuzzy q-learning. In *Computational Intelligence in Robotics and Automation, 2003. Proceedings. 2003 IEEE International Symposium on*, pages 1488– 1491, 2003.

[87] A. Nareyek. Intelligent agents for computer games. In *Second International Conference on Computers and Games*, 2000.

[88] Alexander Nareyek. Computer games: boon or bane for AI research? *Knstliche Intelligenz*, pages 43–44, 2004.

[89] Nintendo. Zelda series, 1987-present. URL: http://www.zelda.com.

[90] University of Ottawa. Paradise laboratory. http://paradise.site.uottawa.ca.

[91] Albert Orriols and Ester Bernadó-Mansilla. The class imbalance problem in learning classifier systems: a preliminary study. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 74–78, New York, NY, USA, 2005. ACM.

[92] Osiris Pérez. Aplicación de técnicas de AI emergentes para la generación de estrategias cooperativas en juegos de video, 2004.

[93] Marc Ponsen. Improving adaptive game AI with evolutionary learning. *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 389–396, 2004.

[94] Marc Ponsen, Héctor Munoz-Ávila, Pieter Spronck, and David W. Aha. Automatically generating game tactics via evolutionary learning. *IAAI-05 : Annual Conference on Innovative Applications of Artificial Intelligence No17*, 2006.

[95] Marc J. V. Ponsen, Stephen Lee-Urban, Hector Munoz-Avila, David W. Aha, and Matthew Molineaux. stratagus: an open-source game engine for research in real-time strategy games. In D. W. Aha, H. Munoz-Avila, and M. van Lent, editors, *Papers from the IJCAI workshop on reasoning representation and learning in computer games*, 2005.

[96] Marc J. V. Ponsen, Hctor Muoz-Avila, Pieter Spronck, and David W. Aha. Automatically acquiring domain knowledge for adaptive game ai using evolutionary learning. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 1535–1540. AAAI Press / The MIT Press, 2005.

[97] The Stratagus Project. Stratagus, 2003. URL: http://www.stratagus.org.

[98] Steve Rabin. *AI Game Programming Wisdom.* Charles River Media, Inc., Rockland, MA, USA, 2002.

[99] B. Ravichandran, Avinash Gandhe, and R. E. Smith. Xcs for robust automatic target recognition. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1803–1810, New York, NY, USA, 2005. ACM.

[100] Timothy E. Revello and Robert McCartney. Generating war game strategies using a genetic algorithm. In *CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress*, pages 1086–1091, Washington, DC, USA, 2002. IEEE Computer Society.

[101] Gabriel Robert, Pierre Portier, and Agns Guillot. Classifier systems as 'animat' architectures for action selection in mmorpg, 2002.

[102] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*, pages 653–664. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second edition, 2003.

[103] Yuji Sato and Ryutaro Kanno. Event-driven learning classifier systems for online soccer games. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 2201–2202, New York, NY, USA, 2005. ACM.

[104] Jorgen Havsberg Seland. A visual programming language for hierarchical finite state machines in game ai, 2007.

[105] Olivier Sigaud and Stewart W. Wilson. Learning classifier systems: a survey. *Soft Comput.*, 11(11):1065–1078, 2007.

[106] Stephen Frederick Smith. *A learning system based on genetic adaptive algorithms.* PhD thesis, Pittsburgh, PA, USA, 1980.

[107] William M. Spears and Diana F. Gordon. *Evolving Finite-State Machine Strategies for Protecting Resources*, pages 5–28. Springer Berlin, 2008.

[108] Pieter Spronck. *Adaptive Game AI.* PhD thesis, University of Maastricht, 2005.

[109] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. *Mach. Learn.*, 63(3), 2006.

[110] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent AI in simulation and in practice. In *Proceedings of the 4th International Conference on Intelligent Games and Simulation*, pages 93–100, 2003.

[111] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Difficulty scaling of game ai. In *GAME'ON'2004, 5th annual European Conference on Simulation and AI in Computer Games*. Eurosis, november 2004.

[112] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653– 668, 2005.

[113] M. Steinberg. Development and simulation of an fa-18 fuzzy logic automaticcarrier landing system. *Second IEEE International Conference on Fuzzy Systems*, 2:797–802, 1993.

[114] Wolfgang Stolzmann. Antizipative classifier systems [anticipatory classifier systems], 1997.

[115] Lionhead Studios. Black and white, 2001. URL: http://www.lionhead.com/bw.

[116] Microsoft Game Studios. Age of mithology, 2002. URL: http://www.microsoft.com/games/ageofmythology.

[117] The Wargus Team. Wargus, 2002. URL: http://wargus.sourceforge.net.

[118] Manuel Valenzuela-Rendón. The fuzzy classifier system: motivations and first results. In *PPSN I: Proceedings of the 1st Workshop on parallel problem solving from nature*, pages 338–342, London, UK, 1991. Springer-Verlag.

[119] HO Wang, K. Tanaka, MF Griffin, U.T.R. Center, and E. Hartford. An approach to fuzzy control of nonlinear systems: stability anddesign issues. *Fuzzy Systems, IEEE Transactions on*, 4(1):14–23, 1996.

[120] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. pages 123–134, 1988.

[121] Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

[122] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[123] Stewart W. Wilson. Generalization in the XCS classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[124] Stewart W. Wilson. State of XCS classifier system research. *Lecture Notes in Computer Science*, 1813:63–81, 2000.

[125] Stewart W. Wilson. Mining oblique data with XCS. *Lecture Notes in Computer Science*, 1996:158–??, 2001.

[126] Stewart W. Wilson. Compact rulesets from xcsi. In *IWLCS '01: Revised Papers from the 4th International Workshop on Advances in Learning Classifier Systems*, pages 197–210, London, UK, 2002. Springer-Verlag.

[127] Sor Ying (Byron) Wong and Sonia Schulenburg. Portfolio allocation using xcs experts in technical analysis, market conditions and options market. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on genetic and evolutionary computation*, pages 2965–2972, New York, NY, USA, 2007. ACM.

[128] Lotfi A. Zadeh. Soft computing and fuzzy logic. *IEEE Softw.*, 11(6):48–56, 1994.

[129] Zhaohua Zhang, Stan Franklin, and Dipankar Dasgupta. Metacognition in software agents using classifier systems. pages 83–88, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.