IMPROVING ADAPTIVE GAME AI WITH EVOLUTIONARY LEARNING

Marc Ponsen
(B.Sc.)

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

Delft, 2004

Faculty of Media & Knowledge Engineering
Delft University of Technology

Bless all forms of intelligence

PREFACE

I have been a 'gamer' my whole life. I started playing computer games on an Atari console when I was just three years old and I am still playing. If it is true that we don't stop playing because we grow old, but we grow old because we stop playing, then I hope I never grow old.

The concept of artificial intelligence has fascinated me for quite some time. Movies such as 'the Matrix' and '2001: A Space Odyssey' have always sparkled my imagination. My interest grew when I was first introduced to 'academic artificial intelligence' during my studies at Delft University of Technology.

In my search for a graduation assignment, I was very fortunate that I was able to combine games and artificial intelligence. My thesis is called: "Improving Adaptive Game AI with Evolutionary Learning". It addresses the application of both adaptive game artificial intelligence and evolutionary learning techniques in computer games.

I would like to thank all the people at the Institute of Knowledge and Agent Technology (IKAT) in Maastricht for providing me with the tools and guidance to complete my master thesis. I especially would like to thank my supervisor, Pieter Spronck for his support and inspiration. Thank you for sacrificing at least a dozen red pens! I would also like to thank the people at Delft University of Technology. Thank you for introducing me to 'academic artificial intelligence' and paving the way for my future career. Special thanks goes to my graduation coordinator Leon Rothkrantz. Furthermore, I would like to thank David Aha for his insightful and lengthily comments on my thesis. Also, much appreciation goes out to the Stratagus development team for their programming support. Last but certainly not least, I would like to express my gratitude to my family and friends for their devotion, support and patience.

Marc Ponsen 2004

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

This Chapter presents the background of this thesis. Section 1.1 provides information on artificial intelligence in games. Section 1.2 discusses the background of the thesis' research, first explaining the role of dynamic scripting in online learning and then discussing the role that offline learning can play in the development of artificial intelligence in games. Section 1.3 discusses the problem statement and research question. Our approach is explained in Section 1.4. Finally, Section 1.5 gives an overview of the remainder of the thesis.

1.1 THE EVOLUTION OF COMPUTER GAME ARTIFICIAL INTELLIGENCE

Since the birth of computer games, artificial intelligence (AI) has been a standard feature of games - especially with developers' emphasis on single-player games, which today still represent the majority of released titles. AI is an element of so-called gameplay, which comprises everything but the visual and auditory presentation of the game. For the gaming industry, AI encompasses many subject areas such as interaction, pathfinding, machine learning, flocking, formations, difficulty scaling and decision-making. The current emphasis in computer game AI is on the *illusion* of human-like behavior. However, there is an increasing demand for a true human-level AI from various perspectives.

The game industry is starting to recognize that sophisticated AI could enhance the entertainment value of their products and consequently increase revenues. Already, many computer games are marketed based on the quality of their AI (e.g., BLACK & WHITE, THE SIMS, FAR CRY). Developers in the past mainly focused on sound and graphics. The implementation of computer opponent's AI was often deferred to the final phase of the project. The trend is shifting; AI is often assigned an equal priority to graphics and sound in the initial game design. Sweetser (2002) states: "As the graphics race subsides and gamers grow weary of predictable and deterministic game characters, game developers must set aside their "old faithful" finite state machines and look to more advanced techniques that give the users the gaming experience they crave. The next industry breakthrough will be with characters that behave realistically and that can learn and adapt, rather than more polygons, higher resolution textures and more frames-per-second".

Military institutions recognize that besides entertainment, computer games can also be used for military training and simulation purposes. Military training and simulation in the real world is too expensive and dangerous. Computer games with ever increasing complex and realistic environments provide a cheap and reliable alternative (Laird 2000).

Interactive computer games are increasingly attractive for academic AI researchers. Laird (2000) states that interactive computer games are the 'killer application' for human-level AI research. "They are the application that will soon need human-level AI, and they can provide the environments for research on the right kinds of problems that lead to the type of the incremental and integrative research needed to achieve human-level AI."

1.2 RESEARCH BACKGROUND

Is it possible to improve computer game AI in commercial computer games by applying machine learning techniques? In this thesis computer game AI will be interpreted solely as the decision-making process of non-player characters (in particular opponents) in a game. Most games resort to scripts for most, if not all of their AI. Scripts, i.e., lists of rules that are executed sequentially (Tozour 2002b) are generally static and tend to be quite long and complex (Brockington and Darrah 2002). Due to this complexity, AI scripts are likely to contain weaknesses, which can be exploited by human players to easily defeat supposedly tough opponents (Spronck, Sprinkhuizen-Kuyper and Postma 2003) Furthermore, because the scripts are static they cannot deal with unforeseen tactics employed by a human player. Machine learning can resolve these shortcomings of static AI and consequently improve the quality of opponent AI. Machine learning can either take place online or offline. We will discuss online and offline learning in computer games in the next subsections.

1.2.1 Online Learning

Online learning entails that the AI will adapt during play after the game has been released. It allows opponents to automatically repair weaknesses in their scripts that are exploited by the human player, and to adapt to changes in human player tactics and playing style. Online learning can be either supervised or unsupervised. Supervised online learning requires that the human player indicate how successful the AI is, which precludes automatic adaptation. Therefore the term "online learning" in this thesis, will be reserved for unsupervised online learning. For online learning to work in practice, it must be fast, effective, robust and efficient. Spronck et al. (2003) explain these requirements as follows:

- Fast: Since online learning takes place during gameplay, the learning algorithm should be computationally fast, lest it will disrupt the pacing of the game.
- Effective: Adapted scripts should be at least as challenging as manually designed ones, and therefore the learning mechanism must guarantee the generation of mostly effective AI. This requirement excludes random learning methods, such as evolutionary algorithms.
- Robust: The learning mechanism must be able to cope with a significant amount of randomness inherent in most commercial gaming mechanisms.
- Efficient: The learning process should learn efficiently, relying on just a small number of trials. This requirement excludes slow-learning techniques, such as neural networks, evolutionary algorithms and reinforcement learning.

Dynamic scripting (Spronck *et al*, 2003) is an unsupervised online learning technique for commercial computer games. Important factors when attempting to achieve high performance for a learning mechanism are using deterministic experiments and adding prior domain knowledge. Because of the non-deterministic nature of game environments in general, dynamic scripting relies heavily on domain knowledge. In dynamic scripting, the rules used in a script that controls an opponent are extracted from an adaptive rulebase that contains only manually designed rules. The probability that a rule is selected for a script is influenced by a weight value that is associated with each rule. The rulebase adapts by changing the weight values to reflect the success or failure rate of the corresponding rules in scripts.

The dynamic scripting technique meets all four requirements. First, it is computationally fast, because it only requires the extraction of rules from a rulebase and the updating of weights once per game. Second, it is effective, because all rules in the rulebase are based on domain knowledge. Third, it is robust because rules are not removed immediately when punished. Finally, Spronck et al. (2003) showed that in a simulated as well as a commercial game environment dynamic scripting can adapt rapidly to static or changing tactics, and therefore it is also efficient.

1.2.2 Offline Learning

Offline learning entails that the AI will adapt by self-play, without human intervention. Adaptive technologies are giving developers a tool that can help them optimize computer game AI parameters offline during the Quality Assurance phase of game development. AI tuning is always somewhat problematic; in commercial games there can be hundreds of parameters that can affect the AI's style of play. Testing every combination is an impossible task, especially given the short amount of time available for AI tuning. Over time, an offline learning mechanism can test out many more AI variations than an individual developer could.

An interesting application of offline learning is creating new strategies and tactics for opponent AI by self-play. Offline learning therefore provides the means of improving the dynamic scripting process by discovering new strategies and tactics that can be added to the dynamic scripting rulebase. This can make the dynamic scripting technique more effective in dealing with human player tactics which the developers did not foresee, and for which they did not add any rules to the rulebase as countermeasures.

1.3 PROBLEM STATEMENT AND RESEARCH QUESTION

In 1.2.2 we proposed that offline learning can potentially enhance the dynamic scripting technique by improving the rulebase through the addition of offline discovered strategies and tactics. However, this has as yet not been shown in practice. This leads to the following problem statement:

Problem Statement: *To what extent can offline learning techniques be used to improve the rulebase used for dynamic scripting, in order to improve the AI in commercial computer games?*

Offline learning does not suffer from any of the four previously mentioned requirements associated with online learning. Therefore, many machine learning techniques are suitable for offline learning. We will focus on evolutionary techniques to enhance the intelligence of opponents by training them against other (scripted) opponents. An evolutionary algorithm, when properly implemented, has the ability to deal with complex environments, such as computer games.

Furthermore, the most complex AI in modern games is found in so-called "computer roleplaying games" (CRPGs) and "real-time strategy" (RTS) games (e.g., war simulations). Dynamic scripting has already been shown to be successful for CRPGs (Spronck *et al.*, 2003), but not yet for RTS games. Since we expect that it is just as applicable to RTS games, we decided to focus on these for our research. The following research questions will therefore guide our research:

Research Question 1: *Is it possible to design and implement an evolutionary algorithm that discovers new tactics and strategies for real-time strategy games?*

Research Question 2: *Will offline discovered tactics and strategies enhance the performance for the dynamic scripting rulebase?*

1.4 APPROACH

To answer the research question, we address four main objectives:

1) Selecting a flexible, state-of-the-art RTS game environment for our experimental research.
2) Designing and implementing the dynamic scripting technique in the selected RTS game and demonstrating that it works against several opponent strategies on several maps.
3) Applying offline learning using an evolutionary algorithm to discover new strategies and tactics in the selected RTS game.
4) Translating offline-discovered strategies and tactics into rules for the rulebase and show that these additions enhance performance for dynamic scripting in the selected RTS game.

1.5 THESIS OVERVIEW

The remainder of the thesis is organized as follows. Chapter 2 discusses various artificial intelligence techniques in commercial computer games relevant to our research. Chapter 3 addresses the first research objective, namely choosing a flexible state-of-the-art RTS game environment for our experiments. The second research objective is discussed in Chapter 4 wherein we will explain how we implemented dynamic scripting in the selected RTS game environment and we will also discuss the results for the dynamic scripting AI against several scripted opponents. Chapter 5 discusses the third research objective, namely applying an evolutionary algorithm in the selected RTS game environment in order to discover new tactics and strategies. Our fourth and final objective is addressed in Chapter 6. We will explain how we translated the offline-discovered tactics and strategies into rules for the dynamic scripting rulebase, and discuss our approach. We will finish the thesis in Chapter 7, where we will answer the research questions and problem statement, as well as give recommendations for future research.

C h a p t e r   2

COMPUTER GAME ARTIFICIAL INTELLIGENCE

In 2.1 we will first introduce rule based AI techniques relevant to our research. In 2.2 we will discuss the different types of machine learning and explain why machine learning can be problematic in computer games. In 2.3 we will give an introduction to evolutionary algorithms.

2.1 RULE BASED ARTIFICIAL INTELLIGENCE IN GAMES

AI programmers have numerous techniques at their disposal to try to simulate human-level-behavior. Of these, rule based approaches have been widely accepted and successfully employed by game developers for a number of years. Rules in a rule based system consist of a condition side (the antecedent) and an action side (the consequent). Rules-based AI is currently the technology of choice for AI development because (1) these approaches are familiar, taking their principles from familiar programming paradigms, (2) rule based designs are generally predictable, hence easy to test and debug and (3) most developers lack any training in, or knowledge of, the more complex AI technologies, and thus don't use them when deadlines are approaching fast. Currently the most dominant rule based AI techniques for computer games include scripting and state machines.

In order to easily implement rules and reactions, over 80% of developers use some kind high-level scripting language (Woodcock 2003). A scripting language is any programming language created to simplify any complex task for a particular program (Sweetser 2002). Scripts are used to control the game engine from the outside. Scripts have four main advantages; they are (1) understandable, (2) easy to implement, (3) easily extendable, and (4) useable by nonprogrammers (Tozour 2002b). Some games use custom scripting languages, such as Bioware’s NWscript, UnrealScript or LUA scripting, to manage the AI.

A finite state machine is a logical hierarchy of rules and conditions that can only be in a finite number of states, each state having its own behavior, and its own trigger. Finite state machines are used more frequently in computer games than any other AI technique (Sweetser 2002) because they are (1) simple to program, (2) easy to understand and debug, and (3) generally enough to be used for any problem (Rabin 2002). One drawback is that using simple finite state machines leads to predictability of game AI. When using fuzzy states and fuzzy transitions rather than a finite set of states and transitions, a variety of different responses to a given set of stimuli can be generated, consequently producing ‘unpredictable’ behavior.

Scripting and state machines are deterministic AI techniques that require the developer to hard-code all aspects of the character behavior. Therefore, the developer has to anticipate all possible game states and situations. Most modern computer games have hundreds of different parameters and scenarios affecting the AI’s behavior. Machine learning techniques may be able to cover the search space in computer games and efficiently search for successful combinations of parameters.

2.2 MACHINE LEARNING AND ADAPTATION IN GAMES

The process of learning in games generally implies the adaptation of behavior for opponent players in order to improve performance. Note that the terms online and offline used in 1.2 apply to the timing of when learning is achieved, i.e. respectively during gameplay against humans or during self-play, and tell nothing on how learning is achieved. Manslow (2002) distinguishes between direct and indirect learning:

Indirect adaptation: Indirect adaptation occurs when alternations are made to certain aspects of behavior based on statistics in the game world. The decision as to what statistics are extracted and their interpretation in terms of necessary changes in behavior are all made by the AI designer (Manslow 2002). The role of the learning mechanism is thus restricted to extracting information from the game world, and plays no direct part in changing the behavior. Indirect adaptation is effective, because its extensive use of prior knowledge makes the learning mechanism simple, highly efficient, and easy to control, test and validate (Manslow 2002). The technique "dynamic difficulty settings" used in MAX PAYNE 2 (Figure 1) is an example of indirect learning in a computer game.

Direct Adaptation: Direct adaptation applies optimization or reinforcement learning algorithms to directly change the AI's behavior on the basis of assessments of its performance in the game world (Manslow 2002). Basically, the learning algorithms searches for AI parameters that offer the best performance, i.e. search for the best behavior. Direct adaptation is generally not efficient and hard to control, making it difficult to debug. Furthermore, it is difficult to design an appropriate performance measure (fitness function). However, direct adaptation has the major advantage of not limiting the opponents' AI behavior and it requires minimal human direction. BLACK & WHITE, illustrated in Figure 1, proves that direct learning can successfully be applied in computer games (Barnes 2002).

Game companies are cautious in using machine learning techniques for their computer game AI because (1) these systems can learn the wrong lessons, (2) they are often very demanding in terms of processing time and (3) they can be difficult to tune and tweak to achieve the desired results. Still, machine learning has the potential of delivering more challenging AI the gaming community craves. Machine learning algorithms can be used to adapt to conditions that cannot be anticipated prior to a game's release, such as the particular styles, tastes, and dispositions of individual players (Manslow 2002). When used correctly, machine learning will help make games more robust and resilient to player exploits and will change the way in which games are played by forcing the player to continually search for new strategies to defeat the AI, rather than perfecting a single technique. Developers can also use machine learning techniques to generate sophisticated AI's 'in-house' before shipping the game, without having to invest significant (human) resources.

In conclusion, learning is expected to be the next big thing in computer game development (Rabin 2002; 2004) and developers are moving away from a hard-coded, rules based approach toward more flexible AI engines based on adaptive technologies e.g., decision trees, neural networks and evolutionary algorithms.

Figure 1: MAX PAYNE 2 (left picture) introduces something called dynamic difficulty settings. Information from the game world is extracted to estimate a player's level of skill and the opponent AI difficulty is set in response. Creatures in BLACK & WHITE (right picture) are trained through the process of rewards and punishments using a reinforcement learning algorithm.

2.3 INTRODUCTION TO EVOLUTIONARY ALGORITHMS IN GAMES

Evolutionary algorithms (EAs) are the broad name given to a group of optimization and search algorithms that are based on the principle of biological evolution. They include genetic algorithms (Goldberg 1989), classifier systems (Goldberg 1989) and genetic programming (Koza 1992).

Traditionally, optimization techniques start with one potential solution to a problem and then gradually adapt this solution in order to reach an optimum. EAs work with a population of solutions. These solutions are often encoded and are then called chromosomes. Each gene in a chromosome represents a variable or aspect of the solution. The chromosomes in the population are assigned a fitness value. The fitness value indicates how successful this solution is in solving the problem, compared to other solutions in the population. To generate new solutions, the EA applies genetic operators to existing solutions. Genetic operators that need only one parent' solution are called mutation operators. Mutation takes one chromosome as parent and inserts, deletes or replaces genes to produce a new child. Genetic operators that need more parents are called crossover operators. Crossover occurs when parent chromosomes are combined to form a child chromosome.

To select parent solutions for a genetic operator, a selection mechanism is applied that is biased to select the fittest solutions. The production of new individuals, called the evolution process, continues until some predefined goal is reached. New solutions either replace existing solutions, or are inserted in a new population. The result is a population of individuals that gradually adapt themselves to the constraints of their digital environments, in effect evolving over time. The fittest individual in the population is considered to be the sought solution to the problem.

EAs are robust search methods i.e. they work well in many different environments and on many different problems. They search effectively in large, complex, or poorly understood search spaces. Once an appropriate representation and fitness function is devised, EAs can be a powerful tool for problems featuring large numbers of variables and chaotic interactions, such as computer games. To our knowledge, EAs have never been used online in commercial computer games. Developers have disregarded EAs because they tend to be computationally expensive and generally produce ineffective behavior. Another drawback is that EAs are not guaranteed to find a good solution, not even a mediocre one. However, EAs have been sporadically used offline in simpler computer games (Demasi & Cruz 2002).

Chapter 3

REAL-TIME STRATEGY GAMES

This Chapter will address our first research objective namely selecting a flexible, state-of-the-art real-time strategy game for our experimental research. We will introduce the real-time strategy genre in 3.1 and address important aspects to its artificial intelligence in 3.2. In 3.3 we will highlight the selection criteria for our game environment. We will introduce Stratagus, the selected game environment, in 3.4. This Chapter is concluded in Section 3.5.

3.1 INTRODUCTION TO REAL-TIME STRATEGY GAMES

Today's RTS games are simple military simulations that require the player to control armies (consisting of different types of units), and defeat all opposing forces. In most RTS games, the key to winning lies in efficiently collecting and managing resources, and appropriately distributing these resources over the various game elements. Typical game elements in RTS games include the construction of buildings, the research of new technologies, and combat. DUNE 2 (Figure 2) is considered as the first RTS game. The genre name was invented by Westwood's Brett Sperry. At first they wanted to classify this game as a war- or strategy game, but Sperry was concerned this might scare players away because of the tremendous complexity in conventional war- and strategy games. Sperry justifies his choice for the name by saying: "Before 1992, war games and strategy games were very much niche markets, so my fears were justified. But in the end, it was best to call it an RTS because that is exactly what it was." The term real-time in the genre name aims at the fact that game time in DUNE 2 progresses at a predefined rate. However, many serious strategy gamers disagree with the use of the word strategy in RTS, arguing that RTS games are nothing more than a cheap imitation of turn-based games because of the tendency of RTS games to devolve into 'clickfests' in which the player who is faster with the mouse generally wins, because they can give orders at a faster rate (Geryk 1998).

Since DUNE 2, plenty of new RTS games were published. In 1994 Blizzard released WARCRAFT, a RTS game set in the realm of a fantasy. Its sequel, WARCRAFT II (1995), would end up being one of the biggest successes the RTS genre has ever seen. The game had a long replay value since Blizzard released a version supporting Windows 95/98 in 1999. This is remarkable because games tend to age fast. New RTS games since WARCRAFT II brought the genre to a higher level, but mainly in terms of graphics and sounds (Figure 2) and not in terms of challenging gameplay.

Figure 2: DUNE 2 (left picture) was the first RTS game ever. Electronic Arts (2003) provides gamers with a realistic perspective on modern warfare in COMMAND & CONQUER GENERALS (right picture).

3.2 ARTIFICIAL INTELLIGENCE IN REAL-TIME STRATEGY GAMES

AI has always been very important feature in strategy games, as strategy games cannot rely on graphics alone and requires good AI to even be playable (Tozour 2002a). Planning for military success in a RTS game can be divided into two separate categories: strategies and tactics. While tactics cover small-scale interactions, such as scouting the battlefield or capturing an enemy city, strategies are all encompassing (Ramsey 2004). Generally, the most valued strategic principles are unity of command (desire for one central leader), control of an objective (having a battle plan and sticking to it), flexibility (the ability to change battle plans), economy of force (divide forces and resources appropriately among potential conflicts), initiative and mass (Dunningan 2003). Some developers argue that a well-structured multi-tiered AI layer in combination with goal-directed reasoning is already fit to tackle some of these real-world military ideologies.

Ramsey (2004) proposes a Multi-Tiered AI Framework, where different levels of managers control the AI, this allowing 'grand strategic decisions' to be made by AI at a higher level, which then has the corresponding manager execute the task. EMPIRE EARTH by Stainless Steel Studios, arguably the game with the most successful RTS AI up to now, decomposed the AI into the following managers:

- Build manager: responsible for placement of structures and towns. Most buildings have requirements on where they can and cannot be placed.
- Unit manager: keeps track of what units are in training at various buildings, monitors the computer player's population limit and prioritizes unit requests.
- Resource manager: responsible for tasking citizens to gather resources in response to requests from both the unit and build managers. This component is also responsible for the expansion to new resource sites.
- Research manager: the research manager examines technologies and selects them based on their usefulness and cost.
- Combat manager: responsible for directing military units on the battlefield. It requests units to be trained via the unit manager and deploys them in whatever offensive or defensive position is beneficial.
- Civilization manager: coordination between build, unit, resource and research managers. It handles player expansion, spending limits, building and units upgrade.

Forward reasoning is impractical in a (RTS) game environment because the sheer number of possible moves from any state is prohibitive (Harmon 2002). Therefore, goal-directed (backward) reasoning is preferred over forward reasoning. While the ultimate goal of an RTS game should be to win the game, this goal is too complicated to address directly. The key is to decompose this goal into sub goals. Sub goals for instance could be to 'expand the base' or 'disable the opponent's resource gathering'.

For AI designers RTS games offer many challenges such as resource management, robust terrain analysis, opponent modeling, influence mapping (Woodcock 2002), the utilization of effective tactics and strategies and more. Providing the computer opponent AI with a variety of subtle and complex tactics will greatly enhance the user's sense of challenge and enjoyment (Kent 2004). Providing it with 'intelligent' adaptive behavior would be nothing less than a revolution.

3.3 SELECTION REQUIREMENTS FOR THE GAME ENVIRONMENT

For our experiments we needed to implement machine learning techniques in a RTS game environment. In our search for an appropriate environment, we took the following list of requirements into account:

1) The game environment required being easily accessible and changeable
2) The game environment should include a scripting language, preferably with a sophisticated AI API, able to support learning techniques.
3) Preferably experiments in the environment should be fast.
4) The game environment must be state-of-the-art in terms of gameplay, meaning that game will have to incorporate non-trivial AI.

At first we investigated the possibility of using commercial computer games such as COMMAND & CONQUER GENERALS or EMPIRE EARTH. Modern commercial games are perfect for research in aspect to their realistic environments and non-trivial AI. Unfortunately, most game companies don't leave scripting hooks in the AI engine to allow academics to build their own AI mods, mainly because developers don't have enough time or simply feel it just isn't worth the effort (Woodcock 2003). Although some commercial computer games do include editors to change game AI, the process of doing this is either too tedious or the possibilities are just too scarce.

We then turned to open-source game engines such as Michel Buro's ORTS (2003), the FREECNC engine and finally Stratagus, formerly known as FREECRAFT. We found Stratagus to be the most stable and appropriate engine for our experiments.

3.4 STRATAGUS

Stratagus is a sophisticated RTS engine that can be used to build real-time strategy games similar to WARCRAFT II, COMMAND & CONQUER, STARCRAFT, AGES OF EMPIRES, etc. It successfully runs under GNU/Linux, BSD, BeOS, MacOS/X, MacOS/Darwin and Windows. Stratagus uses an AI manager, written in low-level C code, along with scripts to control the AI opponents. Stratagus includes several scripted AI opponents, each focusing on different strategies such as attacking over land, sea or air.

Scripts in Stratagus are defined in the high-level scripting language LUA, currently one of the most popular scripting languages for games. LUA is a powerful but surprisingly comfortable scripting language and is perfectly able to implement sophisticated AI techniques such as dynamic scripting and evolutionary algorithms.

Stratagus already incorporates useful features such as a fast forward mode where graphics are partially turned off, resulting in fast experiments (a typical game between two computer controlled armies takes about 1 to 3 minutes). The implementation of an automated self-play environment (machine versus machine) turned out to be easy. During and after the game we can easily access numerous game related data such as time elapsed before winning, the number of killed units or the number of units lost etc. which is useful when designing a performance measure.

Initially Stratagus was developed as an open source alternative to WARCRAFT, hence the former name FREECRAFT. Currently there are many games built on top of the Stratagus engine. We chose the game WARGUS with Stratagus as its underlying engine as the test-bed for our experiments. From this point we will refer to WARGUS as the game used for the experiments. WARGUS is not a stand-alone game but a mod that implements a WARCRAFT II clone. WARGUS is not completely true to the original WARCRAFT II experience because it uses a different engine. However, WARGUS is close enough to the original that WARCRAFT II strategies, which are collected in numerous strategy guides available on the Internet, are applicable to WARGUS.

The fact that Blizzard found it worthwhile to re-release WARCRAFT II four years after the game was first released shows that, despite inferior graphics and sounds, apparently WARCRAFT II was still popular among gamers. Supposedly, the gaming community has become indifferent to the incremental improvements in graphics and sounds in newer RTS games and is demanding a more challenging AI.

3.5 CHAPTER CONCLUSION

In this Chapter we addressed the first objectives as listed in Section 1.4: we selected the game WARGUS with Stratagus as its underlying engine as the RTS environment for our experimental research. This setup meets all 4 selection requirements. First, Stratagus is a sophisticated RTS engine and is easily changeable and extensible. Secondly, all game content (including the AI) is defined in LUA scripts. LUA is a powerful scripting language, allowing the implementation of machine learning techniques such as dynamic scripting and evolutionary algorithms. Thirdly, experiments in the engine are fast because graphics can be partially turned off. Finally, the game WARCRAFT II – and thus WARGUS- can still be considered as state-of-the-art in terms of gameplay.

Chapter 4

DYNAMIC SCRIPTING FOR REAL-TIME STRATEGY GAMES

In 4.1 we will describe the basic principles of the dynamic scripting and how its implementation differs in the RTS genre as opposed to its original implementation in CRPG. In 4.2 we will discuss how we implemented dynamic scripting in WARGUS, the selected game for the experiments. In 4.3 we will present the reader the results for the conducted experiments. We will finish this Chapter with a conclusion in 4.4.

4.1 DYNAMIC SCRIPTING APPLIED IN REAL-TIME STRATEGY GAMES

Dynamic scripting (Spronck *et al.*, 2003) is a direct online learning technique for commercial computer games. The learning mechanism in the dynamic scripting technique is inspired by reinforcement learning techniques (Russell and Norvig 1995). It has been adapted for use in games because regular reinforcement learning techniques do not meet the requirement of efficiency (Manslow 2002). In dynamic scripting an adaptive rulebase is used for the generation of intelligent opponents on the fly. Rules are extracted from a rulebase to form a new script that controls the dynamic players' behavior. The probability that a rule is selected for a script is proportional to a weight value that is associated with each rule i.e., rules with larger weights have a higher probability of being selected. The idea behind dynamic scripting is that the rulebase adapts by changing the weight values to reflect the success or failure rate of the corresponding rules in scripts. After every game, the weights of rules employed in the combat are increased when having a positive contribution to the outcome and decreased when having a negative contribution. The remaining rules get updated so that the total weight of the rules in the rulebase remains unchanged. Through the process of punishments and rewards, the dynamic AI will gradually adapt its strategy to the players. Figure 3 illustrates the dynamic scripting process in RTS games.

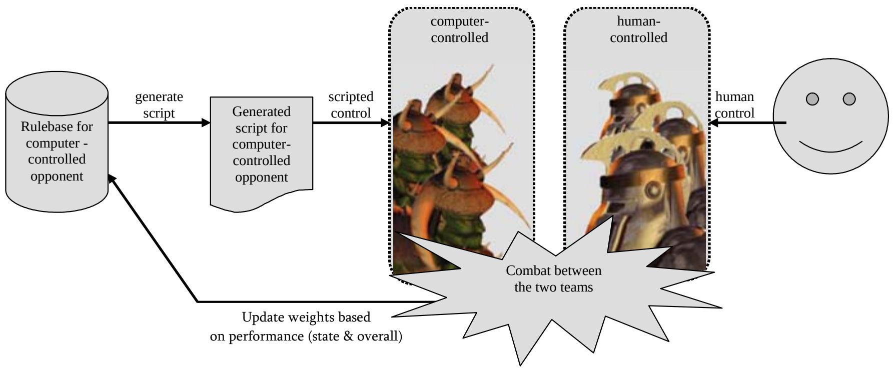

Rulebase for computer - controlled opponent
generate script
Generated script for computer-controlled opponent
scripted control
computer-controlled
human-controlled
human control
Combat between the two teams
Update weights based on performance (state & overall)

Figure 3: The dynamic scripting technique applied in a RTS game. The rulebase generates a new script at the start of a game. After each game, the weights in the rulebase are adapted to reflect the results of the game.

Spronck et al.'s (2003) CRPG implementation of dynamic scripting cannot be transferred to the RTS genre unchanged. We added some new features to the original implementation of dynamic scripting for CRPGs to enable it to work for RTS games. Specifically we introduced 'states' and 'state evaluations'.

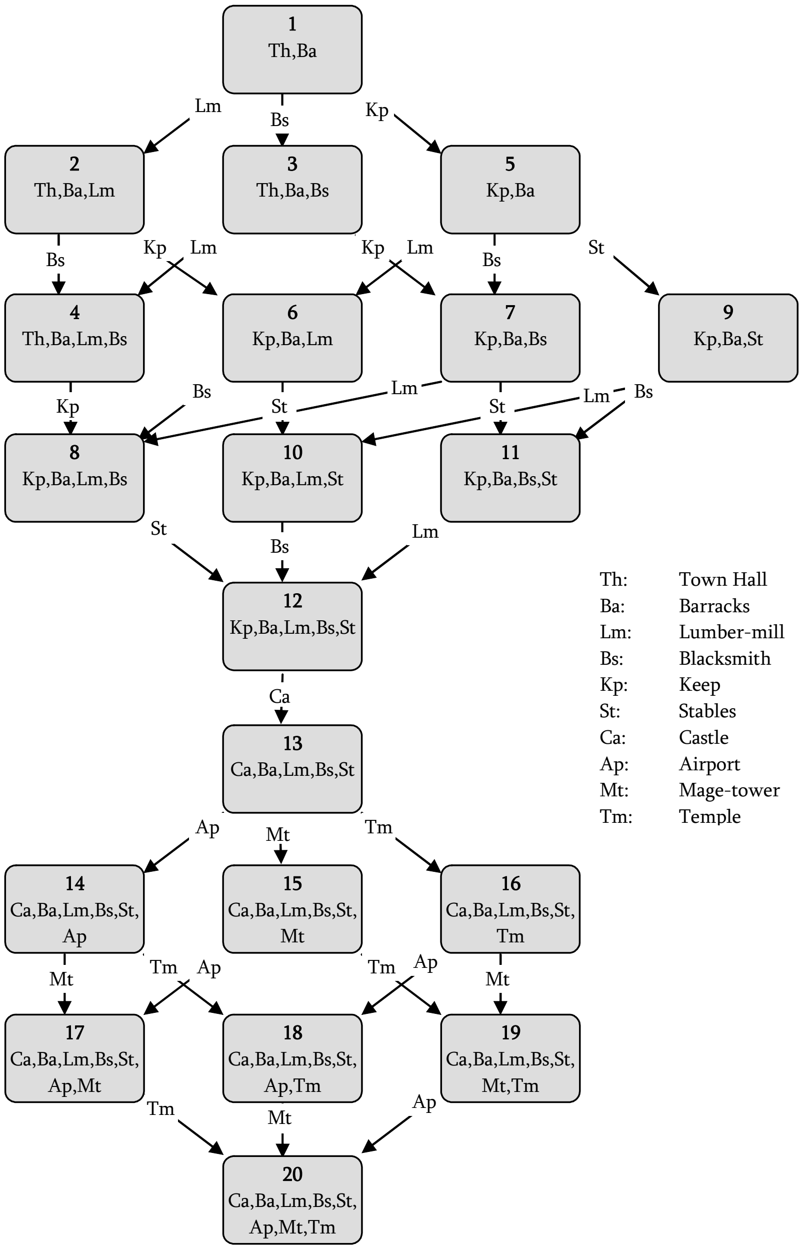
A typical RTS skirmish can be divided into phases. The first phases are traditionally used to get the economy going as well as setting up a base defense. Gradually players will improve their civilization and as time progresses players usually tend to act more offensively. Careful timing of military activities in RTS games is essential to achieve success, e.g., attacking with weak units might be the only viable choice in early game phases, in later game phases, when strong units are available, usually weak units will have become useless. We decided to structure these phases into game states. While dynamic scripting for CRPGs employs different rulebases for different opponent types in the game (Spronck *et al.* 2003), our RTS implementation of dynamic scripting employs different rulebases for the different states of the game. These states will then roughly reflect all distinct game situations in a particular RTS game. In the original dynamic scripting implementation for CRPGs, the success of a rule is reflected by a single weight. This will no longer suffice since we want judge rules based on the temporal state of the game. We need to associate rules with several weights. More specifically, we need to assign each rule with one weight per state (per rulebase).

While dynamic scripting for CRPGs executes weight updates based on an evaluation of a fight, our RTS implementation of dynamic scripting executes weight updates based on both an evaluation of the performance of the game AI during the whole game (called the "overall fitness"), and on an evaluation of the performance of the game AI between state changes (called the "state fitness"). As such, the weight-update function is based on the state fitness, combined with the overall fitness. The use of both evaluations for the weight-updates increases the efficiency of the learning mechanism (Manslow 2004).

4.2 DYNAMIC SCRIPTING IMPLEMENTED IN WARGUS

4.2.1 States and State Evaluations

Manslow (2002) argues that knowledge about the game and the lessons you want AI to learn must be taken into account to structure the state space. For most games it is very hard to draw a line between different phases and determine how many states are appropriate. However, WARGUS has clear distinction in eras. The player starts with a 'town hall'. A 'town hall' can be upgraded to a 'keep', and a 'keep' in its turn can be upgraded to a 'castle'. After doing each of these upgrades, many new build options are available to the player. In preliminary experiments we used three different states. Ideally these would correspond to the town hall, keep and castle era. However, three states proved to be insufficient because of the structure dependencies incorporated this RTS game. In order to upgrade to castle, a list of other buildings are required. So unless we obligate the construction of these dependency buildings, we could never be sure if moving to another era was possible. We felt it was best to avoid compulsory behavior, because eventually the opposing human player will recognize and exploit this. Taking all dependency buildings into account, we found that the 'natural' number of states was 20 (Figure 4). Each state corresponds with a set of buildings the player currently possesses and with a set of potential rules it is allowed to choose from. Some of these rules (e.g. building a blacksmith) cause the player to progress to another state. In our implementation for WARGUS, the dynamic scripting technique evaluates the AI's performance for the current state before it moves to a new state. These state evaluations will be used to update the weights for rules in the rulebase for the state in question.



Th:	Town Hall
Ba:	Barracks
Lm:	Lumber-mill
Bs:	Blacksmith
Kp:	Keep
St:	Stables
Ca:	Castle
Ap:	Airport
Mt:	Mage-tower
Tm:	Temple

Figure 4: Game-states for WARGUS. The boxes represent the states. Inside each box we see the buildings the player already possesses. The arrows represent the state transitions for each state. For example, after building a lumber-mill (Lm) in state 1, the player progresses to state 2. In our setup the player always starts with a town hall and barracks.

4.2.2 Rulebase

The initial rulebase we designed for WARGUS included 50 higher-level rules, each of which exists in all states. We expected it to be crucial to regularly launch firm attacks and to have a steady defensive line at all times. For that reason we inserted more military rules in the rulebase compared to other rules (Table 1).

Rule type	Count
Build Rules (e.g. Build new barracks)	12
Economy Rules (e.g. Train more workers to harvest resources)	4
Military Rules (e.g. Attack the enemy)	25
Research Rules (e.g. Upgrade your weapons)	9

Table 1: Numerical relation between the different types of rules in rulebase

A typical rule in the rulebase allows the dynamic player to launch an attack on his opponent. The domain knowledge here lies in the fact that this rule automatically trains the most advanced units available. In WARGUS it is advisable to always attack with the most advanced units available e.g., a knight can slaughter a group of soldiers. Another form of built-in domain knowledge is incorporated in the building rules. It is important to build more than one barrack. On the other hand, it doesn't really make sense to build more than one blacksmith so we prevent the AI from doing this. For a complete overview of all rules in the initial rulebase see Appendix C.

4.2.3 Creating a dynamic script

As mentioned before, each state corresponds with a set of possible rules. The dynamic scripting technique will start with randomly selecting rules for state 1 and will continue doing this until a rule is selected that spawns a state change. When a rule is selected that spawns a state change, from that point on rules will be selected for the new state. To avoid monotone behavior, we restricted each rule to be selected only once for each state. We allowed a maximum of 100 rules per script. At the end of the scripts, a loop is implemented that initiates continuous attacks against the enemy.

The chance that a rule is chosen depends on the weight of that rule for that state. Since we have a total of 20 states and 50 rules in the rulebase, the total number of weights for the entire rulebase amounts to 1000, which may be too many to achieve fast learning. Taking into consideration that not all rules are applicable for certain states, we have narrowed down the average number of selectable rules per state to 30 (with a minimum of 21 and maximum of 42) by setting the weights in the weight table for non-applicable rules to 0. These rules are disregarded in the selection and weight updating procedure for the state in question. This way the AI only selects appropriate rules. Presenting as little information as possible that is as relevant as possible will speed up the learning process (Manslow 2004).

4.2.4 Fitness and Weight-update Functions

The weight-update function is based on two 'fitness' functions; a fitness function evaluating the game as a whole (overall fitness) and a fitness function evaluating all states visited during the game (state fitness). Both fitness functions yield values in the range of [0,1]. Although not always true, the player controlled by dynamic scripting (henceforth called the "dynamic player") normally has lost (all the players buildings and units were destroyed) when it has an overall fitness score lower than 0.5 and the dynamic player has mostly won the game (it destroyed all the opponents buildings and units) when it has an overall fitness score greater than 0.5. The closer the overall fitness is to 0, the greater the defeat was. An overall fitness evaluation close to 1 represents an overwhelming victory. The same applies for the state performance with the slight difference that the state fitness does not represent win or loss, but merely the performance for that state. A bad start could still lead to victory. The 'overall fitness' function *F* for player *d* controlled by dynamic scripting is defined as:

$$F = \begin{cases} \min\left(\frac{S_d}{S_d + S_o}, b\right) & \{d \quad lost\} \\ \max\left(b, \frac{S_d}{S_d + S_o}\right) & \{d \quad won\} \end{cases} \tag{1}$$

In equation (1), S_d represents the score for the dynamic player, S_o represents the dynamic player's opponent, $b \in [0,1]$ is the break-even point. At the break-even point, weights remain unchanged. The state fitness *F* for state $i \in \{0,20\}$, for dynamic player *d*, is formally defined as:

$$F_i = \begin{cases} \frac{S_{d,i}}{S_{d,i} + S_{o,i}} & \{i = 1\} \\ \frac{S_{d,i}}{S_{d,i} + S_{o,i}} - \frac{S_{d,i-1}}{S_{d,i-1} + S_{o,i-1}} & \{i > 1\} \end{cases} \tag{2}$$

In equation (2), $S_{d,x}$ represents the score of the dynamic player after state *x*, and $S_{o,x}$ represents the score of the dynamic player's opponent after state *x*.

The score function is domain-dependent, and should successfully reflect the relative strength of the two players in the game. We defined the score S_x for player *x* as:

$$S_x = 0.7M_x + 0.3B_x \tag{3}$$

In equation (3), M_x represents the military points for player *x*, i.e. the number of points awarded for killing units and destruction of buildings, and B_x represents the building points for player *x*, i.e. the number of points awarded for training armies and construction of buildings.

After each game, the weights of rules employed are updated. The weight-update function translates the fitness functions into weight adaptations for the rules in the script. The weight-update function *W* for the dynamic player is formally defined as:

$$W = \begin{cases} \max\left(W_{\min}, W_{org} - 0.3\frac{b-F}{b}P - 0.7\frac{b-F_i}{b}P\right) & \{F < b\} \\ \min\left(W_{org} + 0.3\frac{F-b}{1-b}R + 0.7\frac{F_i-b}{1-b}R, W_{\max}\right) & \{F \geq b\} \end{cases} \tag{4}$$

In equation (4), *W* is the new weight value, W_{org} is the original weight value, *P* is the maximum penalty, *R* is the maximum reward, W_{max} is the maximum weight value, W_{min} is the minimum weight value, *F* is the overall fitness of the dynamic player, F_i is the state fitness for the dynamic player in state *i*, and *b* is the break-even point. The equation indicates that we prioritize state performance over the overall performance. The reason is that, even if a game is lost, we wish to prevent rules in states where performance is successful from being punished (too much). In our simulation we set *P* to 175, *R* to 200, W_{max} to 1250, W_{min} to 25 and *b* to 0.5.

4.3 EXPERIMENTS

4.3.1 Experimental Setup

With our experiments we aim at proving that the player controlled by the dynamic scripting AI successfully adapts to a static opponent. Ideally we want the AI controlled by dynamic scripting to be resilient to early attacks as well as long lasting battles; therefore we conducted experiments in both a small map and large map. The small map will most likely be decided with fierce battles in a really early stage, whereas the large map allows both players to advance to other eras, producing interesting battles with advanced units (Figure 5).

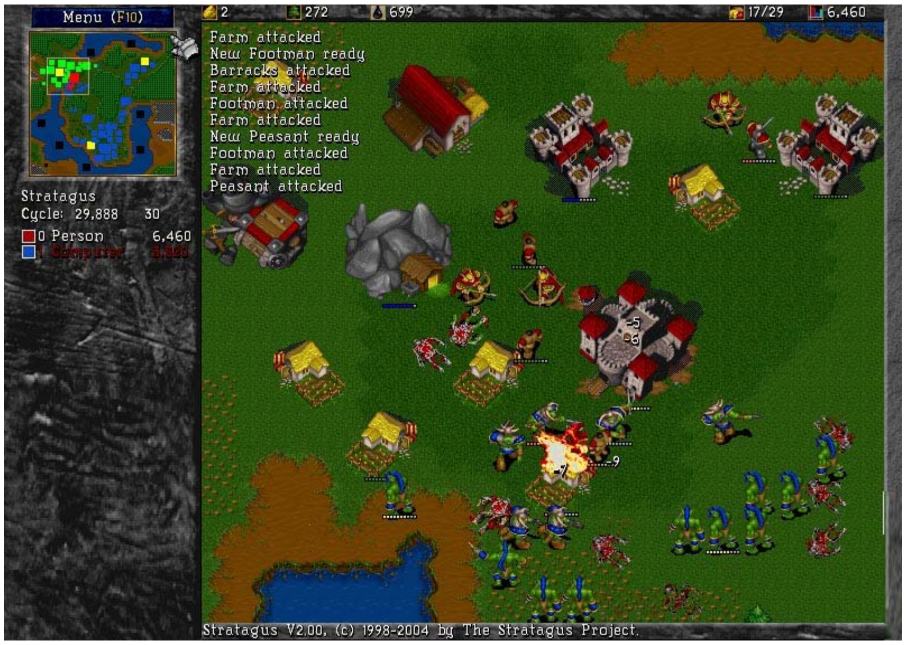
For our first experiment, we used the default land attack AI included with the Stratagus engine as the static opponent AI. We made some moderate improvements (e.g. launching larger offensives) to it because at first the dynamic AI was already outperforming the default land attack AI before any learning could have taken place. The improved land attack AI is an overall balanced strategy focusing on offense, defense and research. It favors ground offenses over air and sea. We employed the improved balanced land attack AI on both the small and large map.

Besides the default land attack AI we also decided to test the dynamic scripting technique against two optimized strategies: the soldier's rush and the knight's rush. The soldier's rush, which we implemented ourselves, aims at overwhelming the opponent with cheap offensive units in the early state of the game. The knight's rush strategy aims at quick technological advancement, launching large offenses as soon as strong offensive units are available. The soldier's rush arguably is most effective on a small map, and the knights' rush on a large map. In summary, the following experimental setups were used:

Name	AI Strategy	Map
Small Balanced Land Attack	Improved default land attack AI	Small Map
Large Balanced Land Attack	Improved default land attack AI	Large Map
Soldier's Rush	Soldier's Rush AI	Small Map
Knight's Rush	Knight's Rush AI	Large Map

Table 2. Experimental setups.

To quantify the relative performance of the dynamic player against the static player, we define two notions of the 'randomization turning point' (RTP) and the 'absolute turning point' (ATP). The RTP is explained as follows: after each game we calculate the average fitness for each of the players over the last ten games. We then use the fitness values over the last ten games to conduct a randomization test (Cohen 1995) with the null hypothesis that both players (dynamic and static) are equally good. The dynamic player is said to outperform the static player at a point when the null hypothesis can be rejected with a probability of 90%. The RTP is the first round in which this is achieved. The ATP is defined as the first game after which a consecutive run of games in which the dynamic player wins is never followed by a longer consecutive run in which the dynamic player loses. Low values for the randomization and absolute turning points indicate good efficiency of dynamic scripting, since they indicate that the opponent player (using dynamic scripting) consistently outperforms the static player within a few games only (Spronck *et al,* 2003). If the dynamic AI is unable to statistically outperform the static player within 100 games, the experiment is stopped and the average fitness is logged. For the Small Balanced Land AI we ran 31 tests. For the Large Balanced Land AI we ran 21 tests. For both the soldier's rush and knight's rush, we ran 10 tests each. The results of these experiments are presented in the next Sections.



```
Farm attacked
New Footman ready
Barracks attacked
Farm attacked
Footman attacked
Farm attacked
New Peasant ready
Footman attacked
Farm attacked
Peasant attacked
```

Figure 5: Screenshot of a battle in WARGUS in the small map 'little ambush' (64x64 tiles). The upper left square in the image above shows an overview of the small map. Because of the relatively small space available, the opponents will be at each other's throats quickly. The second map for our experiments (not illustrated here) is the larger map 'Scandinavian' (128x128 tiles) where longer journeys have to be undertaken to attack the enemy, increasing the chance both players will advance to other eras.

4.3.2 Results

The results for the Small and Large Balanced Land Attack AI presented in Table 3 show that the dynamic scripting technique works in RTS games. With average RTP and ATP values around respectively 50 and 35, the dynamic AI adapts fast to a static opponent. Both the RTP and ATP averages are very similar in both maps. Remarkable are the high outliers in the small map. We will discuss these outliers in the next Section.

	Randomization Test Statistics				Absolute Turning Point Statistics			
Map	Low.	High.	Avg.	Med.	Low.	High.	Avg.	Med.
Small	18	99	50	39	8	91	36	27
Large	19	79	49	47	11	58	34	34

Table 3: Results against the small and large balanced land attack AI. The lowest, highest, average and median values are shown.

The results for the soldier's rush and knight's rush are presented in Table 4. The dynamic scripting was unable to statistically outperform the optimized static AI's within 100 games, resulting in low average fitness scores (AFS). On average dynamic scripting only won approximately 1 out of 100 against the soldier's rush, and 1 out of 50 against the knight's rush.

Soldier's Rush		
Test	Won	AFS
#1	0	0.18
#2	2	0.18
#3	0	0.18
#4	3	0.20
#5	1	0.19
#6	3	0.20
#7	0	0.18
#8	1	0.19
#9	2	0.20
#10	0	0.18
	1.2	**0.19**

Knight's Rush		
Test	Won	AFS
#1	1	0.21
#2	2	0.23
#3	1	0.22
#4	5	0.23
#5	0	0.22
#6	7	0.25
#7	0	0.20
#8	2	0.21
#9	3	0.22
#10	2	0.23
	2.3	**0.22**

Table 4 – Results for the two optimized AI's: the soldier's rush and knight's rush. The numbers in the last row represent the average number of games won, and the average fitness score calculated over all test runs.

4.3.3 Discussion

Although the RTP and ATP averages are very similar for both the Small and Large Balanced Land Attack AI, it can be argued that learning is achieved faster against the Small Balanced Land Attack AI (note the significantly lower median for both performance measures). A typical battle in the small map is decided before either player reaches advanced eras. Consequently weights are updated only for early states. Typical battles in the large map, where early offensive combat is not as decisive, does effect in updating weights for large numbers of states. Therefore, learning is faster in a small map because it involves making changes to a smaller number of states. The fact that the RTP and ATP averages are not significantly lower for the Small Balanced Land Attack AI is most likely due to the high outliers. Unlike in the large map where the dynamic AI has a better chance of recovering after a 'dumb' move, the dynamic AI in the small map has trouble holding on to a winning tactic (Figure 6). Even when the dynamic AI has gained tactical and strategic superiority, an occasional bad start due to randomness will result in total defeat. These are most likely the cause of the high outliers. Novel additions to the dynamic scripting technique, such as a penalty-balancing and a history-fallback mechanism (Spronck *et al*, 2004), enhances the overall performance by preventing a rulebase from deteriorating and reducing the number of outliers.

The results presented in Table 4 clearly indicate that dynamic scripting in the current implementation is not successful in battling optimized strategies. Although dynamic scripting is an adaptive technique, it is still bound to the rules in the rulebase. If the rules offer too few solutions, dynamic scripting is unable to (quickly) discover winning tactics.

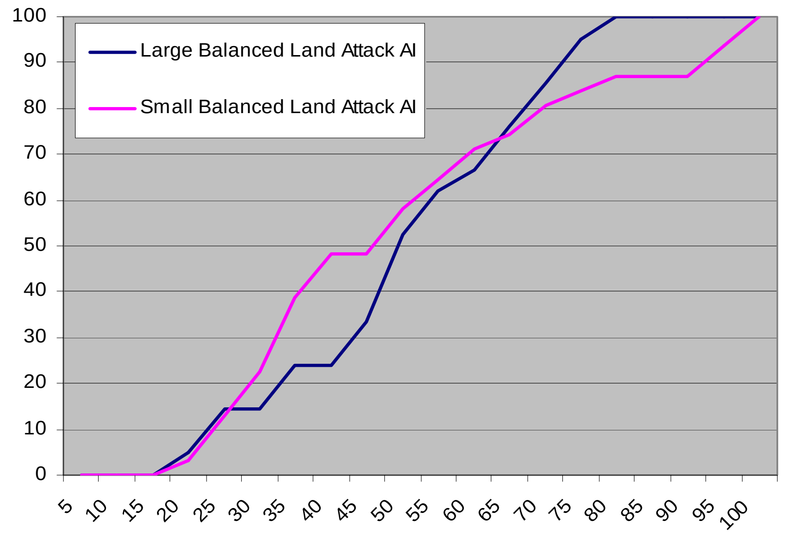


Figure 6: Comparing the average number of games before RTP is reached against the Balanced Land Attack AI for both maps, calculated over all experiments. The x-axis represents the number of games played. The y-axis represents the number of times the RTP has been reached (in percents) after a certain number of games played (e.g. over all experimental runs in the large map, 20% reached the RTP in approximately 35 games). This graph shows that the dynamic AI learns faster in the small map but after considerable training, the dynamic AI is more stable in the large map.

4.4 CHAPTER CONCLUSION

In this Chapter we addressed the second objective as listed in Section 1.4 namely we designed and implemented a method to apply dynamic scripting to RTS games. We extended the original implementation of dynamic scripting (Spronck *et al*, 2003) with states and state evaluations. Against generalized strategies (the improved default land attack AI) dynamic scripting performed well on both the small map and large map. It therefore proved to be resilient to early attacks as well as long lasting battles. Considering the large state space in Wargus, the dynamic scripting technique adapted fast (with RTP averages around 50 and ATP around 35) to its opponent's strategy.

However, with our initial rulebase, dynamic scripting was unable to cope with two optimized AI's (the soldier's rush and the knight's rush). Overcoming extremely optimized strategies as employed by many experienced gamers, can possibly be achieved by creating better rules for the rulebase. Discovering new rules (e.g. changing rule parameters or discovering successful combinations of rules) will be the main focus in the Chapters to come.

Chapter 5

EVOLUTIONARY LEARNING IN REAL-TIME STRATEGY GAMES

In Chapter 4 we noticed that dynamic scripting had trouble coping with optimized strategies. AI's in a RTS game equipped with an evolutionary algorithm can potentially ignore conventional military wisdom and 'think' out-of-the-box. By mimicking the natural process of survival of the fittest and evolution we hope to discover unexpected successful strategies and tactics that can outperform these optimized AI's. In 5.1 and 5.2 we will discuss how evolutionary algorithms can be applied to RTS games in general and in WARGUS specifically. The results of our experiments with the EA in WARGUS are presented in Section 5.3. This Chapter is concluded in Section 5.4.

5.1 EVOLUTIONARY ALGORITHMS APPLIED IN REAL-TIME STRATEGY

When designing an EA for RTS games, the most critical design issues involve the encoding and evaluation. An encoding scheme needs to able to represent any possible solution to the problem, and preferably be designed so that it cannot represent infeasible solutions. Therefore, we give the EA maximal freedom in rule selection and rule parameterization but prevent it from inserting illegal rules into the solution. We do this by using game states that correspond with a set of rules the EA is allowed to choose from.

Designing an appropriate fitness function is essential for the EA to work effectively. Basically the better a chromosome is at solving a specific problem, the higher the fitness score it should receive. An adequate problem definition is therefore crucial when designing the fitness function. In RTS games, the problem can be described as overcoming opposing armies on a specific map. Arguably an overwhelming victory should be awarded a higher fitness than a narrow victory.

Another characteristic that requires special attention when designing an EA for RTS games is the population size. Very determining for the AI's strategy in many RTS games are the building priorities for the AI (e.g. the specific order the AI chooses to construct buildings). The original population should include enough variations in the building priorities to test various strategies and search for an optimal solution. If the population size is chosen too large, the evolution may take too long. However, if it is chosen too small, the EA could converge to a poor solution because of insufficient sampling of the search space.

EAs are initialized by creating a population with a fixed number of sample solutions. New solutions are then played against a static AI and their success is measured. When the population has been filled and all chromosomes have been assigned fitness scores, successful solutions are allowed to breed. The EA will select one of the genetic operators available to the system and then select the appropriate number of child and parent chromosomes. The process of evolution will continue until a certain stopping criterion has been met i.e., when a solution to the problem has been found. Figure 7 illustrates schematically the evolutionary process applied in RTS games.

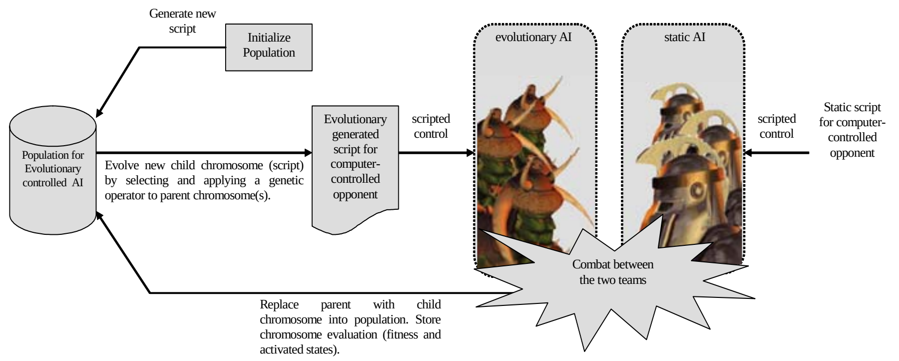

Generate new script
Initialize Population
Population for Evolutionary controlled AI
Evolve new child chromosome (script) by selecting and applying a genetic operator to parent chromosome(s).
Evolutionary generated script for computer-controlled opponent
scripted control
evolutionary AI
static AI
scripted control
Static script for computer-controlled opponent
Combat between the two teams
Replace parent with child chromosome into population. Store chromosome evaluation (fitness and activated states).

Figure 7: The evolutionary process applied in RTS games.

5.2 EVOLUTIONARY ALGORITHM IMPLEMENTED IN WARGUS

5.2.1 Encoding

The genes in a chromosome are grouped into states. A state is activated when the AI has executed at least 1 gene in that state. All chromosomes will at least have state 1 activated while the other states vary depending on the building priority (Figure 4). For WARGUS we encoded 4 types of genes: *build*, *research*, *economy* and *combat* genes.

To construct buildings we introduced build genes. These start with the letter 'B' and are followed by a number ranging from 1 to 12, representing the selected building. Research genes, responsible for researching new technologies to improve civilization, start with the letter 'R' and are followed by a number ranging from 13 to 21. Economy genes are responsible for training workers and start with the letter 'E' followed by the desired number of workers. Military activities are encoded in combat genes. They start with the letter 'C' and a number representing the current state (each state allows fighting with different units). For example, a combat gene in state 1 starts with 'C1', whereas a combat gene in state 20 starts with 'C20'. The first parameter for a combat gene is always the identifier for an army. Stratagus currently supports 10 controllable armies ranging from 0 to 9. The last parameter is always the role of the army: either offensive or defensive. The number of parameters between the first and last vary, depending on the state. For example, state 1 only has one extra parameter (representing the number of soldiers), while state 20 has a total of 6 extra parameters. During the initialization phase these parameters are randomly initialized with a number between 0 and 9. Figure 8 illustrates the design of a chromosome in WARGUS and some example genes. For a complete description of all genes, see Appendix A.

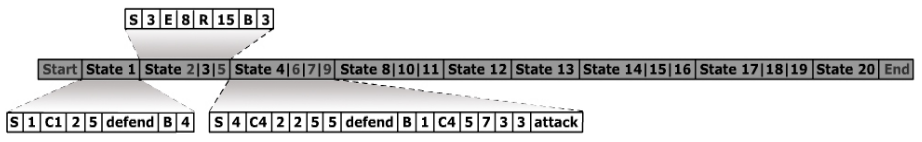

S 3 E 8 R 15 B 3
Start State 1 State 2 3 5 State 4 6 7 9 State 8 10 11 State 12 State 13 State 14 15 16 State 17 18 19 State 20 End
S 1 C1 2 5 defend B 4
S 4 C4 2 2 5 5 defend B 1 C4 5 7 3 3 attack

Figure 8: A chromosome in WARGUS. The gray boxes show the possible states whereas the example genes are listed in the white boxes. The tag 'S' followed by the number 1 implies that the script currently is in state 1. After building a blacksmith with building index 4, we see the script progressing to state 3. Note that the combat gene 'C4' in state 4 has more parameters than the combat gene 'C1' in state 1.

5.2.2 Evaluation

To measure the success of a game AI script represented by a chromosome, the following fitness function *F* for the dynamic player *d*, that yields a value in the range [0,1], is defined:

$$F = \begin{cases} \min\left(\dfrac{GC}{EC} \cdot \dfrac{M_d}{M_d + M_o}, b\right) & \{d \quad lost\} \\ \max\left(b, \dfrac{M_d}{M_d + M_o}\right) & \{d \quad won\} \end{cases} \qquad (5)$$

In equation (5), M_d represents the military points for the dynamic player, M_o represents the military points for the dynamic player's opponent, and *b* is the break-even point. *GC* represents the game cycle, i.e., the time it took before the game is lost by one of the players. *EC* represents the end cycle, i.e. the longest time a game is allowed to continue. When a game reaches the end cycle and neither army has been completely defeated, scores at that time are measured and the game is aborted.

If the evolutionary AI is able to put up a long lasting fight but eventually it still loses, it is probable that this chromosome is close to finding a solution and small changes to the genes might result in a winning chromosome. The factor *GC/EC* ensures that losing solutions that play a long game are awarded higher fitness scores than losing solutions that play a short game.

5.2.3 Genetic Operators

Genetic operators are often designed to fit the specific problem and chromosome design at hand. In WARGUS we designed four genetic operators for the evolution of tactics and strategies in RTS-games: (1) State Crossover, (2) Rule Replace Mutation, (3) Rule Biased Mutation and (4) Randomization. Randomization has a 10% chance of occurring and the remaining genetic operators a 30% chance. We will discuss each of them next.

1) State Crossover: We select 2 parents and check if the selected parents have at least 3 matching activated states for crossover. We make sure that the child chromosome inherits genetic material from both parents to prevent a parent from being copied completely onto the child chromosome. Between two matching states, all states and genes are copied from either parent. This way we prevent the EA from evolving corrupt chromosomes, i.e. illegal state changes. After the last activated state, the remaining part of the chromosome is copied from one of the parents. Figure 9 illustrates an example of a state crossover.

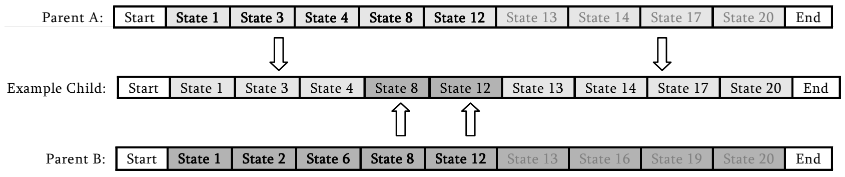


Figure 9: This example illustrates a state crossover. Activate stares are written in bold. The matching activated states between parent A and B are 1, 8 and 12. For this specific example, between 1 and 8, all states are copied from parent A; state 8 is copied from parent B etc.

2) Rule Replace Mutation: Select 1 parent and for every activated state all economy, research or combat rules have a 25% chance of being replaced. Building rules are excluded here both for replacement and as replacement, because these could spawn a state change and could possibly corrupt the chromosome. Genes in inactivated states are ignored as they are considered 'dead' and mutation doesn't really make sense here.

3) Rule Biased Mutation: Select 1 parent and for every activated state the parameters for existing combat or economy rules have a 50% chance of being mutated. The mutation is within a predefined boundary (between a minimum and maximum value). For this genetic operator, we exclude build and research rules. Changing parameters for these specific rule types doesn't make sense and could possibly corrupt the chromosome. We disallowed biased mutation in 'dead' genes.

4) Randomization: Generate a complete new chromosome.

5.2.4 Selection Mechanism

We implemented tournament selection as the selection mechanism. Tournament selection randomly chooses *M* 'winning' chromosomes out of *N* to become parents. The higher the value of *N*, the higher the selection pressure, or in other words, the lower the value of *N*, the more diverse the selection will be (Buckland 2004). This method will most likely select good solutions and prevent early convergence. Since little calculation is involved, this mechanism is also computationally fast.

Many other selection methods exist such as roulette wheel selection, linear ranking or stochastic remainder selection. For WARGUS we found tournament selection to be the most appropriate selection mechanism because (1) it is easy to implement, (2) it most likely will select good solutions, and (3) when choosing a small value for *N* it will prevent the population from converging too soon. In our implementation we set *N* to 3 and *M* to 1, in other words we choose 3 chromosomes and then pick the best solution to become parent.

Since we replace existing chromosomes in the population, we also need a replacement mechanism for which we chose size-3 crowding (Goldberg 1989). We discard the worst solution among the 3 selected chromosomes.

5.2.5 Stop Criteria

If a fitness score exceeds a desired value, a solution has been found. In WARGUS, a fitness score higher than 0.7 almost certainly represents a convincing victory. Since there is no guarantee an EA will find such a solution, we stop the EA after a fixed number of runs (a run equals the generation of a new evolved chromosome). When either stop-criterion (fitness- or run-stop criterion) has been met, the process logs the best solution, resets the population and starts a new search.

5.3 EXPERIMENTS

5.3.1 Experimental Setup

In our experiments we used the soldier's rush and the knight's rush as static AI's since these proved to be problematic for the dynamic scripting technique. We set the population size to 50. This is a fairly small population size, but the search space in WARGUS is not very large and initializing the population with hundreds of solutions is not necessary.

The fitness stop criterion was set to 0.75 and 0.7 for respectively the soldier's rush and the knight's rush. Since we expected evolution to take longer against the knight's rush in the large map, a fitness score higher than 0.7 will suffice. For the soldier's rush we raised the standard to 0.75. Games against the soldier's rush never take long, therefore the EA has ample opportunity to search for a more optimal solution in a relative short time. Our run stop-criterion was set to 250 because preliminary experiments showed that the EA was able to discover solutions within 250 runs.

5.3.2 Results

The EA rapidly discovered solutions. We repeated experiments until we found 10 solutions for both setups. Almost all evolutionary experiments ended before 250 runs with fitness scores exceeding 0.7 in the large map and 0.75 in the small map (Table 5). We can therefore conclude that our EA is able to discover new tactics and strategies to deal with optimized AI's that dynamic scripting was unable to defeat.

	Results fitness scores			
AI	Low.	High.	Avg.	>250
Soldier' Rush	0.73	0.85	0.78	2
Knight's Rush	0.71	0.84	0.75	0

Table 5: The fitness scores for the solutions found by the EA. Respectively the lowest-, highest-, and average fitness scores are shown. The column on the far right lists the number of times an experimental run was stopped by the run-stop criterion, i.e. the desired fitness was not met.

5.3.3 Discussion

We examined the 10 solutions for both setups. As expected, the battle in the small map never took long. Most solutions found by EA included only two activated states. Remarkable was the fact that in 8 out of 10 solutions, the EA chose to first build a blacksmith very early in the game. Furthermore, as soon as the EA reached state 3 (after building a blacksmith) it selected at least 2 out of the 3 possible research advancements. Basically the strategy behind these 8 solutions is to keep a steady line of defense at all times, build a blacksmith as fast as possible, research better weaponry and armor and conclude with large offenses. The remaining 2 solutions overwhelmed the enemy with sheer numbers.

The solutions in the large map offered more genetic diversity and battles took longer compared to battles on the small map (on average 5 or 6 states were activated). Still, we were able to recognize some obvious patterns in the 10 winning chromosomes. A common building order, as employed in 7 out of 10 solutions, was to build a blacksmith, a lumber mill, upgrade to keep and a stable in that precise order.

Two solutions preferred to reach state 11 really fast. This state is special, since it is the first state that allows fighting with advanced units such as knights. A knight arguably is the most powerful unit in the game and the sooner the AI is able to train knights, the higher its chances are for winning the game. That is why in many solutions, whenever the evolutionary AI was in the proximity of a state able to train knights, it progressed to that state really fast.

Boosting up the economy by building additional resource sites and training large number of workers was clearly present in all solutions for the large map.

Another interesting fact is that the evolutionary AI used lots of catapults. This is surprising because most strategy guides for WARCRAFT II tell us that catapults are generally inferior units because of their high costs and high vulnerability. We expect that their impressive damaging abilities and large range make them effective for both defensive and offensive purposes, especially against tightly packed armies, such as large groups of knights.

5.4 CHAPTER CONCLUSION

In this Chapter we addressed the third objective as listed in 1.4: we implemented an EA in Stratagus that successfully evolved chromosomes that were able to beat two optimized AI's (the soldier's rush and the knight's rush). In all evolutionary searches it offered solutions with fitness scores higher than 0.7 and almost always in less than 250 runs. We were able to recognize several strategies and tactics encoded in the chromosomes for both setups. The focus of the next Chapter is to translate these discovered tactics and strategies into improved rules for the dynamic scripting rulebase.

Chapter 6

IMPROVING THE RULEBASE FOR DYNAMIC SCRIPTING

In Chapter 5 we used an EA to discover new tactics and strategies for a RTS game. This Chapter will deal with the translation of these discovered tactics and strategies into rules for the dynamic scripting rulebase. In 6.1 we will explain how we improved the rulebase. In 6.2 we will discuss the experiments with the new rulebase. We will conclude this Chapter in Section 6.3.

6.1 IMPROVING THE DYNAMIC SCRIPTING RULEBASE

In this Chapter we will discuss how we created new rules based on the solutions found by the EA, in order to improve the dynamic scripting rulebase. We aim at proving that the new rulebase will outperform the two optimized AI's (the soldier's rush and the knight's rush AI's) or at least perform better compared to the old rulebase while being at least equal in performance against the non-optimized AI's.

We closely examined and discussed all discovered solutions in Section 5.3.3. Based on our discoveries we decided to make five changes to the old rulebase namely:

1) We recognized a very obvious pattern in most solutions found against the soldier's rush. The AI first built a blacksmith, then researched better weaponry and armor, and finally overwhelmed the enemy with heavily armed soldiers. The first new rule we added under the name 'AntiSoldiersRush', did exactly that.

2) In almost all solutions against the knight's rush, we observed that the EA preferred to train advanced units as fast as possible. This inspired us to create another rule. Whenever the AI was 'one building away' from training advanced units, our second new rule, when selected, constructed this building and then attacked with advanced units.

3) We also learned from solutions found against the knight's rush that boosting the economy by expanding to new resource sites is very important to achieve game success. The original rulebase already offered numerous opportunities for base expansion. However, during experiments with the old rulebase we noticed that new resource sites were often easily destroyed by the opponent AI (Figure 10). Therefore, these rules were often assigned low weights. When we had a closer look at the solutions found by the EA, we saw that the EA first organized its defenses before building a new base. The lesson we should learn from this is: only set up new base if you have the means to defend it. This is why we included the training of a defensive army in our new base expansion rule.

4) For our 4th new rule we selected a winning chromosome (in this case against the knight's rush) and copied all encoded actions in activated states directly to the new rule.

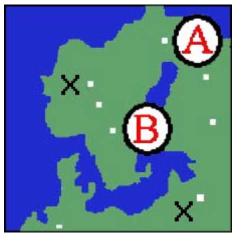


Figure 10: Overview of the large map 'Scandinavian'. When being ordered to expand to a new gold mine, the dynamic AI, which was based on the far left of the map, first chose to expand to gold mine A (the white dot left to A). When having insufficient defensive capabilities, this new base is easily destroyed by nearby opposing forces controlled by the static AI (based in right bottom corner of the map). Arguably expanding to resource site B (the white dot left to B) would be a much safer alternative (out of sight, out of war).

5) For our fifth and final change, we decided not to create an entirely new rule, but to change parameters in existing military rules. We examined all activated states for all chromosomes, and analyzed what type of unit the EA preferred to fight with during a specific temporal state of the game. Based on these statistics we changed parameters in the existing military rules. For instance, we encouraged the use of catapults. The original rulebase hardly included any rules that attacked or defended with large numbers of catapults.

We decided to replace old rules instead of inserting the new rules into the original rulebase. This way we keep the original rulebase size unchanged. The military rules in the original rulebase responsible for air-combat were replaced since these were practically never used in earlier experiments. Besides unused rules, we could also choose to replace unsuccessful rules e.g. rules with low weights.

6.2 EXPERIMENTS

6.2.1 Experimental Setup

With our experiments we aim at proving that the newly optimized rulebase will statistically outperform the two optimized AI's (the soldier's rush and the knight's rush AI's) or at least perform better compared to the old rulebase. For both the soldier's rush and knight's rush, we ran 10 experiments each. We also tested the new rulebase against the Small Balanced Land Attack AI and Large Balanced Land Attack AI. We ran 11 experiments for both. Similar to earlier experiments we will quantify the relative performance of the evolutionary AI against the static player with the 'randomization turning point' (RTP) and the 'absolute turning point' (ATP). If the dynamic AI is unable to statistically outperform the static player within 100 games, the experiments are stopped and the average fitness is logged. For this experiment we set P_{max} to 400, R_{max} to 400. We raised these values compared to earlier experiments to encourage high weights.

6.2.2 Results

The results against the Small Balanced and Large Balanced Land Attack AI with new rulebase are presented in Table 6. The results for the Small Balanced Land Attack AI, with an average ATP of 6, clearly show that dynamic scripting with the new rulebase already is outperforming the static AI before any learning could have taken place. The same more or less applies to the Large Balanced Land Attack AI with an average ATP of 13. The dynamic AI with the old rulebase had RTP averages around 50 and ATP averages around 35. With the new rulebase, the RTP averages dropped to 19 and 24 for respectively the Small and Large Balanced Land Attack AI.

	Randomization Test Statistics				Absolute Turning Point Statistics			
Map	Low.	High.	Avg.	Med.	Low.	High.	Avg.	Med.
Small	10	34	19	14	1	25	6	1
Large	10	61	24	26	1	52	13	10

Table 6: Results against the small and large balanced land attack AI. The lowest, highest, average and median values are shown.

The results against the soldier's rush and the knight's rush with new rulebase are presented in Table 7. The dynamic AI won approximately 1 out of 3 battles (see Table 7) against the soldier's rush, whereas the old rulebase only won 1 out of 100 (see Table 4). The dynamic AI won approximately 1 out of 10 battles (see Table 7) against the knight's rush, whereas the old rulebase only won 1 out of 50 (see Table 4). The average fitness score, calculated over 100 games is approximately 0.3 (see Table 7) for both setups, whereas the average fitness score for the old-rulebase was approximately 0.2 (see Table 4) for both setups. We can therefore conclude that the new rulebase has enabled dynamic scripting to deal better with the optimized AI's.

Soldier's Rush		
Test	Won	AFS
#1	21	0.28
#2	30	0.34
#3	25	0.31
#4	21	0.29
#5	20	0.29
#6	32	0.34
#7	38	0.37
#8	41	0.38
#9	25	0.31
#10	22	0.29
	27.5	**0.32**

Knight's Rush		
Test	Won	AFS
#1	11	0.31
#2	15	0.31
#3	6	0.27
#4	10	0.31
#5	8	0.30
#6	13	0.29
#7	11	0.31
#8	10	0.30
#9	7	0.29
#10	10	0.29
	10.1	**0.30**

Table 7 – Results against the soldier's rush and knight's rush, with the new rulebase. The numbers in the last row represent the average number of games won, and the average fitness score calculated over all test runs.

6.2.3 Discussion

The dynamic scripting AI equipped with the new rulebase is still unable to statistically outperform the two optimized AI's. In order to battle these optimized AI's, there is very little room for variation, requiring the dynamic AI to consistently make a series of appropriate choices. Because of the randomness inherent in the dynamic scripting process, this is unlikely to happen.

However, performance did improve substantially compared to the original rulebase. When examining the weight distribution in the rulebase in more detail, we noticed that new rules were almost always assigned high weights, implying that these new rules proved to be successful and favored by the dynamic scripting technique. The performance increase against the soldier's rush can be subscribed to the new 'AntiSoldiersRush' rule, which had huge weights assigned to it in every experiment. This rule is extremely effective against the soldier's rush to the extent that learning to quickly choose it (this rule is already selectable in the first state) will almost certainly bring victory.

Arguably, no such single effective rule exits against the knight's rush, or at least not that early in the game. We also expect the 'AntiKnightsRush' to be very effective when selected early, but the dynamic AI has to make a series of choices, divided over multiple states, before it even is able to trigger this rule. Since multiple states are involved, learning to select this rule is expected to take longer. This is most likely the reason why the performance increase is not as substantial for the knight's rush compared to the soldier's rush.

The two remaining new rules had larger weights compared to their initial values in almost all experiments. However, neither was as successful as the 'AntiSoldiersRush'.

6.3 CHAPTER CONCLUSION

In this Chapter we addressed the fourth and final objective as listed in 1.4: we were able to translate the offline discovered tactics and strategies into rules for the rulebase. Experiments showed that these changes improved performance of dynamic scripting significantly. We can also conclude that this performance increase can be subscribed to the new rules since these had large weights assigned to them in almost all experiments. In particular, the new 'AntiSoldiersRush' rule had gigantic weights against both the knight and soldier's rush. Apparently the 'AntiSoldiersRush' rule is also effective against the knight's rush, but not as decisive as against the soldier's rush.

Chapter 7

CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

At the end of this thesis we return to the problem statement and research question. In Section 1.3 we presented our problem statement and posed two research questions that should be answered before we address the problem statement. Section 7.1 answers the research questions. In Section 7.2 we will formulate from these answers a reply to the problem statement. We will give future research directions in 7.3.

7.1 ANSWER TO RESEARCH QUESTION

Our research questions were:

Research Question 1: *Is it possible to design and implement an evolutionary algorithm that discovers new tactics and strategies for real-time–strategy games?*

Research Question 2: *Will offline discovered tactics and strategies enhance the performance for the dynamic scripting rulebase?*

In our attempt to answer the research question, we had four objectives:

1) Selecting a flexible, state-of-the-art RTS game-environment for conducting our experimental research in.

The first research objective is discussed in Chapter 3. In accordance with the first research objective, we selected the game WARGUS with Stratagus as its underlying engine as the RTS environment for our experimental research. Stratagus is an appropriate engine for experimental research in game AI. We claim that even today the gameplay for WARGUS can still be considered as state-of-the-art in the RTS genre.

2) Designing and implementing the dynamic scripting technique in the selected RTS game and proving that it works by testing it against several opponent strategies on several maps.

The second research objective is discussed in Chapter 4. In accordance with the second research objective, we modified the original dynamic scripting implementation for CRPG to meet the requirements for RTS games, i.e. we introduced states and state evaluations. We proved that dynamic scripting adapted rapidly to different static AI's on different maps. We additionally discovered that the dynamic scripting technique had trouble coping with extremely optimized AI's.

3) Applying offline learning using an evolutionary algorithm to discover new strategies and tactics in the selected RTS game.

The third research objective is discussed in Chapter 5. In accordance with the third research objective, we implemented an EA for WARGUS that rapidly found solutions to two very optimized AI's. We were able to recognize several strategies and tactics encoded in the chromosomes.

4) Translating offline-discovered strategies and tactics into rules for the rulebase and show that these additions enhance performance for dynamic scripting in the selected RTS game.

The fourth and final research objective is discussed in Chapter 6. In accordance with the fourth research objective, we were able to translate the offline-discovered tactics and strategies into rules for the rulebase and we showed that these changes improved performance for dynamic scripting.

By achieving all our research objectives, we may draw a final conclusion by answering the research questions with an unequivocal yes: (1) it is possible to design and implement an EA that discovers new tactics and strategies for RTS games and (2) these offline discovered tactics and strategies can enhance the dynamic scripting rulebase.

7.2 ANSWER TO PROBLEM STATEMENT

Our problem statement was:

Problem Statement: *To what extent can offline learning techniques be used to improve the rulebase used for dynamic scripting, in order to improve the AI in commercial computer games?*

Taking the answer to the research questions into consideration, we may conclude that offline learning techniques do indeed have the potential to improve the rulebase used for dynamic scripting and consequently improve the AI in commercial computer games. We successfully employed an EA for our offline learning technique. This does not preclude different AI techniques, such as artificial neural networks or decision trees, to achieve good results in this respect.

7.3 RECOMMENDATIONS FOR FUTURE RESEARCH

7.3.1 Improving Dynamic Scripting for Wargus

Dynamic scripting adapted to the static opponents strategy. However, we expect performance to improve (1) when searching for better combinations of learning parameters, (2) when designing more appropriate fitness and update weight functions and (3) when creating better rules for the rulebase.

Experimenting with different values for the maximum penalty, maximum reward, maximum weight value, minimum weight value and break-even point might lead to more efficient learning. The values we selected in our initial setup were chosen intuitively, so more efficient values most likely do exist.

We also expect that further optimizing the fitness and weight update functions will improve performance. Perhaps, the overall score function as stated in 4.2.4 (equation 2) should not reward building points. The goal in a typical RTS game is to destroy all opposing forces. In the end game success is determined by military actions. Arguably, the overall score should not include 'guided rewards' such as points awarded for building actions. The score for the state fitness on the other hand should possibly include even more guided rewards e.g. for resource gathering, research etc. The weight update function (equation 4) indicates we prioritize state performance over the overall performance. We may have overstressed the importance of state evaluations. Employing different values for state and overall evaluations, might achieve faster learning.

Undoubtedly, providing the dynamic scripting algorithm with a very optimized rulebase will have the largest positive contribution to the dynamic scripting performance. We expect performance to increase significantly when using rules for the rulebase that comprises complete tactics e.g., a combination of fine-tuned actions (e.g. build a blacksmith and acquire all related research advancements) rather than single actions (e.g. build a blacksmith). This approach can lead to very effective AI, one that might even be able to tackle optimized AI's such as the soldier's rush and the knight's rush. Our original rulebase included rules with only single actions. Using these 'single rules', and consequently providing the dynamic AI with minimal guidance, does contribute to the diversity for the AI, but has the drawback that dynamic scripting is unable to (quickly) adapt to really optimized AI since these leave practically no room for variation.

Another interesting research objective is testing dynamic scripting against multiple static opponents or even humans. If we want to stage the dynamic AI against multiple opponents, the fitness functions needs to be revised. Currently, the fitness functions are designed for one on one combat.

7.3.2 Improving Evolutionary Algorithm for Wargus

As described in Chapter 5, the EA was able to rapidly find solutions for two supposedly very difficult problems. We may conclude that EAs performance is already extremely high and there is no need for further improvements. Perhaps when testing the EA against even more optimized AI's or multiple opponents, the need for improvements will be more apparent. The EA is potentially improved by applying different EA learning parameters (e.g. larger population size) or by encoding more game options into the chromosome (e.g. include sea warfare in the combat genes).

7.3.3 Improving Translation Algorithm for Wargus

As discussed in 6.3, our translation approach for our fourth rule can be achieved without human intervention and is therefore highly efficient. Additionally, applying this approach we can use the EA to evolve winning chromosomes for distinct opponent strategies on several maps and translate these to rules for the dynamic scripting rulebase. This way we can rebuild the entire rulebase with rules that are fit to battle many different strategies such as rush strategies, defensive strategies, air combat strategies, naval strategies etc. Each rule can be considered as a counter-measure rule for a distinct opponent strategy. Since the rulebase will solely consist of rules that comprise complete tactics i.e., fine-tuned combination of actions, we expect it can produce very effective AI.

7.3.4 Low-level AI Improvements for Stratagus

The low-level AI in Stratagus, e.g. unit AI, robust terrain analyses and pathfinding, has ample room for improvements.

A good local unit performance is crucial to the overall success of the system because generals are overburdened if they have to issue low–level instructions to all objects under their command. Instead, objects are required to handle the most basic problems they face autonomously and quickly (Buro 2003). For instance, in Stratagus the low-level AI for sea units is not very effective. Improving the sea unit AI and including 'sea warfare' rules to the dynamic scripting technique will vastly contribute to the diversity of the AI. Also several other units such as bomb squads and units with magical capabilities exists whose full potential are not utilized by the unit AI.

During our experiments we noticed that in certain situations the AI chose poor locations to construct new resource sites (Figure 10). More robust terrain-analyses for Stratagus e.g. using influence mapping (Woodcock 2002), could provide valuable information for economic planning and prevent situations as illustrated in Figure 10. Robust terrain analysis can also provide the pathfinding algorithm with useful data i.e. to plan an attack route.

Pathfinding in Stratagus is handled with an A* algorithm. Practically all game developers agree that the A* algorithm or some variant is the best answer for both relatively static and dynamic environments and is capable of handling a huge number of possible game designs (Woodcock 2003). However, the Stratagus implementation of the A* is not flexible. For instance, it is impossible to assign armies different attack routes through the higher-level scripts or tell them to attack specific buildings or units. Implementing a more flexible pathfinding algorithm for Stratagus allows AI programmers as well as machine learning techniques to search for smarter tactics and strategies. Armies in COMMAND & CONQUER GENERALS for instance may choose to attack an enemy base using 3 different paths, a frontal, flank or backdoor attack (Electronic Arts 2003). These 3 paths are hard-coded for every map by the designer. Although dynamically determining these way points would increase flexibility even more, this simplistic hard-coded approach already makes it far more difficult for human players to organize their defenses and greatly enhances the sense of challenge.

7.3.5 Machine Learning in Modern Computer Games

Implementing dynamic scripting or other machine learning techniques into modern game environments is certainly another viable research topic. However, most computer games, besides having some mod capabilities, are closed source implying that the implementation of complicated AI techniques is practically impossible. This is unlikely to change unless developers and academics learn to work side by side. Unfortunately, the rift between academics and developers is still far from closed. Developers are under continuing pressure to meet deadlines, and do not find time to implement complicated academic algorithms, let alone to make tools for academics to use in their programs. Academics are therefore forced to invest tremendous efforts in building their own games or turn to often unstable, open-source alternatives. Neither side is served this way while both can benefit from close cooperation.

Developers really do want answers to the harder questions. They are starting to realize that games cannot continually rely on improved graphics and sound alone and that sophisticated AI can produce more interesting gameplay and consequently increase revenues. Academics on the other hand, acknowledge that modern computer games, with ever increasing complexity, are an appropriate tool for integrative human-level AI research. For the near future we expect the rift between developers and academics to shrink. We expect that game companies will soon provide academics with more sophisticated tools to change game AI that enables them to implement machine learning techniques in their games. Consequently we expect academics to show an increasing interest in computer games for experimental research.

Appendix A: Detailed Gene Description

For the EA in Wargus we designed the following gene types:

Build gene := B, Building
Building := [1 ..12]
Example := B,1   ,i.e. build a new town hall

1 :=	BaseExpansion
2 :=	Barracks
3 :=	LumberMill
4 :=	Blacksmith
5 :=	BetterCityCenter
6 :=	STables
7 :=	BestCityCenter
8 :=	Airport
9 :=	MageTower
10 :=	Temple
11 :=	GuardTower
12 :=	CannonTower

Research gene := R, Research
Research := [13..21]
Example := R,13   ,i.e. research better arrows

13 :=	MissileUpgrade
14 :=	ArmorUpgrade
15 :=	WeaponUpgrade
16 :=	CatapultUpgrade
17 :=	MageUpgrade1
18 :=	MageUpgrade2
19 :=	MageUpgrade3
20 :=	MageUpgrade4
21 :=	MageUpgrade5

Economy gene := E, worker_count
worker_count := [0…∞>
Example: E, 10   ,i.e. train an additional 10 workers to harvest resources

Combat := C<current_state>, force_index, {force}, force_role
force_index := [0..9]
force := [unit_type_count, {force}]
unit_type_count := [soldier, shooter, catapult, knight, flyer, mage]
force_role := [attack | defend]
Example: C1, 0,10,attack   ,i.e. assign force 0 to attack with 10 soldiers.

C1,   force_index, soldier, force_role
C2,   force_index, soldier, shooter, force_role
C3,   force_index, soldier, force_role
C4,   force_index, soldier, shooter, catapult, force_role
C5,   force_index, soldier, force_role
C6,   force_index, soldier, shooter, force_role
C7,   force_index, soldier, force_role
C8,   force_index, soldier, shooter, catapult, force_role
C9,   force_index, soldier, force_role
C10, force_index, soldier, shooter, force_role
C11, force_index, soldier, knight, force_role
C12, force_index, soldier, shooter, catapult, knight, force_role
C13, force_index, soldier, shooter, catapult, knight, force_role
C14, force_index, soldier, shooter, catapult, knight, flyer, force_role
C15, force_index, soldier, shooter, catapult, knight, mage, force_role
C16, force_index, soldier, shooter, catapult, knight, force_role
C17, force_index, soldier, shooter, catapult, knight, flyer, mage, force_role
C18, force_index, soldier, shooter, catapult, knight, flyer, force_role
C19, force_index, soldier, shooter, catapult, knight, mage, force_role
C20, force_index, soldier, shooter, catapult, knight, flyer, mage, force_role

Appendix B: AI API Stratagus (http://stratagus.sourceforge.net/)

Stratagus contains the following high-level API commands (which are called from the LUA scripts):

AiNeed: Tells the AI that it should have a unit of this unit-type. The AI builds or trains units in this order of the ai:set/ai:need commands. If the unit or an equivalent unit already exists, the AI does nothing. If the unit is lost, it is automatic rebuild. If the units are requested in wrong order, the AI could hang up. Resources are collected automatic and farms are automatic build, but additional could be requested.

AiSet: This ai:need with a number. Tells the AI that it should have a specified number of a unit of this unit-type. The AI builds or trains units in this order of the ai:set/ai:need commands. If the unit or an equivalent unit already exists, the AI does nothing. If the units are lost, they are automatic rebuild. If the units are requested in wrong order, the AI could hang up. Resources are collected automatic and farms are automatic build, but additional could be requested. In the opposite to ai:need, which always inserts a request, ai:set modifies the last request to the new number.

AiWait: Waits until the *first* request of this unit-type is completed. Don't forget to request a unit-type, before you wait on it.

AiForce: Define a force, what and how many units should belong to a force. Up to 10 forces are currently supported. Force 0 is currently fixed to be the defense force. Forces are automatically sent to a building or unit under attack. If there are unassigned units of requested unit-type, than they are assigned to a force.

AiForceRole: Define the role of a force. Either attack or defend.

AiWaitForce: Wait until a force is complete, the forces are built in force number order. First 0, than 1, last 9.

AiAttackWithForce: Attack the opponent with a force.

AiSleep: Wait some frames, to let the opponent (you) recover.

AiResearch: Let the AI research an upgrade, upgrades are researched in command order. And automatic researched if lost. Building orders have a higher priority. The scriptwriter is responsible for the correct order. AI could hang up when the scriptwriter employs a wrong order.

AiUpgradeTo: Upgrades units or buildings (e.g. upgrade town-hall to keep). Each individual unit or building requires an upgrade command in order to upgrade. The computer automatically searches for the appropriate unit to upgrade.

AiPlayer: Return the player index of the running AI.

Appendix C: Original Rulebase

This is the initially designed rulebase we used for our first experiments with dynamic scripting.

Index	Name	Description
1	BaseExpansion	Expand to new resource site
2	Barracks	Build barracks
3	LumberMill	Build a lumber-mill
4	Blacksmith	Build a blacksmith
5	BetterCityCenter	Upgrade town hall to keep
6	STables	Build sTables
7	BestCityCenter	Upgrade keep to castle
8	Airport	Build an airport
9	MageTower	Build a mage tower
10	Temple	Build a temple
11	GuardTower	Build a guard tower
12	CannonTower	Build a cannon tower
13	MissileUpgrade	Research better arrows
14	ArmorUpgrade	Research better armor
15	WeaponUpgrade	Research better weapons
16	CatapultUpgrade	Research better catapults
17	MageUpgrade1	Research mage spell 1
18	MageUpgrade2	Research mage spell 2
19	MageUpgrade3	Research mage spell 3
20	MageUpgrade4	Research mage spell 4
21	MageUpgrade5	Research mage spell 5
22	LightWorkersExpansion	Train a small amount of new workers
23	NormalWorkersExpansion	Train a medium amount of new workers
24	HeavyWorkersExpansion	Train a large amount of new workers
25	ExtremeWorkersExpansion	Train a extreme large amount of new workers
26	Defense_Squadran	Defend the base with a squadron (smallest-sized force)
27	Defense_Platoon	Defend the base with a platoon (small-sized force)
28	Defense_Battelion	Defend the base with a battalion (medium-sized force)
29	Defense_Company	Defend the base with a company (large-sized force)
30	Defense_Division	Defend the base with a division (largest-sized force)
31	Offense_Squadran	Attack the opponent with a squadron (smallest-sized force)
32	Offense_Platoon	Attack the opponent with a platoon (small-sized force)
33	Offense_Battelion	Attack the opponent with a battalion (medium-sized force)
34	Offense_Company	Attack the opponent with a company (large-sized force)
35	Offense_Division	Attack the opponent with a division (largest-sized force)
36	SoldiersDefense	Defend the base with solely soldiers
37	ShootersDefense	Defend the base with solely archers
38	CatapultDefense	Defend the base with solely catapults
39	KnightsDefense	Defend the base with solely knights
40	MagesDefense	Defend the base with solely mages
41	SoldiersRush	Attack the opponent with solely soldiers
42	ShootersRush	Attack the opponent with solely archers
43	CatapultRush	Attack the opponent with solely catapults
44	KnightsRush	Attack the opponent with solely knights
45	MagesRush	Attack the opponent with solely mages
46	NormalAirDefenseForce	Defend the base with medium-sized air force
47	HeavyAirDefenseForce	Defend the base with large-sized air force
48	NormalAirAttackForce	Attack the opponent with medium-sized air force
49	HeavyAirAttackForce	Attack the opponent with large-sized air force
50	ExtremeAirAttackForce	Attack the opponent with largest-sized air force

Appendix D: Improved Rulebase

The improved rulebase used for our second experiment with dynamic scripting.

Index	Name	Description
1 *	BaseExpansion	Expand to new resource site
2	Barracks	Build barracks
3	LumberMill	Build a lumber-mill
4	Blacksmith	Build a blacksmith
5	BetterCityCenter	Upgrade town hall to keep
6	STables	Build sTables
7	BestCityCenter	Upgrade keep to castle
8	Airport	Build an airport
9	MageTower	Build a mage tower
10	Temple	Build a temple
11	GuardTower	Build a guard tower
12	CannonTower	Build a cannon tower
13	MissileUpgrade	Research better arrows
14	ArmorUpgrade	Research better armor
15	WeaponUpgrade	Research better weapons
16	CatapultUpgrade	Research better catapults
17	MageUpgrade1	Research mage spell 1
18	MageUpgrade2	Research mage spell 2
19	MageUpgrade3	Research mage spell 3
20	MageUpgrade4	Research mage spell 4
21	MageUpgrade5	Research mage spell 5
22	LightWorkersExpansion	Train a small amount of new workers
23	NormalWorkersExpansion	Train a medium amount of new workers
24	HeavyWorkersExpansion	Train a large amount of new workers
25	ExtremeWorkersExpansion	Train a extreme large amount of new workers
26 **	Defense_Squadran	Defend the base with a squadron (smallest-sized force)
27 **	Defense_Platoon	Defend the base with a platoon (small-sized force)
28 **	Defense_Battelion	Defend the base with a battalion (medium-sized force)
29 **	Defense_Company	Defend the base with a company (large-sized force)
30 **	Defense_Division	Defend the base with a division (largest-sized force)
31 **	Offense_Squadran	Attack the opponent with a squadron (smallest-sized force)
32 **	Offense_Platoon	Attack the opponent with a platoon (small-sized force)
33 **	Offense_Battelion	Attack the opponent with a battalion (medium-sized force)
34 **	Offense_Company	Attack the opponent with a company (large-sized force)
35 **	Offense_Division	Attack the opponent with a division (largest-sized force)
36	SoldiersDefense	Defend the base with solely soldiers
37	ShootersDefense	Defend the base with solely archers
38	CatapultDefense	Defend the base with solely catapults
39	KnightsDefense	Defend the base with solely knights
40	MagesDefense	Defend the base with solely mages
41	SoldiersRush	Attack the opponent with solely soldiers
42	ShootersRush	Attack the opponent with solely archers
43	CatapultRush	Attack the opponent with solely catapults
44	KnightsRush	Attack the opponent with solely knights
45	MagesRush	Attack the opponent with solely mages
46 ***	AntiSoldiersRush	- Build a blacksmith - Research better armor - Research better weapons - SoldiersRush
47 ***	AntiKnightsRush	- Build a lumber-mill | blacksmith | sTables - KnightsRush
48 ***	AllYourBaseAreBelongToUs	- Defense_Battalion - BaseExpansion
49 ***	Chromosome_Rule	<genes were copied directly from chromosome>
50 *	Empty	

* Disabled rules ** Parameters modified for existing rules *** New rules added to the rulebase

REFERENCES

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, New York.

Barnes, L. (2002). Testing Undefined Behavior as a Result of Learning. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, 2002, pp. 615-623.

Buckland, M. (2004). Building better Genetic Algorithms. *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, 2002, pp. 649-660.

Buro, M. (2003). RTS Games as Test-Bed for Real-Time AI Research. Department of Computing Science, University of Alberta, Canada. Proceedings of the 7th Joint Conference on Information Science, JCIS 2003,
(Ed. Chen, K., et al.)

Brockington, M. and Darrah, M. (2002). How Not to Implement a Basic Scripting Language. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, pp. 548-554.

Cohen, R.C. (1995). Paragraph 5.3.2: A Randomization of the Paired Sample Test, *Empirical Methods for Artificial Intelligence*, MIT Press, pp. 168-170.

Dunningham J. (2003). *How to make war*, Quill 2003

Demasi, P., Cruz, A.J. de. O. (2002). Online coevolution for action games. Instituto de Matemática, Universidade Federal do Rio de Janeiro

Electronic Arts. (2003). Command & Conquer Generals Worldbuilder.
url: *http://www.generals.ea.com*

Geryk, B (1998). A history of Real-time strategy Games.
url: *http://www.gamespot.com*

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley Publishing Company, Inc.

Harmon, V. (2002). An Economic Approach to Goal-Directed reasoning in an RTS. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, 2002, pp. 402-410.

Kent, T. (2004). Multi-Tiered AI Layers and Terrain Analyses for RTS games. *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, 2002, pp. 447-455.
Koza, J. (1992). *Genetic Programming*, MIT Press, Cambridge, MA

Laird, J. E. van Lent, M (2000). Human-Level AI's Killer Application: Computer Game AI. *Proceedings of AAAI 2000 Fall Symposium on Simulating Human Agents, Technical Report FS-00-03*. AAAI Press 2000, pp. 80-87

Manslow, J. (2002). Learning and Adaptation. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, 2002, pp. 557-566.

Manslow, J. 2004. "Using reinforcement learning to Solve AI Control Problems." *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, pp. 591-601.

Rabin, S. (2002). Implementing a state machine language. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, pp. 314-320

Rabin, S. (2004). *AI Game Programming Wisdom 2*. Charles River Media

Ramsey, M (2004), Designing a Multi-Tiered AI Framework, *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, pp. 457-466

Russel, S. and Norvig, J (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River.

Spronck, P. and Sprinkhuizen-Kuyper, I. and Postma, E. (2003). Online Adaptation of Game Opponent AI in Theory and Practice. *Proceedings of the 4th International Conference on Intelligent Games and Simulation* (GAME-ON 2004) (ed. Q. Mehdi and N. Gough), EUROSIS, pp. 93-100.

Spronck, P. and Sprinkhuizen-Kuyper, I. and Postma, E. (2004). Enhancing the Performance of Dynamic Scripting in Computer Games. *Proceedings of the 4th International Conference on Entertainment Computing (ICEC 2004)*

Sweetser, P. (2002). Current AI in Games: A review. *Australian Journal of Intelligent Information Processing Systems*. Scool of ITEE, University of Queensland

Tozour, P. (2002a). The Evolution of Game AI. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media,pp 3-15

Tozour, P. (2002b). The Perils of AI Scripting. *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, pp. 541-547

Woodcock, S. (2002), Recognizing Strategic Dispositions: Engaging the enemy. *AI Game Programming Wisdom* (ed. S. Rabin). Charles River Media, pp. 221-232.

Woodcock, S. (2003). AI RoundTable Moderator's Report 2003. url: *http://www.gameai.com*