An adaptive AI for real-time strategy games

(HS-IKI-MD-04-101)

Anders Dahlbom (c00andda@student.his.se)

Institutionen för kommunikation och information
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Supervisor: Roger Eriksson
[An adaptive AI for real-time strategy games]
Submitted by Anders Dahlbom to Högskolan i Skövde as a dissertation for the degree of M.Sc., in the School of Humanities and Informatics.

[2004-09-06]
I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: _______________________________________________
An adaptive AI for real-time strategy games

Anders Dahlbom (c00andda@ida.his.se)

Abstract

In real-time strategy (RTS) games, the human player faces tasks such as resource allocation, mission planning, and unit coordination. An Artificial Intelligence (AI) system that acts as an opponent against the human player need to be quite powerful, in order to create one cohesive strategy for victory. Even though the goal for an AI system in a computer game is not to defeat the human player, it might still need to act intelligently and look credible. It might however also need to provide just enough difficulty, so that both novice and expert players appreciates the game. The behavior of computer controlled opponents in RTS games of today has to a large extent been based on static algorithms and structures. Furthermore, the AI in RTS games performs the worst at the strategic level, and many of the problems can be tracked to its static nature. By introducing an adaptive AI at the strategic level, many of the problems could possibly be solved, the illusion of intelligence might be strengthened, and the entertainment value could perhaps be increased.

The aim of this dissertation has been to investigate how dynamic scripting, a technique for achieving adaptation in computer games, possibly could be applied at the strategic level in an RTS game. The dynamic scripting technique proposed by Spronck, et al. (2003), was originally intended for computer role-playing games (CRPGs), where it was used for online creation of scripts to control non-player characters (NPCs). The focus in this dissertation has been to investigate: (1) how the structure of dynamic scripting possibly could be modified to fit the strategic level in an RTS game, (2) how the adaptation time possibly could be lowered, and (3) how the performance of dynamic scripting possibly could be throttled.

A new structure for applying dynamic scripting has been proposed: a goal-rule hierarchy, where goals are used as domain knowledge for selecting rules. A rule is seen as a strategy for achieving a goal, and a goal can in turn be realized by several different rules. The adaptation process operates on the probability of selecting a specific rule as strategy for a specific goal. Rules can be realized by sub-goals, which create a hierarchical system. Further, a rule can be coupled with preconditions, which if false initiates goals with the purpose of fulfilling them. This introduces planning.

Results have shown that it can be more effective, with regard to adaptation time, re-adaptation time, and performance, to have equal punishment and reward factors, or to have higher punishments than rewards, compared to having higher rewards than punishments. It has also been shown that by increasing the learning rate, or including the derivative, both adaptation, and re-adaptation times, can effectively be lowered.

Finally, this dissertation has shown that by applying a fitness-mapping function, the performance of the AI can effectively be throttled. Results have shown that learning rate, and maximum weight setting, also can be used to vary the performance, but not to negative performance levels.

Keywords: real-time strategy games, computer game AI, adaptive AI, RTS game AI.
Acknowledgements

I would like to thank my girlfriend, family, and friends for their support and feedback during the carrying out of this dissertation. I would also like to thank my supervisor, Roger Eriksson, and my examiner, Henrik Engström, for valuable feedback and insightful discussions.

I am also most grateful to Pieter Spronck, for answering my questions concerning the dynamic scripting algorithm.
## Contents

1 **Introduction** .................................................................................................1  
1.1 Difficulty of the AI .........................................................................................1  
1.2 RTS games .........................................................................................................3  
1.3 Problem ............................................................................................................5  
1.4 Dissertation overview ......................................................................................6  

2 **Background** ..............................................................................................7  
2.1 AI in RTS games .............................................................................................7  
  2.1.1 Top-level AI ..............................................................................................7  
  2.1.2 Bottom-level AI .......................................................................................10  
2.2 Adaptive AI in computer games ......................................................................11  
2.3 Dynamic scripting ..........................................................................................12  

3 **Problem description** .............................................................................15  
3.1 Problem delimitation .....................................................................................16  
3.2 Aim and objectives .........................................................................................18  
3.3 Expected results .............................................................................................18  

4 **Focus of investigations** ........................................................................19  
4.1 Strategic level dynamic scripting .................................................................19  
4.2 Problems in dynamic scripting ......................................................................21  
  4.2.1 Punishment or reward .............................................................................22  
  4.2.2 Learning rate ..........................................................................................22  
  4.2.3 Derivative jumping ..................................................................................22  
4.3 Performance of the AI in dynamic scripting .................................................23  
  4.3.1 Learning rate ..........................................................................................23  
  4.3.2 Maximum weight setting .......................................................................24  
  4.3.3 Fitness-mapping function .....................................................................24  

5 **Method** .....................................................................................................26  
5.1 Test environment ...........................................................................................26  
5.2 Measurements ...............................................................................................26  
5.3 Simulations ....................................................................................................28  

6 **Strategic level dynamic scripting** ............................................................29  
6.1 Domain knowledge .......................................................................................29  
6.2 Rules ..............................................................................................................29  
6.3 Planning .........................................................................................................32
6.4 Rule evaluation ........................................................................................................34
6.5 AI in a larger perspective .....................................................................................35

7 Test environment .....................................................................................................38
  7.1 Simulation setup ..................................................................................................38
  7.2 Simulation procedure .........................................................................................39
  7.3 Test cases ...........................................................................................................41
  7.4 Fitness and weight adaptation .............................................................................42

8 Simulation results ....................................................................................................45
  8.1 Convergence time .............................................................................................45
    8.1.1 Punishment or reward ...............................................................................45
    8.1.2 Learning rate .............................................................................................48
    8.1.3 Derivative jumping .....................................................................................49
    8.1.4 Learning rate versus derivative jumping ..................................................51
  8.2 Performance .......................................................................................................53
    8.2.1 Punishment or reward ...............................................................................53
    8.2.2 Learning rate .............................................................................................54
    8.2.3 Derivative jumping .....................................................................................55
  8.3 Varying difficulty ...............................................................................................56
    8.3.1 Learning rate .............................................................................................56
    8.3.2 Maximum weight setting ...........................................................................58
    8.3.3 Fitness-mapping function .........................................................................60

9 Conclusion ...............................................................................................................63
  9.1 Discussion ..........................................................................................................64
  9.2 Future work .......................................................................................................66

References ..................................................................................................................67
1 Introduction

Artificial Intelligence (AI), or rather; the illusion of intelligent behavior, plays a central role in computer games. Tozour (2002a), argues that a more appropriate name for computer game AI would be “agent design”, or “behavioral modeling”, as the word intelligence is rather ambiguous. Laird (2000) defines the role of AI in computer games as follows:

“Just as in military simulation, the role of AIs in computer games is to populate an environment with entities that the human plays with and against.”

The human player can be seen as in control of an agent in the game world, where there usually also exists agents controlled by other players. The other players can either be other human players, or players controlled by the game itself, i.e. by the AI system in the game. Agents not controlled by human players are also known as non-player characters (NPCs). It is often through the interaction with NPCs that the story of a game, which is an important aspect of the entertainment value of a computer game, is delivered to the human player. Hence, the NPCs might need to act intelligently as they are closely coupled with the entertainment value.

The main goal for an AI system in a computer game differs somewhat from that of academic AI systems, which originally strived to reveal the secrets of human level intelligence (Laird & van Lent 2000). In a computer game on the other hand, the main goal is to reach a high level of entertainment, as it is a commercial product whose main task is to entertain its customers. Hence the main goal for the AI system in a computer game is not to be smarter than, and beat the human player; rather it is to increase the level of entertainment experienced by the human player (Spronck, et al. 2002).

The computer games considered in this dissertation are not classical games such as chess, and Othello, but rather the highly interactive, fast-paced computer games, such as first-person shooter (FPS) games like Half-life\textsuperscript{TM} by Valve, or real-time strategy (RTS) games like Command & Conquer: Generals\textsuperscript{TM}, by EA Pacific. Computer games such as those are also commonly known as video games, games made specifically to be played on computers (Fairclough, et al. 2001). More precise, this project is mainly concerned with the AI in RTS games, in which the player is to command an army in battle against other human or AI players. The behavior of opponent players in today’s RTS games are often determined by using scripts (Spronck, et al. 2002). Scripts are usually static and offer few chances for adapting to unforeseen situations. The AI constantly deploys the same set of tactics, and often fall victim to its own predictability. This predictability can make a game monotonous, and also possibly lower the entertainment value. Instead, if the AI could adapt to cater for changing human player tactics, then the illusion of intelligence could perhaps be strengthened. The goal of this project is to investigate how to achieve an adaptive AI at the strategic level in an RTS game.

1.1 Difficulty of the AI

Computer games is a rapidly growing branch within the entertainment industry, which today even has become larger than the movie industry (Laird & van Lent 2000, Nareyek 2002). Many of the games released also tend to be nested together with movies and music around the same theme (Matrix, Lord of the Rings, etc.), in order to increase the profits. Computer game sales have during the past decade been driven by graphical improvements alone (Rubin 2003). Faster graphical processing units
(GPUs) and better algorithms for displaying graphics have constantly been produced in order to push more and more polygons through the graphics rendering pipeline. This has led to a graphical arms race amongst computer game developers and graphics hardware producers, to always achieve better graphical results than their competitors. According to Rubin (2003), the resulting improvements in game play (player experiences during interaction with game systems, TheFreeDictionary.Com 2004) have nicely followed the increase in graphical performance. Rubin (2003) however argues that the graphical arms race cannot continue forever, as the resulting increase in game play no longer is proportional to the increase in graphical performance. Rubin (2003) argues that other aspects of computer games, such as innovation, and attachment (sequels, etc.), will drive game sales in the future. Further, Laird and van Lent (2000) also argue that in some computer games, the AI capabilities are starting to become the point of comparison.

The result of the graphical arms race that has been going on during the last decade is that computer game graphics have evolved to become more and more realistic, and today the graphics have reached a level of quality almost comparable with photorealism. The increased graphical performance has also made it possible to describe far more complex game worlds, containing all sorts of details. Laird and van Lent (2000) suggests that AI systems in computer games can be a part of the continued evolution towards more realistic gaming environments, by creating characters that act more like human beings. Further, Laird and van Lent (2000) also argue that it is the failings of computer game AI players that have pushed many human players towards networked games, where they face other human players intelligent enough to be worth playing against. Hence, reaching more human-level AI could possibly also recreate the experience of playing with or against human players (Laird & van Lent 2000).

Even though the main goal for a computer game AI system differs from that of academic AI systems, agents controlled by a computer game AI still need to act and look credible, even though they according to Laird (2000) in some cases are just meant to be “punching bags” for the human player. Laird (2000) argues that it is important that agents in a computer game seem to possess some intelligence, and that they do not act directly stupid. However, Laird (2000) also argues that it is more important that agents in a computer game neither are too easy nor too hard, than they act realistically. As the virtual worlds in computer games have become more and more complex, the AI systems have also become more powerful. Nareyek (2002) however argue that the complexity of modern computer games has now reached a level where traditional techniques for game AI are no longer directly applicable. Further, Nareyek (2002) believes that techniques from fields such as autonomous agents, and planning are more appropriate to be used in modern computer game AI systems than traditional techniques for game AI.

When creating an AI system for a computer game, developers need to have in mind that there exist different levels of expertise amongst the players. A game should be fun to play for novice players, but it should be equally fun to play for expert players. Hence it is important to have an AI system that scales well with the difficulty level of the game. Lidén (2003) argue that the goal in a computer game is for the human player to win. Pottinger and Laird (2000) however argue that it is not much fun to play against an AI that never has the chance to beat the player either, and that it therefore is important to have an AI which is able to put up a really good fight against the human player. Hence it is interesting to achieve an AI system that is able to compete with the human player, but which also scales well with the performance of the human player.
1.2 RTS games

In a real-time strategy game, the player is to command an army in battle against armies controlled by other players. The game is viewed from a gods-eye perspective, looking down on a battlefield where a number of resources, such as gold, oil, and supplies are located. The task for the player is to collect these resources and construct a base to operate from, in order to create an army. A base consists of buildings of various kinds, such as resource gathering centers, combat unit training facilities, stationary defense buildings, and research facilities. When the game commences, the complete world state is usually unknown, and the player needs to explore it in order to find suitable resource locations and enemy bases. Figure 1 shows a screenshot from the RTS game Age of Mythology™, released by Ensemble Studios in 2003. The resources in this game: gold, wood, and food, can all be seen at various locations in the environment. Another important aspect, which can be seen in the figure, is that the complete game world is not known, until the player has explored it. Even when the complete game world has been visited, it is only the underlying structure of the terrain and past knowledge that remains known. Enemy movement and reconstructions are still unknown, unless they are seen by any friendly units. This is referred to as a fog of war (FOW), in computer games.

![Figure 1: Screenshot from the computer game Age of Mythology™, released by Ensemble Studios in 2003. Different resources can be seen, such as a gold pile, trees, and a cow for food. In the mini-map in the lower right corner, it can also be seen that the complete world have not been discovered, and parts of it is covered in black. Some of the units available in this game can be seen to the left in the picture: cavalry, spearmen, and archers.](image1.jpg)

When the most vital parts of a base have been constructed, the player can start training combat units in order to gain an army. Combat units usually come in many different flavors, such as swordsmen, archers, and cavalry, which are produced at the various training facilities. In a well balanced RTS game, one unit type is usually good at some task but worse at another, and in order to create an army that performs well at many tasks, units must be combined to cover for the weaknesses of each other. For example, archers are good at killing swordsmen that are good at killing cavalry, which in turn are good at killing archers. In order to be strong against all three types of units, a group of archers, swordsmen, and cavalry needs to be combined.
When enough combat units have been trained, the player can start sending assault raids against its enemies. In order for the assaults to be successful, they might need to be based on good strategic decisions. As the complete world state is not known, the player might need to send out scouting units before an actual assault commences, in order to gather information to base the strategic decisions on. Hopefully, the player’s units will breach the enemy defense lines, and start tearing apart the enemy base. However, sometimes the assaults are not as successful, and the units become completely defeated. This is where the stationary defenses, such as walls, bunkers, and turrets can be useful. If the stationary defenses are strategically located, then they can give enough protection against enemy assaults while a new battalion of units can be trained. After a while, when research facilities have been constructed, the combat units can also be upgraded by researching various topics, such as better bows for the archers or thicker armor for the swordsmen.

In Age of Mythology™, the game takes place in the medieval time, with units such as archers, and swordsmen. The RTS game Command & Conquer: Generals™, released by EA Pacific in 2003, takes place in a more modern time, with units such as tanks, and aircraft. In Figure 2, a screenshot from Command & Conquer: Generals™ can be seen. The resources in this game are supplies, which can be seen at the center of the screenshot. Even though the units, buildings, and resources are different in the two games, the overall task remains the same: collect resources, construct buildings, train combat units, and attack the enemies.

Figure 2: Screenshot from the computer game Command & Conquer: Generals™, released by EA Pacific in 2003. The resource in this game, supplies, can be seen in the middle. In this game the underlying terrain of the environment is known in advance, but the location of enemy structures, and units are still unknown, resulting in FOW. Some of the units available in this game, flame tank, and Chinese infantry, can be seen in the center of the screenshot.

The key to victory in an RTS game depends mostly on two factors: good resource handling and strategic thinking. As everything comes to a cost, there must always be a good flow of resources. Resource locations might also need to be secured early on in the game to assure this. Resources must also be used with care, as they come in very limited amounts. Moreover, strategic decisions need to be made concerning how to
best conquer the enemies. Weaknesses in their strategies and defensive lines must be spotted. Advantages in the terrain need to be found. Together, these weaknesses and advantages can be used to construct a good strategy for victory. It is also imperative that the player’s own base is well defended, otherwise it can be overrun, and the game is lost. Hence, strategic decisions concerning base defenses are also an important ingredient for victory. Considering the many complex tasks just described, which a human player faces, it becomes obvious that an AI system for controlling a player in an RTS game need to be quite powerful.

1.3 Problem

The AI system in an RTS game is active at several different levels, with different tasks at each level. At the lower levels, unit/group level, the AI system is to control the behavior of each individual unit in the game, whilst it at the higher levels, strategic level, is to control the behavior of opponent players, facing the same tasks as human players do.

Considering that several hundreds of units might be active in an RTS game at the same time, the AI system at the unit/group level needs to be carefully constructed to meet the timing requirements of a real-time environment. The main task for the AI system at the lower levels is pathfinding, but it might however also need to possess the capabilities of unit coordination, and situation analysis, in order for the units to act intelligently. At the higher levels, the AI system faces tasks such as resource allocation, planning, and opponent modeling. The AI at the strategic level needs to combine many units into one grand strategy for conquering its opponents in battle. All decisions at the strategic level must be realized by issuing low-level unit commands, to all individual units.

Spronck, et al. (2002) argues that the quality of opponent intelligence in computer games such as RTS game primarily lies in the ability of the AI to exhibit human-like behavior. According to Spronck, et al. (2002), this implies that computer controlled opponents should not cheat, they should exploit possibilities offered by the environment, they should learn from mistakes, and they should avoid clearly ineffective behavior. Spronck, et al. (2002) argues that opponents in computer games of today have not yet reached this level of behavior. Buro and Furtak (2003) argue as follows concerning the performance of the AI in RTS games:

"The current AI performance in commercial RTS games is poor by human standards. This may come as a surprise because RTS games have been around for more than ten years already and low-end computers nowadays can execute more than a billion operations per second."

So why is the performance of the AI so poor in RTS games? A main reason for this is that the AI in today’s RTS games mostly is static with very few, if any, methods for adapting to its environment and opponent tactics. The methods used today for creating computer game AI is often based on scripting, finite state machines, and A* pathfinding (Fairclough, et al. 2001, Rabin 2003, Spronck, et al. 2003). This contribute to their inadaptability, as finite state machines are very rigid by nature, and behave poorly when confronted with situations not considered by the designer (Fairclough, et al. 2001). Further, the scripts used are generally static, and because of this they cannot deal with unforeseen tactics, nor do they scale well to the difficulty level exerted by the game, to cover both expert and novice players (Spronck, et al. 2003).
According to Buro and Furtak (2003), one of the biggest shortcomings of current RTS game AI systems, is their inability to learn quickly. A human player on the other hand, can spot weaknesses in the opponent AI within a few games, and then use this knowledge in order to conquer the enemy. Both the problem of human players easily exploiting weaknesses in strategies of the opponent AI, and the inadaptability of the AI, can partly be tracked to the scripts used, as they tend to be large, static, and complex. First, the AI will have difficulties adapting to changed tactics, as the scripts are static, and situations not hard coded by the designers cannot be dealt with in an “intelligent” manner (Spronck, et al. 2003). Secondly, as the scripts become large and complex when trying to cater for large sets of cases, they are likely to contain deficiencies which can be exploited by human players (Spronck, et al. 2002).

When playing against a computer opponent in an RTS game of today, the AI constantly deploys the same strategy: produce units (lots of them), and then move them into the opponents base, following one or more predetermined routes of engagement. This can make the game monotonous very quickly, as the scenario never changes. A human player does not need to consider changed tactics; rather he/she faces the problem of optimizing how to fight small battles. Woodcock (1999), states that the AI in today’s games is capable of winning individual battles, but they still manage to loose the game because of lacking strategic capabilities.

According to Laird and van Lent (2000), it is at the strategic level that the AI in today’s RTS games performs the worst. The AI lacks the ability to develop a high-level plan, which coordinates several units into a grand strategy, based on reasoning, planning, and counter-planning, in order to react to the tactics deployed by the human player. When an AI opponent in an RTS game manages to defeat a human opponent, it is not because of strategic decisions, rather it is a victory based on numbers and/or cheating. Cheating can be used to let the AI have knowledge of enemy locations, etc. in order to simplify decision making. Cheating can be efficient, but if it is detected by the human player, then it can lower the entertainment values (Fairclough, et al. 2001).

To conclude this, the AI in RTS games does not perform well at the strategic level. It lacks the ability to adapt, and deploy strategies which are based on what is feasible at the moment. By introducing an AI which can adapt its higher level strategy to the current situation, many of the weaknesses at the strategic level AI in today’s RTS games could possibly be resolved.

The main goal of this project is to investigate how to achieve an adaptive AI at the strategic level in an RTS game. The techniques used for achieving this must meet the timing constraints coupled with the AI part of a commercial computer game. Further, the techniques used should not be completely based on non-deterministic methods, as it is not feasible to ship a commercial computer game whose AI could turn out to be a failure. This because the entertainment value of a computer game is one of its most important aspects from a commercial point of view.

1.4 Dissertation overview

Chapter two presents relevant background material for understanding this dissertation. This is followed by a problem description, aim and objectives in chapter three. Chapter four sets the focus for what to investigate in this dissertation, and in chapter five, the method for achieving the aim is presented. Chapter six and seven presents theoretical results, and chapter eight presents simulation results. Finally, in chapter nine, conclusions followed by a discussion and statements regarding possible future work, is given.
Background

2 Background

In this chapter, relevant material for understanding the rest of this dissertation is presented. First, in section 2.1, common tasks for the AI in RTS games are presented. Section 2.2 discusses relevant methods for achieving an adaptive AI in computer games. Finally, in section 2.3, a technique for achieving an adaptive AI, dynamic scripting, is presented.

2.1 AI in RTS games

The AI system in an RTS game can be constructed as a hierarchical system (Rabin 2003). At the top level in the hierarchy, the AI system needs to make strategic decisions, and faces tasks such as resource handling, and mission planning. In order to realize the higher level strategies, orders are propagated down the hierarchy, and eventually they reach the low-level AI system through unit-level commands. At the lowest level in the hierarchy, the AI system has the task of controlling each individual unit, or group of units present in the game, according to the orders given from the higher levels in the hierarchy. The AI system in an RTS game can be compared to how real-world armies are structured. At the top, a general or similar decides on an overall strategy for victory. This strategy is then propagated down the hierarchy to different regiments which are concerned with various subparts of the overall strategy. In turn, the major, or commander of a regiment gives orders to individual soldiers, or groups of soldiers (brigades, companies, etc.) under their command, for what they are to do.

2.1.1 Top-level AI

As mentioned above, the AI at the top level in the hierarchy is active at the strategic level, facing the same tasks as a human player would in the same situation. The AI system needs to allocate resources intelligently, place defenses at strategic locations, plan how to conquer the enemies, and recognize what the enemies are currently planning in order to counter plan against it, etc. At this level the AI system needs to combine all these aspects in order to construct a grand strategy plan of how to defeat the enemies. When a plan has been decided, it has to be executed by issuing commands to the lower levels in the AI hierarchy. Some of the problems faced by an AI system at the top-level are now presented and discussed.

A. Resource management

In section 1.2, it was stated that one of the key aspects for achieving victory in an RTS game, is good resource handling. Usually a player in an RTS game initially starts with very limited amounts of resources, only to afford the most vital pieces: resource gathering centers and resource collectors. These pieces can be seen as the most vital, since every aspect of an RTS game depends on the amount of available resources. Buildings, units, and research, all comes to certain costs, and in order to create an army for victory, all these pieces may be needed. It is therefore imperative that a good flow of resources is established early on in the game.

There is however more to resource management then gathering resources. A player needs to decide on what to use the resources for, and also, in what order buildings, units, and research should be invested into. According to Laird and van Lent (2000), resource management is also concerned with production scheduling, so that a player knows in what order to spend the available resources. For example, if a player initially is given 2000 gold and ten infantry units are built to a total cost of 1000 gold. Then, a
resource gathering unit of cost 1500 gold can no longer be afforded, and hence no more gold can be collected. After having lost the ten infantry units, the player becomes an easy prey without any units to defend its base with. It is therefore necessary for a player to properly schedule its production of buildings, units, and research.

Being able to build everything is however not the only aspect that needs to be considered. For example, assuming a scenario where two opponent players, which both have established a good flow of resources, face each other on the battlefield. One of the players decides to invest the whole amount of available resources on building a series of research centers, in order to unlock the heaviest weapon. The other player chooses to go for the cheapest units and attack immediately. The research spending player becomes an easy target, as there are not enough resources available to afford to construct any defensive units. Hence, resource spending need to be well balanced considering temporal aspects as well (Buro & Furtak 2003).

B. Planning

Planning is another key aspect for victory in an RTS game. In order to achieve the overall goal of victory; the AI system might need to plan ahead, and consider future actions that might affect the main goal. A plan needs to be constructed, which may include objectives that are not directly profitable in the near future. The AI might even need to execute unprofitable short term goals, in order to reach the overall goal of the game. For instance, Figure 3 illustrates an example from an RTS game, which is adapted from Buro and Furtak (2003), where each player starts at locations with limited supplies of resources. Each player’s starting position is sealed off from additional resources by strips of trees. A human player (1) realizes rather quickly that in order to defeat the enemy, the resources in the middle need to be secured, and hence starts chopping down trees in order to reach them. An AI player (2) without the ability to plan, starts by collecting the resources next to its base, and only tries to reach the additional resources after the first pile have been exhausted. The human player can, after securing the resources in the middle (a), siege the AI player (b), which is easily defeated (c), when it has run out of resources. In the example just described, only chopping down trees in the beginning can be seen as an unprofitable short-term goal, as there will be a lack of gold to start with. This is however compensated for in the long-run, by a longer lasting flow of gold.

Figure 3: Illustration of a scenario from an RTS game, where the players need to chop down a corridor through the tree line, in order to reach more gold. (Adapted from Buro and Furtak 2003)
Considering that the interface between an AI system and the game world usually consists of basic actions, such as move, attack, build, repair, and collect, longer term planning might not be practical due to the large amount of information available. Given that an RTS game is a dynamic and smart real-time environment, planning in the original world space can be too time-consuming in order to be performed in real-time. Instead, Buro and Furtak (2003) argues that abstractions of the world space might need to be created, which allows forward searches to be conducted in real-time, so that a plan can be formed. This plan can then be translated back into the original world space and executed by issuing unit level commands.

C. Reasoning

Buro and Furtak (2003) argues that spatial-, and temporal-reasoning “is of utmost importance in RTS games”. Spatial reasoning is mainly concerned with analyzing the environment, and temporal reasoning with how actions relate to each other in time.

RTS games are played in an environment where strategic decisions need to be taken. Forbus, et al. (2001) argues that the terrain is of vital importance in war games. In order to make good strategic decisions, the terrain needs to be analyzed. Strategic positions for defenses, weaknesses in enemy defensive lines, and strategic key positions in the environment, might all need to be found for an overall strategy of victory to be established.

Temporal reasoning, how actions relate to, and affects each other, is also of major importance in RTS games. As already mentioned, in sections A and B above, temporal reasoning is of high importance, both for resource management, as well as for planning. In order for an AI system to appear intelligently, it needs to establish what side effects certain actions have; e.g. spend all gold on research, and there is nothing left for building defenses. It is also important to reason about the time it takes from the moment an action is initiated, until it is completed. For example, an order to construct defensive units does not mean that the base is well defended immediately.

Forbus, et al. (2001) argues that spatial reasoning is a major source of difficulties in strategy game AIs, and Buro and Furtak (2003) argue that despite their importance, both spatial-, and temporal-reasoning are largely ignored when building strategy game AIs today, resulting in an AI system that often falls victim to simple common-sense reasoning.

D. Decision making under uncertainty

As mentioned in section 1.2, the complete world state in an RTS game is usually unknown. A player can only see what its units currently see. In some games even the shape of the terrain is initially unknown, and the player needs to send out scouting units in order explore the environment. Considering that the environment is dynamic, the world state as known by one player may not be completely true as another player might have changed it as a result of its actions. It is therefore important to have an AI system which is able to gather intelligence, and make decisions under uncertainty. One property which the AI might need to posses is the ability to model what the other players are currently planning. Buro and Furtak (2003) argue that plausible hypotheses need to be constructed concerning enemy locations, and actions, when there is not enough information available. For example, Figure 4a illustrates a scenario where a player (A) cannot see the overwhelming threat next to its base, shown in Figure 4b, as there is no friendly units close enough to see the approaching army from player (B).
Cheating is a technique which can be used to simplify the decision making for the AI system. The AI system can be given knowledge of how strong its opponents are, together with the structure of their defenses, etc., in order to have a greater chance at victory. If the AI system is relaxed from the task of collecting information by giving it by cheating, decisions can be based on the actual world state instead of a plausible hypothesis about the world state. However, if the cheating is detected than it can ruin the whole gaming experience (Fairclough, et al. 2001).

E. Collaboration

In an RTS game, players can usually choose to cooperate with other players, in order to join forces, or share intelligence. Teams can be created, and together, the individual players on a team can cooperate to defeat other teams, or individual players. Members on a team need to coordinate their actions, and cover for each others weaknesses, in order to achieve an overall team strategy for victory. To coordinate the actions amongst members on a team, there is a need for good communication amongst them. The intentions for other team members need to be known, and the members of a team need to synchronize their actions. The composition of a team can vary between only human players, only AI players, or mixed human and AI player teams.

How teams consisting of only human players coordinate their actions, is not a problem for the AI system in a game. How AI players cooperate with other AI-, and human-players, however is. Different AI players on a team need to reach consensus on what they intend to do, and AI players need to be able to communicate with human players. Reaching consensus, and communicating amongst AI players, can probably be solved by regular communication methods. Communication between an AI player, and a human player, could however pose some fundamental problems. Buro and Furtak (2003) argues that “in case of mixed human/AI teams, the AI player often behaves awkwardly because it does not monitor the human’s actions, cannot infer the human’s intentions, and fails to synchronize its attacks”.

2.1.2 Bottom-level AI

The task for the AI system at the bottom-level is to control each individual unit in the game, including the units possessed by human players. The AI system is to model how the units perceive, think and react to different events in the game world. The main task for the AI at this level can be seen as controlling how the individual units carry out their orders given from a higher strategic level, unaware of if it is a human or computer player issuing them. Most of the orders given to individual units contain some sort of movement instruction. Hence, pathfinding is one of the most fundamental issues for the unit-level AI. If a unit is ordered to move to a certain location, or to attack a specific unit or building, then it needs to find a suitable path to
Background

the location of the target. Some of the problems faced by the AI system at the bottom-level will now briefly be discussed.

A. Unit coordination

Even though pathfinding, which is often realized through the use of AI techniques such as A* pathfinding, is the central issue for the AI at the unit/group level, some other forms of intelligent behavior might also need to be included. For example, Figure 5a, illustrates a situation where a group of units might need to take different routes to a target in order to attack it with a united front. If they all choose the same route, then they will block each other so that only one unit at a time can attack the target, which is illustrated in Figure 5b. The target will in turn have a greater chance of survival, as it only needs to defend itself against one unit at a time. By coordinating their actions, the units can choose different routes to the target, in order to attack it at the same time. This is illustrated in Figure 5c. Hence, unit coordination can be needed for the units to act somewhat intelligently.

Figure 5: Illustrations of how units might need to coordinate their actions. In a), the goal for the units labeled A, is to attack the unit labeled B. In b), it is illustrated how the units might proceed without coordinating their actions, and only relying on the shortest path to the target. It is obvious that they will block each other, making it easier for B to defend itself. In c), it is illustrated how the units can coordinate their actions in order to attack with a united front. (Adapted from Forbus, et al. 2001)

B. Situation analysis

Another aspect which can be important in the bottom-level AI is situation analysis. Consider a situation where a group of units is moving to some location as ordered by the strategic level. If the units are attacked on their way to their target, then they should not continue their journey as if nothing has happened. Instead, they should perform some sort of situation analysis in order to determine how to react to the changed conditions for reaching their target. If the group of units is outnumbered, then they should perhaps re-plan their route, and find a safe path to their target. In case the oppressors are outnumbered, the units could perhaps attack and destroy them before pursuing what they were ordered to do.

2.2 Adaptive AI in computer games

Algorithms for achieving adaptation, also referred to as learning, can be categorized based on the type of feedback available from the environment. Russel and Norvig (1995) identify three distinct categories of adaptive algorithms based on the type of available feedback:

- Supervised learning
- Unsupervised learning
- Reinforcement learning
In supervised learning algorithms, the results produced by the adaptive system are compared with values, which are known to be correct in advance. In unsupervised learning algorithms, the correct results are unknown, and hence the performance of the algorithms needs to be estimated by some criteria. In reinforcement learning, feedback can be received on outputs, concerning if they are good or bad, but nothing is known about what is the correct or optimal output.

When using adaptive algorithms in computer games, two different approaches can be used: online, and offline adaptation (Spronck, et al. 2003, Spronck, et al. 2002). In offline approaches, an adaptive system is used to teach the AI, before an actual game is released to the market. When the game is shipped, the adaptation process is turned off, and the AI becomes static. In online approaches, the adaptation process is turned on after a game is shipped, and the AI hence becomes dynamic, and can adapt to different playing styles and tactics. The two approaches can of course also be combined, so that the AI can be trained in an offline phase, and then also learn from its opponents.

When incorporating an online learning system in a computer game, the timescale on which the learning process operates can also be varied. The AI can either receive feedback during operation, which would mean that the AI could learn during a single game, or it could receive feedback after operation, which would mean that the AI could learn from game to game. This can be compared to nature, where a smaller time scale can be seen as lifetime learning for an animal, and a larger timescale as the evolution of a species, or similar. Both timescales can also be used simultaneously.

According to Spronck, et al. (2003), there are four properties that need to be fulfilled, in order for an unsupervised online learning method to be applicable in practice for the AI in a computer game. The method should be:

1. Fast. Online learning takes place during game play; hence it should be computationally cheap (low algorithmic time complexity).

2. Effective. As the goal for a computer game is to reach a high entertainment value, the method should provide at least as much challenge as a manually designed static AI system would.

3. Robust. Commercial computer game environments often come with a certain degree of inherent randomness. Hence the method needs to be tolerant to some degree of noise.

4. Efficient. The number of occasions in a single game when feedback is available for learning is very limited. Hence the method should learn on a very small number of trials.

Spronck, et al. (2003), also argues that there are two additional properties that commercial game developers would add:

5. Understandable. The resulting AI system should be easy to understand.

6. Non-repetitive. The resulting behavior of the AI should be unpredictable; otherwise it could lower the entertainment value.

### 2.3 Dynamic scripting

Dynamic scripting, presented by Spronck, et al. (2003), is a technique for achieving online adaptation of computer game opponents. In dynamic scripting, scripts are created online based on rules extracted from a rulebase. Figure 6 illustrates an example of how rules are extracted and combined to create a script.
Background

Figure 6: Illustration of how a script is created by extracting rules from a rulebase.

After a script has served its purpose, the rulebase is adapted according to how successful each rule was when applied. Originally, the dynamic scripting algorithm was applied to create scripts that controlled the behavior of individual characters in a CRPG environment.

In a CRPG the human player is usually in control over a single character or a party of characters which are situated in the game world. The characters can usually be of certain types, such as fighter, wizard, or druid, which all have different characteristics. For example, a wizard is wise and can cast spells, but is also weak, has low stamina, and has poor skills in martial arts and weaponry. A fighter on the other hand is strong, has high stamina, is good at martial arts and using weapons, but is not too bright, and cannot cast any spells. The virtual worlds in CRPGs are usually based on different towns, cities, and similar, which usually are connected by the means of wastelands, forests, and caves which are inhabited by various kinds of evil monsters, or similar.

A CRPG usually starts of by introducing the player to a main quest of the game, such as: “The evil wizard Ragnvald has been raised from the dead and is about to bewitch the whole world into evil. He must be stopped before he sets loose all his former companions.” The task for the human player is then usually to track down the evil and save the world from it. While following the story and traveling from village to village, the human player usually carry out smaller sub-quests, which according to Spronck, et al. (2003) includes conversing with the inhabitants of the world, solving puzzles, discovering secrets, and defeating opponent monsters, players, or parties in combat. By defeating enemies in combat, gold, items, and experience is usually earned. Items could for example be more powerful weapons or armor. Gold can be used to buy new equipment in the various towns in the game. When certain levels of experience have been reached, the player is usually given an amount of character points, and skill points. The character points can be used to for example increase vitality, or strength, whilst skill points can be used to unlock new skills, such as a more powerful fireball, or an overhead axe blow.

Dynamic scripting is used to create scripts that controls the opponents faced in combat. Hence the requirements are that the script for an opponent includes rules of use for that character type, such as cast fireball for a wizard. In the dynamic scripting algorithm (Spronck, et al. 2003), a rule is comprised of a single action, with an optional conditional statement (e.g. “if life less than 10 then drink a healing potion”). These rules are created for each type of opponent before the actual game starts. For each enemy unit/group, which the human player faces during the game, a set of rules is extracted from the rulebase in order to create a script to control that opponent. After the encounter has ended, the weights for the rules used from the rulebase are updated according to how much they contributed to the outcome of using the script. Figure 7 illustrates an example of the dynamic scripting process.
A rulebase in dynamic scripting contains several rules, and each rule is associated with a weight value. The weight value for a rule represents the probability that the rule will be selected when creating a script. The adaptation process changes the weight values associated with the rules, in order to reflect how successful they have been. When adapting the weights for rules in a rulebase, a weight-update function is applied for each rule. The weight-update function changes each weight value according to the fitness value achieved for each corresponding rule.

The weight-update function used by Spronck, et al. (2003) calculates the new weight, $W$, for a rule as follows:

$$
W = \begin{cases} 
\max\left(0, W_{\text{org}} - MP \cdot \frac{b - F(p,c)}{b}\right) & \text{if } (F(p,c) < b) \\
\min\left(W_{\text{org}} + MR \cdot \frac{F(p,c) - b}{1-b}, MW\right) & \text{otherwise}
\end{cases}
$$

where $W_{\text{org}}$ is the original weight value, $MP$ the maximum penalty, $MR$ the maximum reward, $MW$ the maximum weight setting, $b$ the break-even point, and $F(p,c)$ the fitness of a character $c$, which is a member of a party $p$.

The fitness $F(p, c)$ for a character $c$ is calculated as a combination of the average remaining health of all party members, the average damage inflicted upon enemies, the remaining health of character $c$ (or time of death if $c$ dead), and the party fitness $F(p)$. The party fitness, $F(p)$, is zero if the party lost the fight, or 0.5 plus half the average remaining health of all party members, if party $p$ won the fight. When changing weights in a rulebase, a weight redistribution function is used, so that the total weight sum is always kept constant. Hence, if the weight for one rule is increased, then the weights for other rules are decreased.

Spronck, et al. (2003) tested the dynamic scripting technique in both simulation and practice, in a computer role-playing game (CRPG) environment, where it was applied at the unit/group level. According to the Spronck, et al. (2003), the technique is fast, effective, robust, and efficient, which are requirements for an online learning algorithm to be useful in a commercial computer game. Finally, the authors concluded that their technique was successfully incorporated into a commercial CRPG game, and that it also should be applicable to other game genres as well.
3 Problem description

The main goal of this project is to investigate how to achieve an adaptive AI at the strategic level in an RTS game. Many of the problems in the AI at the strategic level in an RTS game can possibly be tracked to the static nature of the AI. It is common to build the AI by using scripting (Spronck, et al. 2002). According to Tozour (2002b), there are a number of benefits of using scripting: (1) parallel development, (2) ease of use, (3) reusability, (4) easier development, (5) runs in protected environments, (6) extensibility. Furthermore, scripting also allows behavior to move beyond reactive behavior. Scripts do however also tend to contain deficiencies that can be exploited by human players, when covering for a large number of situations (Spronck, et al. 2002, Spronck, et al. 2003). Furthermore, scripts are also usually static, with few chances to adapt to opponent tactics.

By introducing an adaptive AI at the strategic level in an RTS game, the AI would no longer be static, could possibly cope with situations not thought of by the designers, and could adapt its strategy to the currently deployed human player tactics. Further, an adaptive AI could also help making the game less monotonous, strengthen the illusion of intelligent behavior, and perhaps also increase the entertainment value.

As the purpose is to adapt to cater for changing human player tactics, the results are not known in advance, and hence supervised learning methods cannot be used. Some hint regarding if the results are positive or negative, is however available, and hence reinforcement learning methods are interesting. Both offline and online learning methods are interesting to use, but as the purpose is to let the AI adapt after it has been shipped, offline learning methods are not considered in this dissertation. Offline methods could however be used to train an AI system before it is shipped.

The two different timescales on which a learning algorithm can operate, described in section 2.2, are both interesting to be used in an RTS game. By using a larger timescale, the AI could perhaps learn how different human players play, and evolve its playing style to always keep in phase with the human players. This could create a staggering effect on the difficulty level of a game, which clearly could be an advantage, as the difficulty of a computer game is usually intended to become harder as the player advances. A smaller timescale would be used to let the AI learn to cope with changes in the human playing style during a single game, and it could also be useful in order to give an illusion of intelligence. For example, if the human player attacks with aerial units for three consecutive encounters, then the AI should not continue to use tanks only for defensive purposes, as they are usually not very good at destroying aerial units. If the AI only were to adapt to this tactic in the next game, then it might not be that useful, as the human players grand strategy in the next game could be something drastically different (for example tanks).

It can be considered that a smaller timescale could have a more direct impact on the performance of the AI, and hence a smaller time scale is chosen in this project. Therefore, the investigated adaptation method should be online reinforcement learning, which adapts during operation.

In order for learning to be feasible in a computer game, such as an RTS game, the method for achieving learning should, according to Spronck, et al. (2003), be: fast, effective, robust, efficient, understandable, and non-repetitive (see section 2.2). Spronck, et al. (2003) argues as follows: Computationally expensive learning methods, such as model-based learning, are excluded as they are not fast enough.
Random learning methods such as evolutionary learning are excluded as they are not effective, nor efficient enough. Deterministic learning methods that depend on a gradient search, such as straightforward hill-climbing, are also excluded as they are not robust enough. Slow-learning techniques, such as neural networks, and reinforcement learning are excluded, as they are not efficient enough.

Dynamic scripting is a technique presented by Spronck, et al. (2003), for achieving an online adaptive AI in computer games. It is based on reinforcement learning, and creates scripts online by extracting rules from a rulebase (see section 2.3). Spronck, et al. (2003) argues that dynamic scripting fulfills all six of the requirements for online learning to be feasible in a computer game. The dynamic scripting technique was originally designed to be used in CRPG games, but the authors also argue that it should be applicable to other game genres as well.

Even though Spronck, et al. (2003) concludes that their technique, dynamic scripting, was successful at achieving an adaptive AI, they also stated some potential problems with it, and that some improvements are needed. The adaptation process can in some cases take excessively long time, being the result of undesirable rules having very large weights. According to the authors, the rules which describe undesirable behaviors can be very hard to unlearn, once they have been learned. Spronck, et al. (2003) also observed that sometimes, the dynamic scripting procedure started to generate inferior scripts after a turning point had been reached. Spronck, et al. (2003) believes the reason for this is that the rulebase continues to learn new behavior, even though it is already successful. It is however not an option to turn off the learning procedure once successful behavior have been reached, as the purpose is to adapt to changing human player tactics.

3.1 Problem delimitation

Dynamic scripting is the only algorithm found, for achieving an online reinforcement AI system in computer games, which also fulfills the requirements of fastness, effectiveness, robustness, efficiency, understandability, and non-repetitiveness. The dynamic scripting algorithm, presented by Spronck, et al. (2003), was however originally designed to be used in CRPG games, and hence it needs to be investigated how to apply dynamic scripting at the strategic level in an RTS game. Therefore, the goal of this project is delimited to looking at dynamic scripting, and how it possibly could be used to achieve an adaptive AI at the strategic level in an RTS game.

The dynamic scripting technique was originally designed for CRPG games, where it was used to create scripts that control the behavior of opponent characters. Each created script controls one opponent character. At the strategic level in an RTS game, many units are used together to form one cohesive strategy for defeating opponents. Further, resources need to be collected and managed, buildings need to be constructed, and units created. The differences between controlling an RTS game player at the strategic level, compared to controlling an NPC in a CRPG game, can be quite extensive. Hence it is interesting to investigate how to incorporate dynamic scripting at the strategic level in an RTS game.

At the strategic level in an RTS game, the amount of chances for receiving feedback on the success/failure rate on different scripts can be much lower than in a CRPG game. If the AI fails to adapt to a new opponent strategy within a few encounters, then it will probably have lost the game. Therefore, it is interesting to investigate how to lower the time to reach convergence, in dynamic scripting. The problems in dynamic scripting, described earlier, are of course also interesting to investigate. Potential
problems with reaching convergence, and deteriorating script performance, are both important problems to solve, as they both affect the usability of the algorithm. How to lower the time to reach convergence in dynamic scripting is closely related to one of the problems, sometimes long time to reach convergence. Hence it becomes even more interesting to investigate the problem with time to reach convergence.

At the strategic level in an RTS game, there is a high probability of sudden tactical changes. If the currently deployed tactic does not perform well, then change tactic. This can make the problem with inferior scripts less important. Hence, the potential problem with long time to reach convergence can be more important at the strategic level in an RTS game, compared with the potential problem of inferior script performance.

As mentioned in chapter 1, computer games need to be fun for both novice, and expert players. This can be a problem with most adaptation algorithms, as they usually aim at reaching the best available performance. This problem could however possibly be solved by designing fitness criteria that does not promote good performance, but which promotes high entertainment value. Entertainment value is a quite complex term, which depends on both the medium, as well as the recipient of the entertainment in question. How much fun is it to play a computer game? This is quite individual from player to player, and hence becomes very hard to measure. Spronck, et al. (2003) argues that instead of aiming at good performance, length of fights could be used as evaluation criteria. Spronck et al. (2003), argues that it sometimes even might be useful to punish a rulebase that defeats the human player, or inflicts too much damage on the human player. It could however possibly be considered that an expert player will not have as fun playing against a weak AI, as playing against an AI that puts up a good fight. Further, a novice human player might not have as fun playing against an AI that always wins, as playing against an AI that is beatable. Hence it is interesting to investigate how to throttle the performance of the AI, in order to cope with both expert, as well as novice players.

The computational cost for using the two algorithms is of course also an important aspect when working with computer games. Keeping the framerate stable is a central issue for reaching a high entertainment value, and the time available for processing AI tasks is very limited in each frame, as other parts of a game environment, such as graphics, collision detection, and physics also requires to be processed in each frame. It is therefore of high importance to use computationally cheap algorithms. According to Spronck, et al. (2003), the dynamic scripting algorithm is computationally cheap, and it should be feasible to use in a commercial game environment. Hence, the problem of computational cost is not considered any further in this project.

The issues just described, for applying dynamic scripting at the strategic level in an RTS game, can be summarized in three questions:

1. How can the structure of dynamic scripting be modified to fit the strategic level in an RTS game?
2. How can the potential problem with long time to reach convergence in dynamic scripting be solved?
3. How can the performance of the dynamically scripted AI system be limited?

The main goal of this project is hence delimited to investigating these questions.
3.2 Aim and objectives

The aim of this project is to investigate how dynamic scripting could be applied to achieve an adaptive AI at the strategic level in an RTS game.

In order to achieve the stated aim, a number of objectives have been identified:

(i) Investigate how dynamic scripting could be incorporated at the strategic level in an RTS game.

(ii) Investigate how the potential problem with long time to reach convergence, in dynamic scripting, possibly could be solved.

(iii) Investigate how the performance of the AI possibly could be throttled in dynamic scripting.

(iv) Identify a suitable test environment, and suitable measurements for conducting investigations of the results from objectives (ii) and (iii).

(v) Conduct simulations and analyze the investigated methods.

The first three objectives are directly linked to the questions in section 3.1. The fourth objective aims at constructing a test environment, where the proposed solutions for objectives (ii) and (iii) can be investigated by means of simulations. The fifth and final objective, aims at analyzing the material found in objectives (ii) to (iv).

3.3 Expected results

In this project, it is expected to find a method for applying dynamic scripting at the strategic level in an RTS game. Further, it is also expected to find possible methods for solving the potential problem with long time to reach convergence, in dynamic scripting. The time for reaching adaptation is of special interest at the strategic level in an RTS game, as the number of occasions when feedback is available, can be very limited. Finally, it is also expected to find potential methods for throttling the performance of the AI when using dynamic scripting. This is interesting as the expertise amongst game players span from novice to expert players, and a game should be fun to play for all.
4 Focus of investigations

In this chapter the focus of what to investigate in this dissertation is described. In section 4.1 the focus for investigating the first objective, how to incorporate dynamic scripting at the strategic level in an RTS game, is described. In section 4.2, a discussion is given concerning what to focus on, for possibly solving the potential problem with convergence time, in dynamic scripting. Section 4.3 presents a discussion on what to investigate for possibly throttling the performance of the AI.

4.1 Strategic level dynamic scripting

This section sets the focus for investigating how to incorporate dynamic scripting at the strategic level in an RTS game. Two potential paths for incorporating dynamic scripting at the strategic level in an RTS game are presented. The focus is then set for which of the two paths to investigate more thoroughly in this dissertation.

The key property of dynamic scripting is that it is based on domain knowledge. Spronck, et al. (2003) used character type as domain knowledge when, creating the system in a CRPG environment (see section 2.3). In an RTS game, this could be translated into unit type, which could be used as domain knowledge. This would however not be very useful at a strategic level, where multiple unit types are to be combined into one large strategy. Other sources of domain knowledge need to be found, which fits the strategic level better.

In RTS games, the player can usually chose amongst several different base player types. For example, in Command & Conquer: Generals™, the player can choose to play as the United States, China, or GLA, a fictive terrorist organization. The different kinds of base player types also usually possess drastically different arsenals of weapons, and hence their strategic decisions need to be separated. Using these different player types as domain knowledge would be one approach to apply dynamic scripting at the strategic level in an RTS game. It would however possibly need to be complemented with game state dependant information, such as how good is the current cash flow, how well structured are the defenses, etc., in order for the AI to appear intelligently. Rules could for example be categorized, such as defensive rules, cash collecting rules, and offensive rules. Then, depending on the current game state, a predefined number of rules from each category could be extracted to create a script.

The approach just described would probably be functional, and adaptation would be achieved. It does however largely ignore important tasks to be considered for the AI at the strategic level in an RTS game, such as reasoning, planning, and resource management (see section 2.1.1). These tasks can all be important ingredients for making good strategic decisions. Further, ignoring these tasks could result in an AI that appears unintelligent. Now, a second approach is described, which has a focus on common tasks for the AI at the strategic level.

In section 2.1.1, many tasks, such as resource handling, planning, decision making, and collaboration were presented. These tasks can all be important when creating the AI system at the strategic level in an RTS game, but how can they be combined to make good strategic decisions? The AI can be seen as a collection of subsystems, for handling the tasks, which in combination can be used to form an overall strategy for victory. Dynamic scripting could be used as one subsystem of the AI, in order to implement some of the tasks. In order to investigate which tasks that could be achieved through dynamic scripting; a short recapitulation of the tasks is first given:
Focus of investigations

- **Resource handling.** Resource handling is important in order to collect resources, so that the player can afford to construct units which later can be used on the battlefield. There is however more to resource handling than collecting them. Decisions need to be made concerning what the resources are going to be spent on, what buildings to construct, what units to build, and what upgrades to research. In order to decide on these issues, the resource handling system needs to gather information from other parts of the AI system.

- **Collaboration.** If the AI player is to collaborate with other players, then it needs to take its place on the team. Good communication with the other players is needed in order for the team to combine their forces into one massive group strategy.

- **Spatial reasoning.** The environment needs to be analyzed in order to make decisions concerning where to build defenses, where to attack enemies, where to find resources, etc. A system that can perform good spatial reasoning is therefore important.

- **Temporal reasoning.** Temporal reasoning can also be important, considering how different actions relate to, and affect each other. For example, if the player only possesses two tanks and those are used to launch an attack against an opponent. Then there is nothing left to defend the base with, and the player can be overrun. The reasoning system needs to consider such relations when issuing orders.

- **Modeling.** In an environment where the complete world state is unknown, the AI needs to make decisions of what to do under uncertainty. If the AI only were to consider the exact knowledge it has, then it could easily be defeated by hiding hostile actions from it until it is too late for it to respond on them. For example, if one player builds a huge army, and locates it in such a position that it is hidden from the AI, then the AI would not know of the potential threat if it only considers what it could see (Figure 4, section 2.1.1). It is therefore important that the AI models what the other players are planning, in order for it to play the game.

- **Planning.** Planning can be seen as a key aspect for combining all the subsystems into one massive strategy. A plan can be formed, which is based on reasoning, modeling, learning, and collaboration. The plan can then be used for resource allocation, assaults, and defensive work.

Dynamic scripting, is not a communication system, hence collaboration is ruled out. Further, dynamic scripting is not a spatial reasoning system, nor is it a temporal reasoning system; therefore, both spatial and temporal reasoning is ruled out. Modeling can be divided in explicit and implicit modeling. As dynamic scripting is a machine-learning technique, its weights implicitly models the behavior previously expressed by its enemies, as well as by its allies. Hence, implicit modeling comes with the system. However, dynamic scripting is not a system for making plausible hypotheses concerning enemy actions, when no information is available. Hence explicit modeling is ruled out. As implicit modeling is inherent in the system, focus needs to be on planning, and resource handling.

Nareyek (2002) argue that the most important aspect of an agent’s intelligence is its goal-directed component. As stated in chapter 1, the AI should appear to act intelligently. It can therefore be important that strategic decisions are directly
connected to explicit goals. Goals can be decided by a sub-system of the AI, which analyses the environment, and includes the topics previously discussed. When a goal has been decided, it can be propagated to a planning sub-system, which creates a plan in compliance with the suggested goal. As the plans that are formed are direct results of specific goals, the AI can be seen as goal-driven. This can in turn strengthen the illusion of intelligent behavior, which previously has been stated to be important in a computer game.

Dynamic scripting could be put into a larger perspective, where it is one of several parts working together in order to achieve an AI capable of performing the properties discussed in section 2.1.1. The other parts of the AI could then be used to handle reasoning, communication, implicit modeling, and terrain analysis. This information could later be used as input to the dynamically scripted part of the AI, where it could be used as domain knowledge in form of explicit goals, such as attack, defend, collect supplies, or similar.

Further, Nareyek (2002) also argues that there is a need for “anytime agents”, using planning, and which always have a plan available to be executed. By an “anytime agent”, it is meant that the agent iteratively improves its current plan when there is spare time available. The planning property is important to include, in order for the agent to move beyond being a purely reactive agent, which makes the agent able to attain longer-term goals.

Given that the goal driven aspect is the most important of an agent’s intelligence, and that it is at a strategic level, where adaptation is investigated, the second approach seems more promising. Further, if strategic decisions are made randomly, there is a risk of unintelligence shining through, as in case of the first approach. Hence, a goal driven version of dynamic scripting, which considers planning, resource handling, and implicit modeling, is to be investigated.

4.2 Problems in dynamic scripting

In this section, the focus is set for what to investigate in order to find possible solutions to the potential problem in dynamic scripting, with long time to reach convergence. First, a recapitulation of the potential problem is given:

1. The adaptation process can sometimes take excessively long time.

2. Undesirable behavior can be hard to unlearn, once it has been learned.

Spronck, et al. (2003) argues that the adaptation process occasionally taking long time is a result of undesirable rules having large weights. Hence, the two problems relate to convergence times, and are closely coupled together.

Dynamic scripting is a form of reinforcement learning. Depending on the outcome, some rules are rewarded, whilst others are punished. Investigating the settings of the reward and punishment factors could possibly hold a solution to the problem. A brief investigation of these factors is now given. In section 4.2.1, the relation between the reward and punishment factors is discussed, and how these factors possibly could be used to solve the problem. Section 4.2.2 discusses the relation between the learning rate, and maximum weight setting, and how this possibly relates to the problem. Finally, section 4.2.3 investigates a third potential method for finding a solution to the problem, derivative jumping.
4.2.1 Punishment or reward

In the dynamic scripting algorithm, presented by Spronck, et al. (2003), the punishment factor was set to 30% of the reward factor. Spronck, et al. (2003) did not provide any specific motivation behind the choice of punishment and reward factors. Could the setting of the punishment and reward factors perhaps be related to the problem of unlearning bad behaviors? In case a bad rule gets rewarded, through chance or similar, then it can take more than three punishments, in order to compensate for this. Increasing the punishment factor could possibly solve the problem with bad behaviors being hard to unlearn, and hence different settings of the punishment and reward factors should be investigated.

4.2.2 Learning rate

The learning rate in the dynamic scripting algorithm (Spronck, et al. 2003), is split into the reward, and punishment factors. Hence there are different learning rates for punishment and reward. The learning rate factor could however be introduced, in addition to the punishment and reward factors. In the article by Spronck, et al. (2003), the learning rate parameter would be 5% of the maximum weight value, if introduced. Increasing this factor to 10% or perhaps more, would probably shorten the time to learn new behavior, as fewer rewards and punishments, might be needed before a convergence can be reached. Increasing the learning rate factor, independently from the reward, and punishment factors, might possibly solve the problem of long time to reach adaptation. The learning rate should however not be set too high, as then the behavior could become predictable, and as unpredictability is one of the benefits of using dynamic scripting (Spronck, et al. 2003), this should be avoided. As argued in chapter 3, the number of occasions when feedback is available, can be much fewer at the strategic level in an RTS game, and it is therefore interesting to investigate the learning rate parameter, and if it can be used to lower the adaptation time. Hence different settings of the learning rate parameter will be investigated.

4.2.3 Derivative jumping

A third possible solution to the problem could be to introduce weight jumping according to previous results. Weight jumping, could be achieved by tracking the trend of change over time in the success/failure rates. If the trend points at a large weight increase, then the weights could be “jumped” to higher levels in order to rapidly cater for changing tactics. Similarly, if the trend of the success/failure rate points at a large weight decrease, then the weights could be drastically lowered. A possible realization of this could be to adapt a rule according to the speed of change of the success/failure rate of that rule being used, in addition to the proportional weight changes.

According to classical calculus, the speed is retrieved through the derivative of a function, but this is not directly applicable here, as the success/failure rate cannot be measured as a continuous function. The derivative could however be retrieved by calculating the point-wise derivative between the current result, and the previous result. The point wise derivative could then be used to implement “jumps” on the rule weights. A possible problem with this could however be that the weights could end up oscillating between high and low weight values, resulting in problems to reach convergence of the rulebase, according to the currently deployed opponent strategy. Instead, by inserting the success/failure results into a non-uniform rational b-spline (NURB), or similar, and then solving for the derivative, a weight update rule can be formed. The weight update rule could then additionally to the proportional weight
changes, also update the weights according to the derivative of the success/failure rate. By using a NURB, the influence of historical success/failure results can be weighted so that older results have less influence over the weight change, than more recent results. The weight change would however also depend on previous results, and in case of oscillating results, the impact of the derivative will be lower, resulting in less weight change in case of uncertain derivative direction.

Using the derivative, by means of a NURB curve, in order to implement weight “jumping”, could solve both problems with time to reach convergence. First, the weights of successful rules would be “jumped” to higher levels. Secondly, the weights of unsuccessful rules would be “jumped” to lower levels. This could decrease the adaptation time, and also decrease the time to unlearn unwanted behaviors. Further, in the case of proportional weight changes only, weights are increased when the results are positive, and decreased when the results are negative, on an absolute scale. The “jump” changes would however not depend on if the results are positive, or negative, according to some predefined break-even point, instead they depend on if the results are positive, or negative in relation to previous results. This could make the system less predictable, as weights can be changed independently of how the result relates to a break-even point, i.e. the derivative introduces more noise.

Including the derivative in addition to proportional weight changes, in the adaptation algorithm, should be investigated, and compared with the results achieved when investigating the learning rate, reward, and punishment factors.

4.3 Performance of the AI in dynamic scripting

A computer game should be fun to play for both expert and novice players. In case of a novice player, the game should not be too hard, as then the player could get frustrated for always loosing. In case of an expert player, the game should not be too easy as it could then become boring when no effort needs to be put into winning. This is why a computer game should be difficult enough for an expert player and easy enough for a novice player. Hence, there is a need for having the difficulty level exerted by a computer game variable. Achieving a variable difficulty level can become a problem when using machine-learning techniques, such as dynamic scripting, as they strive to reach the best available solution.

The ability of dynamic scripting to reach the best solution can be limited somewhat, by configuring the rulebase so that a single rule can not dominate and always be chosen. Further, if a single rule is able to dominate, then the behavior might become predictable. As stated earlier, there is a wish to have unpredictable behavior in a computer game, as predictability could lower the entertainment value. The best available solution might however be needed sometimes, to reach high performance levels, in order to cope with an expert player. There is however also a need to be able to limit the performance of the AI, in order to cope with novice players. After all, the AI should not exclusively beat the human player, and hence the need for having a way of delimiting the performance of the AI. This section discusses various approaches for possibly throttling the difficulty level exerted by the AI.

4.3.1 Learning rate

In most machine-learning techniques, the learning rate controls how fast the learning process learns new behavior. Having a low learning rate yields a slow learning AI, and possibly an AI that is not as good as a fast learning AI. Hence, it should be investigated, if the learning rate can be used to vary the performance exerted by the AI.
4.3.2 Maximum weight setting

As previously stated, the maximum weight setting can be used to limit the possibility of always selecting the best available solution. In the dynamic scripting algorithm, the total weight sum for all rules in a rulebase is configured to remain constant. If a weight is decreased, then other weights are increased, in order to keep the total weight sum constant. Having the maximum weight parameter equal to the total weight sum makes it possible to favor a single rule. The single favored rule will be a rule that yields positive results in relation to the break-even point. Having a lower maximum weight setting would yield that the probability of selecting other rules, which might not be as favorable, is higher. Hence, it should be investigated if the performance of the AI can be throttled by varying the maximum weight setting. It should however be kept in mind, that a high maximum weight setting might also yield predictable behavior, which in turn could lower the entertainment value.

4.3.3 Fitness-mapping function

Another approach for limiting the performance of the AI is to investigate the weight update rule. The weight update rule, works as a mapping between fitness scores, and weight changes. In the dynamic scripting algorithm presented by Spronck, et al. (2003), a proportional weight update function is used. The weight update function was described in section 2.3, and Figure 8 illustrates a plot of it.

![Figure 8: A plot of the weight changes for different fitness values, used in the dynamic scripting algorithm by Spronck, et al. (2003).](image)

In Figure 8, it can be observed that the highest available fitness gives the largest weight increase, and the lowest available fitness, yields the largest weight decrease. This kind of function promotes the best available solution. Instead, in order to achieve varying performance of the AI, a weight update function that promotes varying fitness scores, according to the current difficulty level, could be used.

One possible candidate for such a fitness function is to use one revolution of the sine function. The amplitude and frequency of the sine function could be translated to fit the limits of the original fitness function. Further, the function could also be phase-shifted, to center its peak on the fitness value that corresponds to the currently active difficulty setting. The function could be designed as follows:
Focus of investigations

\[ f_n(f_o, f_t) = \begin{cases} 
\sin \left( \frac{2 \pi \cdot (f_o - f_t + 0.5) - \pi}{2} \right) + 0.5 & \text{if } |f_o - f_t| \leq 0.5, \\
0 & \text{otherwise}
\end{cases} \]

where \( f_n(f_o, f_t) \) denotes the fitness-mapping function, \( f_o \), the original fitness value, and \( f_t \) the target fitness value according to difficulty level. By varying the target fitness, the fitness function will promote varying performance levels. Figure 9 illustrates four different plots of the function \( f_n(f_o, f_t) \), with fitness targets of 0, 0.5, 0.75, and 1.0.

![Figure 9: Illustration of how a sine function could be used to translate fitness values, in order to possibly force adaptation in certain directions.](image)

The kind of function just described, could be used as a mapping between the original fitness function and the weight update rule, in order to possibly achieve varying performance, according to some predefined difficulty level. Hence it should be investigated if the performance of the AI can be throttled by using this kind of fitness-mapping function.
5 Method

This chapter describes the method for carrying out the investigations stated in chapter 4. One possible method for investigating properties such as learning rate, and difficulty adjustment, is to conduct simulations in some form of test environment. Conducting simulations in a test environment is considered a good method for investigating the various topics previously discussed, as most of the investigations concern numerical comparison of usability. In section 5.1, a suitable method for finding a test environment is described. This is followed by the method for what to measure during simulations, in section 5.2. Finally, in section 5.3, a method for constructing simulations, with regard to the previously described properties to investigate, is described.

5.1 Test environment

There are mainly two different kinds of environments that can be used:

1. Commercial environment.
2. Artificial environment.

In a commercial computer game environment, all the constraints coupled with developing an algorithm to be used in a game, becomes directly visible. By using a commercial environment, the algorithm to be tested is also used in an environment which it is designed for. This is good as the risk of infusing problems that are not ever going to happen, are limited. A commercial environment also rules out the risk of infusing problems which are specially created for the algorithms to be tested. In a commercial environment, in which the algorithms are planned to operate, the algorithms face the actual problems inherent in the environment.

An artificial environment can be constructed in such a way as to enable different parameter settings, and algorithms to be compared under similar circumstances. Further, specific properties can be compared without having to deal with the complexity of a real computer game. An artificial environment can also be constructed in such a way as to make optimal solutions known in advance, and especially, it can be known that there exist some optimal solutions, which the algorithms can reach. It then becomes easier to compare the time it takes for the two algorithms to reach an optimal solution.

As the purpose of the test environment is to investigate various methods for lowering the convergence time and for varying the performance of the AI, an artificial environment has been selected. This as the conditions, under which the algorithms are to be tested, clearly can be constructed according to a predefined structure. This allows measurements to be performed for different approaches, under equal conditions. In an actual computer game, the inherent randomness makes it hard to take measurements which are easily comparable. Further, an artificial environment can easier be constructed to make the results generalize better to cover for more than one specific environment.

5.2 Measurements

This section is concerned with what to measure during simulations. The purpose of the simulations is to investigate the topics described in section 4.2, and section 4.3. The investigations can be categorized in three different groups:
Method

1. Measurements concerning time to reach convergence:
   - Varying reward and punishment factors
   - Varying learning rates
   - Derivative jumping

2. Measurements concerning performance of the AI:
   - Varying reward and punishment factors
   - Varying learning rates
   - Derivative jumping

3. Measurements concerning difficulty setting.
   - Varying learning rates
   - Varying maximum weight settings
   - Fitness-mapping function with varying target fitness

In the first group there are three main objectives. First, investigate which is more beneficial with respect to convergence time, rewards, punishments, or equal rewards and punishments. Secondly, investigate if the convergence time can be lowered by increasing the learning rate, and/or by using derivative jumping. Finally, compare the effects of increasing the learning rate, compared to using derivative jumping.

When comparing different parameter settings, in order to investigate which settings are more beneficial, the performance of the AI also becomes interesting to investigate. The second group of measurements is concerned with how well the AI performs, both on a discrete scale, as well as on a continuous scale. A discrete performance measure using the discrete values win, and loss, is good, as a victory is still a victory, even if the enemy was not outclassed. Overall it could be considered that this measure is enough, but when faced with an expert player, the quality of solution can also play an important role. The difference of having a supreme solution versus having a mediocre solution might be the decisive factor for if a game is lost or won. This as a supreme solution for one strategic decision may cover for a temporarily weak solution for another strategic decision. Therefore, the performance should be investigated on both a discrete, and a continuous scale.

The third and final group is concerned with investigating if the performance of the AI can be throttled by: varying the learning rate, varying the maximum weight setting, and/or by using a fitness-mapping function. Finally, the three different methods for achieving varying performance should also be compared to each other.

When measuring the time to reach convergence, then it is not the computational time for using the algorithms that is considered. Instead, it is the number of evaluations needed before the rulebase converges, to favor rules that bring victory to the battlefield. Spronck, et al. (2003) defines two metrics for measuring the strength of the dynamic scripting algorithm: the average turning point and the absolute turning point. The average turning point is calculated as the number of encounters needed for the algorithm to produce scripts that outperforms an opponent for at least ten consecutive encounters. The absolute turning point is calculated as the number of encounters needed after which a consecutive run of successful encounters is never followed by a longer consecutive run of unsuccessful encounters.

The second and third categories of investigations are directly dependant on the performance achieved by the AI system. The absolute, and average turning points for measuring the convergence time, are also calculated from the performance of the AI system. Hence it is enough to measure the performance of the AI during simulations.
5.3 Simulations

This section is concerned with how to construct simulations which allows the investigations defined in section 4.2, and section 4.3, to be conducted. Simulations can be constructed in conjunction with the three categories described in section 5.2.

For the first category, time to reach convergence, simulations need to be constructed, which measures both the time to reach adaptation from scratch, and which also measures the time to reach re-adaptation after a previous learning phase. Further, the adaptation time also needs to be investigated against tactics which could resemble some kind of human playing style. Hence five simulation situations have been identified:

1. Time to reach convergence when adapting from scratch.
2. Time to reach convergence when readapting after massive learning.
3. Time to reach convergence under situations which mimics human domain knowledge.
4. Time to reach convergence under situations which mimics human short-term memory.
5. Time to reach convergence under situations which mimics human diversion tactics.

The second category, performance of the AI, can be investigated using the same set of simulation situations, as for the first category.

Finally, for the third category, difficulty setting, the performance of the AI needs to be investigated both under clearly defined circumstances, as well as under situations which simulates more human-like behaviors. Four cases have been identified:

1. Performance under static conditions.
2. Performance under situations which mimics human domain knowledge.
4. Performance under situations which mimics human diversion tactics.

So, what is meant with “more human-like behavior”? In this case, human-like behavior refers to tactics which are likely to be deployed by a human player, in an RTS game. In order to explain each of the three cases which are to resemble some kind of human-like behavior, an example from an RTS game is given for each case.

**Domain knowledge:** In case a human player attacks an enemy, and discovers that the enemy base is only defended with ground units. Then, with the knowledge that ground units perform badly when confronted with aerial units, the human player’s next attack can be carried out using aerial units.

**Short-term memory:** In case a human player is attacked at the right flank by an enemy, for three consecutive encounters, then conclusions can be drawn that it could be beneficial to construct good defenses at the right flank.

**Diversion:** When a human player attacks, it is sometimes preceded by a diversion attack at some other location, in order to draw the enemy’s attention from the real threat.
6 Strategic level dynamic scripting

In this chapter, a theoretical discussion of how dynamic scripting could be applied at the strategic level in an RTS game is presented. Section 6.1 presents how domain knowledge can be used. In section 6.2, an investigation of how rules can be modified, is presented. This is followed, in section 6.3, by a discussion concerning how planning could be incorporated into the dynamic scripting system. In section 6.4 a brief discussion concerning how to evaluate rules is given. Finally, in section 6.5, the dynamically scripted AI is put in relation to other parts of an AI system, as well as to other parts of a computer game engine.

6.1 Domain knowledge

As stated in section 4.1, there are clear advantages of using a goal-directed AI system. The analytical systems of the AI could combine their knowledge into specific goals. These goals could then be fed to the dynamic scripting procedure used as domain knowledge, in order to select rules. Spronck, et al. (2003) maintains several rulebases in their system, one for each character type in the game. When a script is to be created for a character, rules are extracted from the associated rulebase. Now, this structure could be used at the top-level, where each player type is used as superior domain knowledge. Hence one rulebase is created for each player type available in the game. In addition to this, one goalbase, containing all the possible goals for each player type could also be created. Each goal in a goalbase could then be connected to several rules in the associated rulebase, where each rule can be seen as a strategy for achieving that specific goal. When the analytical systems later provides a goal, this goal is also used as domain knowledge, and the selection procedure selects amongst the associated rules for that specific goal. Figure 10 illustrates an example of the setup, where one goalbase and one rulebase are created for each player type. When a goal has been decided for a player, it extracts that goal from its associated goalbase. The extracted goal is then used to select one of its associated rules, from the rulebase belonging to that player type. Together, the player type, and goal information, is used as domain knowledge to retrieve rules that fit the situation at hand.

![Figure 10: Illustration of how one goalbase, and one rulebase, is created for each player type available in a game. Each goal in the goalbases is connected to several rules to choose from when realizing that goal.](image)

6.2 Rules

A rule in the dynamic scripting algorithm is comprised of an optional conditional statement, followed by an action. If the condition is true, then the action should be executed. For instance, Listing 1 shows an example of a rule at the strategic level, for building a tank.
Sample rule for building a tank
1. if CASH > 1000 then
2. BUILD TANK
3. end

Listing 1: Example of a rule for building a tank. If the amount of cash available exceeds 1000, then a tank is constructed.

The rule in Listing 1 would mean that a tank is created if the amount of cash available is greater than 1000. The action to build a tank would however not always be possible to execute, even if the amount of cash exceeds 1000. In order for a tank to be constructed, a heavy weapons factory is also needed. For more advanced weapons, there might even be a need to have reached a certain level of research in the game. Listing 2 shows an example of a rule that takes this into consideration. The rule builds a laser tank, if there is enough cash, there exist a heavy weapons factory to build the tank in, and laser weapons have been discovered.

Sample rule for building a laser tank
1. if CASH > 1000 and NUMBER OF HEAVY WEAPON FACTORIES > 0 and LASER WEAPONS DISCOVERED then
2. BUILD LASER TANK
3. end

Listing 2: An example of a rule for building a laser tank. In this rule, a laser tank is built if the amount of cash is greater than 1000, a heavy weapons factory to build the tank in exists, and laser tanks have been discovered through research.

The rule in Listing 2 would be more useful, but now the rule is not based on a single condition anymore. It is however still rather readable. A final example is given in Listing 3, which shows a rule more suitable for the strategic level AI in an RTS game.

Sample rule for ordering an assault against an enemy
1. if CASH > 12000 and NUMBER OF HEAVY WEAPON FACTORIES > 0 and LASER WEAPONS DISCOVERED and NUMBER OF AERIAL FACTORIES > 0 and HELICOPTERS DISCOVERED and NOT UNDER ATTACK then
2. if NUMBER OF LASER TANKS < 5 then
3. BUILD LASER TANK
4. else if NUMBER OF TANKS < 5 then
5. BUILD TANK
6. else if NUMBER OF HELICOPTERS < 2 then
7. BUILD HELICOPTER
8. else
9. RETRIEVE ENEMY TO ATTACK
10. RETRIEVE ATTACK ROUTE TO ENEMY
11. ORDER ASSAULT
12. end
13. end

Listing 3: Example rule for launching an assault consisting of tanks, laser tanks, and helicopters, against an enemy.

The rule in Listing 3 has six different conditions that need to be fulfilled before an actual assault group can be created, and ordered to attack. In case some of the conditions are not fulfilled, it is up to the analytical parts of the AI to order goals which makes them true. Otherwise the goal to create an assault squad will not be fulfilled until other events make them come true at random. In order to avoid this, and to reach an AI with effective strategic decisions, the analytical parts might need to contain some advanced planning capabilities. In case the AI’s analytical systems do order the fulfillment of the conditions, there may still take some time to reach this change of the internal state. Hence it is not necessary to check the conditions for building an assault group until there have been a change in the internal states for those statements, since they were last checked.
This can be achieved by creating each rule as for example a finite state machine (FSM), with two or more internal states. First, the actual conditions which enables the action to be executed at all, is put into a prestate. This is followed by an active state, in which the action is executed, if its optional main statement is true. Finally, a statement that terminates the active state, decides if the rule is to be terminated. Optionally, this can be followed by a post state, where final actions can be executed, to release resources, etc. Figure 11 illustrates an example of a construction where each rule can be created as a FSM.

Figure 11: Illustration of how a rule can be created as a FSM.

The rule described in Listing 3, can be converted into a rule using the finite state machine setup from Figure 11. This is shown in Listing 4.

Sample rule for ordering an assault against an enemy, as a state machine

1. if STATE = PRE then
2.   if CASH <= 12000 then
3.     wait until CASH > 12000
4.   else if NUMBER OF HEAVY WEAPON FACTORIES = 0 then
5.     wait until HEAVY WEAPON FACTORIES > 0
6.   else if LASER WEAPONS NOT DISCOVERED then
7.     wait until LASER WEAPONS DISCOVERED
8.   else if NUMBER OF AERIAL FACTORIES = 0 then
9.     wait until NUMBER OF AERIAL FACTORIES > 0
10. else if HELICOPTERS NOT DISCOVERED then
11.   wait until HELICOPTERS DISCOVERED
12. else if UNDER ATTACK then
13.   wait until NOT UNDER ATTACK
14. else
15.   SET STATE TO ACTIVE
16. end
17. else if STATE = ACTIVE then
18.   if TARGETS DESTROYED then
19.     SET STATE TO POST
20. else if ALL UNITS DEAD then
21.     SET STATE TO POST
22. else if NUMBER OF LASER TANKS < 5 then
23.     BUILD LASER TANK
24. else if NUMBER OF TANKS < 5 then
25.     BUILD TANK
26. else if NUMBER OF HELICOPTERS < 2 then
27.     BUILD HELICOPTER
28. else
29.     RETRIEVE ENEMY TO ATTACK
30.   retrieve ATTACK ROUTE TO ENEMY
31.   ORDER ASSAULT
32. end
33. else if STATE = POST then
34.   RELEASE ALL UNITS
35. end

Listing 4: Example of an assault rule, similar to Listing 3, but which is based on a FSM. This in order to lower the amount of conditions that needs to be checked in every rule evaluation.
According to Tozour (2002b) one of the advantages of using scripting, and rules, is that it is easy enough to be used by designers, instead of programmers. By using rules which are too complicated, this advantage might be lost. Further, as argued by Spronck, et al. (2003), one of the advantages of using dynamic scripting instead of static scripting, is that it avoids the pitfalls of complex scripts. Complex scripts might include holes not discovered by the development team, which later can be exploited by human players, in order to easily defeat the AI. By comparing the rule in Listing 1, with the rule in Listing 4, it becomes quite obvious that the later is much more complex. As this violates one of the major advantages of using dynamic scripting, the complexity needs to be lowered, and hence this will be discussed in the next section, together with how planning could be implemented in the system.

### 6.3 Planning

By inspecting the rule in Listing 4, section 6.2, it is revealed that some conditions in the prestate are directly related to conditions in the active state. For example, the requirements of an aerial factory to be built, and helicopters being discovered, are coupled with the condition for creating a helicopter. This could be broken down into smaller sub-goals, one for each aspect of creating an assault group. Further, Orkin (2002) argues that if states are designed in a simple, general-purpose, reusable fashion, then they can be used for a variety of situations. If the states are also put together into a hierarchical system, then the lower level states can be concerned with unit-level commands, whilst the higher level states can be used for longer-term planning. The lower level states can also, if kept simple, be reused for many higher level states, such as attacking, or defending. Furthermore, Laird (2000) successfully constructed a goal-directed system, in a hierarchical structure based on the Soar architecture, for agents in FPS games. Figure 12 illustrates an example of the proposed structure, where a rule can point at sub-goals, which in turn are connected to several rules. Lower-level rules can also be linked to even more sub-goals, and this creates a goal-rule hierarchy.

![Figure 12: Illustration of how goals and rules are put together into a hierarchical structure. Several goals can be realized through the same rule, and several rules can call for the same sub-goal. A rule for a goal can be seen as one of possibly several ways of achieving the goal.](image)

The rule to launch an attack, Listing 4 in section 4.2.2, has been broken down into sub-goals, and states. The top goal can be seen in Listing 5.
Sample rule for ordering an assault against an enemy, using heavy units.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>PRE:</td>
</tr>
<tr>
<td>2.</td>
<td>if GROUP NOT CREATED then</td>
</tr>
<tr>
<td>3.</td>
<td>GOAL CREATE HEAVY ASSAULT GROUP</td>
</tr>
<tr>
<td>4.</td>
<td>end</td>
</tr>
<tr>
<td>5.</td>
<td>ACTIVE:</td>
</tr>
<tr>
<td>6.</td>
<td>if NOT UNDER ATTACK then</td>
</tr>
<tr>
<td>7.</td>
<td>RETRIEVE ENEMY TO ATTACK</td>
</tr>
<tr>
<td>8.</td>
<td>ORDER ASSAULT</td>
</tr>
<tr>
<td>9.</td>
<td>end</td>
</tr>
<tr>
<td>10.</td>
<td>TERMINATE:</td>
</tr>
<tr>
<td>11.</td>
<td>TARGETS DESTROYED</td>
</tr>
<tr>
<td>12.</td>
<td>ALL UNITS DEAD</td>
</tr>
<tr>
<td>13.</td>
<td>POST:</td>
</tr>
<tr>
<td>14.</td>
<td>RELEASE ALL UNITS</td>
</tr>
</tbody>
</table>

Listing 5: Example of a rule for launching an assault against an enemy, by using heavy assault units. This could be one rule of several for launching an attack against an enemy.

In Listing 5, a rule for launching an attack with heavy units, the creation of a group has been moved to the prestate of the rule, where it will be executed by calling for a sub-goal. In the active state, the AI would simply check if an enemy is currently attacking it, otherwise it would ask another part of the AI for instructions of whom to attack, and then order the group of units to commence with the assault. Listing 6 shows an example rule for the goal to create an assault group consisting of heavy units.

Sample rule for creating an assault group consisting of heavy units.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>PRE:</td>
</tr>
<tr>
<td>2.</td>
<td>ACTIVE:</td>
</tr>
<tr>
<td>3.</td>
<td>if NUMBER OF UNITS &lt; 12 then</td>
</tr>
<tr>
<td>4.</td>
<td>GOAL CREATE HEAVY ASSAULT UNIT</td>
</tr>
<tr>
<td>5.</td>
<td>end</td>
</tr>
<tr>
<td>6.</td>
<td>TERMINATE:</td>
</tr>
<tr>
<td>7.</td>
<td>NUMBER OF UNITS CREATED = 12</td>
</tr>
<tr>
<td>8.</td>
<td>POST:</td>
</tr>
</tbody>
</table>

Listing 6: Example of a rule for creating an assault group consisting of heavy units. This could be one of several rules for a goal to create a heavy assault group.

The rule in Listing 6 simply calls for another sub-goal, until enough units have been created. The goal to create a heavy assault unit, could in compliance with Figure 12, be realized with rules to create a laser tank, a regular tank, or a helicopter. An example of a rule for creating a laser tank can be seen in Listing 7.

Sample rule for creating a laser tank, which is one of the heavy assault units.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>PRE:</td>
</tr>
<tr>
<td>2.</td>
<td>if NUMBER OF HEAVY WEAPON FactORIES = 0 then</td>
</tr>
<tr>
<td>3.</td>
<td>GOAL CREATE HEAVY WEAPON FACTORY</td>
</tr>
<tr>
<td>4.</td>
<td>end</td>
</tr>
<tr>
<td>5.</td>
<td>if NOT LASER WEAPONS DISCOVERED then</td>
</tr>
<tr>
<td>6.</td>
<td>GOAL RESEARCH LASER WEAPONS</td>
</tr>
<tr>
<td>7.</td>
<td>end</td>
</tr>
<tr>
<td>8.</td>
<td>if CASH &lt; 1000 then</td>
</tr>
<tr>
<td>9.</td>
<td>GOAL COLLECT 1000 CASH</td>
</tr>
<tr>
<td>10.</td>
<td>end</td>
</tr>
<tr>
<td>11.</td>
<td>ACTIVE:</td>
</tr>
<tr>
<td>12.</td>
<td>BUILD LASER TANK</td>
</tr>
<tr>
<td>13.</td>
<td>TERMINATE:</td>
</tr>
<tr>
<td>14.</td>
<td>true</td>
</tr>
<tr>
<td>15.</td>
<td>POST:</td>
</tr>
<tr>
<td>16.</td>
<td>if LASER TANK NOT CREATED then</td>
</tr>
<tr>
<td>17.</td>
<td>RESTART GOAL</td>
</tr>
<tr>
<td>18.</td>
<td>end</td>
</tr>
</tbody>
</table>

Listing 7: Example of a rule to create a laser tank. Regular tanks and helicopters could be other rules for the same goal.
In the rule for creating a laser tank, Listing 7, it can be observed that the conditions for it to be executed are all in the prestate of the rule. The conditions are also negated, and coupled with sub-goals for changing their state. For example, if there is not enough cash available, then a sub-goal for collecting cash is started. When all the preconditions are fulfilled, the state of the rule changes to active. In the active states, a low-level command to build a tank is executed. The terminate condition in the rule is set to true, as the rule has been completed when a tank have been created. In case some other order or environmental event has made it impossible to create a laser tank when it is ordered, this is taken care of in the post statement. The post statement simply restarts the rule if no laser tank has been created.

The layout which previously has been discussed, to create a goal-rule hierarchy, can be seen as a planning system. According to Nareyek (2002), a plan is a solution which brings an initial world description into a goal world description, through a sequence of actions realized by transitions between partial world descriptions in between the initial and goal descriptions of the world. This could for example be translated to the rule system previously presented, as follows: If an assault group is ordered to be created, then the initial world description would be the world without that specific assault group. The goal description of the world would include an extra assault group. Now, the transition between the two world descriptions would be realized by a sequence of transitions between partial world descriptions, in which factories are built, cash collected, and tanks created. Hence the goal-rule hierarchy can be seen as a planning system.

The sequence of rules used for reaching a goal description of the world that contains no enemies; from an initial world description that does, is unknown. This as the learning procedure affects what rules are used. This together with the fact that everything is realized through unit-level commands at the bottom of the hierarchy would make planning in the system become emergent.

### 6.4 Rule evaluation

For the dynamic scripting algorithm to work, the rules need to be evaluated, and their weights adapted. Spronck, et al. (2003) evaluates the rules used, after each encounter had been completed. In the system previously discussed, the evaluation of a rule could be processed when a rule have reached the terminated state. This would however not be feasible for all rules. For example, if the rule to build a laser tank was to be evaluated as soon as the laser tank had been constructed, it would include no information concerning how useful that specific laser tank was on the battlefield. Some information needs to be included in the rules, regarding what they should consider when evaluating. At the higher levels in the hierarchy, it would be enough to consider how successful an individual strategic goal was, but at the lower levels, evaluation might need to be delayed until the goals at the higher levels have been terminated. For example, the rule to build a laser tank could be evaluated as soon as that tank, together with all other units belonging to the same group of units, have been destroyed, their group dissolved, or their goal achieved.

Timing properties might also need to be included in rules for major strategic decisions. If results for a rule have not been reached when a certain amount of time have elapsed, then that specific rule might not be effective enough. It could take the whole game to reach results, and then there would be no learning taking place. Good measures for how successful a rule for achieving a goal was could be the amount of damage dealt, number of units destroyed versus lost, or amount of cash collected.
6.5 AI in a larger perspective

Previously in this chapter it has been discussed how dynamic scripting could be applied at the strategic level in an RTS game. The goal-rule system covers four of the important properties described in section 4.1:

1. Learning. Dynamic scripting is a technique for learning.
2. Resource management. Goals to collect resources, construct tanks, and invest into research are included in the preconditions of rules.
3. Implicit modeling. Weights implicitly reflect how the opponents previously have been playing.

In order for the dynamic scripting system previously described, to be operational, there is a need to provide it with goals. As mentioned, these goals can be created by another part of the AI, which considers temporal-, and spatial-reasoning, modeling, etc. Goals decided by the analytical system could then be fed into the dynamic scripting system, where various rules are extracted to realize each decided goal.

In each frame processed in a computer game, there are lots of computations to perform. Figure 13 illustrates an example of a very simplistic game loop, which only includes some of the most basic sub-systems which needs to be processed in each frame.

![Figure 13: Illustration of a simple game loop, which needs to be processed in each frame.](image)

In order to reach a framerate of 60 fps, each frame gets around 1/60th of a second, or 0.016 seconds, to perform its calculations. It then becomes obvious that each sub-system needs to be very computationally cheap. According to Woodcock (2000), the amount of time reserved for the AI system in a computer game of today, is usually around 25% of the total frame time. Each individual unit in an RTS game needs to be given processing time, for reactive behaviors, in each frame, or at the very least in each other frame, in order for them to act realistically. This together with the assumption that each player might possess a couple of hundred units makes it clear that there is not much room for expensive algorithms. Pottinger (2000), states that approximately 60-70% of the time used by the AI in their RTS game Age of Empires 2™, is used for pathfinding. Hence there is even less time left for the dynamically scripted part of the AI. The remaining time can however be considered enough, as strategic decisions usually don’t need to be made once every 60th second. Instead, strategic decisions span a slower timescale intended for longer-term goals, which makes it suitable for running in a separate thread that can be interrupted by higher priority threads.
Laird and van Lent (2000) argues that controlling an RTS game with only a mouse, is a significant part of the challenge for human players. In order to make the battle fair, they also argue that a strategic opponent must obey to the same limitations a human player would, such as reaction times. This together with the fact that the amount of time available for checking rules is very limited makes it suitable to break up the dynamic scripting system into different layers.

In a larger perspective, there could be two major layers created, unit behavior, and strategic behavior. The unit layer could then be shared by both an AI player, and a human player, and it would have the task to control how the units behave on the battlefield. As it is a real-time environment, rules at the unit-level would also need to be evaluated in real-time. The strategic-level behavior would on the other hand only be used for the AI, as the human player would control this behavior itself. In order for the AI to obey the same limitations as a human player, the strategic-level would also need to be executed on a slower time scale, to reflect the time it takes for a human player to execute a strategic decision (for example: scroll map to position of weapons factory, select which unit to build, etc.).

The layer for strategic behavior could further be broken down in two individual layers, strategic decisions, and group decisions. The layer for strategic decisions would include the analytical parts of the AI, as well as the top-most goals in the goal-rule hierarchy. The layer for group decisions would be used for coordinating group behavior, and hence provides units belonging to a group with goals.

Figure 14 illustrates an example construction of how the AI could be divided into these specific layers. At the strategic level, which is only used by the AI, an analysis system would provide an army general with goals, based on modeling, communication, and terrain analysis. The army general would in turn realize its plans with sub-goals, which eventually reaches the group-level, where a commander is in control. The group-level is also only used by the AI, and it would provide the unit-level with further sub-goals, to coordinate the behavior of a group. At the unit-level, used by both the AI, and the human player, a captain would be in control. The captain has the task to control the behavior of the unit it commands. An army general can also bypass the group level, if it needs to control the behavior of individual units, by sending goals directly to the captain for a unit. All three levels in the dynamically scripted part of the AI can also send queries to the analytical system, regarding positions in the environment.

![Figure 14: Illustration of how dynamic scripting can be used together with other AI systems, such as terrain analysis, in order to create a goal-directed strategic planning system, which is able to adapt according to changing human player tactics.](image-url)
In a larger perspective, each captain at the unit-level in the AI hierarchy, Figure 14, would be connected to an actual object in the game world. It would provide this object with instructions of how to move, and in turn it would receive sensory input from the object. All objects in the game world would when ordered to a specific location, ask the pathfinding system in the AI system for instructions of how to get there.

In order for the game to be played in real-time, the currently active rules for all units would be evaluated in a separate system. Rules would then be inserted to, and extracted from this system as they reach, and leave their active state. The rules in this system would translate into reactive behavior of the units in the game. Figure 15 illustrates how the suggested AI system could be connected to the other systems of a game environment, in a very simplified construction.

Spronck, et al. (2003), also mentions the use of application specific knowledge in the dynamic scripting algorithm. Rules are created with priorities, so that the order in which to check and evaluate the rules becomes dependant on their priorities which are set in an offline phase depending on their use. A possible translation of this to the goal-rule hierarchy would be to let the analytical systems decide, set, and change the priorities for different rules during run-time. For example, a rule to create a cash collecting unit could be given higher priority than a rule to create a tank, when the amount of available cash is low. Without any priorities, the two rules would be competing to spend the available cash, and if the rule to create a tank is executed before the rule to create a cash collecting unit, then the flow of cash could possibly be jeopardized. Hence, rules are given different priorities, and when several rules are active at the same time, the rule with the highest priority is processed first.
7 Test environment

This chapter presents the test environment that will be used to carry out simulations. In section 7.1, the setup of the test environment is described. This is followed, in section 7.2, by the procedure used for carrying out a simulation in the environment. The test cases that will be used during simulations are presented in section 7.3. Finally, section 7.4 presents the fitness evaluation and weight adaptation formulas.

7.1 Simulation setup

As this dissertation is concerned with RTS games, the test environment will be constructed to resemble some form of RTS game. Two players will be created in each simulation, where they are facing each other on a battlefield. In section 5.1, the method for constructing a test environment was described. An artificial environment, which allows comparison of various parameters under similar circumstances, was chosen as method. Hence the test environment will be very simple in its construction. There will be no graphics, no resources gathering, no construction of buildings, and no advanced mission planning. Instead, the two players will combat each other by launching assault raids against each other in an alternating fashion. In the beginning of each simulation, each player will be given 1500 in cash, and during each encounter, cash will be withdraw according to the outcome of the encounter. Before an actual assault commences, the attacker will select a rule for attacking, and the defender will select a rule for defending. A simulation will end when one of the two players has run out of cash. Figure 16 shows an illustration of the constructed battle procedure.

![Figure 16: Illustration of the battle procedure used during simulations. First, both players will be initialized with 1500 in cash. This is followed by a battle, in which the players launch assault raids against each other in an alternating fashion. The battle ends when one of the two players has run out of cash.](image-url)

The structure for applying dynamic scripting, presented in chapter 6, will be delimited to cover two levels in a goal-rule hierarchy (one goal level, and one rule level). The result of this is that adaptation only occurs at one level (rule level). By limiting the size of the structure, simulations can be carried out under more similar circumstances,
as the effect of chance is also limited. This can in turn make the results generalize better to other situations.

As stated, the goal-rule hierarchy will be limited to two levels. Two topmost goals will be used, attack and defend. Each of these goals will have a total of eight rules each for achieving the goal. In order for adaptation to be possible in the environment, some rules need to be stronger, and some rules need to be weaker, according to a predefined scheme. If the effect of using a rule was to be set by chance, various parameter settings could possibly not be measured under similar conditions, and hence the effectiveness of rules need to be set in advance. Furthermore, if the system does not contain some rules that are more effective, and other rules which are less effective, then the system could possibly end up in a state where convergence is not possible. This is also one of the reasons for choosing a simple environment; it is known in advance that reaching convergence is always possible, and hence the time to reach convergence can always be measured. Figure 17 illustrates how two goals will be created in the test environment, and how they will be connected to eight rules each, which are setup to be strong and weak against certain rules according to a predefined scheme.

Figure 17: Illustration of how the goals and their respective rules are setup in order to assure that each strategy is strong at some point, but also weak at another. Example ratios are highlighted. For example, attack rule 2 has an 80% chance of victory against defensive rule 2. Defensive rule 2, on the other hand, only has a 20% chance of victory against offensive rule 2. It can also be observed that attack rule 2 is strong versus the defensive rule that is strong against attack rule 1.

In Figure 17, it can be observed that each rule will have an 80% chance for victory in battle against one rule, 60% chance against a second, 40% against a third, 20% against a fourth. Rules not connected will have a 50% chance against each other.

7.2 Simulation procedure

When simulating a battle between two players, one will be a dynamically scripted player, and the other will be a manually designed player. The manually designed AIs are referred to as opponent players, and the dynamically scripted AI is simply referred to as the AI. During simulation, both players will be given instructions to always activate the defend goal. This gives that both players always will have a rule ready for defending themselves. As already stated, the simulations will be carried out by ordering the two players to attack each other, in an alternating fashion. This is done by giving them an attack goal, which then activates an attack rule.
When an opponent player is assigned a goal, it will select the appropriate rule according to a predefined scheme. A dynamically scripted AI will select a rule according to the dynamic scripting procedure. Furthermore, for each rule that is activated for a player, an amount of ten cash will be withdrawn from that player’s stash of cash, in order to assemble a group. The amount of cash given to a group symbolizes a number of individual units. Hence, ten in cash gives that a group consists of ten units.

By activating an attack rule, an encounter between two players is initiated. The player having the attack rule starts attacking, and then the defending player counter-attack. When one of the two players attacks the other, the probability coupled with the currently activated attack, or defend rule, will be used to randomly decide the results of the attack with the probability of the rule. If the result is positive, the cash for the targeted group will be decreased by one. The encounter will proceed until one of the two groups has run out of cash. Figure 18 illustrates an example of the procedure for an encounter between two players.

As stated, an encounter between two groups of units will last until one of the two groups has run out of units, and during each encounter, the groups for the players will attack each other in an alternating fashion, starting with the offensive group. An encounter between an offensive group $A$ using rule $A_1$, and a defensive group $B$ using rule $D_1$, where $P_1$ is the probability that $A_1$ hits $D_1$, and $P_2$ is the probability that $D_1$ hits $A_1$, proceeds, in pseudo-code, according to Listing 8.

```
Encounter between two groups A, and B
1. Let P1 be the probability that Group A hits Group B //i.e. the probability that //the currently used rule for group A hits the currently used rule for group B
2. Let P2 be the probability that group B hits group A //Same as above
3. Let L1 be the amount of units for group A
4. Let L2 be the amount of units for group B
5. while L1 > 0 and L2 > 0 do
6. Set P to random value between 0 and 1
7. if P <= P1 then
8. Withdraw one from L2
9. end
10. if L2 > 0 then
11. Set P to random value between 0 and 1
12. if P <= P2 then
13. Withdraw one from L1
14. end
15. end
16. Perform other game stuff
17. end
```

![Figure 18: Illustration of how an encounter between two players proceeds.](image-url)
Test environment

| 18. if L1 > L2 then |
| 19. victory for group A |
| 20. else |
| 21. victory for group B |
| 22. end |

Listing 8: Example of how an encounter proceeds between two groups A and B.

7.3 Test cases

In section 5.3, a total of 5 different test situations were identified for investigating adaptation times, performance, and performance throttling:

1. Adaptation under static circumstances.
2. Re-adaptation under static circumstances.
3. Adaptation under human-like diversion tactic.

An obvious solution for the first situation is to let the opponent always deploy the same tactic, in order to become static. The time to reach adaptation against the constant tactic should then become clearly visible. In order to carry out measurements on a re-adaptation phase, a clearly identifiable switch of tactic is needed. A solution for this is to let the opponent use two different rules. The first rule is deployed for a predefined number of encounters (80), after which the second rule is deployed. During the use of the first rule, the dynamically scripted AI should be able to learn how to defeat that rule. When the second rule starts being deployed, a re-adaptation phase begins, which can be measured. In order to magnify the results of the change in tactic, the second rule can be selected as the best available rule. Further, this kind of behavior could also be used to resemble a behavior where the human player tries to trick the learning procedure into learning a behavior which later can be exploited. For the third situation, diversion, one rule could be used as main rule. At a regular interval, another rule could be deployed to simulate a diversion. Diversions are also something often being deployed by human players.

The fourth situation, domain knowledge, is to resemble some kind of human-like domain knowledge. For example, if the human player is attacked by foot soldiers a couple of times in a row, than the human player could draw the conclusion that the enemy is likely to attack with foot soldiers again. Hence, troop-mines are deployed in a valley where the enemy has passed every time.

One possible realization of this would be to exploit how the rules are structured in the test environment. The next consecutive rule is always strong against the rule that is strong against the currently deployed rule. By measuring the fitness, the next consecutive rule could be deployed when the performance drops below the break-even point, i.e. a loss. The next consecutive rule should represent beneficial behavior in the current situation, as it has an 80% chance at victory against the rule that the currently deployed rule has a 20% chance against.

According to Spronck, et al. (2003), this kind of behavior is also the most similar to how a human player would play: if the currently used strategy was unsuccessful; try something else. Furthermore, a human player is also likely to put some faith into currently deployed tactics; hence a short memory of the previous five results will be stored. If the average fitness of previous results becomes negative, then the next consecutive rule will be used.
Test environment

The fifth situation is to resemble some form of short-term memory. A human player is likely to remember which tactics that previously have been good. To realize this, the fitness results for all rules are stored. Every time a rule should be selected, the rule with the best average fitness will be selected. To make the memory short-term, fitness results for ten consecutive encounters are stored. This kind of information is also something that a human player is likely to have, as the most beneficial tactic recently used, is likely to be remembered.

The different opponent tactics just described can be divided in two groups, static and dynamic opponent types. The static opponents will select rules according to a predefined scheme, and the dynamic opponents according to previously experienced results. The different opponents are summarized in Table 1.

Table 1: Descriptions of the different manually designed opponent types that will be used during simulations, as opponents against the dynamically scripted AI

<table>
<thead>
<tr>
<th>Enemy tactic</th>
<th>Description</th>
</tr>
</thead>
</table>
| Constant     | Type: Static  
               Purpose: To allow measurements on adaptation phase alone.  
               Operation: This opponent will always deploy the first rule for each assigned goal. |
| Changing     | Type: Static  
               Purpose: To allow measurements on re-adaptation phase alone.  
               Operation: This opponent will deploy the first rule for each assigned goal, during the first 80 encounter pairs. After 80 encounter pairs have elapsed, the second rule will be deployed for each assigned goal. The second rule has an 80% chance of beating the rule that is strong against the first rule. |
| Diversion    | Type: Static  
               Purpose: Simulate the human-like behavior of a diversion.  
               Operation: During four consecutive encounter pairs, the first rule is deployed. For the fifth encounter pair, the fourth rule will be selected as a diversion. This is repeated throughout the simulation. |
| Consecutive  | Type: Dynamic  
               Purpose: Simulate the human-like behavior of domain knowledge.  
               Operation: This opponent will calculate an average fitness over the last five consecutive encounters. If the result is below 0.5 i.e. the break-even point, then a change of rule will occur, and the next consecutive rule is used. |
| Best         | Type: Dynamic  
               Purpose: Simulate the human-like behavior of short-term memory.  
               Operation: This opponent will store the last ten fitness results for every rule. An average will be calculated for every rule, and the rule with the highest average will be selected. |

7.4 Fitness and weight adaptation

In order for the dynamic scripting algorithms to work, there needs to be a fitness function, whose results are used when adapting the rulebases. The fitness function will be kept simple, in compliance with the other parts of the test environment.

As everything is based on individual units combating each other, a fitness function will be created which is based on the amount of units left, $u_i$, versus the amount of units killed, $u_k$. The fitness is calculated as a value in the range from 0 to 1, and as every group of units initially contains ten units, the following formula has been constructed:

$$f = \begin{cases} 
0.5 + 0.05u_i & \text{if } u_i > 0 \\
0.05u_k & \text{otherwise} 
\end{cases}$$
When the fitness for a rule has been calculated, it’s associated weight need to be adjusted. This is done by applying a weight-update function. The weight-update function used by Spronck, et al. (2003) will be used, and a new weight, \( W_{NEW}^p \), for a rule is calculated as follows:

\[
W_{NEW}^p = \begin{cases} 
\max \left( 0, W_{OLD} - M_p \left( \frac{b-f}{b} \right) \right) & \text{if } (f < b) \\
\min \left( W_{OLD} + M_R \left( \frac{f-b}{1-b} \right), W_{MAX} \right) & \text{otherwise}
\end{cases}
\]

where \( W_{OLD} \) is the old weight, \( M_p, M_R, \) and \( W_{MAX} \) are constants describing the maximum penalty, the maximum reward, and the maximum weight value, respectively, \( f \) the fitness, and \( b \) the break-even point. Max and min are functions that return the highest, and the lowest value, respectively. The break-even point is defined as the fitness score that separates a victory from a loss, and it will be set to 0.5 in all simulations.

When including the derivative in the weight-update function, the new weight, \( W_{NEW}^{P+D} \), for a rule, will be calculated as a sum of the proportional weight change, and the derivative of the fitness scores, as follows:

\[
W_{NEW}^{P+D} = W_{NEW}^p + W_{MAX} \cdot W_{NEW}^D \left( 0 \leq W_{NEW}^{P+D} \leq 1 \right),
\]

where \( W_{NEW}^p \) is the weight calculation according to the old algorithm, and \( W_{NEW}^D \) is the weight calculation according to the derivative of the fitness values.

The derivative of the fitness values, \( W_{NEW}^D \), will be calculated by creating a Non-Uniform-Rational-B-spline, NURB, of degree four with evenly distributed knots, and then extracting the derivative from it. This will now be explained top-down starting with calculating \( W_{NEW}^D \), as follows:

\[
W_{NEW}^D = \begin{cases} 
\frac{1}{n} \cdot d(n-1, f, w_{normal}) & \text{if } (\text{sign}(d(n, f, w_{normal})) \neq \text{sign}(d(n-1, f, w_{normal}))) \\
\frac{1}{n} \cdot d(n-1, f, w_{normal}) & \text{otherwise}
\end{cases}
\]

where \( n \) is the number of old fitness scores available, \( d(m, f, w) \) is the derivative in point \( m \) on a NURB curve based on a fitness vector \( f \), and a weight vector \( w \). Sign is a function that returns true if the value is positive and false otherwise. Both \( w_{normal} \), and \( w_{turn} \) represents two different weight vectors for a NURB curve. Observe that \( w \), \( w_{turn} \), and \( w_{normal} \) is not to be confused with the rule weights. They describe weights for pulling the NURB curve towards the NURB control points, here constituted of the fitness values.

Defining \( w_{normal}, w_{turn} \), and \( f \) as follows:

\[
w_{normal} = \left\{ \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n}{n} \right\},
\]

\[
w_{turn} = \left\{ \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}, 10 \right\},
\]

\[
f = \left\{ f_{i-\delta}, \ldots, f_{i-1}, f_i \right\},
\]
where $f_t$ represents the fitness result at time, or encounter $t$.

Figure 19 illustrates three examples of how NURB curves are created from various fitness and weight vectors. The derivative is then extracted from each NURB. A NURB is a b-spline with certain properties (non-uniform, and rational). A b-spline in turn, is a piecewise polynomial, based on control points. For further information concerning NURBs, b-splines, and for information concerning how to calculate the derivative in a point on a NURB curve, see Piegl and Tiller (1995).

The reason for using two different weight vectors for the NURBs will now be explained. As the derivative used for the weight adaptation is calculated in the point of the previous fitness score, in order to reach a smoothing effect, the derivative might point in the wrong direction when there is a local maximum, or minimum on the fitness curve between the most recent result, and the previous result. If there is a maximum, or minimum in between the two points, then the $wv_{\text{turn}}$ will be used, which makes the most recent fitness result have much more impact on the resulting NURB curve, which in turn impacts on the derivative of the curve. Basically, this lowers the risk for increasing, or decreasing a weight value, whose fitness value represents the opposite action. This can also be seen in Figure 19.

After a new weight value has been established, the old weight value is replaced with the newly calculated value. The weights for the other rules for the same goal are also changed, in order to keep the total weight sum constant. Hence, lowering the weight for one rule will increase the weights for the other rules.

At the initialization phase of each simulation, the weights for all rules under a specific goal will be equally distributed such as to make the total weight sum equal to 1.0. As previously stated, each goal will have a total of eight rules, hence, initially all rules will have a weight of 0.125. In all simulations, the maximum weight value will be set to the maximum of 0.5, and 1.0 divided by the number of rules present for a specific goal. This yields that the maximum weight value will be 0.5 for all rules that have sister rules under the same goal. The break-even point in all simulations will be set to 0.5, except where specified otherwise. Fitness values below 0.5 means that an encounter has been lost.
8 Simulation results

In this chapter, the results from the simulations are presented. Section 8.1 presents results concerning convergence time, and section 8.2 presents results concerning performance. Finally, in section 8.3, results concerning throttling the performance of the AI are presented.

All results in this chapter are presented together with their respective 95% confidence values. Under the assumption that the results are normally distributed, the confidence values have been used to perform significance tests on the results. For example, assuming that the results are normally distributed, then an average turning point of 16, with a 3.9 confidence value, gives that the average turning point is to 95% certainty located in the interval [12, 20]. A second average turning point of 24, with a 3.0 confidence value, gives that the average turning point to a 95% certainty is located in the interval [21, 27]. As the two intervals do not overlap, the former average turning point (16) is significantly lower than the later (24), to a 95% certainty, but only as long as the assumption of the results being normally distributed holds.

8.1 Convergence time

This section presents results concerning convergence time. First, in section 8.1.1, the punishment and reward factors are compared. This is followed, in section 8.1.2, by results obtained when varying the learning rate. In section 8.1.3, convergence times for derivative jumping, are presented. Finally, in section 8.1.4, using derivative jumping is compared to changing the learning rate. All fitness values presented in this section are calculated as an average of the actual fitness for the last ten consecutive encounters, similar to Spronck, et al. (2003).

To measure the adaptation time, a turning point has been calculated for each conducted simulation. The turning point measure that has been used is a combination of the average, and absolute turning point calculations used by Spronck, et al. (2003) (see section 5.2). The turning point is calculated as the number of the first encounter: $x$ that is followed by at least ten consecutive successful encounters (inclusive). After which the amount of consecutive successful encounters is never followed by a longer run of consecutive unsuccessful encounters.

To quantify the re-adaptation time against the tactic changing opponent, another calculation has been used, which measures the length of the longest interval of unsuccessful encounters occurring after the turning point have been reached.

8.1.1 Punishment or reward

Three different punishment and reward settings have been used during simulations. Equal punishments and rewards, half punishments compared to rewards, and half rewards compared to punishments. Simulations have been conducted against five different opponents: constant, changing, diversion, consecutive, and best (see section 7.3). For each opponent, and parameter setting, 20 simulations have been carried out. For each encounter, the fitness of the dynamically scripted opponent has been recorded.

Table 2a shows the average turning points, calculated from each set of 20 simulations, against all five opponent. In Table 2b, the lowest and highest turning points are found. Median turning points have also been calculated, but are excluded as they were similar to the average values.
Simulation results

Table 2: Average turning points in table a, low and high turning points in table b, for three different punishment and reward settings, against five different opponents

<table>
<thead>
<tr>
<th>Setting</th>
<th>Opponent type</th>
<th>Opponent</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Constant</td>
<td>Changing Diversion</td>
<td>Consecutive Best</td>
</tr>
<tr>
<td>0.5p, 1.0r, 1.0lr</td>
<td>Avg. 95%</td>
<td>12</td>
<td>2.7 16 3.9 16 4.9</td>
<td>116 33.3 132 32.1</td>
</tr>
<tr>
<td></td>
<td>Avg. 95%</td>
<td>16</td>
<td>5.0 15 4.1 14 3.6</td>
<td>97 35.6 97 31.4</td>
</tr>
</tbody>
</table>

It can be observed in Table 2 that there are clear differences between the static and dynamic opponent types. The static types are coupled with much lower turning points than the dynamic opponents. The highest turning point for both the consecutive, as well as for the best opponents, is indefinite, meaning that a turning point could never be reached before the amount of available cash had been consumed. This clearly indicates that the AI had more problems defeating dynamic opponents than static opponents. It can also be observed in Table 2, that it can be slightly more beneficial to have equal punishment, and reward factors, or higher punishments than rewards. No significant differences are however found, as the 95% confidence intervals overlap. In Figure 20, the average fitness development for the various punishment and reward settings against the constant opponent is plotted.

Figure 20: Average fitness development for three different punishment and reward settings, against the constant opponent. Error bars represent 95% confidence intervals.

In Figure 20, it can be observed that the average performance development is similar for all three parameter configurations, and that the 95% confidence intervals mostly overlap. All three configurations reach performance levels between 0.7, and 0.8. This can be considered good, as the most productive rule at any time, only has an 80% chance at victory. Further, as the maximum weight setting has been kept at 0.5, one single rule cannot dominate, and hence rules with less than 80% chance at victory will
Simulation results

also be selected sometimes, resulting in performance below 0.8. Figure 21 illustrates the average fitness development for the three parameter configurations, against the tactic changing opponent.

![Figure 21: Average fitness development for three punishment and reward settings, against the tactic changing opponent. Error bars represent 95% confidence intervals.](image)

From Figure 21 it can be concluded that the performance development is similar for all three parameter configurations, when readapting against the tactic changing opponent. It can however be seen that having equal punishment, and reward factors, is slightly more beneficial, but not to a significant extent. Further, it can also be observed that the setup of the tactic changing opponent seems to be functional, as the performance drops drastically when the tactic is changed. Figure 22 illustrates the average re-adaptation time for three different punishment and reward settings, against the tactic changing opponent.

![Figure 22: Average number of encounters to readapt, for three different punishment and reward settings. 95% confidence intervals are shown by the error bars.](image)

Again, it can be observed in Figure 22 that it can be slightly more beneficial to have equal parameter settings, or higher punishments than rewards, compared to having higher rewards. The results are however not significant, as the confidence intervals overlap for all three configurations.
8.1.2 Learning rate

In this section, 20 simulations have been carried out against all five opponent types, for three different learning rates, 1.0, 2.0, and 4.0. The punishment, and reward factors have been set equal in all simulations. In Table 3, the average, low, and high turning points are presented, for each learning rate, and opponent. Median values have been excluded as they were similar to the average values.

Table 3: Average turning points in table a, low and high turning points in table b, against five different opponents, for three different learning rates

<table>
<thead>
<tr>
<th>Opponent type</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opponent</td>
<td>Constant</td>
<td>Changing</td>
</tr>
<tr>
<td>Setting</td>
<td>Avg.</td>
<td>95%</td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr</td>
<td>11</td>
<td>1.9</td>
</tr>
<tr>
<td>1.0p, 1.0r, 2.0lr</td>
<td>8</td>
<td>2.2</td>
</tr>
<tr>
<td>1.0p, 1.0r, 4.0lr</td>
<td>8</td>
<td>2.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Turning point - high, low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opponent type</td>
</tr>
<tr>
<td>Opponent</td>
</tr>
<tr>
<td>Setting</td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr</td>
</tr>
<tr>
<td>1.0p, 1.0r, 2.0lr</td>
</tr>
<tr>
<td>1.0p, 1.0r, 4.0lr</td>
</tr>
</tbody>
</table>

In Table 3, it can be observed that increasing the learning rate yields less time to reach convergence, against the dynamic opponents. For the static opponents, the adaptation times again overlap, and no significant results were achieved. It can also be observed that the highest turning point against the dynamic opponent types, yet again, is indefinite. This strengthens the fact that the dynamic scripting algorithm had clear difficulties in beating the dynamic opponents. Figure 23, illustrates the average fitness development against the constant opponent, for all three learning rates.

In Figure 23, it can be observed that the performance development is similar for all three learning rates. It can also be seen that the performance development becomes slightly steeper, when increasing the learning rate, but it does however not produce
any significant differences against the constant opponent. Again, the performance stabilized around 0.7 to 0.8. In Figure 24, the average fitness development against the tactic changing opponent, for three different learning rates, is plotted.

![Figure 24: Average fitness development for three different learning rates against the tactic changing opponent. 95% confidence intervals are shown at every 50th encounter.](image)

In Figure 24, it can be observed that increasing the learning rate, significantly lowers the re-adaptation time, against the tactic changing opponent. Figure 25, illustrates the average re-adaptation time against the tactic changing opponent, for three different learning rates.

![Figure 25: Average number of encounters to readapt against tactic changing opponent, for three different learning rates. 95% confidence intervals are included.](image)

Again, it can be observed in Figure 25 that increasing the learning rate significantly lowers the re-adaptation time against the tactic changing opponent. In Figure 25, it can however also be observed that having a learning rate of 2.0, and 4.0, both lowers the re-adaptation time, compared to having a learning rate of 1.0. Another observation to be made is that the confidence interval seems to become tighter, when increasing the learning rate.

### 8.1.3 Derivative jumping

In this section, results concerning convergence time when using derivative jumping is presented. Three different settings of the punishment and reward settings have been used, which is also complemented with the derivative of the fitness history. All five opponents have been used during simulation, and 20 simulations have been conducted.
Simulation results

for each opponent. Table 4 contains the average, low, and high turning points against all five opponents, for three different parameter configurations. Median values have also been calculated, but are excluded as they were similar to the average values.

Table 4: Average turning points, in table a, low and high turning point, in table b, for three settings of derivative jumping, against five different opponents

<table>
<thead>
<tr>
<th>Setting</th>
<th>Opponent type</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opponent</td>
<td>Constant</td>
<td>Changing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
</tr>
<tr>
<td>0.5p, 1.0r, 1.0lr +df/dt</td>
<td>12</td>
<td>3.6</td>
<td>11</td>
</tr>
<tr>
<td>1.0p, 0.5r, 1.0lr +df/dt</td>
<td>12</td>
<td>2.7</td>
<td>12</td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr +df/dt</td>
<td>12</td>
<td>3.2</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Opponent type</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opponent</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>0.5p, 1.0r, 1.0lr +df/dt</td>
<td>1</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>1.0p, 0.5r, 1.0lr +df/dt</td>
<td>3</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr +df/dt</td>
<td>2</td>
<td>32</td>
<td>1</td>
</tr>
</tbody>
</table>

From Table 4, it can be observed that it can be slightly more beneficial to have equal punishment and reward factors, or higher punishments than rewards, even when including the derivative. It can also be observed that the convergence time is much higher for the dynamic opponents, compared to the static opponents. Further, it can be seen that the highest turning point against the dynamic opponents, is again indefinite. In Figure 26, the average fitness development against the constant opponent is plotted, for different parameter configurations.

![Figure 26: Average fitness development against constant opponent for three different settings of derivative jumping. Error bars represents 95% confidence intervals.](image)

In Figure 26, it can again be seen that having equal parameter settings, is slightly more beneficial, than having either punishments, or rewards higher, against the constant opponent. As before, performance stabilized around 0.7 to 0.8. Figure 27 illustrates the average fitness development when using derivative jumping, and different punishment and reward settings, against the tactic changing opponent.
In Figure 27 it can be observed that it is slightly more beneficial to have equal punishment and reward settings, when including the derivative, against the tactic changing opponent, with regard to re-adaptation time. Figure 28 illustrates the average re-adaptation time when including the derivative, for three different punishment and reward settings, against the tactic changing opponent.

Again, it can be observed in Figure 28, that having equal punishments and rewards, or having higher punishments than rewards, is slightly more beneficial with regard to convergence time against the tactic changing opponent, when including the derivative.

### 8.1.4 Learning rate versus derivative jumping

In this section, results when varying the learning rate, is compared to results when including the derivative. Figures are based on material from sections 8.1.1, 8.1.2, and 8.1.3. In Figure 29, the average performance development against the constant opponent, for three different learning rates, and one setting of including the derivative, is illustrated.
Simulation results

Figure 29: Average fitness development against constant opponent, for three different learning rates, and for derivative jumping. 95% confidence intervals are included.

In Figure 29, it can be observed that only increasing the learning rate, results in slightly better adaptation behavior, compared to including the derivative, against the constant opponent. Figure 30 illustrates the average performance development against the tactic changing opponent, for different learning rates, and when including the derivative.

Figure 30: Average fitness development against tactic changing opponent, for three different learning rates, and derivative jumping. 95% confidence intervals are shown by error bars.

From Figure 30, it can be observed that including the derivative results in lower re-adaptation time against the tactic changing opponent, compared to having a learning rate of 1.0. Including the derivative however behaves similar to having a learning rate of 2.0, or 4.0. Figure 31, illustrates the average re-adaptation time against the tactic changing opponent, for various punishment and reward settings, learning rates, and when including the derivative.
Simulation results

Figure 31: Average number of encounters to readapt, for various punishment and reward factors, and for derivative jumping. Error bars are 95% confidence intervals.

In Figure 31, it can be seen that increasing the learning rate, significantly lowers the re-adaptation time against the tactic changing opponent. It can also be seen that including the derivative, and having equal punishments, and rewards, yields similar re-adaptation times to having a learning rate of 2.0. Including the derivative for equal punishment and reward settings, hence also yields significantly lower re-adaptation times, compared to not including the derivative.

8.2 Performance

The results in this section are based on the same set of simulations that were conducted for measuring the convergence time, in section 8.1. This section however presents the performance values achieved during simulations. Both average and median values have been calculated, but the median values are excluded as they were similar to the average values.

8.2.1 Punishment or reward

As in section 8.1.1, three different settings of the punishment and reward factors have been used. 20 simulations have been carried out, against five different opponents.

Table 5: Average fitness against five opponents, for three settings of punishment and reward factors

<table>
<thead>
<tr>
<th>Opposition</th>
<th>Static</th>
<th>Changing</th>
<th>Diversion</th>
<th>Consecutive</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opponent type</td>
<td>Setting</td>
<td>Avg.</td>
<td>95%</td>
<td>Avg.</td>
<td>95%</td>
</tr>
<tr>
<td>0.5p, 1.0r, 1.0lr</td>
<td>0.688</td>
<td>0.008</td>
<td>0.632</td>
<td>0.009</td>
<td>0.625</td>
</tr>
<tr>
<td>1.0p, 0.5r, 1.0lr</td>
<td>0.687</td>
<td>0.009</td>
<td>0.637</td>
<td>0.009</td>
<td>0.619</td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr</td>
<td>0.687</td>
<td>0.013</td>
<td>0.658</td>
<td>0.010</td>
<td>0.640</td>
</tr>
</tbody>
</table>

In Table 5, it can be observed that varying punishment and reward settings behave similarly. Having equal punishments and rewards, is however significantly more effective against the tactic changing opponent, considering the average fitness. The other parameter settings are however never significantly better than having equal factors and hence having equal factors can be considered slightly more beneficial. The performance against dynamic opponent types lies around 0.5, in conjunction with previous results, where the dynamic scripting technique showed difficulties beating the dynamic opponent types.

Figure 32 illustrates the average percent of successful victories, against all five opponents, when varying the punishment and reward factors.
Simulation results

Figure 32: Plot of the average percent of victories against five opponents, for different punishment and reward settings. 95% confidence intervals are shown.

Again, it can be seen in Figure 32, that having equal parameter settings against the tactic changing opponent, is significantly better, when considering the average percent of successful encounters against the tactic changing opponent.

8.2.2 Learning rate

As in section 8.1.2, three different learning rates have been used, 1.0, 2.0, and 4.0. For each parameter setting, 20 Simulations have been carried out against five opponents.

Table 6: Average fitness against five opponents, for three different learning rates

<table>
<thead>
<tr>
<th>Opponent type</th>
<th>Static</th>
<th>Changing</th>
<th>Diversion</th>
<th>Consecutive</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opponent</td>
<td>Avg.</td>
<td>95%</td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
</tr>
<tr>
<td>Setting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr</td>
<td>0.703</td>
<td>0.009</td>
<td>0.660 0.009</td>
<td>0.638 0.012</td>
<td>0.490 0.007</td>
</tr>
<tr>
<td>1.0p, 1.0r, 2.0lr</td>
<td>0.710</td>
<td>0.009</td>
<td>0.682 0.010</td>
<td>0.631 0.012</td>
<td>0.498 0.006</td>
</tr>
<tr>
<td>1.0p, 1.0r, 4.0lr</td>
<td>0.722</td>
<td>0.011</td>
<td>0.681 0.008</td>
<td>0.636 0.012</td>
<td>0.515 0.006</td>
</tr>
</tbody>
</table>

In Table 6, it can be observed that performance mostly increases when increasing the learning rate. Again, performance against dynamic opponent types lies around 0.5. Inspecting the setup for the dynamic opponents more thoroughly provides the reason for only achieving a performance of around 0.5. Both the consecutive, as well as the best opponent, selects rules according to how the AI previously has been playing. Both opponents stores a number of fitness results in a memory. For the consecutive opponent, a rule change will occur if the average of the previous results is negative (below 0.5). If the AI (dynamic scripting) reaches performance above 0.5, then the opponent will achieve performance below 0.5, and a rule change will occur. Similar discussion can be made concerning the best opponent. Hence, when the performance of the dynamic scripting system centers on 0.5, it can actually be considered quite good.

Figure 33 illustrates the percent of successful encounters for various learning rates, against the five different opponents.
Simulation results

Figure 33: Diagram with the percent of victories against various opponents, over learning rates. Error bars represent 95% confidence intervals.

From Figure 33 it can be observed that the percent of successful encounters, also mostly increases, when increasing the learning rate. Against the best and constant opponents, the percent successful encounters are significantly higher when increasing the learning rate from 1.0, to 4.0.

8.2.3 Derivative jumping

As in section 8.1.3, three different settings of punishment and reward factors have been used, together with the derivative of previous results. For each parameter setting, 20 simulations have been carried out, against five different opponents.

Table 7: Average fitness against five opponents, for three settings of derivative jumping

<table>
<thead>
<tr>
<th>Average fitness, 95% confidence</th>
<th>Setting</th>
<th>Opponent</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Constant</td>
<td>Changing</td>
<td>Diversion</td>
</tr>
<tr>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
</tr>
<tr>
<td>0.5p, 1.0r, 1.0lr +df/dt</td>
<td>0.689 0.008</td>
<td>0.662 0.013</td>
<td>0.642 0.008</td>
<td>0.494 0.007</td>
</tr>
<tr>
<td>1.0p, 0.5r, 1.0lr +df/dt</td>
<td>0.696 0.008</td>
<td>0.671 0.008</td>
<td>0.630 0.008</td>
<td>0.505 0.006</td>
</tr>
<tr>
<td>1.0p, 1.0r, 1.0lr +df/dt</td>
<td>0.692 0.009</td>
<td>0.678 0.011</td>
<td>0.636 0.012</td>
<td>0.504 0.005</td>
</tr>
</tbody>
</table>

From Table 7, it can be concluded that the average fitness results, are largely independent from the individual settings of the punishment and reward factors. As before, performance against dynamic opponent types, centers on 0.5. Figure 34 shows the average percent of successful encounters against five different opponent tactics, when including the derivative, and when varying the punishment and reward factors.

Figure 34: Plot of the percent victories against various opponents for derivative jumping. Error bars represent 95% confidence intervals.
It can be observed in Figure 34, that the percent of victories mostly is independent from the variation of punishment and reward settings, when including the derivative.

8.3 Varying difficulty

In this section, results when investigating three different methods for throttling the performance, is presented. For each investigated configuration, 20 simulations have been conducted against five different opponents (20 simulations for each opponent, and parameter configuration). Section 8.3.1 presents results when manipulating the learning rate, in order to throttle the performance. In section 8.3.2, results when varying the maximum weight settings, is presented. Finally, section 8.3.3, presents results when using a fitness-mapping function with different fitness target values.

8.3.1 Learning rate

When investigating if the learning rate can be used to throttle the performance exhibited by the AI, ten different learning rates have been used during simulation, [0.1, 0.2, 0.4, 0.6, 0.8, 1.0, 2.0, 4.0, and 8.0]. For each learning rate configuration, the average, and median fitness values achieved for 20 simulations have been calculated. The median values are however not presented as they were similar to the average values. In all simulations, the maximum weight has been set to 0.5.

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Static opponent types</th>
<th>Dynamic opponent types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg. 95%</td>
<td>Avg. 95%</td>
</tr>
<tr>
<td>0.10</td>
<td>0.559</td>
<td>0.007</td>
</tr>
<tr>
<td>0.20</td>
<td>0.609</td>
<td>0.010</td>
</tr>
<tr>
<td>0.40</td>
<td>0.656</td>
<td>0.012</td>
</tr>
<tr>
<td>0.60</td>
<td>0.671</td>
<td>0.011</td>
</tr>
<tr>
<td>0.80</td>
<td>0.689</td>
<td>0.009</td>
</tr>
<tr>
<td>1.00</td>
<td>0.696</td>
<td>0.013</td>
</tr>
<tr>
<td>2.00</td>
<td>0.712</td>
<td>0.010</td>
</tr>
<tr>
<td>3.00</td>
<td>0.698</td>
<td>0.016</td>
</tr>
<tr>
<td>4.00</td>
<td>0.701</td>
<td>0.014</td>
</tr>
<tr>
<td>8.00</td>
<td>0.706</td>
<td>0.009</td>
</tr>
</tbody>
</table>

A significant increase in average performance when increasing the learning rate, for the static opponent types, can be seen in Table 8. The performance against the consecutive and best opponents seems to increase slightly, and then stabilize around a fitness value of 0.5. This conveys with previously presented results. Increasing the learning rate too much, does however seem to lower the performance somewhat, against the diversion opponent (maximum at 2.0, then decreasing). The reason for achieving worse performance can probably be tracked to a rule being used for diversion, as follows. During four consecutive encounters, the opponent deploys one rule. At every fifth encounter, an alternate rule is used. At high learning rates, this single use of an alternate rule makes the AI unlearn the behavior learned against the first rule. At lower learning rates, behaviors are not unlearned based on a single encounter, and hence the productive behavior against the first rule remains active. As the first rule is deployed in 80% of the encounters, the slow learning AI is more likely to achieve higher performance than the fast learning AI, which is more likely to unlearn behaviors that are productive in 80% of the encounters.
### Table 9: Percent successful encounters against five opponents, for different learning rates

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Static opponent types</th>
<th>Dynamic opponent types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Constant PV 95%</td>
<td>Changing PV 95%</td>
</tr>
<tr>
<td>0.10</td>
<td>0.605 0.016</td>
<td>0.555 0.017</td>
</tr>
<tr>
<td>0.20</td>
<td>0.666 0.017</td>
<td>0.588 0.018</td>
</tr>
<tr>
<td>0.40</td>
<td>0.733 0.025</td>
<td>0.648 0.020</td>
</tr>
<tr>
<td>0.60</td>
<td>0.767 0.017</td>
<td>0.683 0.014</td>
</tr>
<tr>
<td>0.80</td>
<td>0.788 0.017</td>
<td>0.704 0.020</td>
</tr>
<tr>
<td>1.00</td>
<td>0.806 0.023</td>
<td>0.719 0.018</td>
</tr>
<tr>
<td>2.00</td>
<td>0.827 0.018</td>
<td>0.756 0.016</td>
</tr>
<tr>
<td>3.00</td>
<td>0.825 0.023</td>
<td>0.743 0.019</td>
</tr>
<tr>
<td>4.00</td>
<td>0.828 0.020</td>
<td>0.758 0.018</td>
</tr>
<tr>
<td>8.00</td>
<td>0.819 0.015</td>
<td>0.738 0.020</td>
</tr>
</tbody>
</table>

The percent of successful encounters also significantly increases, when increasing the learning rate, for static opponent types. This can be seen in Table 9. Similar as before, the average percent of successful encounters against dynamic opponent types also seem to stabilize around 0.5, when increasing the learning rate. Again, increasing the learning rate too much seems to lower the performance somewhat, against the diversion opponent. Figure 35 illustrates a plot of the average fitness development, against the constant opponent type, for five different learning rates.

![Figure 35: Average fitness development for five different learning rates, against the constant opponent. 95% confidence intervals are shown by errors bars.](image)

In Figure 35, it can be observed that there are significant differences in performance development, when increasing the learning rate. All configurations do however reach performance values higher than the break-even point, and hence the learning rate cannot be used to let the AI loose. Figure 36 illustrates the average fitness over learning rates, against four different opponents.
Simulation results

Figure 36: Average achieved fitness against four different opponents, over learning rate. 95% confidence intervals are shown by error bars.

In Figure 36, it can be seen that the performance against static opponents increases the most when changing the learning rate between 0.1, and 1.0. The increase in performance takes the shape of a logarithmic function. For dynamic opponents, the average achieved fitness stabilizes around a fitness score of 0.5, in conjunction with earlier results. Against the diversion opponent, the average fitness decreases slightly, when increasing the learning rate too much.

8.3.2 Maximum weight setting

In this section, results for varying the performance, by changing the maximum weight setting, are presented. Ten different maximum weight settings have been used: 0.125, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. The reason for starting at 0.125 is that the rulebases only consists of 8 different rules for each goal. Using 0.125 hence represents completely random behavior, as no learning can take place. Learning cannot occur as the total weight sum must remain constant at a value of 1.0. In all simulations, the learning rate has been kept at 1.0, that is, 1.0 in punishment, and 1.0 in reward. As before, both average and median values have been calculated. The median values are however not presented as they showed similar behavior to the average values.

Table 10: Average fitness over maximum weight setting, against five different opponents

<table>
<thead>
<tr>
<th>Maximum weight</th>
<th>Static opponent types</th>
<th>Dynamic opponent types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg. 95% Avg. 95%</td>
<td>Avg. 95% Avg. 95%</td>
</tr>
<tr>
<td>0.125</td>
<td>0.503 0.009 0.503 0.006 0.512 0.007 0.490 0.007 0.488 0.009</td>
<td>0.511 0.008 0.502 0.007 0.511 0.008 0.502 0.007 0.511 0.008 0.502 0.007</td>
</tr>
<tr>
<td>0.200</td>
<td>0.577 0.006 0.570 0.009 0.557 0.008 0.511 0.008 0.502 0.007</td>
<td>0.511 0.008 0.502 0.007 0.511 0.008 0.502 0.007 0.511 0.008 0.502 0.007</td>
</tr>
<tr>
<td>0.300</td>
<td>0.628 0.009 0.610 0.007 0.594 0.009 0.498 0.008 0.508 0.007</td>
<td>0.502 0.009 0.502 0.008 0.502 0.008 0.502 0.008 0.502 0.008 0.502 0.008</td>
</tr>
<tr>
<td>0.400</td>
<td>0.671 0.009 0.634 0.010 0.609 0.014 0.502 0.009 0.502 0.008</td>
<td>0.496 0.005 0.501 0.006 0.496 0.005 0.501 0.006 0.496 0.005 0.501 0.006</td>
</tr>
<tr>
<td>0.500</td>
<td>0.698 0.011 0.650 0.009 0.609 0.010 0.496 0.005 0.501 0.006</td>
<td>0.494 0.009 0.497 0.005 0.494 0.009 0.497 0.005 0.494 0.009 0.497 0.005</td>
</tr>
<tr>
<td>0.600</td>
<td>0.713 0.010 0.659 0.010 0.617 0.010 0.494 0.009 0.497 0.005</td>
<td>0.492 0.010 0.497 0.008 0.492 0.010 0.497 0.008 0.492 0.010 0.497 0.008</td>
</tr>
<tr>
<td>0.700</td>
<td>0.736 0.015 0.687 0.012 0.626 0.010 0.492 0.010 0.497 0.008</td>
<td>0.487 0.008 0.504 0.008 0.487 0.008 0.504 0.008 0.487 0.008 0.504 0.008</td>
</tr>
<tr>
<td>0.800</td>
<td>0.755 0.009 0.684 0.011 0.621 0.010 0.492 0.011 0.501 0.011</td>
<td>0.492 0.011 0.501 0.011 0.492 0.011 0.501 0.011 0.492 0.011 0.501 0.011</td>
</tr>
<tr>
<td>0.900</td>
<td>0.760 0.019 0.696 0.011 0.625 0.012 0.487 0.008 0.504 0.008</td>
<td>0.494 0.008 0.494 0.008 0.494 0.008 0.494 0.008 0.494 0.008 0.494 0.008</td>
</tr>
<tr>
<td>1.000</td>
<td>0.773 0.019 0.687 0.014 0.630 0.010 0.494 0.008 0.494 0.008</td>
<td>0.494 0.008 0.494 0.008 0.494 0.008 0.494 0.008 0.494 0.008 0.494 0.008</td>
</tr>
</tbody>
</table>
Simulation results

In Table 10, it can be seen that the average performance significantly increases, when increasing the maximum weight setting, against static opponent types. For the dynamic opponent types, similar results as earlier, fitness lies around 0.5, and no significant increase in performance can be observed.

Table 11: Percent successful encounters against five different opponents, over variable maximum weight setting

<table>
<thead>
<tr>
<th>Successful encounters - percent victories, 95% confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static opponent types</td>
</tr>
<tr>
<td>Maximum weight</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>0.125</td>
</tr>
<tr>
<td>0.200</td>
</tr>
<tr>
<td>0.300</td>
</tr>
<tr>
<td>0.400</td>
</tr>
<tr>
<td>0.500</td>
</tr>
<tr>
<td>0.600</td>
</tr>
<tr>
<td>0.700</td>
</tr>
<tr>
<td>0.800</td>
</tr>
<tr>
<td>0.900</td>
</tr>
<tr>
<td>1.000</td>
</tr>
</tbody>
</table>

Again, results in Table 11, are similar as before. The percent of successful victories significantly increases, when increasing the maximum weight setting, against static opponent types. The percent of victories against dynamic opponents however centers on the break-even point, 0.5. Figure 37 shows the average fitness development against the constant opponent, for five different maximum weight settings.

Figure 37: Average fitness development for five different maximum weight settings, against constant opponent. Error bars represent 95% confidence intervals.

In Figure 37, it can be seen that the performance increases when increasing the maximum weight setting, against the constant opponent. The different levels of performance are significantly separated. The performance can however not be throttled below the break-even point, similar as when varying the learning rate. It can also be observed that for a maximum weight of 0.125, the performance centers on 0.5, against the static opponent. This conveys with theory, where a maximum weight of 0.125 should represent random behavior.
Simulation results

In Figure 38, the average performance over varying maximum weight settings, are plotted against four different opponents.

![Figure 38: Average fitness over variable maximum weight setting, against four different opponents. Error bars represent 95% confidence intervals.](image)

From Figure 38, it can be concluded that the performance increases when increasing the maximum weight setting, for the static opponent types. Again, the performance against the dynamic opponent types seems to center around the break-even point.

8.3.3 Fitness-mapping function

In this section, results when applying a fitness-mapping function, described in section 4.3.3, are presented. Ten different fitness-mapping targets have been used: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. For each fitness-mapping target, 20 simulations have been conducted against five different opponents. In all simulations, the learning rate has been kept constant at 1.0, and the maximum weight has been kept constant at 0.5. Both average and median fitness values have been calculated, but only the average values are shown, as the median values were similar.

<table>
<thead>
<tr>
<th>Target performance</th>
<th>Static opponent types</th>
<th>Dynamic opponent types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>95%</td>
</tr>
<tr>
<td>0.1</td>
<td>0.323</td>
<td>0.009</td>
</tr>
<tr>
<td>0.2</td>
<td>0.305</td>
<td>0.009</td>
</tr>
<tr>
<td>0.3</td>
<td>0.370</td>
<td>0.017</td>
</tr>
<tr>
<td>0.4</td>
<td>0.473</td>
<td>0.007</td>
</tr>
<tr>
<td>0.5</td>
<td>0.507</td>
<td>0.009</td>
</tr>
<tr>
<td>0.6</td>
<td>0.550</td>
<td>0.009</td>
</tr>
<tr>
<td>0.7</td>
<td>0.629</td>
<td>0.014</td>
</tr>
<tr>
<td>0.8</td>
<td>0.711</td>
<td>0.009</td>
</tr>
<tr>
<td>0.9</td>
<td>0.707</td>
<td>0.010</td>
</tr>
<tr>
<td>1.0</td>
<td>0.678</td>
<td>0.012</td>
</tr>
</tbody>
</table>

It can be observed in Table 12, that for the static opponent types, the performance significantly increases when varying the fitness-mapping target in the interval [0.2, 0.8]. Not reaching below 0.2, and not reaching above 0.8, can be explained by the setup of the test environment. The best rule has an 80% chance at victory and the
Simulation results

worst a 20% chance at victory. Hence performance above 0.8, and below 0.2, should not be theoretically available. For the dynamic opponent types, the performance significantly increases when varying the fitness-mapping target in the interval [0.2, 0.5]. The fitness then stabilizes around 0.5, similar to previously presented results. In contrast to varying performance by learning rate, or by maximum weight settings, the performance can be throttled below the break-even point, when using a fitness-mapping function.

Table 13: Percent victories against five different opponents, for ten fitness-mapping targets

<table>
<thead>
<tr>
<th>Target performance</th>
<th>Static opponent types</th>
<th>Dynamic opponent types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PV 95%</td>
<td>PV 95%</td>
</tr>
<tr>
<td>0.1</td>
<td>0.245 0.017</td>
<td>0.255 0.014</td>
</tr>
<tr>
<td>0.2</td>
<td>0.221 0.018</td>
<td>0.287 0.013</td>
</tr>
<tr>
<td>0.3</td>
<td>0.293 0.020</td>
<td>0.370 0.018</td>
</tr>
<tr>
<td>0.4</td>
<td>0.438 0.016</td>
<td>0.495 0.024</td>
</tr>
<tr>
<td>0.5</td>
<td>0.514 0.025</td>
<td>0.542 0.026</td>
</tr>
<tr>
<td>0.6</td>
<td>0.616 0.023</td>
<td>0.610 0.025</td>
</tr>
<tr>
<td>0.7</td>
<td>0.727 0.019</td>
<td>0.718 0.018</td>
</tr>
<tr>
<td>0.8</td>
<td>0.837 0.018</td>
<td>0.743 0.018</td>
</tr>
<tr>
<td>0.9</td>
<td>0.823 0.015</td>
<td>0.740 0.010</td>
</tr>
<tr>
<td>1.0</td>
<td>0.748 0.021</td>
<td>0.696 0.013</td>
</tr>
</tbody>
</table>

In Table 13, percent successful encounters, similar observations as for the average, values can be made. For the static opponent types, the performance increases when increasing the target performance in the interval [0.2, 0.8], and for the dynamic opponent types, the performance increases when increasing the target performance in the interval [0.2, 0.5].

The interval [0.2, 0.5] for the dynamic opponents can be related to previous results, in which the performance against dynamic opponents has centered on the break-even point, 0.5. This can, as stated before, be considered good against the dynamic opponent types, as they change tactic when experiencing negative performance levels. The interval [0.2, 0.8], for the static opponent types, can probably be explained as a combination of the test environment, shape of the fitness-mapping function, and maximum weight setting.

In order to reach the lowest available performance, the worst available rule needs to be selected, which has a 0.2 chance of being defeated. Hence, performance values below 0.2 should not be theoretically available. Similar, in order to reach the highest available performance, the best available rule needs to be selected. The chance for the best available rule to win is 0.8, and hence performance values above 0.8 should not be available in theory. Further, when using a maximum weight setting of 0.5, a single rule cannot be favored alone, as there is always a 50% chance of selecting a worse or better rule. Further, when aiming at a target fitness of 1.0, the sine function which has been used, translates fitness values of 0.75, and below as values below the break-even point, and hence they are penalized. This means that victories can receive punishment, and bad behavior becomes promoted. Hence the lower performance for higher fitness-mapping targets and the higher performance for lower fitness-mapping targets.

Figure 39 illustrates a plot of the average fitness development for six different fitness-mapping targets, against the constant opponent.
Simulation results

**Figure 39:** Average fitness development for six different fitness-mapping targets, against constant opponent. Error bars represent 95% confidence intervals.

In Figure 39, it can be observed that different levels of performance, which are significantly separated, can be achieved by varying the fitness-mapping target, against the constant opponent. In contrast to the results when varying the learning rate, or the maximum weight value, performance levels below the break-even point can be achieved. Figure 40 illustrates the average fitness against four different opponents, over varying fitness-mapping targets.

![Fitness development, constant opponent - Fitness mapping function](image)

**Figure 40:** Average fitness over fitness-mapping target, against four different opponents. 95% confidence intervals are shown by error bars.

In Figure 40, it can be observed that the performance increases, when increasing the fitness-mapping target, against the static opponent types, in the interval [0.2, 0.8]. For the dynamic opponent types, it can again be observed that the performance increases, when increasing the fitness-mapping target in the interval [0.2, 0.5].
9 Conclusion

Picture an RTS game, in which the computer controlled opponent even after a hundred games, still surprises the human player, and acts as irrational as some human players do. The aim of this dissertation has been to investigate how dynamic scripting, a technique for achieving online adaptation in computer games, possibly could be applied at the strategic level in an RTS game. Three research questions have been investigated:

1. How can the structure of dynamic scripting possibly be modified to fit the strategic level in an RTS game?
2. How can the adaptation time in dynamic scripting possibly be lowered?
3. How can the performance of dynamic scripting possibly be throttled?

The first question is interesting as the dynamic scripting technique originally was designed to be used in CRPG games. The differences between the strategic level in an RTS game and the character level in a CRPG can be extensive. Further, Spronck, et al. (2003) reported that the dynamic scripting technique sometimes experienced long times to reach convergence. In an RTS game, the number of occasions to receive feedback can be rather low, hence the interest for the second question. Amongst the players of computer games, such as RTS games, there exist many different levels of expertise, and a computer game should be fun to play for both expert and novice players, hence the interest for the third question.

In this dissertation, a new structure for applying dynamic scripting has been proposed: goal-rule hierarchy. In this structure, goals are used as domain knowledge, in order to select suitable rules at each instance. A rule is seen as a strategy for achieving a specific goal, and one goal can be achieved through several different rules. The adaptation process operates on the probability that a specific rule is chosen as strategy for achieving a goal at any given time. Each rule can in turn be realized by other goals, and this creates a hierarchy of goals and rules. Planning is also introduced in the structure, as each rule is coupled with preconditions that need to be fulfilled for a rule to be activated. In case the preconditions are false, other goals are initiated with the purpose of fulfilling the preconditions.

Further, it has been shown that having equal punishment and reward factors, or having higher punishments than rewards, can be more effective, compared to having higher rewards, regarding adaptation time, re-adaptation time, and performance. Having equal punishment and reward factors has at some occasions even been shown significantly more effective than having higher rewards. It has also been shown that increasing the learning rate, or including the derivative, are two efficient methods for decreasing both adaptation, and re-adaptation times.

This dissertation has investigated three different approaches for varying the performance of the dynamic scripting technique. It has been shown that the performance effectively can be varied by:

1. Varying the learning rate
2. Varying the maximum weight setting
3. Applying a fitness-mapping function, and varying the target fitness value.

It has however been shown that the third approach, applying a fitness-mapping function, is superior, as the performance also can be throttled to negative levels.


9.1 Discussion

Even though the goal-rule hierarchy proposed in this dissertation has not been thoroughly evaluated, it should still provide a good fundament for constructing an RTS game AI system. The system covers not only the strategic level, but also all levels of the AI, down to every single unit. Hence, the system also serves as an interface between different levels of the AI. Given that the AI in an RTS game is preferably built in a hierarchical fashion, the goal-rule hierarchy provides a good structure for achieving a goal-directed behavior, which includes adaptation.

The dynamic scripting technique presented by Spronck, et al. (2003) is based on rules which are created by humans. The adaptation process operates on the selection criteria for individual rules that are hand-designed. This is considered a very important aspect, as the quality of individual rules can be assured in advance. Hence the dynamic scripting technique will provide at least as much challenge as manually designed scripts, but it will at the same time also provide means for adaptation, in order to cater for changing human-player tactics, and deficiencies in complex sets of scripts.

The suggested structure for using dynamic scripting however suffers from not being as simple, and easily understood, as the original structure suggested by Spronck, et al. (2003). The process of designing rules also becomes more complex, as the chain of different events might need to be constructed and evaluated in advance.

There is however some advantages gained from building the suggested structure. First, some degree of planning, and resource allocation is managed implicitly by the system. Further, it provides means for the planning to become emergent, as adaptation is introduced in the planning process. The most important advantage is however the inclusion of goal-directed behavior. Using individual goals to dictate the behavior of the AI, can serve as an excellent approach for achieving an illusion of intelligence.

The original structure of dynamic scripting can however be achieved in the new structure, by only having one goal, and assigning all rules to that goal. Further, the preconditions, post statements, and terminate conditions can be removed to achieve simple rules. This would results in the original algorithm, but whilst at the same time keeping the possibility to use the new structure. The new structure could for example be used at the strategic level, and then simplifying further down the hierarchy, to use the original structure at the lower levels for unit behavior.

The main goal in this dissertation has been to investigate how dynamic scripting possibly could be applied at the strategic level in an RTS game. The suggested goal-rule hierarchy should however also be applicable in other game genres as well, where adaptation, goal-directed behavior, and strategic thinking are needed. It could for example be used to manage strategic decisions in first-person shooter games, where many individual agents are to exhibit some form of cooperative behavior.

Spronck, et al. (2003) reported that the dynamic scripting technique sometimes could have problems with long adaptation times. They argued that the reason for this was that it could take long time to unlearn unwanted behaviors. In this dissertation, it has been investigated how varying the punishment and reward factors, increasing the learning rate, and including the derivative, affects the adaptation times, re-adaptation times, and the performance in dynamic scripting.

In an RTS game, where the adaptation process is to operate at the strategic level, there might not be room for the adaptation process to take excessively long time. Having equal punishment and reward factors has however been shown to be more effective,
compared to giving higher rewards for good behavior. Having equal factors, hence limits the risk of unwanted behaviors being hard to unlearn. When operating at the strategic level, there is also a high risk of several sudden changes in tactical behavior: if the first tactic is not productive, then change to another tactic. This is something constantly being deployed by human players, and hence the AI needs to be capable of dealing with sudden changes. Having equal factors can be significantly more effective, when readapting to a new opponent tactic, and hence this result is considered important.

More important, the learning rate has been shown to significantly lower the re-adaptation times. Hence, at the strategic level, a high learning rate could be used to cater for sudden tactical changes. There is however one potential problem with increasing the learning rate too much: the behavior might become predictable. According to Spronck (E-mail contact, 2004-05-11), the dynamic scripting technique do not only aim at learning better behavior, it also aims at being unpredictable. Hence, increasing the learning rate too much violates one of the benefits of using dynamic scripting. There are at least two reasons for wanting an algorithm in a commercial computer to be unpredictable. First, if the behavior is predictable, then the human player can learn this predictable behavior, and exploit it. Secondly, the game becomes non-repetitive, which can increase the time for how long a game is enjoyable.

It has been shown that by including the derivative, the adaptation, and re-adaptation times, can also be lowered. Including the derivative should however not become as predictable as increasing the learning rate too much, as some degree of noise is also introduced with the derivative. Small changes in fitness can produce weight changes in opposite direction of what the fitness represents on a discrete scale. This because the derivative operates on a scale relative to previous results, compared to an absolute scale around the break-even point, for the punishment and reward factors.

Finally, this project has also been concerned with potential methods for varying the performance in the dynamic scripting technique. Being able to vary the performance exhibited by the AI in a computer game can be of high importance. Commercial computer games are usually intended for many different players, covering both experts, and novice players. If an adaptive AI system would show to be able to beat human players, this could introduce some problems, as the human-players are usually intended to emerge victorious. A very difficult AI system could of course be very fun to play against for an expert player, but in order to also cater for novice players, means for limiting the performance need to be included.

Applying a fitness-mapping function, which converts fitness values onto a scale that promotes a fitness target, has been shown to be the most effective method for limiting the performance. The performance can even be throttled to negative performance levels, which can be very important if it is assumed that the human player should win. The other two methods for varying the difficulty, (1) varying the learning rate, and (2) varying the maximum weight, has also been shown to be effective for varying the performance. These two methods can however not be used to throttle the performance to negative levels.

As argued earlier, increasing the learning rate too much could result in predictable behavior. The same problem exists for the maximum weight value. Increasing the maximum weight value could result in predictable behavior. By having a low maximum weight value, more rules will have a higher probability to be selected, in relation to other rules. Increasing the maximum weight value, will increase the chance of promoting individual rules, so that they can dominate. This in turn could result in
predictable behavior, as the most beneficial rules are likely to be selected. The fitness-mapping function should however not result in this kind of predictability.

Furthermore, by using a fitness-mapping function, the original fitness calculations can become easier to design. The fitness calculations could simply be created to promote the best available performance. The calculated fitness would then be mapped into a fitness space designed for the current difficulty level of a game.

9.2 Future work

Future work includes developing the suggested structure further and also to perform investigations on the structure in a larger and more complex environment, and also to investigate the inclusion of the derivative in the adaptation process, more thoroughly.

The best way of evaluating the usability of the suggested structure, would probably be to conduct investigations in a commercial RTS game environment. This could also provide knowledge concerning if the adaptation process operates fast enough to be used at the strategic level.

Another approach to follow would be to evaluate how adaptation could be achieved at a slower timescale, where the opponent AI adapts its strategies in-between games. Furthermore, it is also interesting to investigate how machine-learning techniques possibly could be used to automate the process of creating rules.

Finally, it would also be interesting to investigate the applicability of the suggested structure in other game genres.
References

Age of Mythology (2003) [Computer game], Ensemble Studios, Microsoft.


Command and Conquer: Generals (2003) [Computer game], EA Pacific, EA.


Half-life (1999) [Computer game], Valve, Sierra.


