Adaptive reinforcement learning agents in RTS games

Eric Kok BSc
erickok@gmail.com

Supervised by Dr. Frank Dignum and Joost Westra MSc

Intelligent Systems group
University Utrecht, The Netherlands

Thesis number INF/SCR-07-73
10-06-2008

Abstract

Computer game AI nowadays still relies heavily on scripted behaviour. The resulting computer players are therefore predictable and non-adaptive to the opponent. An approach to create challenging, learning computer players, called Dynamic Scripting, has been proposed by Pieter Spronck. Based on this work I developed reinforcement learning agents for a real-time strategy game called Bos Wars. These agents, implemented using the 2apl platform, can autonomously reason on goals and world beliefs. Several approaches to acquire a winning strategy have been tested, including Dynamic Scripting and Monte Carlo methods. To better cope with opponents that switch strategies, implicit and explicit adaptation is tried out. Due to the integration of agent technology and reinforcement learning, agents have proven to be able to quickly and consistently learn to outperform the scripted fixed and strategy switching players. It is shown that incorporating opponent statistics into the learning process of a Monte Carlo agent gives the best learning results.

Contents

1 Introduction

The document you are reading is the result of my master's graduation project at the Intelligent Systems group of the Computer Science department of the Utrecht University, The Netherlands. It describes the project's motivations, design and implementation considerations and the results of the research. In this first chapter I will give an introduction to the main topics and explain how the project is organised.

1.1 Project background and motivation

As part of the Agent Technology master at the Utrecht University, a seminar Games and Agents is organised by Dr. Frank Dignum. It was during this course, and especially during the talk of invited speaker Dr. Pieter Spronck, that I became most interested in research on learning in computer games. It seemed very obvious to apply the agent paradigm combined with reinforcement learning to virtual characters or computer-controlled players in games. Games offer an attractive, practical and challenging AI research platform since complex and high-level decisions should be made whilst having a large set of requirements including time constraints and an abundance of environmental data.

Of course, building realistic intelligent agents is a hard task in which many researchers worldwide operate in a wide area of subfields. What I will cover in this project is the specific topic of learning top-level strategic decision making. As mentioned, Dr. Pieter Spronck inspired me on this idea through his Dynamic Scripting project. It uses a reinforcement learning-like technique that generates flat, static scripts to create winning strategies for computer players. He used role-playing games such as Baldur's Gate and Neverwinter Nights which are relatively short and straightforward games. Chapter 3.2 explains more about Dynamic Scripting. My intuition, which was supported by Dr. Frank Dignum, was that longer and more complex games need a more refined and flexible learning technique. It seems reasonably to include environmental data on its own and its opponent game state, for example. I will use a real-time strategy game to test this idea.

1.2 Real-time strategy games

A real-time strategy game, further abbreviated to RTS game, is a strategic war game in which multiple players operate on a virtual battlefield, controlling bases and armies of military units. Actions are mostly tactical decisions which typically end with the destruction of the enemy. Some of the best known RTS games are Command and Conquer, Starcraft and Age of Empires.

RTS games provide an interesting platform for researchers. They offer complex and partially unknown environments in which the main focus is on high-level decision making. Strategy selection is a complex problem in which a human may be very good but a computer is not. It is hard to make sound derivations from all the available data and to learn from past interactions quickly. This sets high requirements on the learning capacities of the computer player. Therefore it is the perfect game type to use in this project.

1.3 Bos Wars

Like most commercial computer games, modern RTS games are closed source and can thus not be changed, or at least not enough for our purpose. We will need to extract game data from the engine and take control over the computer players' behaviour. Bos Wars however is an open source game similar to the well known classics of the genre, developed by enthusiastic volunteers. It doesn't offer the

complexity and graphic detail of modern RTS games, but it does offer a suitable test environment for this project.

Figure 1: Screen shot of a Bos Wars game

Bos Wars has a futuristic feel and clear game setup. The players start on an uninhabited map. Using engineers, they can start constructing a base with buildings such as vehicle factories, soldier training camps, power stations, gun turrets and magma pumps. Several different types of units may be trained to use in defence or offense such as engineers, assault soldiers, tanks and rocket tanks. Because all of these units and buildings have costs, resources should be maintained. Bos Wars uses an economic-like resource model, disallowing storing of large quantities of resources, so a steady input and output flow should be preserved. This can be done by harvesting them from trees and rocks using engineers or by buildings such as power plants and magna pumps. The player who destroys all of the enemy's buildings and units wins the game.

The tactical decision making in Bos Wars is currently controlled by static scripts. It is expected of these that they control the constructing of the buildings and training of units, the order in which to create buildings and units, how attacking and defending groups are organised and when to wait for something to finish or attack the enemy base. They do not control the spot where buildings are placed, the planning of paths that units take, what specific buildings or units to attack or what resources will be gathered by engineers that aren't constructing buildings. This still leaves a task that is complex and challenging. Hundreds of unique strategies may be used of which only a small percentage is strong enough to win against most others. Although there are some known strong lines of attack, no single definitive solution exists. For every tactic there is always a counter-tactic, yet some may be hard to find. The learning computer players that I will introduce should be able to replace the existing scripts, handling all of its current jobs.

As an alternative to Bos Wars, the ORTS game environment has been considered [Buro and Furtak, 2003]. This RTS game is open source as well and has been specifically designed for research purposes.

The main problem with this platform though is the lack of a fully functioning agent to start development with and compare results to. Also, it features a very limited set of buildings and units, focussing more on low-level tasks of pathfinding, resource gathering and the forming of attack and defence formations.

1.4 Agent technology

The new computer players developed in this project will be treated as intelligent software agents. Agents are artificial and intelligent entities that act autonomously and pro-actively in an environment. It is easy to think of them as virtual robots or computer-controlled actors. Agent technology has been widely research by computer scientists. It is a fundamentally different approach to software development. With this design stance, it is the software agent that has the will to perform a task. They are not scripted and have no fixed execution plan. Rather, they have a task that is not enforced by a parent program. This has two benefits. First, they allow for a high-level definition of the problem without the need to explicitly state how to solve the (often complex) problem. Secondly, agents are better reusable and more robust than fixed programs.

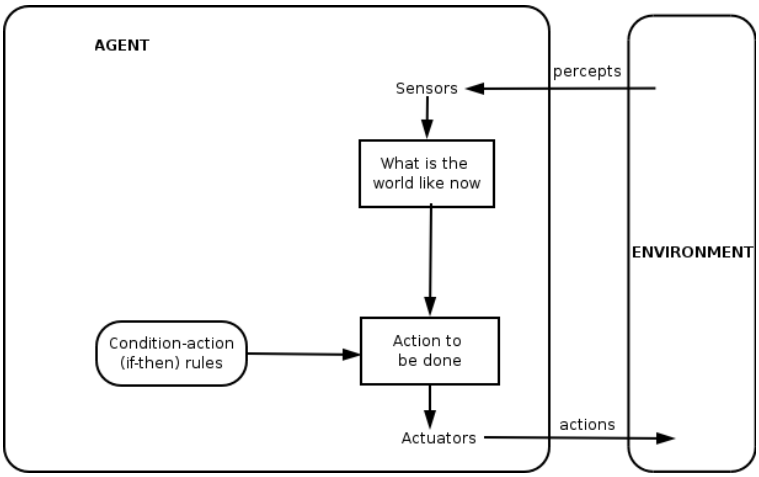


Figure 2: Model of a simple reflex agent

Multiple types of agents exist, including simple reflex agents, model-based agents and goal-directed agents. There is no strict language in which agents should be developed. For example a model-based agent can effectively be programmed in an object-oriented language.

This project however uses goal-based agents, also called rational agents, which are build using an agent programming language with a BDI architecture. BDI stands for beliefs, desires and intensions. The agents reason for themselves on these high-level concepts and operate in an unknown environment. It has the desire to solve a problem (the task) and adopt a goal (has the intention) to accomplish this. To solve how this may be done, it may use beliefs about the environment and itself as well as known plans. The plans ultimately specify which actions it should take to achieve the goal. In Bos Wars these are the strategy actions such as constructing a building and attacking with an army. These are analogous to the ones available to the fixed scripts.

An agent in a computer game can thus work out how to perform a task itself. Often this is not trivial. Plans may be layered into subplans and may consists of basic actions that operate into the environment, but can also use introspection or speech acts to communicate with other agents. New goals and beliefs may be adopted or dropped during the execution. Also, it may know of several plans to use for one

single goal. If multiple plans are available it should decide which are best in the unique situation, according to its beliefs. Ideally it can learn from its own past experiences. If such reflection is used it can learn the best approach achieve the original task.

1.5 Reinforcement Learning

Allowing a computer to learn is called machine learning. Many learning techniques exist, being inductive (generalisations over a data set) or deductive (inferred details from known facts). Whilst many approaches use a supervised approach, having one process control learning from a large subset of data, agents can only use their limited experiences from agent-environment interactions. It does not know if these experiences are valid representative samples. To learn in such an unknown environment, it may use reinforcement learning.

Reinforcement learning, or RL, has a large proven base in computer science. The conceptual basics and main classes of algorithms have been formally described and proven to converge to the optimal solution of a task. It works much like human learning. Through trial and error it may learn what actions are successful and which are not. The procedure is designed as a Markov Decision Process. The agent operates in a model that consists of distinctly identifiable states between which it can transit by selection from a discrete set of available actions. Depending on the state, which is a representation of the unique place of the agent in the world, it may select different actions with different results. Action execution gives rewards to the agent. These are numerical indications on how well the agent is performing the task. If a learning algorithm converges to the optimal policy, the optimal action selection is known for the task.

However, the agent has no prior knowledge on what the optimal actions are. To learn the best action selection it needs to explore the state-action search space. From the rewards provided by the environment, the agent can calculate the true expected reward when selecting an action from a state, which is called the weight. It is algorithm-specific on how the weights of state-action pairs are calculated from a received reward.

The dilemma on when to explore the world for optimal actions and when to select the best, or greedy, action is called the exploration-exploitation problem. If only greedy actions are chosen it may not learn if there are more beneficial actions. Exploration is therefore needed, which can be done in multiple ways. The main used approaches are ε-greedy selection, choosing a non-greedy action in some percentage of the cases, and Softmax action selection that uses a weighted probability according to the state-action weight.

The main RL algorithms are Monte Carlo methods and Temporal Difference learning. Monte Carlo methods, or MC, can be best understood by averaging over random samples. After each full episode, performing the task once, from each visited state the weight is updated by the total gathered reward by averaging it over the existing weights and visits. For example in an action is selected from the base state tree times with rewards 1, 1 and 4 the state-action pair gets updated to a weight of 2. Important is that MC needs a full episode to finish before updating weights and it may directly update all visited state-actions pair in the episode.

Temporal Difference learning, or TD, is the main alternative to Monte Carlo methods. TD works on non-episodic tasks as well because it uses bootstrapping. Weights get updated immediately after a

reward was received. However, the state-action pair directly receiving the reward is the only one that gets updated. It is not until a next visit that the action previous to it will be updated to reflect this new knowledge. This reflects a more realistic assignment of rewards to actions compared to MC, which updates all actions in an episode directly.

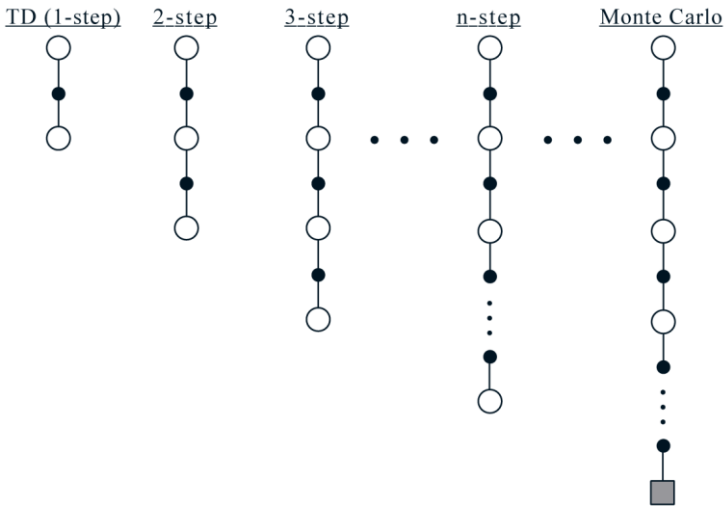


Figure 3: Backup diagrams: TD learning updates directly after each step, MC not until the episode is terminated

In an RTS game, the unknown environment is the game map in which the agent operates. When a game has been won or lost, a reward should be returned to the agent. This can be used to adjust the selected state-action weights. Because the RL algorithm tries to optimize the reward, it will learn to optimize the task of winning the game.

Intermediate approaches to MC and TD also exist, as well as many extensions to speed up learning, cope with uncertainty in environments, integrate temporal aspects of actions, and many more. The approach however is to work with the basic, distinct algorithms to get an expectation on their applicability in this project.

1.6 Document overview

Bos Wars is not build as an agent platform. I will integrate the concept of agents into this game environment and supply them with reinforcement learning capabilities. This will give rise to questions such as how agents operate in games, what they need in order to learn winning strategies and if they can adapt to their opponents.

In this document I will first formalize the research topic in chapter 2. What are the challenges and how to we approach the problems in this project? I will discuss the problems with static scripts and what project approach is to solving this. A study of the existing research, found in chapter 3, provides the starting point for development on learning agents.

The second part, chapters 4 to 7, explains how reinforcement learning was applied in Bos Wars to learn winning strategies. First we model Bos Wars players as agents and use basic learning techniques. Later it is shown how they can be optimized to better meet the set requirements. Also, I will examine if we can adapt our strategy to that of the opponent.

In the last part I will oversee the results and importance of the project. Chapter 8 makes the conclusions based on the formal work and the experiments ran. Have we succeeded in creating sufficiently intelligent and interesting RTS game playing agents? Finally, chapter 9 shows how future research into this topic may be directed.

2 Motivation and research approach

Having more intelligent and interesting computer players in RTS games seems like an obvious objective However, it is important to see what the actual goals are in creating such players. This chapter explains what the problem is with the existing approach and how this project aims to solve this.

2.1 Challenges in RTS game AI

Computer games are becoming increasingly more popular to use in computer science. However, most of the research is still either immature or very specialized. On the other hand game developers are investing enormous amount of time and money into the development of new techniques and play-testing. Games are traditionally above all focused on storytelling and graphics, but nowadays 'the AI' is becoming more important as well. 'The AI' in the game developer's view is not what we consider here. It often includes things like path finding, character animation, finite state machines and static player scripts. Online learning of high-level strategies, without user direction or help, has not yet been used in commercial products.

Playing Bos Wars as a human is fun and challenging. The game compels you to think of astute and novel strategies to defeat the computer opponent. This compels you to reason on many specific and more general aspects of the game simultaneously. Decisions that need to be made during a game include:

- What kind of army will I need to defeat the opponent?
- What is needed to train such an army?
- How will I cope with counter attacks?
- What do I know of the opponent and can I use this to my benefit?

Combining all information from this and previous games is challenging for a human player. For a computer, it is even harder. Even if (which might not be true) all the information is present, it is not clear what is most important and how to make sound derivations from past experiences. How do I attack a player that keeps hitting me with small groups of soldiers? Can I adapt my strategy during the game if I know he has a training camp but no vehicle factory?

Besides being complex, the decisions also need to be taken in real-time. Because RTS games use continuous time, as opposed to for example chess or turn-based strategy games, the players have very limited time to calculate new best actions. Therefore strategy selection should be efficient and computationally feasible. In practice this means high-level decisions such as described above can only consume about 15% of the CPU time available [Buro and Furtak, 2003].

2.2 Problems with game AI scripts

Currently, computer games almost exclusively use static, linear scripts. Their behaviour is thus fixed and predictable. This is a major advantage for the game developer perspective. Especially in modern games where most effort goes into the graphics, animations and sounds and where high-level strategy control structures are mostly added afterwards. For example in RTS games the computer opponent behaviour is added after the actual game play and graphics have been realised.

However, the need for more challenging and interesting players rises. To enrich the game experience, the computer should be able to challenge a human and introduce more variety. At this point in time, it is

hard to build in more intelligent, reasoning computer controlled players or characters. To make higher level decisions we need a lot of information on the environment, including all opponents. But even simple statistics of the player itself are often hard to come by during this stage of development; nicely packed away into the object-oriented classes. At this stage simple scripting behaviour can still be employed since it needs no such (complex) data. Also, scripts are easy to implement, give very predictable results as the individual steps and their outcomes can be monitored and they are understandable by non-programmers [Tozour, 2002].

However, scripts cannot be used to create complex and flexible behaviour. The human is confronted with repetitive tactics that are predictable and thus easy to beat. The game experience is limited to the preset and linear actions provided by the game developers which are fixed after a game release. Weaknesses in the game AI can be exploited by a human since it cannot be solved without patching a game. Play testing for AI weaknesses is a very money and time consuming task. Also, it is unable to adapt to the human player, which would give a boost to the feel of intelligence [Buro and Furtak, 2003].

2.3 Problem statement

Considering the shortcomings as described in the previous section, we can define the problem statement:

Existing RTS game AI is fixed, repetitive and predictable. A feasible approach to model intelligent and flexible opponents does not exist. To cope with all types of players, to reduce behavioural errors after a game release and to provide a more fun and challenging game experience, computer players should learn to optimize and adapt their strategies after a game has been released.

2.4 Project approach to better RTS game AI

In this project I will introduce flexible and learning high-level strategy decision making computer players. The resulting agent should outperform both static scripts as well as strategy-changing opponents through adaptation. This agent is:

- More fun to play against; its strategy isn't static so the game provides varied and interesting new perspectives, which prolongs 'half-life' (the time it takes for players to lose interest in the game, [Fogel *et. al.*, 2004])
- More challenging; it can exploit human weaknesses [Manslow, 2002], is less predictable and come up with novel new tactics to challenge the human
- Less error-prone; it can fix its own design flaws by learning what is poor behaviour and avoiding this [Manslow, 2002], potentially decreasing the need for expensive play testing

A learning BDI agent will be coupled to the Bos Wars RTS game and extended with reinforcement learning capabilities. To prove the effectiveness of the learning agents, experiments will be run. Because Dynamic Scripting is the most promising existing technique, this will be the reference point. If this can be effectively used, different RL algorithms will be tested and compared to DS.

2.5 Requirements on online strategy learning

It is possible to define the requirements that an online strategy learning algorithm should meet. All these are directly related to the real-time, competitive and entertaining nature of RTS games. The computational requirements are:

- Speed; learning should take little CPU time, having modest impact on the game-play [Manslow, 2002].
- Effectiveness; learning AI should create strategies at least as effective as static scripts and may not generate defective or unrealistic behaviour [Barnes and Hutchens, 2002].
- Robustness; it should be able to deal with unexpected game situations and randomness [Chan *et. al.*, 2004].
- Efficiency; the learning speed should be high so it takes little episodes (or encounters) to learn effective strategies [Spronck, 2005].

The functional requirements are:

- Clarity; the method should be easily understandable, even for non-programmers, to be suitable for use in actual computer games [Spronck, 2005].
- Variety; strategies generated should be divers and interesting, avoiding predictable, static behaviour [Spronck, 2005], but refraining from seemingly random behaviour as well.
- Consistency; the learning rate and results should be predictable, without strange exceptions such as learning taking exceptionally long [Spronck, 2005].
- Scalability; the algorithm should work well on all possible opponents [Liden, 2004]

The learning algorithms tested during this project will be checked on these requirements. For example, to prove a decent learning speed, consistency and scalability, experiments will be executed against multiple opponents and on multiple game maps.

2.6 Formalisation of the research topic

How does an agent acquire a winning RTS game strategy? An agent needs information from its environment to make decisions. Therefore it should be supplied with (or have access to) this information to reason about. During the reasoning it should decide what actions to take in the environment and in what order. The available actions and when and how to apply them are part of the agent program. It may use information from past experiences as well. Finally, the agent executes the actions in the environment, queuing the strategic commands in Bos Wars.

Figure 4: A learning agent in Bos Wars should map the game states to appropriate actions

Commands that are processed in the environment have results. During a game the results are not immediately clear, but at the end of the game the agent knows if it has won or lost. This information should be used to update its own beliefs so it can make better decisions in the future. If it successfully reflects on its own effectiveness over multiple games, it will eventually find a strategy that will win the game.

Strictly put, the agents in Bos Wars should map the input (environment data and game results) to correct output (select strategic actions). That means it should select the right actions in the right order. If the mapping of this is effective, it has learned a winning RTS game strategy. In the next chapter we will see how the interaction between the environment and agent is modelled.

3 Existing research

Online learning of high-level strategies, without human intervention, is never used in a commercial game. Existing research into machine learning in computer games is also sparse. However, a strong starting point is needed for this project. This offers a working development base and provides a reference point to compare new approaches.

3.1 Adaptive game AI using Dynamic Scripting

As explained Dynamic Scripting was the inspiration for this project. This technique is developed by Dr. Pieter Spronck at the Maastricht University. It was first described in [Spronck *et. al.*, 2003] in which the idea was tested in the role playing game Baldur's Gate.

Dynamic Scripting is a reinforcement learning-inspired algorithm to generate scripts for computer players that maximize the player's fitness. From a player-unique rule base consisting of known useful actions (basic actions as well as more complex strategy blocks) a script is generated. The computer linearly follows this script which results in an episode fitness. The fitness is, as in reinforcement learning, an indication on the effectiveness of the last actions in relation to the task we want to learn. The fitness value is calculated back into the available actions' weights. The next episode a new script is generated according to the new weight values.

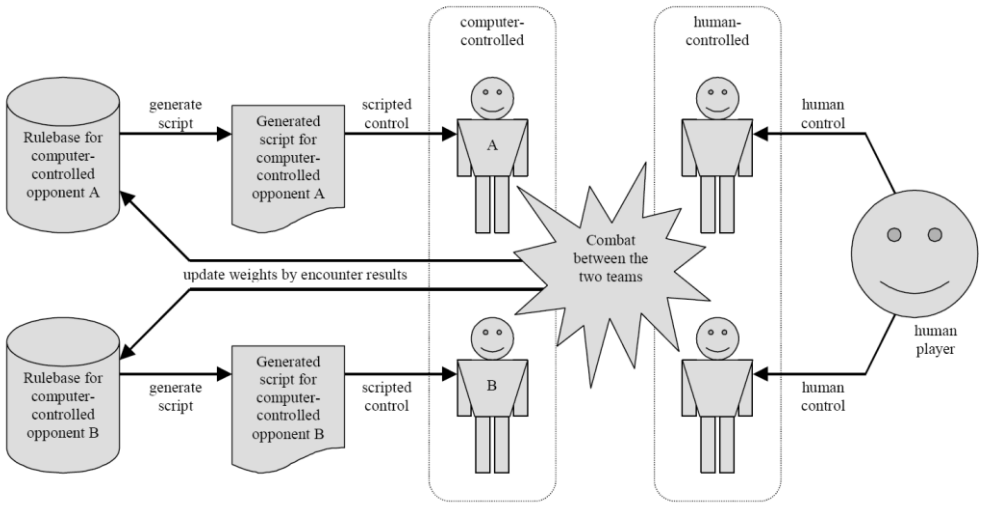


Figure 5: Dynamic Scripting process model [Spronck *et. al.*, 2003]

For example take the Baldur's Gate role playing game where two teams of two characters battle against each other. Classic scripted computer players follow a linear fixed script so they cannot change the behaviour. This is predictable and easy to beat. Dynamic Scripting agents however can adjust themselves by using another script, based on the previous experience. The very first episode it has none, so it randomly selects a fixed number of rules from its rule base. The rule base contains all rules that the player may use in a fight. The selected rules are combined into a script, order as predefined by the game developer. This script will be used to actually play the game, just like a classic script. At the end the fitness of the player is determined. This value is used to update the weights in the rule base. Rules that were used will thus get higher or lower weights assigned. This will determine the probability that they get selected again next time. All non-used rules get updated as well to compensate for the increased or

decreased total weight of the whole rule base. For example the randomly chosen rules were unsuccessful and received a negative fitness. All the rules that were not selected in the script will now get higher rewards. The next game, or encounter, a new script is generated. Again it selects a fixed number of rules from the rule base, but this time it has a better indication on the rules' realistic effectiveness. Linear Softmax selection is used to weight the rule selection probability. After several encounters it will be clear which rules are effective and since they get chosen (mush) more often, the player will outperform a non-dynamic opponent.

The generated scripts are easy to understand, because they resemble classic static scripts as still used in most modern computer games. From the experiments it is shown that it allows computer players to learn a winning strategy fairy efficiently, for the Baldur's Gate game as well as for the games MiniGate and Neverwinter Nights [Spronck *et. al.*, 2003]. However, it may still need many episodes (up to 50) to defeat certain strategies on average. It is concluded that in action role-playing games such as MiniGate this is feasible since many encounters with the same opponent type may be expected.

The strength of DS clearly can be attributed to the way it reduces the search space of finding winning strategies enormously. This, although not explicitly stated in the articles, can be explained through these characteristics:

- Rules in the rule base contain a large amount of domain knowledge
- Rules are explicitly ordered and may only be used once in a script
- The script is of a fixed length
- The script is static during the game execution
- There is no differentiation between unique environmental states

Aside from these clear differences between reinforcement learning and DS, it still has the same act – reward – update weights loop. It actually resembles Actor-Critic Temporal Difference Learning, but with the on-episode-end updating of weights (like Monte Carlo) and the translation of the result difference (TD error) is done via the generated script.

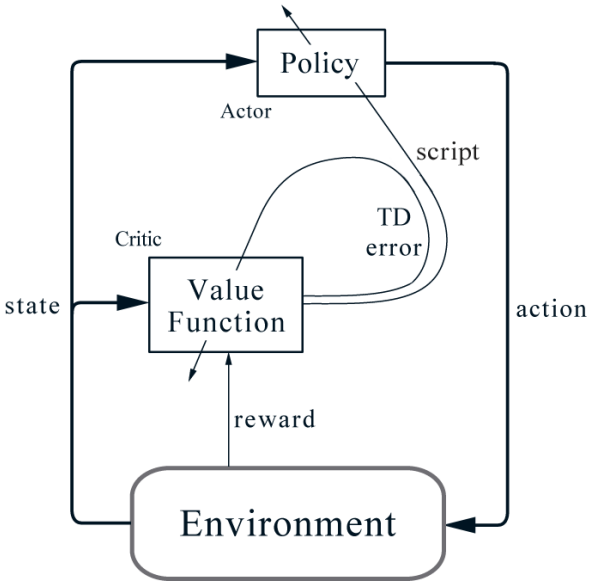


Figure 6: Dynamic Scripting's policy update architecture [Spronck *et. al.*, 2006]

Spronck argues that applying regular reinforcement learning such as Monte Carlo methods may be infeasible. The search space is huge if we have all basic actions to try and it is unclear what environmental information should be put in the state vector. In this project we will try to overcome these problems by combining agent technology, reinforcement learning and Dynamic Scripting strengths.

The research that led to the Dynamic Scripting technique was part of Dr. Pieter Spronck's PhD-thesis [Spronck, 2005]. This thesis includes an experiment with a real-time strategy game called Stratagus, which actually is the predecessor of Bos Wars used in this project. Spronck reasons that RTS games as opposed to role-playing games need a state representation of the world, corresponding to the state in reinforcement learning. He uses states defined by the owned buildings and each state has a unique rule base that should be learned. The dynamic player was able to find winning strategies against balanced static opponents within 50 game episodes on average.

Dynamic scripting is a very interesting technique that arguably may be used in actual commercial computer games. In learning situations with sufficient similar encounters it may online learn winning strategies. However, I believe the algorithm does have some downsides:

- No discrimination between unique world states may be used (except in the RTS game experiment). For example it cannot switch strategies when it received new opponent information such as what the actual type of opponent it is fighting.
- Rule ordering is pre-set which may be non-optimal and requires tweaking at development time. In rare cases this might even prevent the algorithm to find certain best counter tactics. An attempt at solving this was made in [Timuri, 2007] which unfortunately complicates the algorithm.
- Rules may not be used multiple times in one script. For example it cannot use a 'build vehicle factory' rule more than once. If needed, a rule should be added to the rule base to build another (or two, or more). This prevents loops but severely limits the script possibilities.
- There is a fixed script size. If chosen too small it may end up having no more rules to execute. If chosen too large it may need to include unbeneficial rules to get to the size (especially in small rule bases). Preliminary experiments were needed for all games and game characters to determine a useful script length, which requires a lot of time.
- The algorithm depends highly on the actual numbers set for the possible game results whilst ideally you want the environment to handle this. Also, the experiments use complex fitness functions to calculate scores. It is unclear whether this speeds up the learning process or is needed at all. At least it may be hard to construct the complex functions.
- DS only works on short episodic tasks; if an agent deals with a continuous task the episode will never end and with long episodes the script might run out of actions.
- Learned weights are specific to that rule base; if multiple rule bases are used for different game opponents, the learned weights of one character may not be used with another. Learning speed and effectiveness would be greatly increased if such parallel learning would be possible.

Overall Dynamic Scripting is an original and useful approach into making more intelligent and adaptive game AI though the introduction of machine learning. It is a solid base for this projects' work on the

creation of flexible learning RTS game playing agents. It is highly interesting to see if DS can feasibly be employed in Bos Wars to generate winning strategies against both static and strategy-switching opponents.

3.2 Other related work

The reference and starting point of this project, Dynamic Scripting, is the predominant example on scientific game AI research. However, there have been several other approaches including some specific on high-level real-time strategy game tactics learning. The most relevant papers are discussed here.

3.2.1 Generating Game Tactics via Evolutionary Learning

Ponsen and Spronck [2004] proposed a new method to search for winning RTS game tactics in Wargus. This method uses an offline evolutionary algorithm. Chromosomes define the full strategy and the best chromosomes in an experimental session are mutated to ultimately come up with a chromosome that has a fitness that meets the target value. Experiments showed that the algorithm is able to find strong strategies that outperformed the used rush-type static scripts.

Figure 7: Screen shot of a Wargus game

Although the approach was able to generate strategies with surprisingly high fitness values, it has a limited use. Since it uses an offline evolutionary algorithm it needs many experimental games to test the large number of possible chromosomes. The learning wasn't halted until a minimal threshold fitness value was found or 250 solutions (750 games played) had been tried. This makes the approach useful in optimizing static scripts and finding useful rule base rules (to be used by Dynamic Scripting) [Ponsen *et. al.*, 2006] during development time, when it doesn't need to quickly react to new experiences. However it is far too slow to feasibly use in the online learning to be able to adapt to human players during a game experience. This makes the generation of tactics via evolutionary algorithms unsuitable for use in this project.

3.2.2 Case-Based Plan Selection

A learning approach to better cope with changing opponents has been introduced by Aha *et. al.* [2005]. It praises the Dynamic Scripting approach's efficiency and simplicity but criticizes its inability to cope well with different opponents because no knowledge can be transferred between rule bases. Instead a Case-based Tactician (CaT) is implemented which can learn even from games with random consecutive opponents.

CaT uses three sources of domain knowledge; the abstract world state and available strategy actions from [Ponsen and Spronck, 2004] (which however include opponent statistics) and a mapping of specific game situations to known tactics and their performance. During a game, it evaluates taken actions such as unit training to reflect on the performance according to the Wargus game score before the action was executed, right thereafter and at the end of the game. This resembles a reinforcement learning approach towards action effectiveness measurement and resembles hierarchical task network (HTN) planning towards the use of sub-tasks to speed up learning as well as allowing for the transfer of knowledge on action performance between game states.

The implementation uses TIELT (a general framework for the testing and comparison of learning techniques) to attach CaT to Wargus. This middleware supplies CaT with a generalized game model as input and handles the proper executes of actions in the Wargus environment. Unfortunately it is no longer actively supported and was therefore not a viable platform to use for my own project. Also, TIELT introduces an unwanted degree of non-determinism which makes experimenting harder.

Experiments show CaT outperforms the static opponents (a combination of rush-like scripts and strong student contributions). On average it wins 80% of the games after around 75 played episodes. The most notable difference with Dynamic Scripting being that it is able to learn against multiple different opponents against which it can differentiate whilst still using learned knowledge between different opponent scripts. Unfortunately the experimental results are unclear about performance differences between opponents that actually switch during a series of episodes. This might show exactly to what degree it is beneficial to do this.

3.2.3 Concurrent Hierarchical Reinforcement Learning

Marthi *et. al.* [2005] aim to use concurrent hierarchical reinforcement learning to play an RTS game. This approach uses reinforcement learning augmented with prior knowledge about the high-level structure of behaviour. This constrains the possibilities of the learning agent and may thus greatly reduce the search space.

The actions to learn are not only high-level abstract actions ('train army of 4 soldiers and attack') but it also attempts to learn lower level actions ('move north' in order to come to a gold mine, where they can 'pick gold'). The experiment on the higher-level strategy domain still includes more low-level decisions than used by the previously discussed papers. It shows it is still able to learn to train multiple peasants (for resources), handle resource management, construct barracks and train footman. Most notably, this is a task that couldn't be well learned by classic single-threaded reinforcement learning. Rather, it effectively distributes the learning over small subtasks.

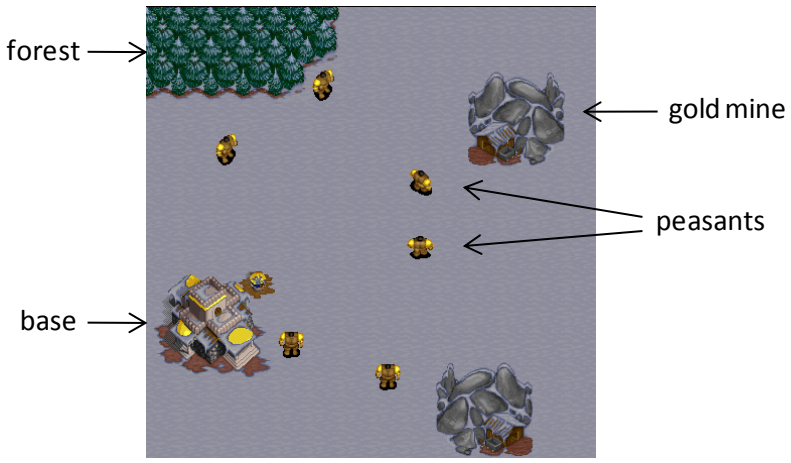


Figure 8: The resource-gathering subgame within Stratagus

However, the focus in the paper is on the ability to learn rather than to quickly online react to changes on the opponent. Therefore it is interesting to see that such complex tasks can be learned by mere trial and error, but this approach is infeasible when we want to online learn full RTS game strategies that can adapt to the (human) opponent.

4 Agent technology in RTS game AI

The intelligent computer opponents will be designed as software agents. It is easy to see the benefits of taking this design stance. An agent is by definition autonomous, pro-active and rational, just as a human player is. This chapter explains the benefits of incorporating agent technology in games and how this is achieved in Bos Wars.

4.1 Combining agents and games

Introducing agents in games ideally requires an upfront planning of the agent-game interaction model. During game design, environmental data should be kept accessible for agents to use. This means a change in philosophy on data enclosure. However, this doesn't mean anything on the actual game AI has to be planned or implemented prematurely. It is merely a matter of providing the computer agents with the same information as the human player has.

The combination of agents and games has not yet been studied a lot. The first attempt can be found in [van Lent *et. al.*, 1999] in which the Soar agent architecture is coupled to the Quake 2 and Descent 2 first-person shooter games. The classic agent loop of perceive (sensor input), think (reasoning) and act (action output) is already present here. According to the tasks that the agent needs to perform and to the actual game, different levels of information need to be supplied to the agent. Similar games may use similar domain knowledge and thus reuse parts of the agent code. Soar was primarily used for low-level decision making.

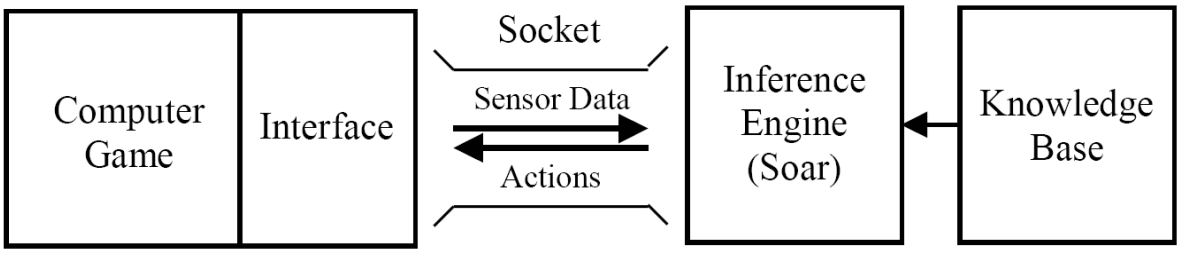


Figure 9: Soar's agent-game interaction model [van Lent *et. al.*, 1999]

In this project we will use a similar design to model interaction between an agent and the game. For this we will use our own agent programming platform and couple this to the Bos Wars RTS game.

4.2 Introducing the 2apl platform

At the Utrecht University, the Intelligent Systems group has developed an agent programming language called 2apl [Dastani *et. al.*, 2007]. It provides a platform for intelligent agents and multi-agent communication. The agents use the well-known BDI architecture to reason over their beliefs and goals. 2apl agents can such expose complex and intelligent behaviour.

A 2apl agent typically has one or more goals which it aims to achieve. It also has beliefs about itself and the world which it may use in the deliberation process. Beliefs and goals may be adopted or dropped during the processing of goals. To achieve a goal it can use plan-goal and procedural rules to elaborate on abstract plans. The platform by design supports methods for interacting with environments. Information from the environment can be send to an agent causing an external event that may trigger new procedural rules. If the agent wants to act in the environment it can use an external action plan.

The whole of selecting goals, finding plans and acting in the environment makes a complex, autonomous and pro-active software agent.

4.3 Implementing 2apl agents in Bos Wars

To allow computer players in Bos Wars to be controlled by 2apl agents, I coupled Bos Wars to the 2apl platform. Similar to the model from [van Lent *et. al.*, 1999] an interface between both applications was written, which is called Bos22apl ('Bos Wars to 2apl'). The C++ version of 2apl was used since Bos Wars is written in C++.

2apl agents are autonomous within the 2apl platform on which they run. Each agent has its own deliberation cycle and handles external events and actions at preset times in their loop. Bos Wars has its own game loop in which it handles graphics, sound, low-level AI such as path finding, etc. At the pre-set position in the game loop it deals with the execution of computer player scripts. These separate loops need to be synchronized to ensure proper interaction without data loss.

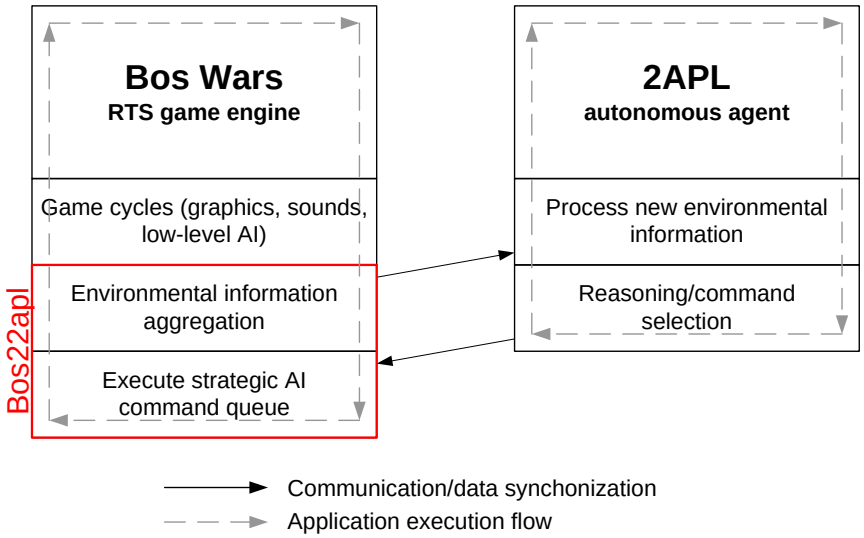


Figure 10: Functional design of the Bos22apl agent-game interaction model

To cope with the synchronization, queues are used. Information from the Bos Wars game environment is aggregated and send to an agent through the 2apl platform, where it is queued in its external event queue. An agent uses a procedural rule to handle the parsing of new information. It is up to the programmer of the agent to write such a rule, possibly with a corresponding belief update. To reduce the information stream to the agent, only changes to the agent's world state are send (closed-world assumption).

When an agent uses an external action, the Bos22apl module queues this in the agent's strategic AI command queue. For each agent such a queue is maintained together with the environmental data on the agent and several statistics such as if the game has been won. When Bos Wars' game loop would normally handle AI scripts, it will now also look at each of the agents' command queues to see if a new command should be handled. The actual strategic commands will be processed by the same functions as for the static scripts.

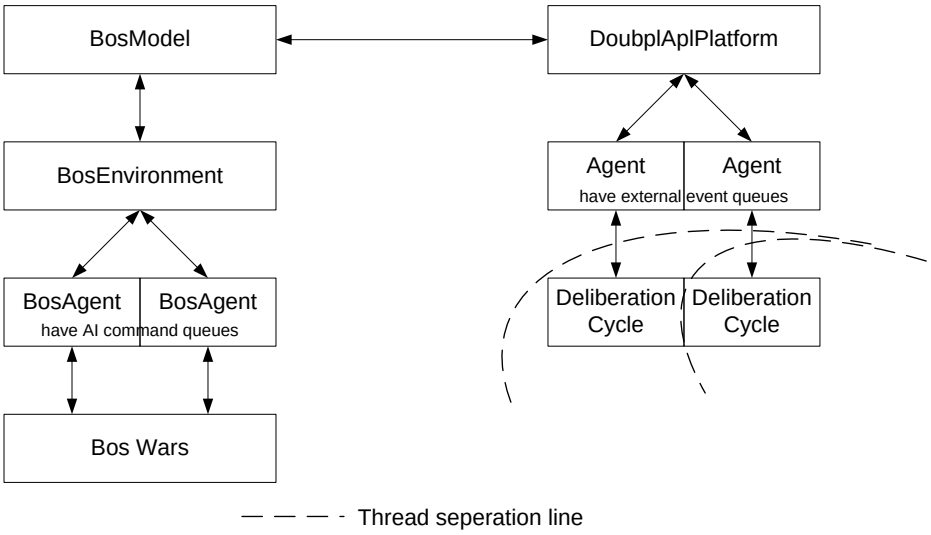


Figure 11: Technical design of the Bos22apl agent-game interaction model

Individual Bos Wars playing agents may communicate with each other using the regular message send command plan. This way agents may negotiate tactics or just chat.

During the development it was considered to directly infuse an agent with beliefs from Bos Wars. Sending environmental information via events as modelled here does bring some overhead. Each message from the environment has to be processes separately within the agent. This means that it needs to select an appropriate pc-rule each time and execute a belief update plan to store the new information into the agent's belief base. This is a quiet time and resource consuming process that may become too have for just the purpose of storing beliefs. Alternatively a direct infusing of beliefs into the agent (without pc-rules and belief updates) could be modelled. This is a lot faster. However, it does make the model fuzzier. The agent paradigm, where agents are autonomous and are not directly controlled by a hierarchical parent, would be distorted. Also, in Bos Wars we needed to send only a very limited amount of data to the agents. Therefore no such functionality was introduced.

In future game applications a separate agent module might be the best approach. This may collect the data from the game environment actively as part of the agent and does not need event-based communication. Such a model may also filter the perceived data before saving them as beliefs, which should be ready-to-use. Such a model may for example simplify a perceived head, body, limps and sword as a knight of a certain type. The actual useable knowledge about the knight can be stored as a belief.

After a command has been queued and when it was executed, an event will be send to the agent to notify of this. Also an event is send when the command queue is empty. Although reacting on an empty queue seems a reasonable approach, much like how a human plays the game, the programmer isn't restricted to using this approach.

Also, the command queue may be filled up at the start of the game and be used until the game ends, or it may rather send individual commands and wait for them to finish before sending new commands.

A small example agent can be found in Appendix I.

4.4 Winning game AI using BDI agents

Linear script as used in Bos Wars may use several commands to control the player behaviour. These include ordering to construct buildings or train units and to attack the enemy with a specified army. The new BDI agents are also restricted to these commands, but do not have such a deterministic execution process. An agent progress is conceptually structured with only goals, beliefs and plans. Since the usage of plans is determined at runtime, there is no strict relation between a strategy and these plans.

The strategy of a Bos Wars agent is thus contained yet not explicit in its plan base. Not the plans form the strategy, but rather the behaviour that results from selecting the plans in a specific order. To win an RTS game, it needs to learn what plans to utilize in what order. During this plan selection, it can use the goals and beliefs to flexibly differentiate between unique game situations.

It is up to the programmer of the agent to define what strategy to use and how this is achieved. It is totally left open to use complex goals with abstract plans or just simple reactive, flat agents. A flat agent has a plan base of rules that are all similar; they may all be selected at all times. It needs to use the belief base or learning approaches (weighting the action selection) to be able to properly select the rules in the right order. A goal-directed agent with a layered structure can use goals to select a subset of rules to use instead of considering the full plan base. Also, plans itself may consist of subplans that have multiple options on achieving this. It is not obvious what the best approach is in building a winning RTS game agent. This will be studied in this project.

If the agent successfully selected the right actions in the right order, it will be able to defeat the opponent by destroying all the enemy's buildings and units. Thus, this is a winning RTS game strategy.

4.5 Discussion on benefits and scientific relevance

The modelling of computer opponents as agents gives us several benefits over the use of static scripts.

- Natural design standpoint; computer players may be modelled just as a human player, one that needs to be provided with information and may use several commands to control strategies. Since their reasoning is human-like it is easy to program the decision making process for computer players. They may reason about high-level goals and plans as opposed to follow clear but unnatural scripts.
- More complex decision making; the agent may use more than one approach by using in-game knowledge to select different tactics. If during game play we received information that the opponent has a vehicle factory, we may use different rules, resulting is different behaviour that better counters the human player. Also there is no fixed script length. This creates more flexible and intelligently looking opponents.
- Additional modules attached to the agent platform may be used; modules such as perception filtering, machine learning, explicit opponent modelling or even subcontracting of tasks to other agents. Since the same platform may be reused, this requires no extra programming efforts.

Agents still have the benefits of linear scripts.

- Agent programs are easy to read and understand, even by non-programmers; they consist of abstract high-level commands that are similar or equal to that used in scripts.

- Rule execution can be monitored and explained; their behaviour is deterministic and can be followed using a view on the agent's goals, beliefs and plans.
- Agents are easy to implement; if during development the environmental information is made accessible to the agent (information that is already present in the game), it is easy to supply this to the agent platform. Since the platform itself may be reused in every game, the additional development time in relation to scripts in minimal.

The resulting agents in Bos Wars may now depict intelligent, flexible and fun agent to play against. However it is not only useful for game developers to employ agent in games. Games can now also easily be employed for use in agent technology research.

In several papers [Buro, 2003; van Lent, 1999] computer games has been proposed as the ideal testbed for AI research. With the generic platform such as presented here, it is possible to test newly developed technologies easily in a competitive, complex and realistic environment. Some examples of research possibilities are perception data filtering such as with neural networks and agent negotiation and cooperation. In the next chapter, I will describe how I use the Bos Wars/2apl setup to try and compare different online reinforcement learning techniques.

5 Learning winning RTS game strategies

Bringing agents in RTS games already gives an improved flexibility and increases the game experience. The introduction of learning computer players has some additional advantages. They may provide even more challenges, can generate novel new strategies and fix design flaws. This chapter describes how reinforcement learning was implemented and will compare different learning algorithms.

To allow Bos Wars-playing agents to learn how to win, we should provide it with the capability to reflect on past experiences. If certain actions proved to be beneficial (because we got a high reward score from the environment), we should in the future focus on these actions. If some employed strategy caused a game loss however, the selected actions should be downgraded. This can be done using reinforcement learning.

5.1 Learning in 2apl agents

Classically, reinforcement learning has been used directly as an algorithmic approach to problems. Within the domain of board games for example this has been proven very effective. However, the approach in this project is to thoroughly integrate learning with the agent technology. I implemented a reinforcement learning framework into the 2apl agent programming language itself, called 2aplLearn. This way every agent written using 2apl can benefit from this framework. An agent programmer doesn't need to implement the algorithms each time again but can use the generic framework, with all the existing advantages in using an agent programming language such as reasoning on high-level goals and beliefs.

The conceptual model of reinforcement learning fits very nicely to that of agents as used in 2apl. In both cases there is a decision making agent who operates in an unknown environment at which it tries to accomplish a task. The perceive – think – act loop is present in both models. The only addition to the learning model is receiving a feedback from the environment on the effectiveness of the performed actions in relation to the task we are learning.

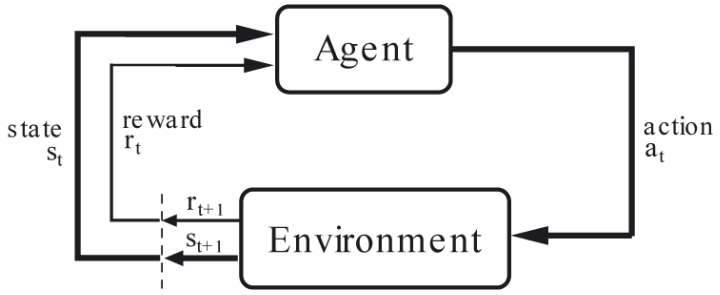


Figure 12: Agent-Environment interaction in reinforcement learning [Sutton and Barto, 1998]

The concepts from RL map well to 2apl. The agent is autonomous and operates in an environment, which in this project is Bos Wars. Actions are executed in the environment, which map to procedural rules of the 2apl agent. Such pc-rules may thus consist of multiple basic commands directed to the environment and may also include communication or abstract plans. A pc-rule to learn can thus be a very basic action (such as normally used with reinforcement learning) but may also be a full complex strategy. Later we will see how this may be very beneficial for the learning speed.

From the environment comes a new game state. This basically is the unique world state in which the agent is. This way it may differentiate between actions. For example it may learn distinct strategies for what actions to select when we already have a vehicle factory (it probably doesn't want to build another). Finally, a reward may be returned from the game which is the numeric feedback to the selected rules. With this reward we can calculate back the weights of the selected actions so they will be chosen less or more often.

For the 2apl platform to know what pc-rules it should learn, they should be marked. When the deliberation cycle wants to select a rule that is learnable, it does not select the first applicable rule found directly. Rather it will collect all the applicable pc-rules at that moment. These are all the rules with the specific head (name) and of which the guard can be satisfied. This will select all the possible options for the agent to solve the problem. Depending on the RL algorithm used, it will now pick greedy or non-greedy. Greedy means it picks the one with the highest weight of all applicable rules. This rule has been proven most beneficial in this specific state from past experiences. Non-greedy means it selects another rule, probably a random one or the second-best. This is used for exploration; sometimes we should try rules to learn how well they do.

The pc-rule weights are unique for every distinct game state. Building a power plant at the start of the game might be more beneficial then later on when a full army has already been trained. The agent's state should reflect this. The state is represented by the beliefs that the agent has at a certain point in time. Every unique combination of beliefs is a unique game state. Beliefs that are missing or new beliefs that are added make for a new game state. For example, not only a belief base with 'a(0)' is different from 'a(1)' but it is also different from 'a(0) b(0)' and from an empty belief base.

A problem occurs now when the agent has beliefs that do not directly reflect the game state, but are rather to use in the program flow or are merely for debugging purposes. Such beliefs may be excluded explicitly from the game state by marking them. This way we can precisely control what beliefs define the unique state as used by the RL algorithm used.

Every learning rule uses the same game state. If multiple distinct pc-rule selections need to be learned, learning multiple distinct tasks, the same game state is used. In the future it might be interesting to be able to separate game states for multiple tasks. This would allow an agent to, for example, use a different game state (different beliefs) for a winning-a-game task and a managing-resources task. Also, all game rewards are assigned to all visited learning pc-rules. It is thus currently not possible to reward an agent for a specific set of pc-rules. For true distinct task learning, this should be possible in the future. To win a Bos Wars game, it is not.

The only additional code an agent programmer needs to write is to define an episode beginning and ending, rule visits and where rewards are received. Most reinforcement learning algorithms need feedback on when a new state has been reached. This is done by using the visit learning action plan in an abstract plan of a pc-rule. Begin and end episode learning action plans can be used to mark when a game is started or ended, which some RL algorithms need. Finally, a reward action plan may be used in an abstract plan to indicate that a reward has been received. This is always a numeric value and will typically directly come from the environment.

A Bos Wars playing agent with simple learning capabilities can be found in Appendix I. Beliefs marked with a caret (^) are excluded from the game state and pc-rules marked with a caret will be learned. The ^beginEpisode, ^endEpisode, ^log, ^visit and ^reward actions handle the feedback to the learning algorithm.

5.2 From game rewards to pc-rule selection

To translate the rewards from the game environment into valid pc-rule selection, a translation has to be made. Weight updating of plan base rules is specific to the way agents are programmed. Here the usefulness of the BDI architecture is especially apparent. If rules would be stored in a linear script, such as with the Dynamic Scripting implementation of Pieter Spronck, the original rule selection model would now be disturbed. In other words, how rules are selected and executed is now significantly changed. Alternatively, rules may be selected by case or if-then statements such as with agents written in object-oriented procedural languages like Java. Here the weights need to be translated into the precondition for rule execution, which is unnatural and distorts the program logical flow.

However, rule selection in BDI agents is already separated from the actual rule logic. Because the pc-rules are substituted in the agent's plan base, picking a rule is already controlled by a rule selector. Non-learning agents always select the first applicable rule, but it is very natural and easy to make rule selection weighted according to the learned rule success.

Calculating weights into the pc-rule weights is now as easy as updating their numeric values for the unique game state. In a reward was received, the learning algorithm can update the values of the visited rules as simple numbers. The rule selector as used when requesting a new pc-rule can now use these numbers to weight the selection probability. Both steps are separated and fit into the natural concept of a BDI agent's sense-reason-act loop.

5.3 Reinforcement learning algorithms in Bos Wars

For this project, I have implemented three distinct RL algorithms. Both TD learning and Monte Carlo methods are the most general approaches to learning and differ mostly in how rewards are calculated back into the rule weights. To compare these classic approaches with the work of Pieter Spronck [2005], I have also implemented Dynamic Scripting.

The algorithms will be tested on their usability with Bos Wars. They will play games on different maps and will need to learn with the minimal feedback the environment gives. Actually the only feedback that is received is whether the games have been won, lost or if it was a draw. The rewards given to the agent were 125, 0 and -20 respectively. These numbers are a little arbitrarily but have been optimized for the Dynamic Scripting algorithm to work best, as established in some preliminary experiments; see 5.5. This also implicitly says that winning is a lot more important than not losing and may even make up for an unfortunate loss.

Important however is that it is not until the very end of a game that the agent receives a reflection on its performance. During a game no indication on the effectiveness of individual rules can thus be given. Although there is an in-game score maintained by Bos Wars, which even updates during an episode, this isn't used as reward. This score is not just an indication whether the task of winning is effective. It also reflects on the performance of the opponent, the number of units killed and on the own army size. These are interesting, but do not directly reflect the actual task we want to learn, namely to win. For

example a very quickly won game may not have a significantly higher score then a very long and close-call but lost game. It is possible to account for this using complex fitness functions as done in [Spronck, 2005], but it is unclear on what the effect of this is. It has not been proven that rewarding subtasks helps increasing the learning speed of the actual task to win. Instead I chose to simply use the only actual reward on this task: if we succeeded in winning.

In the book of Sutton and Barto [1998], two main classes of RL algorithms are identified: Temporal-Difference learning and Monte Carlo methods. As described, they are the best known and most distinct approaches to the resolving of rewards into state-action (belief base-pc-rule) pair weights. When learning action selection, this is called the control version of the algorithms. I use the Sarsa on-policy TD control and on-policy Monte Carlo control algorithms. On-policy means it learns the same policy as it uses in action selection. The latest rewards are directly used in the next pc-rule selection. Because game AI needs to make optimal use of past experiences, on-policy algorithms are most suitable.

5.3.1 Temporal-Difference learning with Sarsa

Temporal Difference learning using on-policy control is called Sarsa. This name comes from the events that make one weigh update in the algorithm: s(tate)-a(tion)-r(eward)-s(tate)-a(ction). In other words, a weight update takes place after each action (except the very first). Only the very last action leading to the reward is updated. The weights are thus updated directly and based on the previous weight rather than a set of stored past rewards. This is typical for TD learning and is called bootstrapping. For some tasks this is very beneficial, for example in games with very long or never ending episodes.

In a previous project before this thesis, this has been proven successful in a simple path finding task. An immediate reward for every step of -1 was given, which could directly be used in both the policy and since it uses on-policy learning, also directly in the action selection.

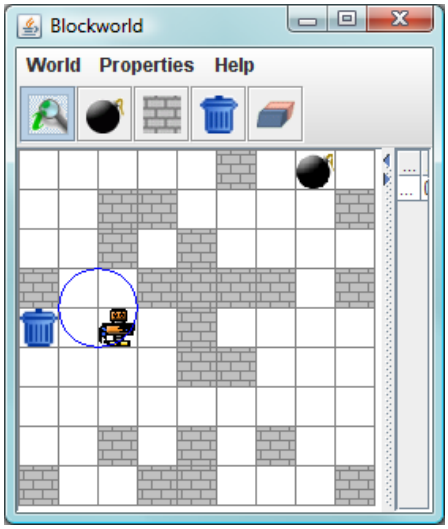


Figure 13: A path finding task in the 2apl block world environment

The problem in using Sarsa to learn winning strategies in Bos Wars is that there is just one reward at the very end of the game. Sarsa (or at least the basic version used here) only calculates a received reward back into the last executed action preceding it. This means that only the very last action made before winning (or losing) a game gets an updated weight. It takes another episode to get this value into the before-last action and a very long time to calculate this back to the very start of the game. Since we want

to very quickly adapt to the opponent we need quick changes and make maximum use of the rewards received. Therefore, Sarsa is unusable for learning strategies in Bos Wars.

It should be noted that there are ways to attribute this weakness in Sarsa. Eligibility traces could be used to propagate rewards back further than just the last action; even all the way back to the very first. However, implementing eligibility traces is out of the scope of this project, especially since the other methods worked well early on.

5.3.2 Monte Carlo methods

The second main class of RL algorithms is that of the Monte Carlo methods. The on-policy action selection version is called Monte Carlo control. With MC control an episode is always finished first and the total collected reward is then employed to update all the used rules' weights. In its most simple form this results in an average of all rewards for every unique state-action pair ever visited. Already after the first episode an indicative value is known for the very first action done in this episode. This allows for very quick adjustments, although there is of course no guarantee that this is a realistic value that can rightfully be ascribed to this action.

Monte Carlo methods have been formally proven to converge into the optimal policy, maximizing the rewards. The learned weights of state-action pairs reflect the actual expected reward. MC does not have a fixed exploration method. To determine when to select the best action (greedy or when to explore (non-greedy), multiple methods exist, like Softmax and ε-Greedy. As described earlier, Softmax uses a weighted selection probability whilst ε-Greedy selects non-greedy in a set percentage of all cases. If no exploration was to be used it would always select greedy. However, exploring new cases in vital in learning a task, especially when there is none or still very little experience.

Bos Wars only returns rewards at the very end of an episode and MC works well with that. Still, there are three problems in using MC in real-time computer games [Manslow, 2002].

- It is often hard to specify what information should be included in the unique game state; for example is it important to include information about the opponent's buildings? This will be discussed further in sections 6.4 and 7.2.
- Execution is less controlled and cannot easily be debugged; this is especially true if we would learn only the basic actions and without an agent which reasons rationally. In section 5.7 it is discussed how problem is solved.
- Convergence to effective, winning RTS game strategies may take a long time; if too many episodes are needed to fully adapt, the opponent may not even notice the effect at all. This depends high on the game. Learning may be speeded up by reducing the search space. Section 2.5 set the requirements on the learning speed for Bos Wars players.

In general, the Monte Carlo implementation is a simple and computationally inexpensive way to learn pc-rule selection in 2apl agents. Bos Wars playing agents can learn to differentiate between unique game situations and adapt to the opponent if the agent program is properly designed.

5.3.3 Dynamic Scripting

To be able to compare classic reinforcement learning techniques and to see if it can be effectively used in an RTS game, Dynamic Scripting was implemented. The algorithm is taken from [Spronck, 2005] and

fitted to the general RL framework of 2aplLearn. This provided some challenges in resolving model incompatibilities.

Since DS only generates one linear script at the very start of each episode, the interaction model with the environment should support the sending of all commands at the game start. Luckily, the Bos22apl module supports this through the queuing of strategy commands. However, this doesn't resolve the apparent problem of having a fixed script length and allowing every pc-rule to be selected only once in a script; see 3.1. These restrictions needed to be hard-coded and this for instance results in not being able to learn multiple distinct tasks in one agent. The rules that may be included in the script, and will thus be learned, need to be specifically marked (by naming them 'script').

DS uses linear Softmax action selection and minimum and maximum rule weights. Rules with weights twice as high get selected twice as often. Although this seems reasonable, it lays a big influence on the setting of the algorithm parameters. The rewards for winning, losing or getting a draw, the script length and minimum and maximum weights need to be set just right for DS to make best use of the algorithm. It may take a lot of experimentation to come up with useful parameters. In [Spronck *et. al.*, 2003] for example, preliminary experiments had to be run for all games and all rule bases of unique opponent characters.

A typical aspect of Dynamic Scripting is that rules that were not selected in an episode still get updated. For example if the latest episode return causes the used rules to be lowered in weight, the weights of unused rules get increased. This feels unnatural but is needed for DS to run. For example, if the last used rules were lowered below zero the Softmax selection is corrupted, and if the unused rules wouldn't get higher selection probabilities at some point no rules would be selected any more. Since Monte Carlo doesn't use minimum and maximum weights (and weights reflect the actual expected reward), this doesn't need such a feature.

Rules in the rule base may contain preconditions in the original implementation. In 2apl this is mimicked by the use of pc-rule guards. The guards may use checks on the agent's belief base to see if certain constraints hold. Although this can be used to distinguish between some specific situations, it is not sufficient to truly learn distinct opponent tactics because it would then need to include rules for all these tactics in one script. This is not a feasible approach and has not been tested.

A very strong argument for using Dynamic Scripting is its learning speed. The main reason for learning so quickly is because the amount of domain knowledge in the rule base and the very narrow search space as a consequence. The rest of this chapter will reveal if it can learn efficiently enough to be used to learn winning RTS game strategies.

5.4 Learning speed experiments setup

To test if the implemented algorithms meet the requirements on learning speed and scalability (see 2.5), an experimental setup is made. Learning speed, or algorithm efficiency, will be tested by measuring how long it takes for the algorithm to learn a winning strategy against a static AI script. The amount of episodes this takes should be small to make online learning feasible in Bos Wars. Against a human Bos Wars player, it should adapt within the scope of around 10 to 25 games, depending on the skill and type of player. If learning takes longer, the adaptation results are not visible or ineffective.

The experiment will record how many episodes the algorithm needs to 'outperform' a static AI script. The evaluation on when it outperforms the opponent is not trivial. From a human perspective, it should be dominant, winning most games in a row. Therefore we design the performance measure as having to win most games over the last few. More precise, the learning agent should gather 12 points over the last 7 games, being awarded 2 points for a win, 1 for a draw and 0 for a loss. A draw occurs when none of the agents has won within 200.000 game cycles (most games end within 100.000 cycles). Effectively, the learning agent should win at least 6 and lose 1 or win 5 and have 2 draws. The first episode in such a winning series is called the Turning Point.

We do not only want the agent to perform well against one specific type of script. According to the requirement on scalability, it should do well against many different opponents. In the experiments, the algorithms are tested against many different static AI scripts (and in the next chapter even against opponents that change tactics). These include:

- Force; a moderately strong tank-biased, but balanced attack.
- Blitz; a moderately strong soldier-biased, but balanced attack, supplied with Bos Wars.
- EasyRush; a fairly strong soldier rush (quickly build one type of unit and attack with this), supplied with Bos Wars.
- SoliderRush; a very strong soldier rush, attacking with soldiers as early as possible.
- TankRush; a strong tank rush, attacking with tanks as early as possible.

As can be inferred from the list, rush-type tactics are often strong because there are little counter-tactics for them. Although for each unique tactic a counter strategy does exist, it is especially hard to find them against the soldier rush script. We can expect the longest Turning Points here. If an algorithm is able to find counter-tactics against all opponent types, we may consider it to meet the scalability requirement.

5.5 Programming an experimental Bos Wars agent

The Bos Wars/2aplLearn environment allows for a great variety of agents to be programmed. They may use complex goals with hierarchically structured abstract plans and negotiation with other playing agents. However, the agent that is used in this project is a much simpler flat, reactive agent. The most important argument is that this allows for comparison of Monte Carlo and Dynamic Scripting. DS can't learn multiple distinct goals simultaneously and uses a simple rule base in which rules may have only a precondition and some basic commands. Also, a reactive and flat agent is fully free in the order in which to select actions allowing for more different and fine-grained strategies.

All the algorithms will use the same 2apl agent code. It effectively works like a reactive agent, sending commands to Bos Wars when the agent's strategy command queue is empty. When an action needs to be selected, it may use all the pc-rules available, as long as the precondition (the rule guard) holds (see 6.1). In the case of Dynamic Scripting, the generated script's order is chosen in respect to the agent program's pc-rule order.

The rules that are used are chosen to allow for different types and levels of strategies. Some of the actions are likely not very good. For example the building of defence gun turrets and attacking with just one soldier is rarely beneficial. Some others are very likely to get chosen. For example at least one rule of building a vehicle factory or building a training camp has to be chosen to train any attack units at all.

Also, none of the rules are by definition required in general (but may be against certain specific opponent scripts). To somewhat regulate the experiments and reduce the complexity of the game, a limited subset of buildings and units are available. The static scripts do not have access to more or less buildings and units then the learning agents do.

The agent program used in all experiments can be found in Appendix II. The changes made for the specific algorithms:

- The beliefs included in the game state; MC has information on the number of buildings and units owned and DS has no game state at all by design.
- The preconditions used; MC excludes the selection of impossible actions.
- The algorithm parameters; the best settings for the parameters were established during preliminary experiments. DS has several parameters which took more time to tweak than setting the MC values. For example setting the DS script length from 6 to 8 increased the learning speed fourfold. The rewards for a win, a loss and a draw were also of significance. Increasing the win reward, tested with DS, from 50 to 125 doubled the learning speed. MC's Softmax temperature was set at 3.

To test if the learning in the agents is directed and non-incidental, the Base Performance has been established. In these experiments a fully random agent was tested. Since this has no reflection on its own performance, it will always select a random action. If no actual learning would take place, this is the minimal performance to be expected.

5.6 Basic algorithm comparison

The Dynamic Scripting, Monte Carlo and Base Performance agents are tested against all the static AI scripts. For each run of game simulations, the Turning Point is measured. If no Turning Point was reached within 200 subsequent episodes, the agent could not find a strategy that outperformed the static script within 200 played games, it was halted and recorded as a Fail. Algorithms that have few Fails meet the consistency requirement of learning agents.

The result charts show the average Turning Point and the Fail Rate for each algorithm against each static opponent. For the average Turning Point number, lower is better, since this would learn faster and thus be more efficient, which is a requirement on online learning RTS game agents. The consistency requirement is expressed in the Fail Rate. This is the percentage of runs that failed of all experiment runs. A lower value indicates it is more consistent in reaching a Turning Point in time and is thus better.

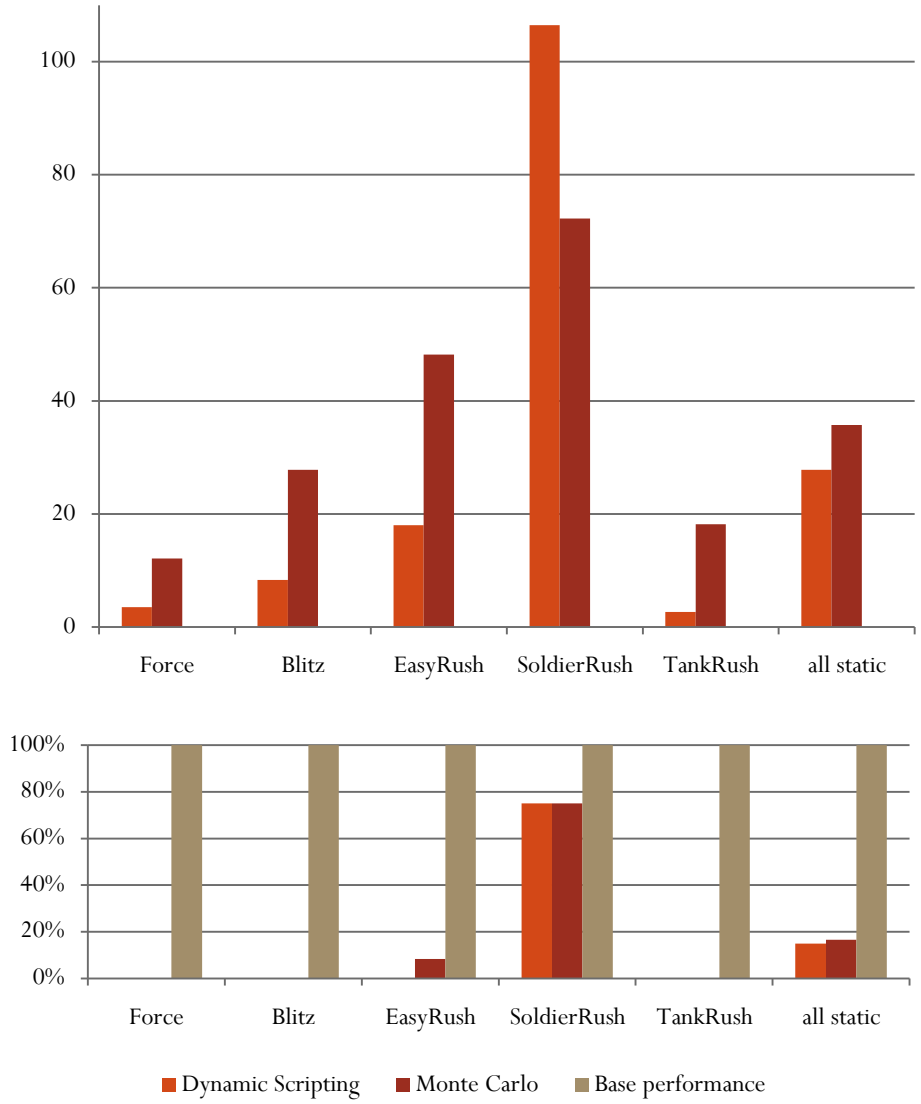


Figure 14: Average Turning Points and Fail Rates of the basic learning algorithms against all static scripts

	Dynamic Scripting			Monte Carlo			Base performance		
	# Exps	ATP	Fail Rate	# Exps	ATP	Fail Rate	# Exps	ATP	Fail Rate
Force	16	4	0%	16	12	0%	8	0	100%
Blitz	12	8	0%	12	28	0%	8	0	100%
EasyRush	12	18	0%	12	48	8%	8	0	100%
SoldierRush	16	107	75%	16	72	75%	8	0	100%
TankRush	19	3	0%	17	18	0%	8	0	100%
all static average	**75**	**28**	**15%**	**73**	**36**	**17%**	**40**	**0**	**100%**
all but SoliderRush	59	8	0%	57	27	2%	32	0	100%

Table 1: Basic learning algorithm experiment statistics against all static scripts

The base performance shows that random selection of commands will never results in being able to outperform the static opponents. Even though this may seem obvious, this does prove that is takes more to win then just pursuing a random strategy without reflection on previous performances. Agents that do improve and can outperform the static scripts are thus non-incidental and do truly learn.

Dynamic Scripting performs very well against most static opponents. Only the SoldierRush tactic is hard to beat, failing in 75% of the cases. This also greatly increases the average against all static opponents. On average it needs 28 episodes to get at a sequence where it outperforms the any static opponent on average. Only 8 episodes are needed and it never fails if SoliderRush is excluded. It thus seems that it is not strictly needed to keep your own possessions into account, having no game state, when making decisions against static opponent scripts. Also, the restrictions of DS such as the limited script length and only being able to select every action just once are not a problem when the parameters of the algorithm are set right. Dynamic Scripting can thus successfully be employed in the online learning of winning strategies against static RTS players.

Monte Carlo is also able to outperform all static scripts as well, having problems with SoldierRush just as DS does, but it needs longer runs to accomplish this. The average Turning Point over all scripts is 36 episodes and 27 when SoliderRush is excluded. There is no single reason for this, or at least this is not deductable from the results, but an important factor is the much larger search space compared to DS. It uses unique rule weights for every possible game state and does not use a fixed rule order. In the next chapter this issue will be addressed. Still, it is a motivating result that MC is actually usable in the online learning in an RTS game.

5.7 Conclusions on learning agents

The Bos Wars playing agents are now not only capable of playing goal-based games, but can also learn from past experiences. Section 2.5 contained the requirements on the learning agents to determine if the algorithms can be feasibly used to online learn winning RTS game tactics. We can now reflect on the success of both the Dynamic Scripting and Monte Carlo implementations.

- Speed; both algorithms consume very little CPU time. Decisions on rule selection are as simple as comparing each applicable rule's weight. DS does all these at the game start, but even MC has little decisions to make (5-30) because it only needs to send commands when the command queue is empty. The updating of weights is done after an episode is finished, when there are no more real-time performance issues.
- Effectiveness; both the algorithms can come up with winning strategies, but there are differences. Because of DS's fixed script length, it may run out of rules to select, stopping any actions at all, which is unrealistic and bad behaviour. There is no simple solution to this problem, see 6.6.1. MC may rarely use a slightly illogical order, but will never run out of actions.
- Robustness; if strategies employed seem unsuccessful, it will learn from this experience and fix itself by choosing new, possibly effective tactics. Therefore it overcomes problems with handling of randomness and unexpected situations, as opposed to static scripts or deterministic agents.
- Efficiency; the above experiments show that DS is already able to learn winning strategies efficiently. It takes just 8 episodes to reach a series where it outperforms the static opponents on average (SoldierRush excluded). This is within the scope of a human player's RTS game playing session. The results for MC are encouraging. Without the DS limitations, it is still able to win against static opponents in 27 episodes on average (SoldierRush excluded). The next chapter will try to improve this number, especially against the hard SoldierRush opponent script.

- Clarity; both algorithms are easy to understand. The generated scripts can be inspected beforehand with DS or at least afterwards with MC. Actually, with the right tools a simple and powerful decision tree could be generated from saved MC weights to explain in-game decisions. This would make it more understandable for non-programmers. The actual agent code is already very clear because of the use of the agent framework; a benefit of having DS build in 2apl instead of a game-specific stand-alone implementation.
- Variability; all learning approached will explore new strategies. Even when it outperforms the static opponent it may make small adjustments, depending on the exploration parameter settings. If the opponent would switch to a truly different tactic it may even implicitly learn to adapt to this. This idea is further explored in chapter 7.
- Consistency; results should be predictable. Exceptionally long learning runs should not occur. This is reflected in the fail rate of the experiments ran. Both DS and MC rarely encountered series of games where it took more than 200 episodes before the static script was outperformed. Only the SoldierRush tactic is the exception, being very hard to beat. This is no surprise, given that this is a near-optimal policy. Only very specific and thus hard to find tactics can oppose it. Both algorithms were still able to find these in time in 25% of the cases. Although there are some outliers, the overall result is very consistent. Due to inherit randomness in Bos Wars the learning process can never be fully deterministic and this isn't even a desired property.
- Scalability; Both DS and MC proved to be able to cope with multiple distinct strategies, including moderate and strong opponents. The algorithms thus scaled well. Chapter 6 discusses how they cope with opponents that change their strategies during a series of games.

Within the limited setup of this experiment, both Dynamic Scripting and Monte Carlo methods have been proven feasible in learning winning strategies. Monte Carlo methods may need to be further improved to increase learning speed. Dynamic Scripting has shown some design limitations that make it less promising and less practical to focus development on.

The most noteworthy result is that MC, in spite of the apparent problems, can be effectively used on high-level tasks in an RTS game. This outcome be can attributed to the integration with agent technology. This allows for learning of strategy blocks (complex rules) rather than basic actions to make full use of domain knowledge, as well as to the absence of a game state. We will further elaborate on the problem of what domain knowledge to put in the game state later.

6 Improving basic strategy learning

Until now we have used only basic strategy learning. It can find a suitable opposing strategy that will eventually outperform a static script. We have also seen that the Monte Carlo algorithm's learning efficiency is somewhat lacking. Independent of the learning algorithm there are several techniques which can help improving the learning efficiency and effectiveness.

The speed in which a solution may be found is mostly influenced by the amount of possibilities it has to explore during the learning process, the amount of domain knowledge it can use and how clever the rule weights are updated. An attempt to reduce the state space, incorporate more game design information and improve the action selection is described in this chapter.

6.1 Rule guards

Reducing the number of actions that an agent may select in a certain state decrease the amount of possibilities to try. Fewer possibilities mean higher chances of selecting the right actions for that moment. It may thus be very beneficial to deny the selection of all impossible or by design infeasible action.

When analysing preliminary experiments it was clear that the selection of infeasible and unusable actions consumed a lot of learning time. Restrictions on rules were thus already used during the further development of the Bos Wars agent used for the basic learning experiments. The Monte Carlo agent uses the rule guards of 2apl pc-rules to restrict the selection in specific game states. Rule guards are already a feature of regular non-learning 2apl agents. It is used to reflect on the belief base to test if a rule is appropriate to use. It was very logical to employ this mechanism for the exclusion of actions from the learning algorithm as well. The 2apl action selector previously only selected the top-most action that was applicable. The new learning algorithm action selector may also filter the applicable actions first before looking at the weights and selecting a greedy or non-greedy action.

For example, if the state includes a belief unit-camp(0) (we own zero training camps), all the rules that command to train assault soldiers can be made unavailable. The pc-rule may for example be:

```
^attack(enemy) <- unit-camp(T) and T > 0 | @boswars(defineForce(0, "unit-assault,2;"))
```

When the abstract plan attack(enemy) is now to be resolved (which uses learning pc-rules), this rule won't be applicable and thus another rule will be selected.

The use of abundant but valid rule guards can be very effective. Little experiments have been run without the use of rule guards (because these last very long) but very early results show a speed decrease of around 300%. Therefore already the basic MC agent was programmed to use rule guards.

The Dynamic Scripting algorithm as designed by Pieter Spronck also uses rule preconditions. However, this wouldn't exclude the rule from being selected in a script. Rather it will include the rule and when executed it will be checked for applicability at that moment and skipped if it was not. This behaviour is mimicked in the Bos Wars/2apl DS implementation. Invalid script commands are ignored and removed from the command queue. Using 2apl rule guards with DS is impossible since the full script is already generated at the episode start, when no game state is known. Adding state information to the algorithm and dynamically check this during game time is out of the project scope (see 6.6.1). Rather, DS actually

takes advantage of the selection of invalid rules. The fixed script length forces the algorithm to choose a set number of rules. If fewer rules would be better, for instance when a simple and small tactic works best, it still needs to select some other, possibly redundant rules. In practice, the algorithm then picks some invalid rules to fill the script without breaking the strategy.

All together, using rule guards is a very strong enhancement in making online learning in Bos Wars more feasible. The learning speed and consistency can be greatly increased. This can be explained by the introduction of domain knowledge. Within the rule guards an enormous amount of information of what not to do is included. The use of 2apl as agent platform is thus highly advantageous, because this already include the necessary features to make use of rule guards.

6.2 Softmax exploration policy

The second measure to improve learning that was already used during the basic learning experiments, was the use of Softmax as exploration policy. Dynamic Scripting uses a linear Softmax action selection method. Analysis of the early Monte Carlo agents showed that it often selected very unviable or even stupid actions when exploring. Although it already learned that certain non-greedy actions were very ineffective, having been assigned very low weights, it would still pick them in non-greedy cases. To solve this problem, the DS approach of using Softmax selection was tested.

Since Softmax selects near-optimal actions on non-greedy cases rather than random actions, it may explore more promising options. Softmax uses the actual rule weights to determine a weighted selection probability. Linear Softmax such as used in DS selects a rule with a weight of 100 two times as often as a rule with a weight of 50. Softmax selection using a Gibbs or Boltzmann distribution was used with MC. Here a parameter, called the temperature, can be set to increase or decrease the influence of rule weight differences. High temperatures results in a near deterministic selection of greedy actions. Very low temperatures may give a near-equal selection probability to all applicable options.

Selecting near-optimal rather than random actions may improve the learning speed as well [Sutton and Barto, 1998]. A Monte Carlo implementation that uses ε-Greedy selection has been developed and compared to MC with Softmax selection with a Gibbs distribution. Both learning agents use rule guards. The exploration rate ε was set to 10% and the Softmax temperature is still set to 3.

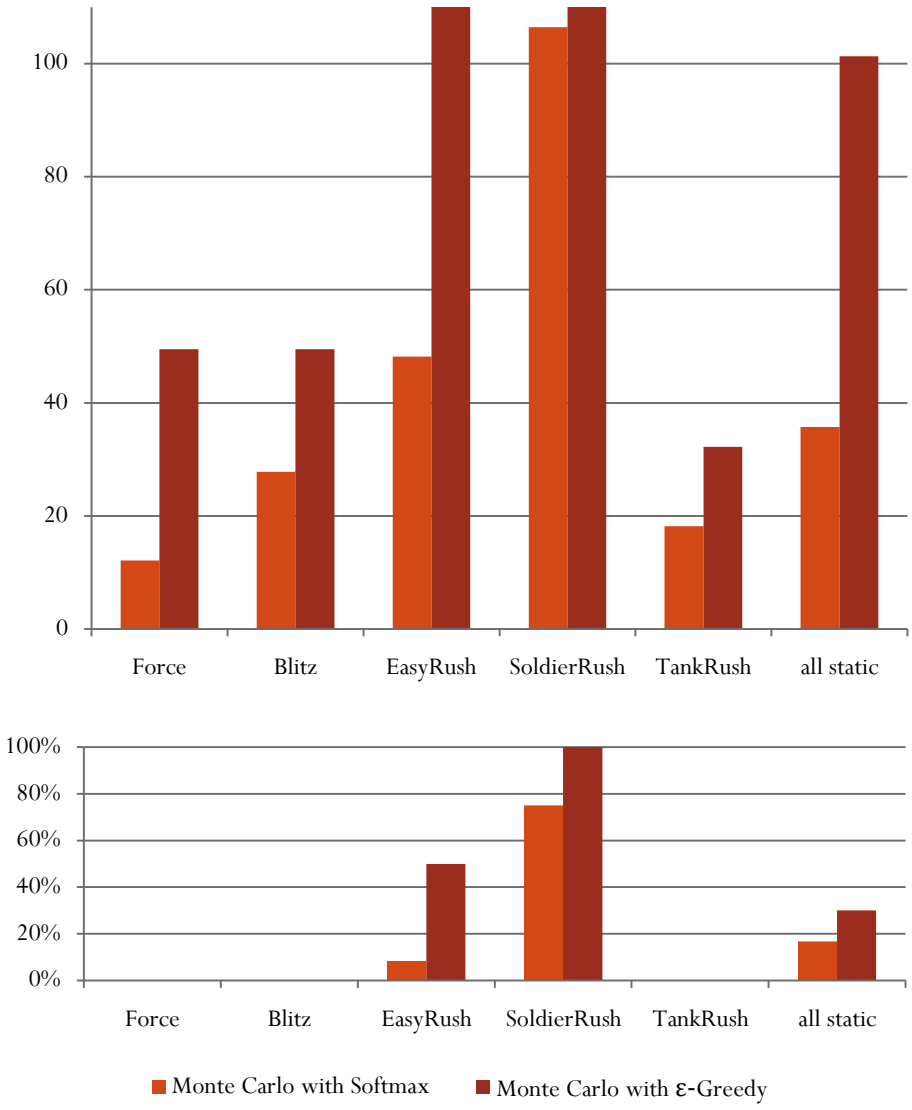


Figure 15: Average Turning Points and Fail Rates of Softmax and ε-Greedy exploration against all static scripts

The Softmax-using agent (as also used in the basic experiments) was able to outperform the static scripts 180% faster on average than the agent that used an ε-Greedy policy, which needed 101 episodes and had a fail rate of 30%. (The ε-Greedy agent was not able to outperform the SoldierRush opponent, so an average Turning Point of 200 was used, since this is the minimal actual Turning Point when the games weren't cut off at 200 played episodes.)

Although in general learning algorithms that use Softmax exploration do not necessarily achieve higher learning speeds, the Monte Carlo implementation in Bos Wars does benefit greatly from it. It not only learns faster, but also has the benefit of selecting more promising and less potentially stupid actions during exploration. It should be noted that the setting of the temperature parameter is less intuitive than setting the epsilon parameter in ε-Greedy selection. However, it may still be logically deduced without the need for extensive preliminary experimentation.

6.3 Strategy hierarchy

The agent used in the basic learning algorithm comparison experiments used a flat, reactive rule structure. It doesn't adopt and drop goals dynamically or use multiple levels of abstract plans. Rather, it learns to select the right rule at the right moment. For DS there is no alternative since it only generates a fixed script from which it cannot deviate during a game. The Monte Carlo implementation however can learn multiple distinct tasks and can use obtained environmental information. This way it may adopt smaller goals and monitor on when it achieved those to set new subsequent goals.

During the development of the flat action, it was considered to use goals and plan hierarchy, but this wasn't used to create a more flexible agent and to be able to better compare it to DS. However, to test if it is beneficial in Bos Wars to divide the task of winning, I decided to set up an experimental agent with this approach and test it against the flat counterpart. This agent uses a more hierarchical setup. It first has a goal to build a base and when this is achieved, it drops it and sets a new goal to conquer the enemy. The underlying idea is that it doesn't need to bother with rules to train units when building a base or to decide if it should construct more buildings when creating an army. This might speed up the learning task.

Of course, many more alternatives on how to structure the task exist and some may work better than others. However, since the current learning agents can only use a single game state for all learning pc-rules, it cannot use different beliefs to learn base building and to learn unit training. Also, there is only one reward that can be used to reflect on both tasks.

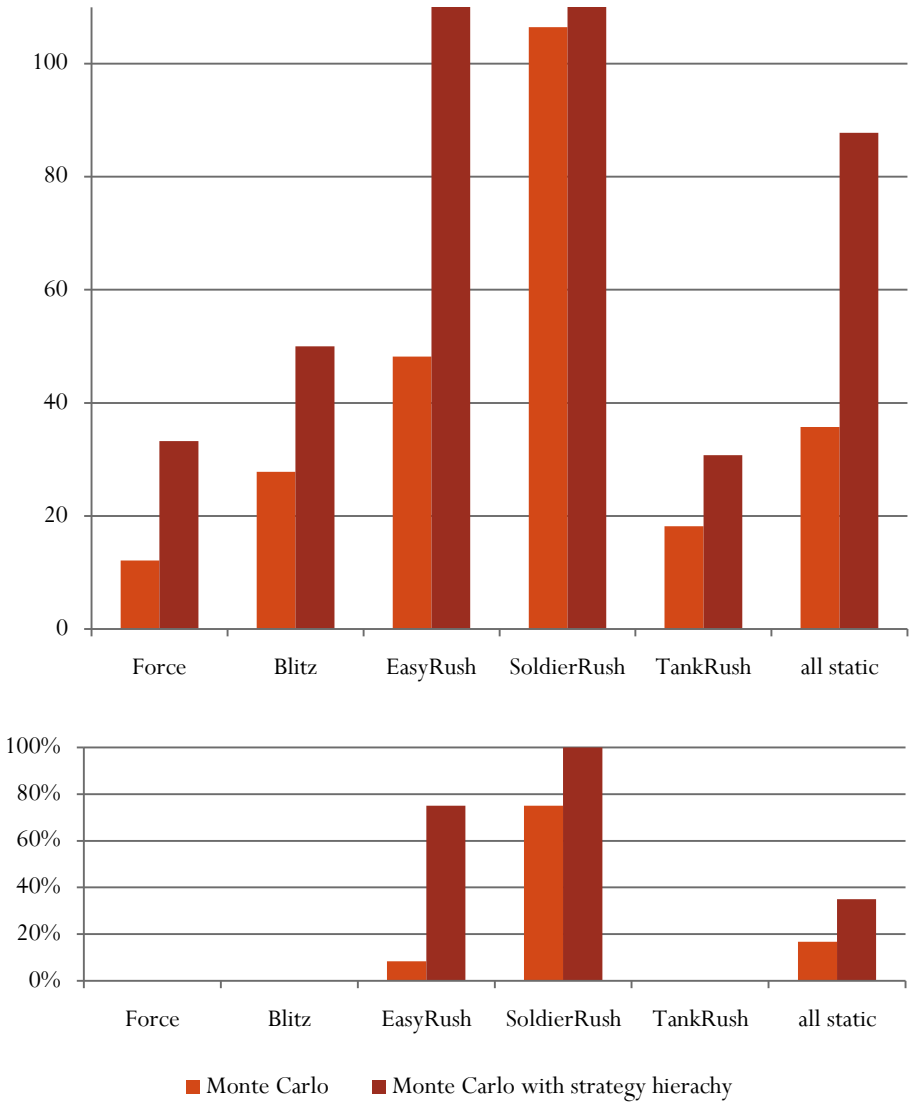


Figure 16: Average Turning Points and Fail Rates for strategy hierarchy and reactive agents against all static scripts

The agent that uses the strategy hierarchy is compared to the regular reactive agent. Tested again against all static scripts, the agent shows a decreased average learning speed of 144% as well as a fail rate that is 106% higher. The effect of the task division is thus adverse. (Since it wasn't able to outperform the SoldierRush opponent at all, again an average Turning Point of 200 episodes was used.)

Constructing a strategy is a hard yet not deeply structured task in Bos Wars. Although some causal relations exist, within the limited rule base used there are no long chains of consecutive actions. A relationship between first building a base and training an army afterwards is not too strictly connected but rather a consequence of the prerequisites of unit construction. It is often unclear where the building of a base stops and the training of the units begins. Also the rule guards already steer the action selection away from unusable rules depending in the game state.

With longer and more complex tasks I expect the hierarchical division of tasks to be more valuable. For example the low-level task of moving towards the enemy base using a ship to cross water. This task has discrete steps, selecting a transport ship, moving to it and loading, crossing the water and unloading and

continuing the journey to the enemy base. Later steps are restricted to former steps in the process. Actually this is similar to the multi-threaded task learning approach used in [Marthi *et. al.*, 2005] to learn the resource-gathering subgame.

6.4 MC-DS Hybrid using rule order as game state

One of the problems of using Monte Carlo learning in computer games is that it is often unclear what information to put in the game state vector [Manslow, 2002]. The original Monte Carlo agent uses data on the number of buildings and units owned as unique game state. This allows to, for example, give higher weights to the 'build a vehicle factory' rule when it doesn't already posses a vehicle factory.

Having a complex game state allows for a more refined action selection policy and thus a more refined game strategy. On the other hand it greatly increases the number of unique game states possible and thus the search space. Ideally, we would want to employ a minimal but still effective game state.

Dynamic Scripting has no unique game state at all but only has a fixed rule order, which is one of the main reasons for its fast learning. Is it possible to use this approach in classic reinforcement learning as well? To improve Monte Carlo learning, the idea of ordered rule selection has been incorporated into the MC agent. This will no longer use the environmental data in its game state, but only a sequence indicator mimicking the rule selection order. The environmental data will still be available to rule guards to test applicability during an episode.

The agent using a rule order as game state is called the MC-DS Hybrid, using the DS characteristic of non-informed rule selection with the more flexible and less-restricted MC algorithm. It is again tested against the static scripts to compare the learning performance.

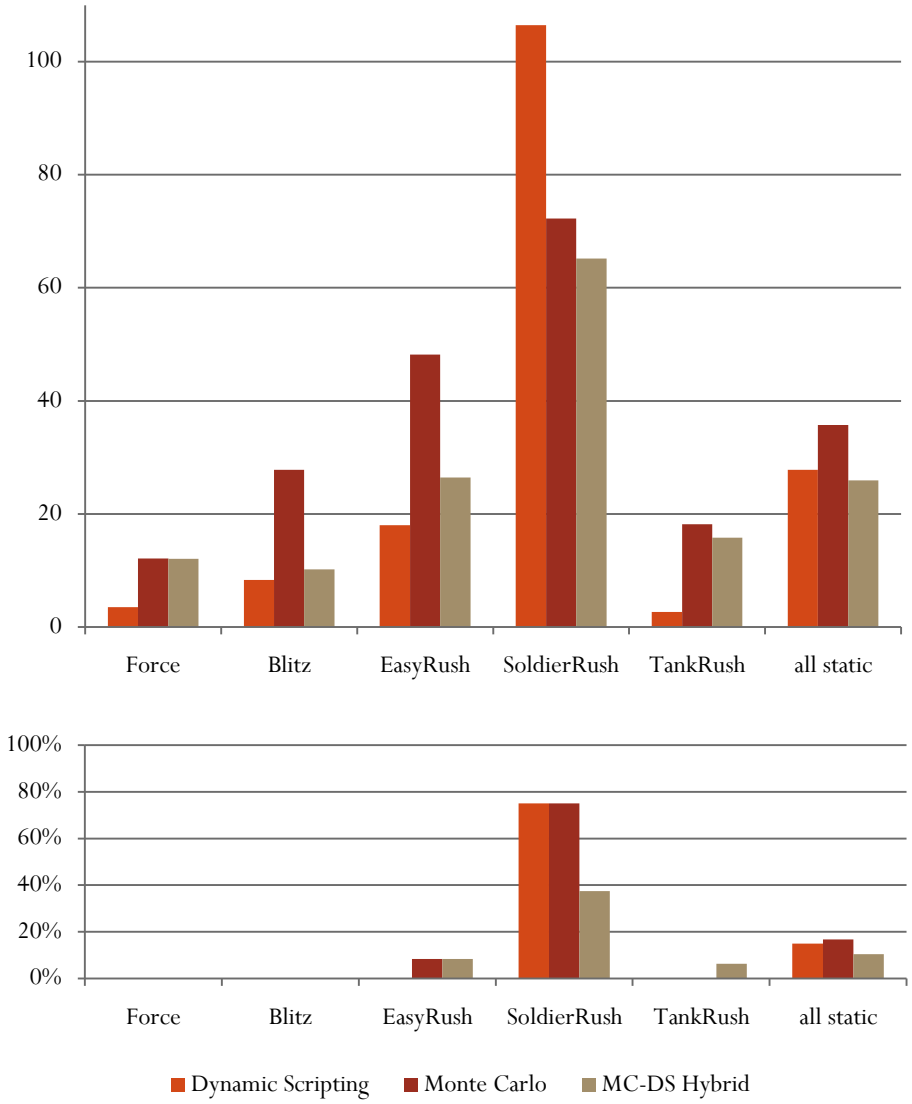


Figure 17: Average Turning Points and Fail Rates of the MC-DS Hybrid against all static scripts

Both the learning speed and fail rate are positively influenced compared to the regular Monte Carlo agent. The limited game state as used by DS can thus be utilized to improve the learning speed of Monte Carlo as well. An interesting observation is that the MC-DS Hybrid has a significantly lower Fail Rate against the SoldierRush compared to DS. The overall fail rate of the Dynamic Scripting is 50% higher than that of the MC-DS Hybrid agent. Also the learning speed is now similar to that of DS, even a little lower with a Turning Points at 26 instead of 28 episodes on average against all static opponents.

Compared to DS, the agent has several design benefits as well:

- It has no fixed script length, being flexible in the strategy size. There is no need to select invalid rules.
- Rules may be reselected. This makes rule base designs more intuitive and might be crucial for certain tasks. The effect of this should be investigated further.
- The game state may still be used if necessary or beneficial. An example of this will be shown later, including information from the opponent into the game state.

Using the rule order as game state in the Monte Carlo implementation is not the final answer to increasing learning efficiency. It can match the learning speed of Dynamic Scripting against the static opponents on average, but lacks a little against the easiest opponents Force and TankRush. Still, it is an interesting approach that may be useful, especially where it is unclear what information to put into the game state. With the rule order as game state, Monte Carlo can still effectively be applied.

6.5 Conclusions on learning improvements

The original agent's performance enhancement measures, rule guards and Softmax selection, have been proven very effective. Compared to their featureless counterparts, a major increase in learning speed and consistency has been achieved. The selection of unusable or unrealistic actions has been reduced significantly, improving the algorithm's effectiveness.

The introduction of more structure into the strategy planning has not been proven effective. The task of learning a winning strategy in Bos Wars is not very strictly ordered or layered. Therefore the agent's performance degrades. Yet the technique still seems very useful where higher and lower level tasks are combined, for example on building placement or on overseas unit path planning, especially when contained within the overall task of winning a game.

The combination of Dynamic Scripting with Monte Carlo is an interesting approach. It allows the MC algorithm to learn faster while needing no game state at all. This way it can match or even surpass the DS agent, making this algorithm redundant. It focuses on the order of actions rather than placing it in a specific context. This seems most useful with straightforward tasks with a complex environmental model, such as the winning-a-game task in Bos Wars as used in the experiments. There is no fixed script length or manually set rule order and is still easy to understand and implement.

Overall, Monte Carlo can be effectively used to learn a winning Bos Wars strategy. An agent using the considered enhancements may better meet the efficiency and consistency requirements set on RTS game playing agents. Also, a MC-DS Hybrid may replace a Dynamic Scripting agent.

6.6 Improvement ideas out of project scope

Many more ideas on how to improve the learning speed have come up during the project. Unfortunately, most could not be tested because of the project's time span. Here are some of the most promising ideas considered and discarded.

6.6.1 Enhancing Dynamic Scripting to overcome limitations

At some point during the project it was clear that Dynamic Scripting has several severe limitations in its design. As stated earlier it is bound to episodic tasks, has a fixed script size, uses once-only rule selection and can't reuse possibly mutually effective rules between situations. It is tempting to try to resolve these limitations by enhancing the algorithm. For example, it might learn the size of the script rather than have this fixed. For this however, a distinct learning process would have been needed which itself needs to be tested thoroughly. Allowing it to reuse already selected rules or adopt a game state to differentiate between unique world situations would fundamentally change the algorithm architecture, even tampering with its core idea.

After such radical changes Dynamic Scripting would have been incomparable to the original. Also, the outcome of such changes is very unpredictable. Since Monte Carlo learning already doesn't have these

downsides it was more logical to continue improving Monte Carlo to perform more like Dynamic Scripting. Although interesting and possibly effective, for this project it was more important to compare the original algorithms and trying to apply them in an RTS game playing agent. Therefore, during the project the limitations of Dynamic Scripting have not been minimized but rather taken as a fact, or even strength.

6.6.2 Comparison of more learning algorithms

During the development of 2aplLearn and its connection to Bos Wars, I implemented several reinforcement learning algorithms to test the learning of opposing strategies. Since there are many interesting methods to choose from, I decided to select the more general, dissimilar ones.

There are many more algorithms that might be interesting. For example since regular Sarsa proved to be far too inefficient to effectively apply in Bos Wars it was very tempting to implement eligibility traces through Sarsa(λ). This will likely overcome the original problem of learning only from immediate rewards. However, implementation of this or the many other interesting methods would become a task too large to cover during this project. Since I wanted to concentrate more on the overall learning performance and strategy switching techniques, no further time was available to implement additional learning algorithms. Also, the performance might have improved but the result would have been effectively the same thereby not directly contributing to the research questions stated.

6.6.3 Game state function approximation

For the implementation of all learning algorithms, but most importantly for Monte Carlo learning, the state space had to be reduced as much as possible. One of the ideas to reduce the search space was to introduce function approximation. This technique is commonly applied to reinforcement learning problems, most often using neural networks.

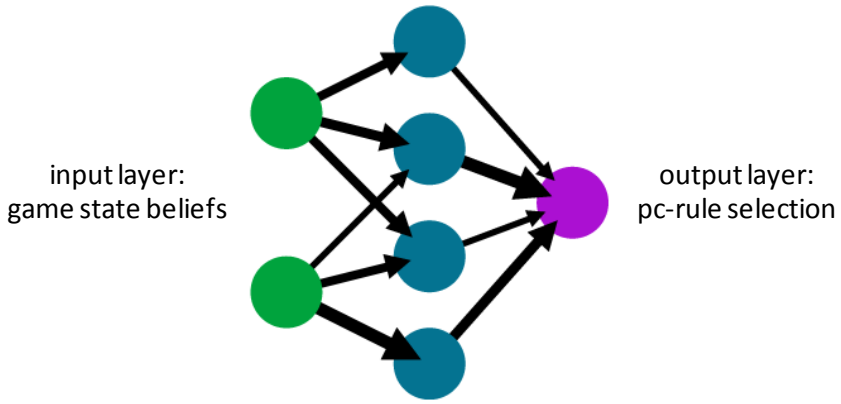


Figure 18: A neural network may be used as function approximator to generalize over the game state

However, implementation, testing and tweaking of such components can take a lot of time. Also, the effect it has in the domain of Bos Wars agents is uncertain since the generalization it makes over game states may not be very effectual or learning takes too long. There is a profound danger of overfitting which is larger than the likely benefits function approximation will bring for the limited RTS game domain. Also, the project's time constraints disallowed me to run preliminary experiments to better predict effectiveness of game state function approximators.

6.6.4 Semi-MDP reinforcement learning

If a 2apl agent learns a task in a computer game, not all actions it selects take the same amount of time. When it is learning which actions are best, through environmental rewards, it should ideally take this into account. If it would consider two strategies for example that receive equal rewards, it should prefer the one that took less time. When considering the temporal aspect of actions, the problem is called a semi-MDP. The effect of time for reinforcement learning algorithms has been extensively researched like in [Sutton and Barto, 1999].

For Bos Wars playing agents however, the cost of introducing the temporal aspect in their learning is too high to adopt it, most importantly because the agent will only receive one single reward at the end of an episode. Individual rules will thus all be reinforced with the same received reward. Although the temporal aspect is still interesting, the effect in the learning process is likely to be insignificant. For one thing, there is no need to optimize for one strategy opposed to another which lasts longer since no additional reward is given for this and thus strictly speaking it is not better. Considering all this, and the amount of time needed to introduce semi-MDP learning enhancements, this idea was not continued. In future work on learning 2APL agents it would still be a very interesting topic.

7 Adaptation to opponent strategies

In commercial computer games the game AI will likely not play against static scripts but rather against a human player. Humans will employ more challenging and changing tactics. They may use one tactic until this seems to fail and then switch to a new, distinct tactic, or keep switching their tactics between each game. This will cause learning computer opponents to fail using their just-learned rule selection. It is therefore interesting to see if the computer can learn not a static tactic but adapt to distinct opposing tactics itself.

In this chapter I describe the ideas and implementations behind several techniques tested, to adapt to opponents that switch between fixed, distinct strategies.

7.1 Strategy switching opponents

For the experiments on adaptive agents, several opponents were designed that switch their strategies. Unlike the static scripts, these opponents may switch tactics after each episode. This makes it harder to win a series of games. If a successful counterstrategy was found against a tank rush for example, this wouldn't necessarily be successful in the next game since the opponent may now employ a soldier rush. The switching computer players used are:

- Changing; switches among EasyRush, Force and Blitz tactics after it lost a game. This resembles an approach commonly used by intermediately skilled human players.
- Flipping; switches among EasyRush, Force and Blitz tactics after every 4 games
- FlipShort; switches among EasyRush, Force and Blitz tactics after every 2 games. Fast switching is more common amongst higher skilled human players.
- FlipLong; switches between EasyRush and Force tactics after every 8 games.
- RushFlip; switches between SoldierRush and TankRush after every 8 games.

First, all the previously used agents are tested against the new strategy switching opponents.

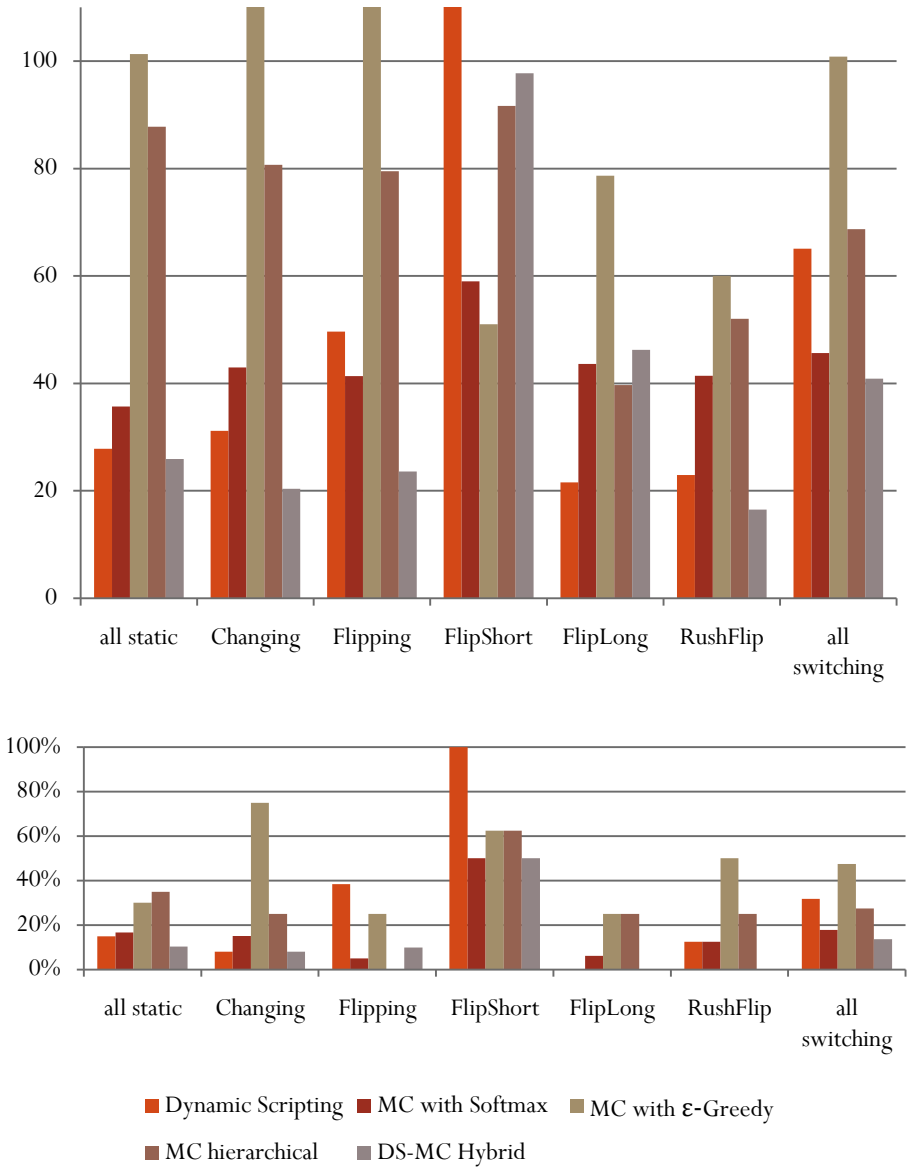


Figure 19: Average Turning Points and Fail Rates of non-adaptive agents against switching scripts

The results indeed show an increased difficulty in finding a winning strategy. The Dynamic Scripting agent has a significantly higher fail rate at 32%, whilst the MC-DS Hybrid has only 14% and Monte Carlo with Softmax has a 28% Fail Rate. It does especially poor against the faster switching opponents Flipping and FlipShort. Since it isn't able to outperform the FlipShort strategy, an average Turning Point of 200 episodes has again been used. The learning speed of MC (with Softmax) and the MC-DS Hybrid are substantially lower than that of DS, with 41 and 46 episodes as opposed to DS's 65 episodes on average against all switching opponents.

Next, we will look if we can use opponent adaptation to improve the results against strategy switching opponents. This can be achieved with implicit or explicit adaptation. Explicit adaptation has a separate model or tracking system for the opponent. It explicitly states the adaptation to that specific opponent by adjusting the values to new experiences with the player. On the other hand implicit adaptation has no additional model but uses the existing learning mechanism. It may use new game state data or new rules to the learning process so it can incorporate this into the decision process.

7.2 Implicit adaptation through learning

The original Monte Carlo algorithm needs game state information to differentiate which rules to select in what situations. The MC-DS Hybrid can be used if no state space is obvious or needed. However, there is a lot of environmental data that the agent can gather and use in its decision making. Can this be used in the adaptation to the strategy applied by the opponent?

My intuition is that you can better adapt to an opponent, creating more fit counter strategies, if you know what the opponent is going to do. If you incorporate this information you might adjust your decision making to the opponent strategy. Since it is already possible to include information about the world, it was natural to use this mechanism to realize opponent data inclusion.

An experiment has been set up to test if the Monte Carlo algorithm can implicitly learn to adapt. By including opponent information into the belief base, it is able to learn different weights for the rules on different opponent states. For instance, it assigns a different weight to the 'build a vehicle factory' rule when the opponent has a vehicle factory or not. For both the default Monte Carlo agents as well as the MC-DS Hybrid, a version was programmed that includes whether the opponent has a vehicle factory or a soldier training camp (or both or none). This is expected to be useful to better reason what to build ourselves. If the enemy hasn't got a training camp, he will likely attack with tanks so I might rush some soldiers to attack with before he has finished his tanks.

This approach is very easy to implement. Bos Wars had to send this information (which works exactly like the messages on its own possessions) and then it is as easy as including this in the agent's belief base. The MC and hybrid agents are again put to the test against all opponents to measure the learning speed and fail rate.

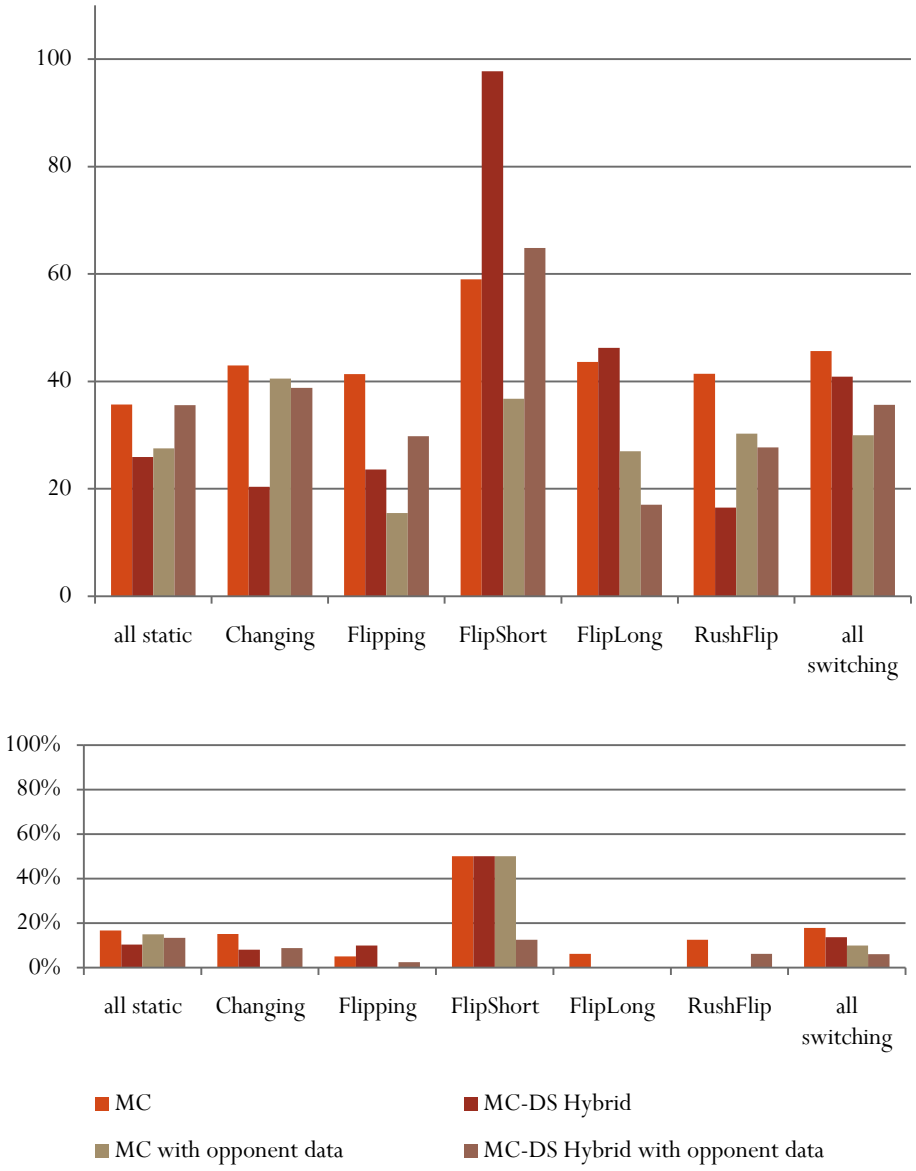


Figure 20: Average Turning Points and Fail Rates for opponent data-including agents against switching scripts

It seems that including opponent data can have considerable positive effect on the learning speed and stability. The average number of episodes to reach a Turning Point in MC and MC-DS Hybrid agents against switching scripts is 53% and 14% higher if opponent data is not used in the game state. Both also perform better in respect to the consistency, having a fail rate that is respectively 80% and 67% higher when no opponent data is used. If the actual learned strategies are analysed, we see that it certainly makes use of the differentiation between specific situations. It will often prefer a tank-based attack when the opponent has tanks as well, and a soldier-based attack when the opponent has a unit training camp. Dynamic Scripting has a fixed script during an episode and can thus not use acquired opponent data in its decisions.

A note should be made on how the information on the enemy buildings is gathered. This is extracted directly from the game engine and send to the agent. Since it hasn't actually seen these buildings on the map, a 'fog of war' hides the unexplored territories, one might call this cheating. Ideally, the agent should have no more knowledge than a human player. However, game developers often use such

computer player advantages. They are safe as long as it is not unrealistic or noticeably unfair and helps the game play. Alternatively, the agent could send one cheap, simple unit to the opponent base. The human player may identify this as a spy sniffing around its base, giving credit to the game for being smart, which adds to the game experience..

7.3 Explicit adaptation through expected result monitoring

Instead of including opponent data into the agent's game state, we could employ an explicit approach to adaptation. If it is possible to notice when the opponent switches tactics, we can change our own as well. Since we want this to work against human players as well, we can't just look at the opponent's AI statistics.

With strategy switching opponents, there is a strong distinction between used tactics. I expected that, after establishing a strong counter strategy, the agent would suddenly lose because the tactic doesn't work any against the new opponent strategy. This would be a nice moment to stop the learning and start learning against the new opponent strategy. Ideally we can thus lay a direct connection between an unexpected loss and the opponent switching strategies. Although there are more ways to model explicit adaptation, it seems as an interesting idea.

The approach used here is to monitor our own results and detect when we suddenly lose. The idea is that, the opponent may switch and at that point making our tactic useless. However we would want to use this in the future. If the opponent later changes back to the original strategy, we could reuse the specific counter-tactic. For each distinct opponent strategy, we will learn a counter-tactic.

To recognize when we established a winning strategy, and thereafter whether we suddenly lost, isn't trivial. The only information on this is the rewards that have been received at the end of each episode. The new agent will thus monitor the rewards and calculates if it has suddenly dropped. To allow for a certain unexpected but not necessarily bad episode, it should moderately assign an opponent strategy switch. This is a complex computation which is highly domain specific. Several functions have been considered for use with Bos Wars. The one that was finally implemented compares the results of the last two episodes with the two before that. To moderate effects it has a fail margin parameter that can be set.

$$unexpectedLow = \frac{(\sum Returns(LastTwo)/2)^2}{(\sum Returns(PreviousTwo)/2)^2 \times failMargin}$$

If an unexpected low return has been identified, it stores the current strategy in the known strategy base. If any other strategies were already stored, it will test these again, for these might now be again applicable. If none of them are indeed usable, or there were no saved strategies, it will start learning again. A MC-DS Hybrid-based agent, called the MC-DS Hybrid XR agent, that uses this expected return monitoring was tested against all opponent scripts. The fail margin was set to 0,2.

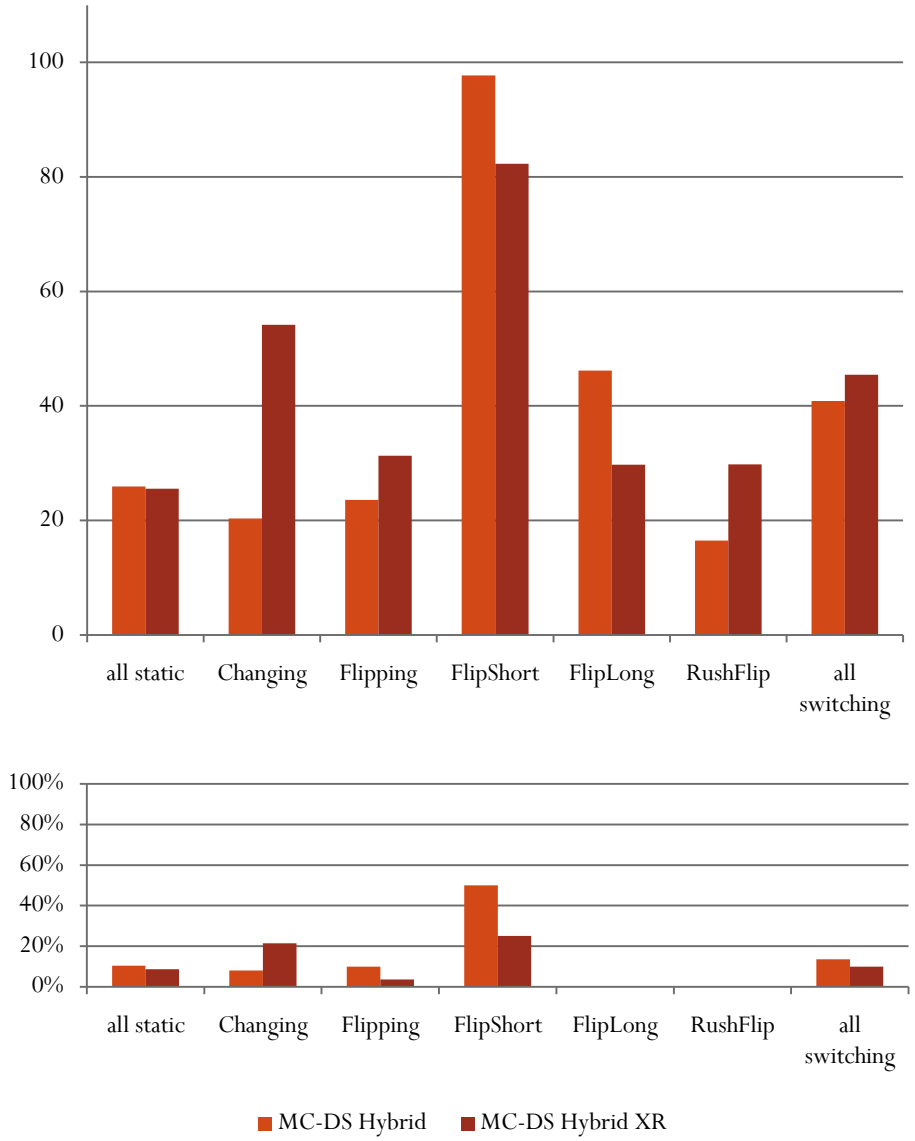


Figure 21: Average Turning Points and Fail Rates of the MC-DS Hybrid XR against all switching scripts

The explicit adaptation through expected result monitoring seems to have no solid beneficial effect on the learning speed. Except for the FlipLong and FlipShort tactics, it actually takes longer to outperform the opponent. Over all switching scripts, the average Turning Point and Fail rate have not considerably been improved.

More importantly, when examining the agent's behaviour and saved strategies, it doesn't seem to actually correctly recognize opponent strategy switches. In most cases it found a tactic that was just good overall, wining the series from there on without losing. In the cases when it did work out distinct strategies, these were not always proper counter-tactics but rather a half-developed strategy based on mixed experiences. In rare cases everything worked as designed, mainly against the FlipLong tactic where it has more time to adjust and opponent changes are more dramatic, but never very quickly.

The results can be explained in various ways. Most importantly, the opponent scripts will often change too fast for the script to learn a distinct strategy. This makes it hard to assign a certain winning streak to

a specific opponent strategy, or a sudden loss to a strategy switch. Also, the rewards received do not always reflect the actual applicability of the employed strategy at that moment. This makes it very hard to reasonably make sound conclusions on just a series of rewards, despite the fact that the actual detection of sudden losses works well. Finally, it frequently happened that the first winning episode was a strategy that worked well against the opponent's other strategies as well. It will then win a series long enough to outperform the opponent and end up not using the adaptation mechanism at all.

7.4 Multi-agent learning to test performance

To further analyse the actual strategies that learning agents employ as well as to compare their adversary performances, multi-agent experiments are carried out. For this, the Bos Wars environment was extended to allow multiple 2apl agents to compete in Bos Wars against each other.

The most effective agents were put to the test. In the first experiment it was tested if some of the algorithms significantly outperformed the others. The Dynamic Scripting, Monte Carlo, MC-DS Hybrid and MC-DS Hybrid with opponent data agents each played 8 games against each other. An agent wins if it outperforms the opponent, having a series where 12 points are gathered in a 7 episodes scope.

	Dynamic Scripting	Monte Carlo	MC-DS Hybrid	Opponent data	won	lost	result
Dynamic Scripting	X	X	X	X	11	13	-2
Monte Carlo	4 - 4	X	X	X	12	12	0
MC-DS Hybrid	3 - 5	3 - 5	X	X	11	13	-2
Opponent data	6 - 2	5 - 3	3 - 5	X	14	10	4

Table 2: Results of the multi-agent competition to outperform the adversary

None of the algorithms was much stronger than learning too quick for the other to come up with counter-tactics. The MC-DS Hybrid agent that uses opponent data (if it has a vehicle factory or a training camp) has a little more points, but with these numbers it cannot be argued that this is substantial. It seems to be mostly a result of who happens to find a near-optimal tactic first.

To study this in more detail, it might be interesting to see if any of the agents would gather much more points in several games that are not finished on a winning series of 6-7 episodes. Instead they will compete against each other for the full 200 episodes to see if they keep finding new counter-tactics over and over again.

Shown below are the results of the 200-episode-long series of games where two agents fight to collect the highest total reward. The average rewards for the full game are shown.

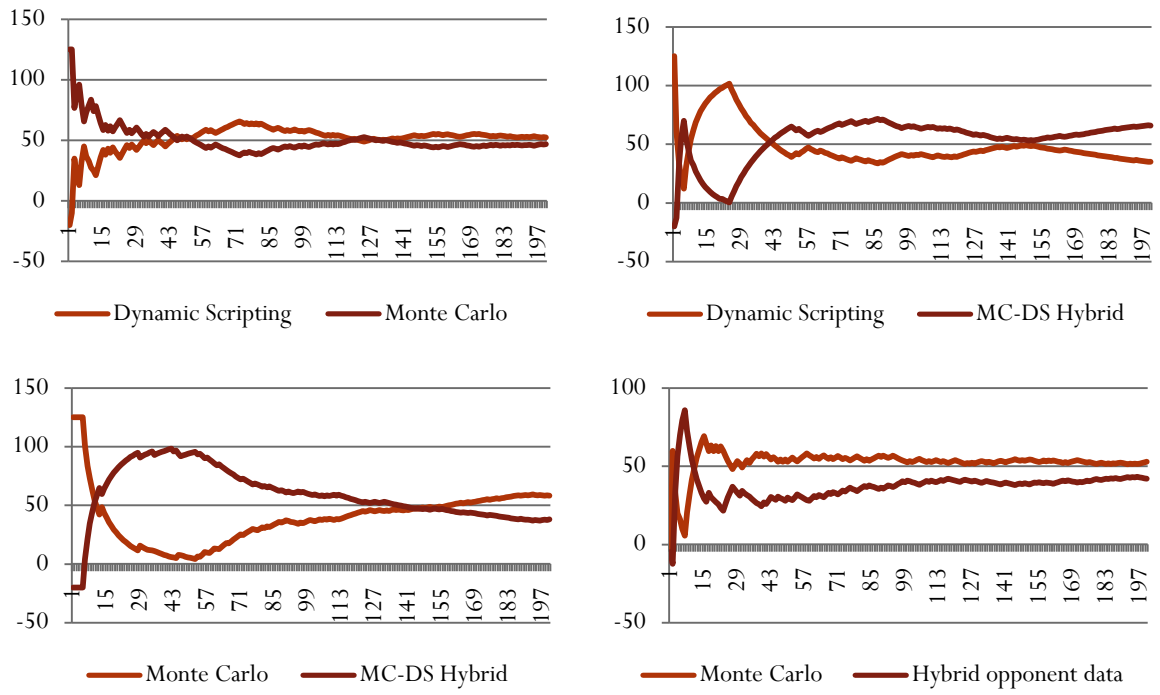


Figure 22: Average rewards during some example competition runs between all agents

Very interesting to see is how agents learn a winning strategy and keep using this as long as they win. The opponent will instead search for a counter-tactic in which all agents will succeed eventually. There has not been a single strategy that dominates.

From the experiments against static and switching opponent scripts it was shown that agents using opponent data performed best. From the long multi-agent experiment series not a single agent shows to be dominant over the other strategies. After 200 episodes the average rewards for all agents are very similar. The examples shown can just as easily be reversed in a next run.

The most important result from the multi-agent experiments is that all agents can learn against other learning agents. If one of them would be actually used in Bos Wars, it will be able to form counter-tactics against any human player. If the human continues to develop its strategy, even switching drastically, all agents will still be able to come up with a relevant answer.

7.5 Conclusions on strategy adaptation

Adapting to the opponent strategy is not an easy task. Information about the opponent's game state or the actual game results may be used. Both approaches were tested in Bos Wars. Explicit modelling by just monitoring the game results seems infeasible. It is too hard to accurately identify opponent strategy switches and to assign distinct tactics to this. An approach using explicit player modelling might be more viable, however this is untested.

Implicit modelling by inclusion of opponent data in the game state does considerably increase the learning efficiency and consistency. Both the average Turning Points and Fail Rate numbers are lower than their non-opponent-aware counterparts. We have only used a simple statistic on whether the opponent has a vehicle factory or training camp, which works well. More information might or might

not be helpful. Interestingly, Dynamic Scripting cannot make use of opponent data since it has a fixed script during an episode, and has a much higher Fail Rate against the strategy switching opponents.

7.6 Adaptation ideas out of project scope

During the project, several ideas have spun off from the original adaptation idea. Those of which I decided not to continue with are described here.

7.6.1 Explicit adaptation through player modelling

An interesting approach to adapt explicitly to the opponent is by monitoring its tactics. We could focus on the actions he takes to reason about what he will do next. This can either be done using model fitting or using prediction techniques.

With model fitting we have several known models of distinct strategy classes. A model may be the soldier rush or a defence-first approach. If we could identify the opponent's actions to fit into one class, we can use a known counter-tactic. An alternative is to predict the opponent's next action based on his state and what he has previously done. This could for example be done using a separate reinforcement learning algorithm, where we don't select actions ourselves, or just a neural network. This may be more fine-grained and doesn't need pre-known models, but it may take a lot of episodes to predict effectively. Both can be implemented as learning 2apl agents using complex rules as strategy models and rewards when the model has successfully predicted the result. Unfortunately, this is a major topic on its own and far out of the reach of this project.

7.6.2 Opponent strength adaptation

The original Dynamic Scripting algorithm has already been extended with opponent strength adaptation. The learning game AI tries not to outperform the opponent but rather offer a challenge by adjusting its performance to the measured opponent strength. From the human perspective this is 'more fun' since it makes the computer player seem intelligent without being unbeatable. From the methods suggested in [Spronck, 2006], Top Culling proved most efficient. Dynamic Scripting equipped with Top Culling was able to adjust its own performance against several weak and strong computer role-playing game tactics.

Learning Bos Wars agents could adapt this functionality as well, for both Dynamic Scripting and Monte Carlo algorithms. Rather than selecting the rules with the highest weights all the time, it should exclude rules with weights that are too high. That is, rules that have a weight above some threshold value determined by experimentation. It is also interesting to attempt to integrate this technique into the Monte Carlo algorithm. However, because of the focus of the project as well as the time that it would take to build and test the enhancements, opponent strength adaptation was not worked out.

8 Project conclusions

In this project we have attempted to create better high-level winning real-time strategy game AI. In this chapter I will overlook the project results and form conclusions on this.

First, I will reflect on the requirements on learning in RTS games as set at the project start. If the agents meet the requirements, a reflection on whether we solved the original problem can be made. Finally, I will discuss how this work may be used in real-world computer games and in research.

8.1 Meeting the learning requirements

To cope with the online and entertaining nature of an RTS game, several requirements on the learning process have been set. These should be met to make the learning behaviour feasible and usable in a real real-time strategy game such as Bos Wars.

First, the modelling of computer players as software agents already brings several advantages. Due to the nature of how agents work and how the platform is designed, 2apl agents inherit the features to meet the requirements on clarity, variability, robustness, effectiveness and computational speed. The rules that define an agent's behaviour are easy to program and explain. Unwanted action selection can be prevented by using Softmax and rule guards with domain knowledge. Higher level notions may be used to create more flexible and robust decision making and very little computational resources are used, only when a new choice has to be made. Agent programming in 2apl is very natural and the platform is easy to attach to a computer game.

In regard to the performance of the learning approaches, all agents were tested against both static and strategy switching scripts. The final performance graph shows the learning efficiency and consistency. Through the switching opponents the adaptability was tested.

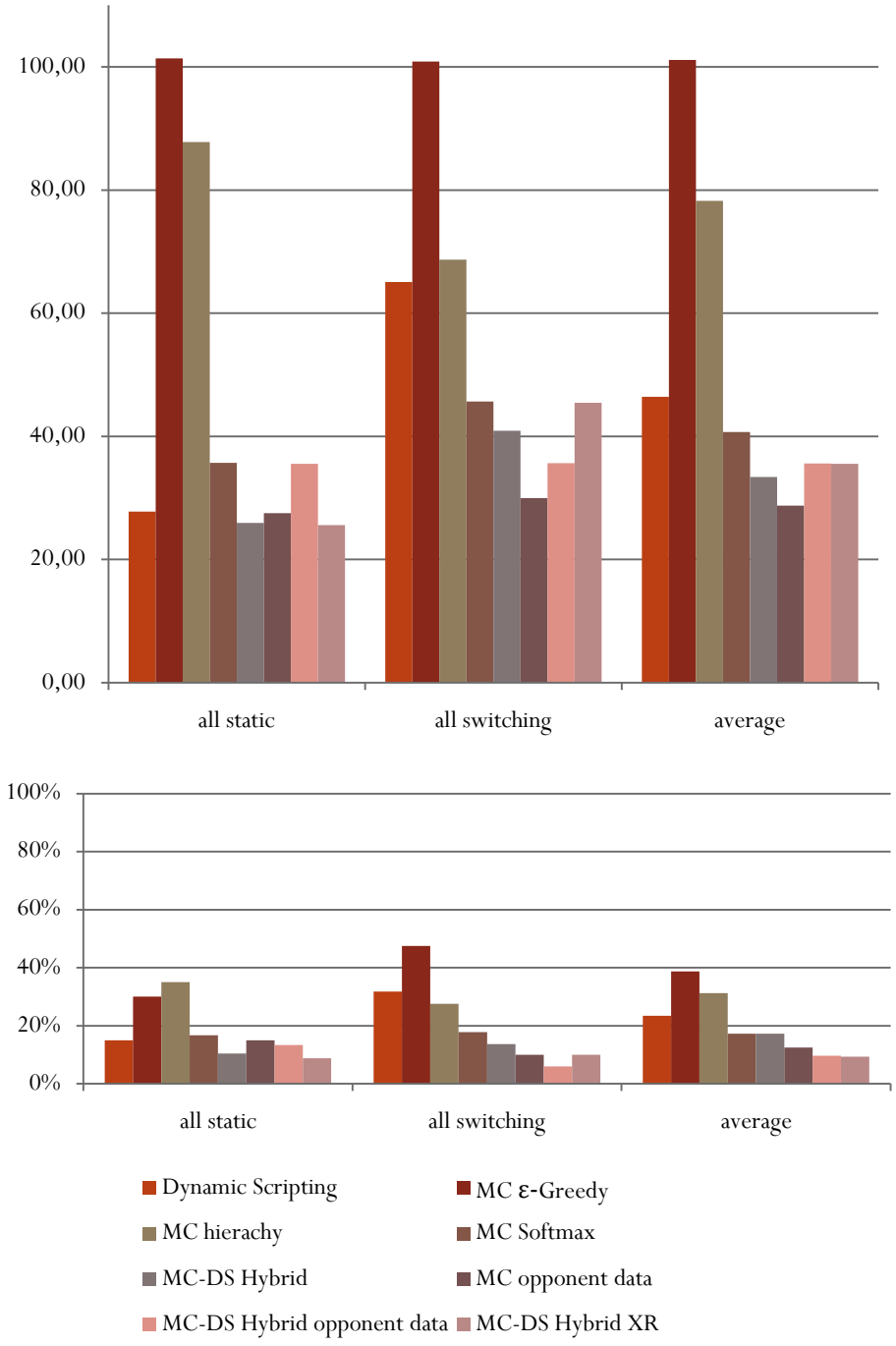


Figure 23: Average Turning Points and Fail Rates for all agents against static and switching scripts

The best performing agent against static scripts considering the average Turning Point was the MC-DS Hybrid. This agent uses the distinctive beneficial features of the Monte Carlo and Dynamic Scripting agents. This way it has a learning performance similar to DS without any of this algorithm's limitations. The static opponent script can be outperformed on average in 26 episodes, including easy and very hard to beat static scripts.

Looking at the adaptation to strategy switching agents, we see the implicitly adaptive agents that use opponent data perform best. They have average Turning Points and Fail Rates that are significantly lower than there non-opponent-aware counterparts. Dynamic Scripting performs even worse than the

plain MC agents, failing to outperform the switching opponents in 32% of the cases. Clearly this technique is not very well suited to cope with switching strategies. The MC agent with opponent data needed just 30 episodes on average to outperform the complex, switching opponents. Its strength lies in the ability to adapt to the opponent's distinct tactics used. Furthermore, since it performs just as well as DS against static opponents, this agent makes the Dynamic Scripting technique redundant.

Evaluating the results, we may conclude that at least the best performing agent, MC with opponent data, is efficient, needing only very little episodes to outperform any opponent. Both easy (Force) and very hard (SoldierRush) scripts were used as well as more challenging and human-like opponents of varying difficulty (Changing and FlipShort), proving the agent's scalability. Since the agent has a very low Fail Rate of only 13%, it meets the consistency requirement as well.

In general, the efficient and effective learning of MC agents can be attributes to the integration with agent technology. The BDI design is capable of learning plan blocks rather than basic actions and may use rule guards and clever game states to accelerate the learning speed. The final agents have thus overcome the classic game AI related problems of Monte Carlo, that is learning too slow, having a too complex game state and being hard to monitor. The final result graphs with data tables can be found in Appendix III.

8.2 Reflection on problem statement

Since the learning agents meet the requirements on learning game AI players, they may actually be integrated in commercial computer games. The learning agent we designed and tested was build to solve the problems with classic game AI approaches, static scripts in particular. Most modern computer games still use static scripts for the high-level strategy decision making. However, such scripts are fixed, repetitive and predictable. To provide a more fun and challenging opponent, one that can fix its own strategy, a new approach must be used.

To offer a fun computer player, the agent should use varied and novel strategies. This is achieved in the learning agent by allowing it to change the employed actions and by using exploration to find unused approaches. It will keep varying its strategy, increasing the game experience. This achievement can be assigned to the BDI agent design.

To provide a challenging computer opponent to all skill levels, the learning agent should employ strong tactics that are unpredictable. Because the agent keeps improving its game play, dismissing unsuccessful tactics, it will eventually outperform the human player. Also, because it keeps trying different and original combinations, it will confront the human with unforeseen tactics. On top of this it can exploit human weaknesses by using those unique strategies that proved to win against the human. He will thus be challenged to improve itself to again defeat the game AI player.

As already hinted at, it learns what ineffective behaviour is and avoids this. This way it may reject such tactics on new games, effectively repairing its own flaws. With the measures taken such as using Softmax and rule guards, stupid action selection is minimized.

The learning agents can thus effectively overcome all limitations of static scripts.

8.3 Usability of learning agents in computer games

The new fun and challenging agents can be use in real-world RTS games. The approach to use an agent platform such as 2apl is highly recommended, since this provides an effective general framework which is easy to connect with and supplies the mandatory features such as learning. Also, the agent programs are understandable and can be tested and altered without recompiling the game code, as with scripts.

Similarly important is that the learning agents offer a more fun and challenging experience to a Bos Wars player. They adapt their strategies, coming up with interesting and tough new tactics to counter. This greatly enhances the game play. Also, the agents will discard any inferior strategies employed; fixing themselves after the game has been released.

Dynamic Scripting has been the project's reference point. This method has proven to be successful in role-playing games. In Bos Wars, it still works reasonably well, but lacks on the ability to adapt to strategy switching opponents. An approach using Monte Carlo learning with opponent data in its game state is better suitable for the RTS game domain.

For problems where a game state vector is hard to design, we may use the MC-DS Hybrid agent. This needs no full game state but is still as flexible as a regular Monte Carlo agent and maintains a solid learning performance.

In the experiments, a flat agent structure has been used to be able to compare DS to other approaches. However, the full featured and flexible 2apl framework also allows a more structured design. Reasoning on high-level aspects such as goals may be used, as well as sub-plans. Hierarchical learning may be adopted to learn higher and lower level tasks at the same time.

In conclusion, the resulting Bos Wars playing agents form effective, fun and challenging opponents, overcoming the problems with static scripts and enriching the RTS game experience. They vary their tactics and demand the human to come up with novel new plans to defeat the computer adversary.

9 Future research

Although a lot of research has already been done, there are numerous new ways to further improve the learning agents or to make them even more useable for commercial games. Some hints towards the most promising future studies are explained here.

9.1 Learning on a full complex, layered game task

Throughout the project we have used agents that can only use a limited subset of buildings and units. This made it easier to run experiments and compare algorithms. It would be very interesting to see how learning agents will do when a full belief base was to be implemented. It would be able to build more rich bases and create more complex and varied attack squads. Since the amount of possibilities will be much larger, it is expected yet not implied that learning will take longer.

The expanding of the task could be extended further, incorporating not only high-level strategy decisions, but also some low-level actions. For example it would be able to direct troops via certain paths or have more control over building placement. New questions may be answered such as what novel new tactics it may learn to use and if it would benefit from hierarchical learning in relation to the more layered search space.

If in the future agents want to learn truly distinctive tasks it will probably be necessary to refine the learning architecture. Currently, an agent may learn over multiple distinct sets of pc-rules but the same game state is used for both problems. Also, rewards from the environment are always calculated back into all visited rules. It is likely beneficial if a separate game state and specific rewarding of pc-rules are needed to allow solid multi-task learning within one agent. For example, it a Bos Wars engineer would be designed as an agent and both the building placement task as well as the resource gathering task are to be learned, the agent programmer will want to assign distinct rewards and game states. A future project should be set up to model multi-task learning in the agent architecture.

9.2 Strategy visualisation tool

The point has often been raised that reinforcement learning has the problem that it is unclear why certain decisions are made. Although the regular Monte Carlo algorithm has been proven formally to converge to the optimal strategy, the weights of rules offer no human-understandable explanation to an agent's behaviour. This is most apparent in the classical form where basic actions are learned. However, already with the integration of the agent platform, a lot of clarity is brought back. Agents no longer follow an incomprehensible pattern of low-level actions but learn to select chunks of strategy. The decisions are never totally out of place as well, due to the use of rule guards and domain knowledge.

On the other hand, it may still not always be fully predictable what an agent does. To make decision making more explainable and accordingly understandable, an easy view on the decisions-to-be-expected may be given. One approach is to build a visual tool that analyses the agent's rule weights and provides a hierarchical decision diagram that shows what choices it will make in which unique situations. It may visualize what it will differentiate on based on the game state. This way the path an agent will take can be analysed before it is actually ran (but after several training episodes). In the worst case such a tool can be used to explain the action selections afterwards. This will improve the insight into the learning process, especially for non-programmers.

9.3 Explicit player modelling

The use of opponent data in an agent's game state allowed for implicit adaptation in the learning process. Another way to better adapt to the opponent is by the use of an explicit player model. This model may use several known classes into which the opponent can be fitted, or adopt several characteristics parameters that are altered to the specific opponent.

An explicit player model has several uses. Performance against strategy switching opponents may be improved, a skill level can be learned to which it may adapt and the regular learning speed can be increased by using known specific counter-tactics.

9.4 Bos Wars as an agent research platform

The main project focus was to replace scripts in computer games by adaptive reinforcement learning agents. However, the resulting game platform can also be effectively used in agent technology research. The Bos Wars RTS game provides a challenging and multi-facetted environment for research in a large number of topics. This includes for instance high-performance reinforcement learning and planning, multi-agent communication and negotiation, agent societies, data mining and human player modelling. The existing framework provides the essential basis for all of these.

10 References

[Aha *et. al.*, 2005] David Aha, Matthew Molineaux and Marc Ponsen (2005). *Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game*. Proceedings of the Sixth International Conference on Case-Based Reasoning, pp. 5-20.

[Buro and Furtak, 2003] Michael Buro and Timothy Furtak (2003). *RTS Games as Test-Bed for Real-Time Research*. Invited Paper at the Workshop on Game AI, Joint Conference on Information Sciences 2003, pp. 481-484

[Barnes and Hutchens, 2002] Jonty Barnes and Jason Hutchens (2002). Testing Undefined Behavior as a Result of Learning. AI Game Programming Wisdom (ed. S. Rabin), Charles River Media, pp. 615-623

[Chan *et. al.*, 2004] Ben Chan, Jörg Denzinger, Darryl Gates, Kevin Loose and John Buchanan (2004). *Evolutionary behavior testing of commercial computer games*. Proceedings of the Conference on E-Commerce Technology, Portland, 2004, pp. 125-132.

[Dastani *et. al.*, 2007] Mehdi Dastani, Dirk Hobo and John-Jules Meyer. *Practical Extensions in Agent Programming Languages*. In Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems, 2007.

[Fogel *et. al.*, 2004] David Fogel, Timothy Hays and Douglas Johnson. *A platform for evolving characters in competitive games*. In Proceedings of the Congress on Evolutionary Computation, pp. 1420-1426.

[van Lent *et. al.*, 1999] Michael van Lent, John Laird, Josh Buckman, Joe Hartford, Steve Houchard, Kurt Steinkraus and Russ Tedrake (1999). *Intelligent Agents in Computer Games*. Proceedings of the National Conference on Artificial Intelligence, July 1999, Orlando, FL, pp. 929-930.

[Liden, 2004] Lars Lidén (2004). *Artificial Stupidity: The art of making intentional mistakes*. AI Game Programming Wisdom 2 (ed. S. Rabin), Charles River Media, pp. 41-48.

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

[Marthi *et. al.*, 2005] Bhaskara Marthi, Stuart Russell, David Latham and Carlos Guestrin (2005). *Concurrent hierarchical reinforcement learning*. In International Joint Conference on Artificial Intelligence, Edinburgh, Scotland.

[Ponsen and Spronck, 2004] Marc Ponsen and Pieter Spronck (2004). *Improving Adaptive Game AI with Evolutionary Learning*. Computer Games: Artificial Intelligence, Design and Education (eds. Quasim Mehdi, Norman Gough, Stéphane Natkin and David Al-Dabass), pp. 389-396.

[Ponsen *et. al.*, 2006] Marc Ponsen, Héctor Muñoz-Avila, Pieter Spronck, and David Aha (2006). *Automatically Generating Game Tactics with Evolutionary Learning*. AI Magazine, Vol.27, No. 3, pp.75-84.

[Spronck *et. al.*, 2003] Pieter Spronck, Ida Sprinkhuizen-Kuyper and Eric Postma (2003). *Online Adaptation of Game Opponent AI in Simulation and in Practice*. Proceedings of the 4th International Conference on Intelligent Games and Simulation (eds. Quasim Mehdi and Norman Gough), pp. 93-100.

[Spronck, 2005] Pieter Spronck (2005). *Adaptive Game AI*. Ph.D. thesis, Maastricht University Press, Maastricht, The Netherlands.

[Spronck *et. al.*, 2006] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper and Eric Postma (2006). *Adaptive Game AI with Dynamic Scripting*. Machine Learning, Vol. 63, No. 3, pp. 217-248.

[Sutton and Barto, 1998] Richard Sutton and Andrew Barto (1998). *Reinforcement learning: An introduction*. Cambridge, MA. MIT Press.

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

Appendix I. A simple example Bos Wars agent

```
Beliefs:
^gameCycle(0).

BeliefUpdates:
{ gameCycle(O) } UpdateCycle(N) { not gameCycle(O), gameCycle(N) }

Plans:
{
  ^setting("epsilon", 0.1);
  ^startEpisode();
  adopta(attack(enemy))
}

PG-rules:
  attack(enemy) <-
    true | {
    prepare(base); }

PC-rules:
  ^prepare(base) <-
    true | {
    ^log("Constructing base");
    @boswars(setUnit("unit-engineer", 4), R);
    @boswars(setUnit("unit-powerplant", 1), R);
    @boswars(setUnit("unit-magmapump", 2), R);
    @boswars(setUnit("unit-vfac", 1), R);
    @boswars(setUnit("unit-gturrent", 1), R);
    @boswars(waitUnit("unit-vfac"), R);
    launch(attack); }

  ^launch(attack) <-
    true | {
    ^visit();
    ^log("Single tank attack");
    @boswars(defineForce(0, "unit-tank,1;"), R);
    @boswars(waitForce(0), R);
    @boswars(attackWithForce(0), R);
    dropGoal(attack(enemy)); }

  ^launch(attack) <-
    true | {
    ^visit();
    ^log("Balanced tank attack");
    @boswars(defineForce(0, "unit-rtank,2;unit-tank,2;"), R);
    @boswars(waitForce(0), R);
    @boswars(attackWithForce(0), R);
    dropGoal(attack(enemy)); }

  ^launch(attack) <-
    true | {
    ^visit();
    ^log("Large tank attack");
    @boswars(defineForce(0, "unit-tank,6;"), R);
    @boswars(waitForce(0), R);
    @boswars(attackWithForce(0), R);
    dropGoal(attack(enemy)); }

  updateGameCycle(NewCycle) <-
    true | {
    UpdateCycle(NewCycle); }

  updateGameResult(Result, Fitness) <-
    true | {
    ^reward(Fitness);
    ^endEpisode();
    ^log("End game fitness:", Fitness); }
```

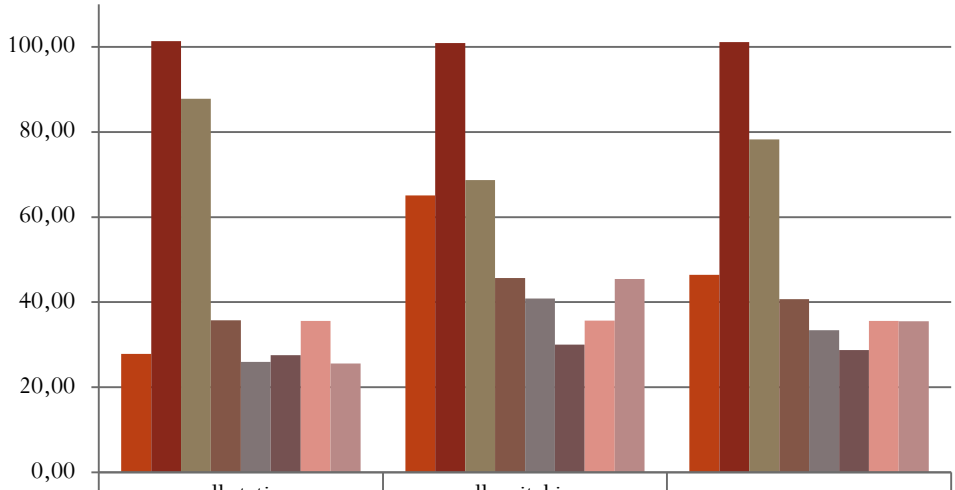
Appendix II. The experimental Bos Wars agent

```
Beliefs:
^gameCycle(0).
unit(unit-vault,0).
unit(unit-powerplant,0).
unit(unit-magmapump,0).
unit(unit-camp,0).
unit(unit-vfac,0).
unit(unit-gturret,0).
unit(unit-engineer,0).
unit(unit-assault,0).
unit(unit-tank,0).
unit(unit-rtank,0).

BeliefUpdates:
{ gameCycle(O) }              UpdateCycle(N) { not gameCycle(O), gameCycle(N) }
{ unit(UnitType, OldCount) } UpdateOwnUnit(UnitType, NewCount) { not unit(UnitType,
OldCount), unit(UnitType, NewCount) }

Plans:
{
  ^startEpisode();
  ^setting("tau", 3);
  script(game);
}

PC-rules:
  ^script(game) <-
    unit("unit-engineer",X) and X < 2 | {
    ^visit();
    ^log("unit-engineer", 2);
    @boswars(setUnit("unit-engineer", 2), R); }

  ^script(game) <-
    unit("unit-engineer",X) and X < 4 | {
    ^visit();
    ^log("unit-engineer", 4);
    @boswars(setUnit("unit-engineer", 4), R); }

  ^script(game) <-
    unit("unit-engineer",X) and X < 6 | {
    ^visit();
    ^log("unit-engineer", 6);
    @boswars(setUnit("unit-engineer", 6), R); }

  ^script(game) <-
    unit("unit-powerplant",X) and X < 1 | {
    ^visit();
    ^log("unit-powerplant", 1);
    @boswars(setUnit("unit-powerplant", 1), R); }

  ^script(game) <-
    unit("unit-powerplant",X) and X < 2 | {
    ^visit();
    ^log("unit-powerplant", 2);
    @boswars(setUnit("unit-powerplant", 2), R); }

  ^script(game) <-
    unit("unit-magmapump",X) and X < 2 | {
    ^visit();
    ^log("unit-magmapump", 2);
    @boswars(setUnit("unit-magmapump", 2), R); }

  ^script(game) <-
    unit("unit-camp",X) and X < 1 | {
    ^visit();
    ^log("unit-camp", 1);
    @boswars(setUnit("unit-camp", 1), R);
    @boswars(waitUnit("unit-camp"), R); }

  ^script(game) <-
```

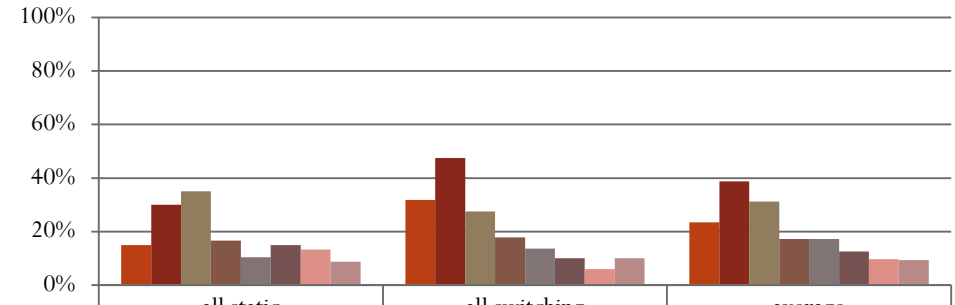
```
  unit("unit-vfac",X) and X < 1 | {
  ^visit();
  ^log("unit-vfac", 1);
  @boswars(setUnit("unit-vfac", 1), R);
  @boswars(waitUnit("unit-vfac"), R); }

^script(game) <-
  unit("unit-gturret",X) and X < 1 | {
  ^visit();
  ^log("unit-gturret", 1);
  @boswars(setUnit("unit-gturret", 1), R); }

^script(game) <-
  unit("unit-gturret",X) and X < 2 | {
  ^visit();
  ^log("unit-gturret", 2);
  @boswars(setUnit("unit-gturret", 2), R); }

^script(game) <-
  unit("unit-camp",X) and X > 0 | {
  ^visit();
  ^log("unit-assault", 2);
  @boswars(defineForce(0, "unit-assault,2;"), R);
  @boswars(waitForce(0), R);
  @boswars(attackWithForce(0), R); }

^script(game) <-
  unit("unit-camp",X) and X > 0 | {
  ^visit();
  ^log("unit-assault", 4);
  @boswars(defineForce(7, "unit-assault,4;"), R);
  @boswars(waitForce(7), R);
  @boswars(attackWithForce(7), R); }

^script(game) <-
  unit("unit-camp",X) and X > 0 | {
  ^visit();
  ^log("unit-assault", 8);
  @boswars(defineForce(1, "unit-assault,8;"), R);
  @boswars(waitForce(1), R);
  @boswars(attackWithForce(1), R); }

^script(game) <-
  unit("unit-camp",X) and X > 0 | {
  ^visit();
  ^log("unit-assault", 12);
  @boswars(defineForce(2, "unit-assault,12;"), R);
  @boswars(waitForce(2), R);
  @boswars(attackWithForce(2), R); }

^script(game) <-
  unit("unit-camp",X) and X > 0 | {
  ^visit();
  ^log("unit-assault", 20);
  @boswars(defineForce(6, "unit-assault,20;"), R);
  @boswars(waitForce(6), R);
  @boswars(attackWithForce(6), R); }

^script(game) <-
  unit("unit-vfac",X) and X > 0 | {
  ^visit();
  ^log("unit-tank", 1);
  @boswars(defineForce(3, "unit-tank,1;"), R);
  @boswars(waitForce(3), R);
  @boswars(attackWithForce(3), R); }

^script(game) <-
  unit("unit-vfac",X) and X > 0 | {
  ^visit();
  ^log("unit-rtank;unit-tank", 1);
  @boswars(defineForce(9, "unit-rtank,1;unit-tank,1;"), R);
  @boswars(waitForce(9), R);
  @boswars(attackWithForce(9), R); }

^script(game) <-
  unit("unit-vfac",X) and X > 0 | {
```

```
  ^visit();
  ^log("unit-rtank", 4);
  @boswars(defineForce(8, "unit-rtank,4;"), R);
  @boswars(waitForce(8), R);
  @boswars(attackWithForce(8), R); }

^script(game) <-
  unit("unit-vfac",X) and X > 0 | {
  ^visit();
  ^log("unit-tank", 4);
  @boswars(defineForce(4, "unit-tank,4;"), R);
  @boswars(waitForce(4), R);
  @boswars(attackWithForce(4), R); }

^script(game) <-
  unit("unit-vfac",X) and X > 0 | {
  ^visit();
  ^log("unit-tank", 8);
  @boswars(defineForce(5, "unit-tank,8;"), R);
  @boswars(waitForce(5), R);
  @boswars(attackWithForce(5), R); }

^script(game) <-
  true | {
  ^visit();
  @boswars(waitOneSecond(), R);
  ^log("empty action"); }

queueEmpty() <-
  true | {
  script(game); }

updateOwnUnit(UnitType, UnitCount) <-
  true | {
  UpdateOwnUnit(UnitType, UnitCount); }

updateGameResult(Result, Fitness) <-
  true | {
  ^reward(Fitness);
  ^endEpisode();
  ^log("End game fitness:", Fitness);
  @boswars(gameResultHandled(), R);
  dropGoal(win(game)); }
```

Appendix III. Final result graphs with data tables



	all static	all switching	average
Dynamic Scripting	27,80	65,06	46,43
MC ε-Greedy	101,35	100,87	101,11
MC hierachy	87,80	68,70	78,25
MC Softmax	35,71	45,67	40,69
MC-DS Hybrid	25,94	40,89	33,42
MC opponent data	27,55	30,00	28,78
MC-DS Hybrid opponent data	35,55	35,66	35,60
MC-DS Hybrid XR	25,58	45,47	35,53



	all static	all switching	average
Dynamic Scripting	15%	32%	23%
MC ε-Greedy	30%	48%	39%
MC hierachy	35%	28%	31%
MC Softmax	17%	18%	17%
MC-DS Hybrid	10%	14%	17%
MC opponent data	15%	10%	13%
MC-DS Hybrid opponent data	13%	6%	10%
MC-DS Hybrid XR	9%	10%	9%

Figure 24: Average Turning Points and Fail Rates for all agents against all static and switching opponents