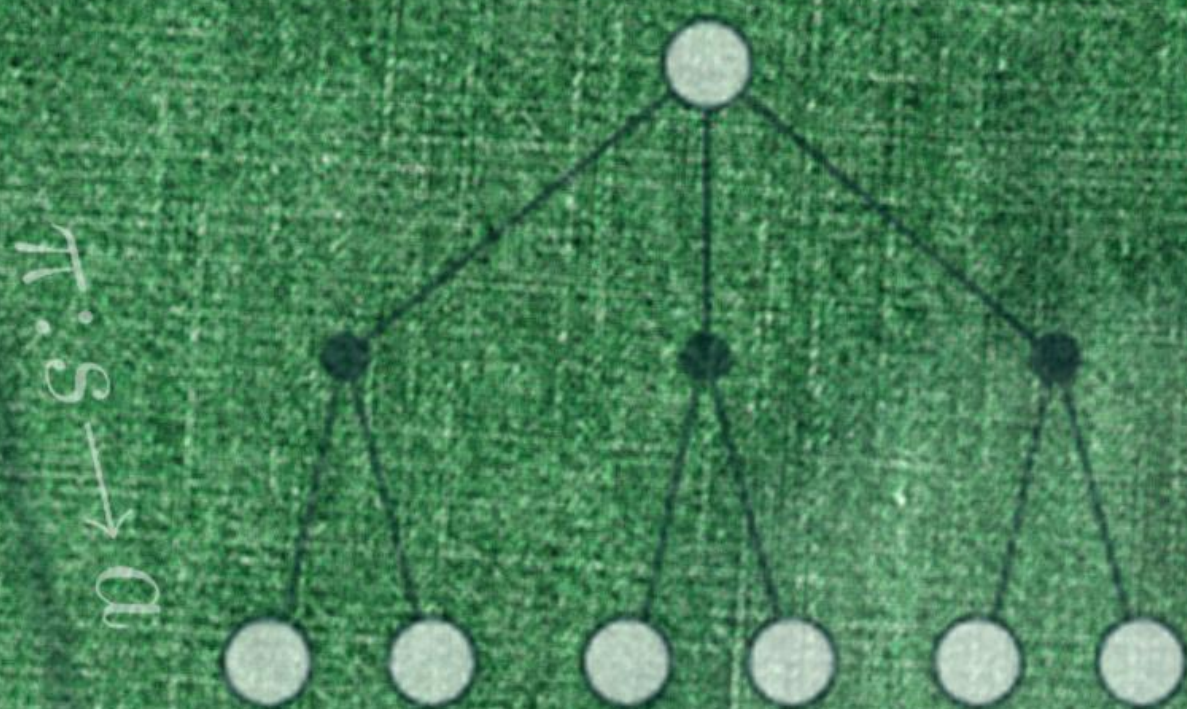


REINFORCEMENT LEARNING IN RTS GAMES

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$



$$Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

A MASTER THESIS
BY 0633A

Title: Reinforcement Learning in RTS games

Theme: Reinforcement Learning

Semester: DAT6

Project group: d633a

Date: June 12th 2008

Synopsis:

Group members:

Kresten Toftgaard Andersen

Anders Buch

Dennis Dahl Christensen

Dung Tran

Advisor:
Yifeng Zeng

Number printed: 6
Number of pages: 76 + appendices

In this master thesis we want to apply Reinforcement Learning, which is a well known academic Machine Learning method, to a Real Time Strategy (RTS) game. Our goal is to investigate the feasibility of monitoring human opponents' strategy in an RTS game to counter it by predesigned rewards and policies, where policies are updated using Reinforcement Learning.

We propose three extensions of the Reinforcement Learning structure to improve the use of RL in RTS games. We propose a simple and effective multi layered structure, which together with our player modelling technique called profiler, makes it easy to swap the current set of rewards and policy used by the Reinforcement Learning AI. We also propose a modification to the update function of RL algorithms which we call backtracking, which updates several steps back, instead of one step as TD(0) algorithms do.

These proposed extensions constitute a framework, which can be applied to any RTS game. To verify that our proposals work in a practical setting, we have applied our framework to the RTS game Tank General, which mainly was developed in the preparation semester.

While the backtracking does not give a performance boost, the multi layered structure and the profiler performs very well in an RTS game. Together they reduce the state space drastically, while reducing the time it takes for the RL algorithms to converge.

Therefore the conclusion of this thesis is that it definitely is feasible to use our proposed framework in an RTS game.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Content	6
1.3	Project goal	6
2	Related Work	9
2.1	Introduction to Reinforcement Learning	9
2.2	AI Techniques In Games	15
2.3	Reinforcement Learning Limitations and Improvement Techniques . . .	19
2.4	Reinforcement Learning Articles	24
2.5	Summary	27
3	Framework	29
3.1	Structural Overview	29
3.2	Multi Layer	31
3.3	Player Modeling	33
3.4	Backtracking	34
3.5	Summary	36
4	Case study	39
4.1	Tank General Analysis	39
4.2	Multi Layer	41
4.3	Player Modeling	44

4.4	Backtracking	47
4.5	Standard Single Layered Reinforcement Learning	48
4.6	General Application	49
4.7	Summary	54
5	Results	57
5.1	Test Setup	57
5.2	Test 1: SARSA versus Q-Learning	60
5.3	Test 2: Profiling versus Non-profiling	62
5.4	Test 3: Single Layered versus Multi Layered	64
5.5	Test 4: Backtracking versus Non-backtracking	65
5.6	Test 5: Framework versus Standard	68
5.7	Examples of Learning	69
5.8	Summary	70
6	Conclusion	73
7	Appendices	77
A	Example of a policy file	79
B	AI Scripting Framework	81
B.1	Consistency	81
B.2	Anti-cheat	83
B.3	Fairness	83
B.4	Test AI's	84
C	Reinforcement Learning	87
C.1	Action-Value Method	87
C.2	Solution Methods	90
D	The Game	95

CONTENTS

D.1 Game Description	95
D.2 Game Manual	99
D.3 Hot Keys	103
E Reinforcement Learning Attributes in Tank General	105
F Tests	109
G Screen Shots	117

Chapter 1

INTRODUCTION

This report documents our research and results during our master thesis in which we propose a framework for successfully implementing Reinforcement Learning in a Real Time Strategy game. Even though the content of this report is based on the work from our preparation semester [3] this report still stand alone and can be read without reading the previous report.

In the following section we will describe the motivation for our master thesis and afterwards present our goal which we investigate and have sought to achieve.

1.1 MOTIVATION

Traditionally there has been a significant gap between game developers and researchers in the field of Machine Learning [16]. Beside classical board games like chess, backgammon etc. researchers have rarely used computer games for their research of sophisticated Machine Learning methods. Meanwhile game developers have not paid much attention to advanced Machine Learning techniques, they instead have focused on simple and fast Artificial Intelligence (AI) methods like scripting and path planning. Traditionally game developers have prioritized other areas such as beautiful graphics above intelligent computer adversaries.

But looking at the trend of the recent years this gap between the research community and the fast developing computer game business seems to be narrowing [16]. The processing power available in modern CPU's increases in line with Moore's law and the new generation of advanced graphics cards have released significant processing power for other tasks like AI. In the following years, advanced Machine Learning techniques will be considered just as important as graphics, gameplay and other areas. AI is no longer relegated to the backwater of the schedule, something to be done by a part-time intern over the summer [30].

In this master thesis we want to apply Reinforcement Learning, which is a well known academic Machine Learning method, to a Real Time Strategy (RTS) game. During our preparation semester we developed our own RTS game called "Tank General" and examined several different Machine Learning techniques in order to find out, which

techniques were suitable for our game. "Tank General - Reinforcement Learning in an RTS game" [3] documents this work and we concluded that Reinforcement Learning indeed is suitable for our game. We also concluded that in order to achieve the best results using the Reinforcement Learning methods, some changes in the architecture are needed.

Even though our conclusion from our previous report [3] states Reinforcement Learning is suitable for this project, there were also other reasons and motivation factors, which influenced our decision. We are quite familiar with the Reinforcement Learning technique since we also used it in our previous project "Murder Of Crows - Emergence and Learning" [2] and are fascinated by its ability to learn from experience without any supervision.

Requirements such as small but information-rich representation of the state space and occasionally random actions might look like big disadvantages for using Reinforcement Learning in games, but still it can be an extremely powerful technique [5]. We want to take on the challenge and contribute with a framework for successfully applying Reinforcement Learning into an RTS game.

1.2 CONTENT

In this master thesis we first present and examine what we refer to as the "standard" architecture of Reinforcement Learning. Then we investigate previously work regarding Reinforcement Learning in computer games and examines the limitations of Reinforcement Learning in order to find out how to improve it for RTS games (Chapter 2). Based on this knowledge we are ready to propose a new framework which includes a new architecture and ideas of how to improve the update function (Chapter 3). Having presented our contribution we are ready to show how to apply it in an actual game (Chapter 4). Then we perform a comprehensive amount of tests in order to find out how our framework and other ideas perform, compared to the "standard" Reinforcement Learning architecture and other simple and common solutions such as scripting (Chapter 5). Finally we conclude our results (Chapter 6).

It is our intentions and hope that showing how to successfully implement and use an advanced Machine Learning technique in an RTS game, can help bridge the gap between AI game developers and Machine Learning researchers.

First we define the goal of this project.

1.3 PROJECT GOAL

In this master thesis we want to carry out an:

"Investigation of the *feasibility* of *monitoring* human opponents' strategy in an RTS game to *counter* it by *predefined rewards and policies*, where policies are updated using Reinforcement Learning."

1.3. Project goal

The key elements of this project proposal will now be described in further details. Several predefined policies are learned in advance, which corresponds to certain strategies of the human opponent. These policies are not static in nature but are continuously updated using Reinforcement Learning. Each policy is paired with a set of rewards, which are used to skew the specific policy towards a desired strategy.

During the game, the AI is monitoring the human player in order to categorize his current strategy and select the corresponding pair of counter policy and set of rewards. The substitution of predefined pairs of policies and rewards makes it possible for the AI to adapt to the strategy of the human player.

We define feasibility as whether the convergence rate of the Reinforcement Learning is sufficient to be able to learn a good strategy within a set number of games, or if an introduction of techniques to alleviate the potentially poor learning speed is necessary. This will be measured by comparing the convergence rate of different implementations of Reinforcement Learning against simpler AI's, and also by letting the different implementations play against each other.

Chapter 2

RELATED WORK

In this chapter we will look into the applications and limitations of the use of Reinforcement Learning (RL) and other Machine Learning techniques in computer games.

First we will give a short introduction to Reinforcement Learning and mention the solution methods we are using in this report.

Then we will describe how other advanced AI techniques could and have been used in existing computer games.

In our previously written report [3], which can be found on the attached CD-ROM, we have described approaches to use RL in common computer game genres, which we will briefly summarize to give an overview of the uses of RL in computer games.

We will then look at some general limitations of Reinforcement Learning. Since some of these issues can prevent the use of RL as a learning algorithm in applications, we will also look at different articles regarding some specific limitations and how to overcome these.

Finally we will look at some previously written articles regarding the use of Reinforcement Learning in various applications. These articles are all fairly new, and we will discuss the relevance in terms of general use of RL, innovation in the use of RL, and correlation to our work in this project.

2.1 INTRODUCTION TO REINFORCEMENT LEARNING

In this section we will investigate the learning approach called Reinforcement Learning. RL is a promising new approach to the learning algorithms, as it is easy to design and implement. The hard part is to adjust the algorithm to fit the game, but this will be discussed in detail later. This section is written using [25] and is a short version of the chapter written in our previous report [3]. In appendix C the issues of action-value method and solution methods are examined in details.

2.1.1 INTRODUCTION

Reinforcement Learning is a learn-by-doing, or trial-and-error, algorithm. It connects states and actions to maximize a numeric reward. The algorithm does not know in advance which actions are good to take, but have to explore which actions lead to the greatest rewards by trying out all available actions in a given state. Often the actions in a certain state will not yield an immediate reward, but a reward at some point in the future.

These two characteristics, trial-and-error and delayed rewards, are the most important properties of Reinforcement Learning.

RL separates itself from learning methods like Supervised Learning [27], which makes decisions based on previously experienced states provided by an intelligent and external supervisor. Supervised Learning does have advantages over RL, especially when data from previous experiments are available, but since this is not always the case, an agent need to be able to learn from its own experiences and not others.

A challenge when using RL is to balance exploration and exploitation. To maximize the reward in a given state the agent need to choose actions previously found to yield a high reward. But in order to know the rewards of the actions, the agent need to try out the previously untried actions. The agent need to exploit what it already know, but it should also try out untried actions. The challenge consist of balancing exploration and exploitation, since the exclusion of either one would shut down the concept of RL, which relies on trying out new actions and preferring big rewards. This property of RL will be discussed in further detail later.

Another important aspect of RL is the ability to consider the entire problem at once, unlike other learning approaches which considers parts of the problem and not the big picture of which they are used. This problem can be solved with planning or generalization of goals, but is very deterministic and rigid in structure. Their focus on the solution of parts of the problem might prohibit them from maximizing the final goal.

RL is directly opposite, since it considers the entire goal and how to reach it. All RL agents have precise goals, detects part of their environment and choose the actions which affects their surroundings, making them act well in unknown and changing environments.

2.1.2 ELEMENTS OF REINFORCEMENT LEARNING

In a Reinforcement Learning system four important elements exist: a *policy*, a *reward function*, a *value function* and a *model of the environment*.

A *policy* defines how an agent should behave at a specific time. A policy is a mapping-function between the state of the environment and an action, which is taken in the specific state.

The policy is written: $\pi : s \rightarrow a$, where π is the policy, S is all possible states of the surroundings, s is a specific state where $s \in S$, A is all actions, and a is a specific action, where $a \in A$.

Sometimes the policy is a simple function, and sometimes it involves a lot of calculations

2.1. Introduction to Reinforcement Learning

and searches. The policy is the central part of an agent, since the policy decides the behavior of the agent almost entirely.

A *reward function* defines the goal of the RL problem. The reward function is a mapping-function between the state of the environment and a numeric (reward) indication of how good a specific state is. This is written: $R : s \rightarrow r$, where R is the reward function, s is a specific state being mapped into a specific reward r . The only purpose of an RL agent is to maximize the total amount of reward received during an entire run/game. The reward function defines which actions are good or bad for an agent to take in a specific state. A reward function can not be changed by an agent, but can be used to change the policy of an agent. When an action chosen by a policy yields a low reward, the policy must change so the low-yielding action is not chosen so often in the future.

Whereas a reward function evaluates which action is good in a specific state, a *value function* decides which action is best when considering the future states and their rewards. A value function provides the total amount of reward, an agent can expect in the future when being in a specific state. This can be written: $U : s \rightarrow R$, where U is the value function, s is a specific state being mapped into a total amount of reward being R .

An action might yield a big reward at some point in the future, but a low reward immediately, and opposite. The value function is used to ensure that the future big reward is taken into consideration when a decision about which action to take is made. This makes the value function a very important part of an agent, since a good value function will make sure that the best decisions are made and best actions chosen.

The final element of an RL system is a *model of the environment*. The model is used to compare the behavior of the environment, and can, when provided with a certain state and action, predict the future state and corresponding reward. Models are used for planning, helping the agent make a decision regarding which action to take by considering the possible future states of the surroundings, before they occur.

The use of models of the environment is a relatively new subject regarding RL, since the earliest RL only used trial-and-error learning, which is the opposite of planning. Modern RL systems use trial-and-error to learn a model of the environment, and then use this model for planning.

2.1.3 SOLUTION METHODS

In this section we will examine the most used solution method to the RL problem. The older and more cumbersome solution methods like Dynamic Programming (DP) and Monte Carlo (MC) will be described in section C.2 in Appendix C. We will here look at Temporal Difference and list its functionality, strengths and weaknesses.

TEMPORAL-DIFFERENCE LEARNING

Temporal-Difference (TD) learning is regarded the most central idea to RL. TD learning is a combination of MC and DP methods. TD learn from experience like MC, but bootstrap like DP.

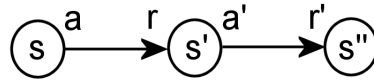


Figure 2.1: Episode consisting of states, actions and rewards

When calculating the value of a state, TD does not wait until the end of an episode to update, like MC, but updates the value, each time a nonterminal state is reached. The simplest TD method, called TD(0), is: $V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$. The α value is a step-size parameter like describe on page 87 in section C.1, and γ is a discount value used to downplay the value of the coming state. This update is of course only available after the value of $V(s_t)$ is observed. The following figure shows TD(0) in a completely procedural form:

```

Initialize for all  $s \in S$  and  $a \in A$  the value  $V(s)$  to a constant value,
and  $\pi$  the policy to be evaluated
Repeat for each episode:
    Initialize  $s$ 
    Repeat for each step in the episode, until  $s$  is terminal:
        Choose  $a$  for  $s$  using  $\pi$ 
        Execute action  $a$  and save reward  $r$  and next state  $s'$ 
         $V(s) = V(s) + \alpha[r + \gamma V(s') - V(s)]$ 
         $s = s'$ 

```

One issue to note is that TD and MC use sample-backups instead of full-backups like DP. Instead of tracing the reward all the way back to the start like DP, TD only goes back one step to update that value. This is not an disadvantage, since it reduces the number of calculations significantly, and eventually the value will be traced all the way back.

When using TD in practical applications, there are two primary algorithms to solve this problem: SARSA and Q-Learning. To improve the performance of TD, we will learn the value of action-values instead of state-values. This will make it easier to choose the best action in a specific state.

SARSA

In SARSA we use the state s , the best action a for state s , the reward r for the following state s' , and the best action a' for state s' , to calculate the action-value for s, a , called $Q(s, a)$. This is easier understood when looking at figure 2.1 showing an episode consisting of states, values and rewards. The quintuple (s, a, r, s', a') has given SARSA its name. The algorithm for SARSA is shown here:

2.1. Introduction to Reinforcement Learning

```
Initialize for all  $s \in S$  and  $a \in A$  the value  $Q(s, a)$  to a constant value
Repeat for each episode:
    Initialize  $s$ 
    Choose  $a$  for  $s$  using  $\pi$ 
    Repeat for each step in the episode, until  $s$  is terminal:
        Execute action  $a$  and save reward  $r$  and next state  $s'$ 
        Choose  $a'$  for  $s'$  using  $\pi$ 
         $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$ 
         $s = s'$  and  $a = a'$ 
```

Compared to the simple TD(0), the biggest difference is that SARSA uses the action-value instead of state-value.

Q-Learning

Q-learning differs from SARSA by not using the action-value of the next state, but instead the value of the action with the highest value. Q-learning does not wait until a second action is chosen before updating the value of the chosen action. Here is the Q-learning algorithm:

```
Initialize for all  $s \in S$  and  $a \in A$  the value  $Q(s, a)$  to a constant value
Repeat for each episode:
    Initialize  $s$ 
    Repeat for each step in the episode, until  $s$  is terminal:
        Choose  $a$  for  $s$  using  $\pi$ 
        Execute action  $a$  and save reward  $r$  and next state  $s'$ 
         $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
         $s = s'$ 
```

Q-learning is shown to converge earlier than SARSA, because it utilizes a greedy method for determining the action-value instead of selecting a random action with probability ϵ .

We will now apply the Q-learning method in an example similar to the example used in section C.2.1 on page 91. Since the SARSA and Q-learning will converge to the same policy, we will show a few steps on the way to convergence for each of the two solution methods.

The example was extended to include action-values instead of state-values, since this is one of the elements of Q-learning and SARSA. The rules are altered so it is possible to take an action which leads "outside" the area, which will just return the "unit" to the previous tile. The example can also only terminate in the upper left corner, which is changed to make the example slightly more interesting. The discount and stepsize parameter are both set to 1 to produce integer values. The action selection method is 100% greedy, so there are no random actions, making Q-Learning and SARSA very similar.

For Q-learning the first step will be random, and could go in all directions. The first state is (15) and to find an action we roll a dice and the direction will be upwards, resulting in action UP. The new state is (11), and now we update the $Q((15), UP)$ value. The new value is:

$Q((15), UP) = 0 + 1 \cdot [-1 + 1 \cdot 0 - 0] = -1$. The $\max_{a'} Q((11), a')$ is 0, because all $Q(s, a)$

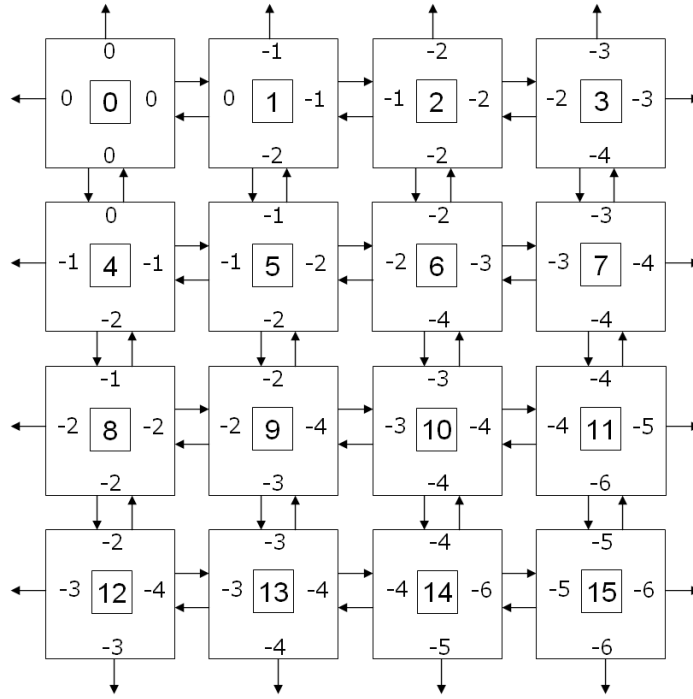


Figure 2.2: Converged values of Q-Learning.

for all action-values are still 0.

Now we select a new action, making it *DOWN* resulting in the new state (15). Now we update the value of $Q((11), \text{DOWN})$ like this:

$Q((11), \text{DOWN}) = 0 + 1 \cdot [-1 + 1 \cdot 0 - 0] = -1$. The $\max_{a'} Q((15), a')$ is calculated to 0, since three action-values are 0 and one -1.

For SARSA we assume the same trace of steps to show where the difference is situated. After the first state (15) and action *UP*, we end up in state 11 where we choose the action *DOWN*. First now we will update the value of $Q((15), \text{UP})$ as this:

$Q((15), \text{UP}) = 0 + 1 \cdot [-1 + 1 \cdot 0 - 0] = -1$. No calculation for the best action is needed, since it is already known.

The value for $Q((11), \text{DOWN})$ can first be updated when a new action in state 15 is selected. This clearly shows the biggest difference of Q-Learning and SARSA. Since the ϵ -value is 0, the traces will be similar for both methods, and should it change, only the SARSA method would differ slightly.

The converged example can be seen in figure 2.2.

2.1.4 SUMMARY

In this section, we have shortly described Reinforcement Learning. We found out that RL contains two important properties: trial-and-error and delayed reward.

We learned that RL consists of four parts:

2.2. AI Techniques In Games

- The *policy* which decides which action to take in a specific state.
- The *reward function* which gives a positive or negative reward when executing a specific action in a specific state.
- The *value function* used to find out which action yields the best reward over time.
- The *model of the environment* containing all relevant information regarding the current state.

We found out that RL uses a method called ϵ -greedy to choose the best action in a specific state while still exploring other actions to maximize the total amount of reward.

We examined TD-learning which uses experience and bootstraps, which makes it very useful for almost all RL problems. The two most important solution methods using TD are Q-Learning and SARSA.

With the theory of Reinforcement Learning established, the use of this learning technique in different game genres will now be examined, along with the use of other Machine Learning techniques in commercial games.

2.2 AI TECHNIQUES IN GAMES

The use of Machine Learning (ML) techniques for AI is a fairly new subject introduced in the computer game industry. In the past ML techniques have often been discarded because of their high computational use of the processor. In recent years the raw processing power of the home computers have increased by a large margin [15], providing more CPU power to the AI part of a game. Therefore it has become of great interest to fit the traditional Machine Learning techniques into suitable computer genres.

First we will lay out some potential and practical examples from the gaming industry, where other Machine Learning techniques have been utilized for reasons, which we will explain in details.

In our previous report we mentioned genres suitable for the use of Reinforcement Learning, and therefore we will shortly summarize these to give an overview of genres where RL could provide the game with a significant change the AI structure.

2.2.1 MACHINE LEARNING IN COMPUTER GAMES

Machine Learning techniques in their basic forms are often oversized and cumbersome compared to the size of the problems, most AI programmers have when creating games. Therefore it is often a requirement that the game contains a certain degree of complexity before such ML techniques are suitable for use. One example of this could be to apply almost any ML technique into an arcade game, which are often so simple that the benefit compared to the cost would not be enough for a gaming company to spend time or money on such.

But this is not to say that games in general can not benefit from ML. Several games have incorporated ML techniques to some or full extent, and we will display some of

these along with their potential uses. Although BDI (Belief-Desire-Intention) and Fuzzy Logic are not per se Machine Learning techniques, since they do not have a learning element built into their structure, they can be expanded to incorporate learning, and along with their popularity among AI programmers, we have chosen to include them anyway.

For a full description of the ML techniques, see [3] for more information.

BDI

The BDI technique, which is nothing else but more expressive *if-then-else* sentences, can be used in almost any kind of game genre. To make it reasonable to implement BDI instead of a scripted AI, some complexity of the game and AI is required, but the required bar is not very difficult to reach. Most computer controlled opponents, being football players, tanks, emperors or soldiers, could use BDI to make better and more clear decisions based on a broader variety of environmental variables.

Black & White [24] used BDI for the large creatures [22], which are controlled and influenced by the actions of the player. BDI is used to determine whether certain thresholds have been exceeded and thereby expressing the desire of the creature. The creature then reviews its desire and choose actions (intentions) to satisfy these desires. Black & White used a simple feedback learning technique in order for the creature to learn which actions were the most appropriate to take. This mimics the foundation of Reinforcement Learning a great deal, although more simple and rigid in structure.

In Black & White the use of learning techniques was viewed upon with great excitement in the gaming community, but was let down by the extremities of the learning. Players often found that when the creature was punished or rewarded for something, it would disregard all else or never do this action again, making it hard to teach fine-grained tasks to the creature.

FUZZY LOGIC

Fuzzy Logic offers a broader variety of interpreting the environmental variables of a game, simplifying a lot of information into more understandable knowledge. Again the environment must contain a sufficient number of important variables for this technique to be suitable and worth the implementation time. In most strategy games, the use of Fuzzy Logic could help out the commander of a group of units, cities, or a country by making it easier to handle a great amount of information and output more usable data. Civilization: Call to Power used Fuzzy Logic [31] for the strategic decisions made by the AI-controlled players in the game. If an AI would be in doubt regarding which strategic route to take, it would rely on the set of Fuzzy State Machines to guide it toward a sensible goal. If more conflicting strategies were selected, the AI would again rely on Fuzzy Logic to evaluate the best strategy at the present time.

Call to Power was the first strategy game of the Civilization series [20] which incorporated Fuzzy Logic, also making it possible for the player to look at some of the computed values the AI use for decision making. The use of Fuzzy Logic resulted in a more detailed and challenging AI, making the individual AI players behave more intelligent and sensible than ever before seen in the Civilization series.

2.2. AI Techniques In Games

NEURAL NETWORKS

Neural Networks (NN) provides a complex model which is able to adapt to a changing environment and thereby learn better strategies. The NN would flourish in the real-time strategy and turn-based strategy genres, where adaptable computer opponents, basing decisions upon a large quantity of parameters, should provide all kinds of players with a suitable challenge, matching their way to playing the game and their level of expertise.

Creatures, which included a lot of different ML techniques, used Neural Networks as part of the learning elements of the game [32]. The use of NN served two purposes. First it allowed for the individual creature to learn preferable abilities and actions while living, and secondly it allowed for knowledge to be parsed down to its descendants. Creatures, although not as much a game as an ant farm, is a good example of the use of NN for intelligent behavior.

GENETIC ALGORITHMS

Genetic Algorithms (GA) can be harder to implement in a game, since the basic genetic code can be hard to "find" in a game. If the game contains some kind of MDP¹, the genetic string could consist of state-action pairs, which after a game would be evaluated. This type of implementation would be useful in a first person shooter or football manager game.

Although hard to find actual use of Genetic Algorithms in commercial games, this does not mean that they are not used. GA have e.g. been used to create intelligent behaving bots in the First Person Shooter Unreal [13]. Although not processed into an actual game, the use of GA in games is undisputable. The biggest reason for our lack of references to commercial games using GA, might be the common practice of the technique.

Although rare and hard to come by, examples of the use of ML in commercial games do exist, and with the increasing CPU power we will definitely see more and more games incorporate these or other techniques in some or full extent.

One may ask why Reinforcement Learning has not been mentioned in the above list of ML techniques in commercial games, and the reason for this is quite simple: there are none. RL as a technique is not as old as other ML techniques and can be somewhat more difficult to implement. This is the reason for looking at some game genres and find out what the use of RL could bring to those.

2.2.2 REINFORCEMENT LEARNING IN GAME GENRES

A lot of game developers use RL and other complex learning algorithms during the development of their game [17]. More specifically during the test phase or somewhere around this phase, since the learning algorithms need to have as much information as possible to be able to learn the most. When the learning algorithm has converged toward one or more usable strategies, these strategies will be hardcoded into scripts for

¹http://en.wikipedia.org/wiki/Markov_decision_process

the AI to read before the shipping of the game.

Next to none of the game developers use RL during the actual game, i.e. post-shipping, because it tends to require a lot of CPU power and because it puts the AI out of the control of the programmers. Since the first issue is becoming less important in the future, the second issue of loss of control is very valid and requires a shift of paradigm in the minds of programmers and developers. Usually game AI can not take unexpected paths toward the goal, because it is not designed for that. This makes the AI rigid in structure, and to let go of this control is difficult, but might inevitably be a necessary step toward better game AI.

We will now take a look at some of the most important genres of the computer game industry at the moment in which RL could be used. A more elaborate discussion can be found in [3].

GENRE EXAMPLES

Not all genres of computer games might be suited for the use of RL. It depends a lot on the complexity of the game, the number of different entities needing an AI, the room for computation of RL, and a lot of other issues. We will now discuss the currently most dominating genres in the industry and leave out some of the simple genres like sidescrollers, puzzle games, etc.

In a conventional First Person Shooter (FPS) game vast amounts of adversaries must be shot down or neutralized by different means. This per default makes RL unattractive to use for the AI of these enemies, because of their short lifespan and inability to pass on knowledge to each other.

One counter example would be the very successful game *Counter-Strike* [26] featuring human players fighting against AI controlled bots or other human players. So in a FPS like Counter-Strike, RL would fit nicely.

The RTS games is often split into a campaign and a skirmish part. In the campaign part you are playing against an AI with a lower or greater amount of units/buildings than yourself which you are to overcome. In a skirmish game both sides start on equal terms, and therefore the strategy to beat the AI depends greatly on the skill of the player unlike the campaign part where it depends on the setup of the level.

Using RL on the campaign part of a RTS game seems like overkill, because it is essentially not needed, because the AI can be hardcoded without being dull or easy for the player to beat. The skirmish part on the other hand needs to have an AI which will provide a challenge. Here RL would fit perfectly, since the policy could be primed again for certain types of behavior, while still improving after a single skirmish game is finished.

Turn Based Strategy (TBS) games are divided into turns, during which only one player/opponent is allowed to make decisions regarding their play. In this kind of game it is easy to determine the state of the world, but hard to categorize it into a reasonable amount of states for the policy to use. One big advantage in TBS is the time available to create a good policy and executing it.

Because of the structure of TBS it gives the RL opportunity to have reasonable time to make policy updating calculations because the RL does not have to update during the turn of the player. This however must still be fast, since the player of a TBS will

2.3. Reinforcement Learning Limitations and Improvement Techniques

not be pleased with waiting for too long, before he can take another turn.

The use of RL depends greatly on the genre of a game. This is also why we chose to make an RTS game for our implementation of RL. We could have chosen any of the above mentioned genres, but the RTS seemed like a good choice for us.

Now that we have looked on how different Machine Learning techniques are and can be used in different game genres, we will now look at some of the reasons which are preventing the use of RL in games, and how to relieve some of these.

2.3 REINFORCEMENT LEARNING LIMITATIONS AND IMPROVEMENT TECHNIQUES

Now we will look at the limitations by the use of Reinforcement Learning in general. There are always pros and cons to all Machine Learning techniques, and since we have chosen to focus on Reinforcement Learning, it would be wise of us to examine its weaknesses.

When we have looked at limitations, we will present some articles which will relieve some specific constraints. These articles were discovered subsequently of our project proposal, but they still represent the foundation of the proposal.

2.3.1 REINFORCEMENT LEARNING LIMITATIONS

When dealing with Machine Learning techniques, one must balance the strengths and weaknesses to choose the most appropriate ML technique for a game. Reinforcement Learning is, although among ML techniques still new, well examined and scrutinized and therefore it is relatively easy to list some of the issues, one encounter when implementing RL. This list is not exhaustive, but should provide a fairly thorough survey of RL. The list is a shortened version of a list found in [29].

TEMPORAL CREDIT ASSIGNMENT PROBLEM

This problem is caused by having very large state-action spaces, where a significant number of state changes can occur, before a reward is given. This problem can be illustrated in a robot example, where a robot would have to perform many moves with rewards close to zero with a rather distant rewards set in the future. The result of this will have very weak influence on all the temporal states preceding it.

This will require several games with similar episodes of state-action pairs to bootstrap these rewards. This of course only holds true for the TD algorithms, as DP bootstraps all steps and MC does not bootstrap at all. But the use of DP and MC is computational very questionable in most choices of solution method, so the problem, though TD specific, is still very relevant.

CURSE OF DIMENSIONALITY

In general it is very easy to define the state-action spaces in an RL implementation, but for it to be practical and even possible to implement, one must use function discretization to reduce the number of spaces. The naïve state-action space contains all possible states being possible for the AI to use, but this would result in a vast and not implementable number. Instead it is most reasonable only to include the states needed to describe the relevant situations.

To bring down the number of states, a method called function discretization is used. Although not a method or even a function, this technique is used to combine states, e.g. by using a grid instead of coordinates or using intervals instead of precise integers. The number of actions can also be brought down by combining similar actions into a single action and then let this general action make the call to select the precise action to carry out.

EXPLORATION-EXPLOITATION DILEMMA

The use of RL requires agents to explore their environment in order to build up the policy table containing rewarding decisions, which then can be exploited. The difficult task is to decide exactly when the optimal actions have been found and then must be exploited. If the length of the task is known in advance, parameters concerning the ϵ -factor can be used to downgrade the importance of the random actions over time. Other parameters like waypoints during a task can also be used, assuming that the task is fundamentally linear.

This RL issue resides in all implementations of RL and must be considered in every individual use, because of the fluctuating impact it has on the learning task.

NON-STATIONARY ENVIRONMENTS

Like many other ML techniques, RL will only perform at best in approximated stationary environments, where the dynamics does not change or change very slowly. This is a fundamental problem and is very hard to circumvent. If the environment changes too fast, RL does not have the time to learn a task. Even in slowly changing environments RL methods can fail to learn if their state space is too big.

INCOMPLETE STATE INFORMATION

In a real world scenario an RL agent can often end up in a state which it could not identify unambiguously, and therefore the ability to take an optimal action is impaired. Basic RL methods can not be used in these situations, since they require full observability and history independent states. Often Partial Observable Markov Decision Process (POMDP) methods are used to solve the problem [18], and many solutions have been designed for this.

INCORPORATING PRIOR KNOWLEDGE

To use knowledge, which is known in advance and which makes sense for the RL to prelearn, is very hard to put into a policy, because of the large number of states and actions. Not to say that this is impossible, but one could devise some algorithm to find the state-action pairs and put in a reasonable amount of rewards to simulate the prelearned knowledge. If a smaller state-action space is used, this can even be done by hand.

Another approach is to make the RL play its first games in an environment which would convey this information to the policy. This solution is more in style with the ML mindset, but it is still not a convenient method.

Now that we have examined some of the limitations and problems with the use of RL, we will look at two common and very powerful techniques used to alleviate some of these issues.

2.3.2 REINFORCEMENT LEARNING IMPROVEMENT TECHNIQUES

There are many techniques to improve the performance of RL, and we have selected two techniques and will focus on these. They are both fairly easy to understand and implement, which makes them perfect candidates to use in our game.

ELIGIBILITY TRACES

This technique deals with the issue of *Temporal Credit Assignment Problem* which describes how rewards received after a long period of state changes are hard to backtrack. This chapter is based on chapter 7 in [25].

This technique provides every state or state-action pair with an extra parameter called *eligibility* which describes how much a reward is allowed to influence the value of the state or state-action pair. The parameter is noted λ and is defined within the interval of 0 to 1, describing how much the rewards is backupt. If the λ is 0, then the algorithm will function like the TD(0) method, and if 1 then the method will act like Monte Carlo. λ is called the *trace-decay* parameter because it describes how much the reward decays back in steps.

Each state contains this extra λ parameter which must be updated after each state change. The value is updated like this:

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma \lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

In this update the value of $e_t(s)$, being the eligibility e of the state s at the time t , is decreased with the factor $\gamma \lambda$, unless the state chosen s is the current state s_t , at which it is still decreased with the factor $\gamma \lambda$ but incremented with the value 1. The $t - 1$ part is indicating that the old value is used, since the use of t would make the algorithm recursive which it is not.

This is done for all $s \in S$, where γ is the discount factor and λ is the *trace-decay* parameter. This kind of trace is called *accumulating* trace because of the +1 part of the equation. Another trace is the *replacing* trace which, instead of adding 1, replaces

the previous value with 1. The question of which kind of trace to use depends deeply on the problem.

The complete algorithm for use of eligibility traces in an online TD(λ) method is shown here:

```

Initialize  $V(s)$  arbitrarily and  $e(s) = 0$ , for all  $s \in S$ 
Repeat for each episode:
  Initialize  $s$ 
  Repeat for each step of the episode:
     $a \leftarrow$  action given by  $\pi$  for  $s$ 
    Take action  $a$ , observe reward  $r$  and next state  $s'$ 
     $\delta \leftarrow r + \gamma V(s') - V(s)$ 
     $e(s) \leftarrow e(s) + 1$ 
    For all  $s$ :
       $V(s) \leftarrow V(s) + \alpha \delta e(s)$ 
       $e(s) \leftarrow \gamma \lambda e(s)$ 
     $s \leftarrow s'$ 
  until  $s$  is terminal

```

The δ parameter is used to calculate a TD-error, which indicates the difference between the current state and the future state added the rewards given between the states.

Consider what happens with various values of λ . If $\lambda = 0$, all traces are zero except the visited one, which reduces the algorithm to a simple TD(0) update. If $\lambda = 1$ along with $\gamma = 1$, then the algorithm resembles the basic structure of Monte Carlo very much. This version is called TD(1) and is often used to implement MC in a more approachable fashion.

To apply eligibility traces to the state-action world, small changes are needed. We will not show how to use this in Q-learning, since these update functions are too complicated and cumbersome to describe here. Instead we will show the SARSA(λ).

The obvious reason for applying eligibility traces to SARSA is the ability to learn action-values instead of state-values. The eligibility must now be updated in this fashion:

$$e_t(s, a) = \begin{cases} \gamma \lambda e_{t-1}(s, a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma \lambda e_{t-1}(s, a) & \text{otherwise} \end{cases}$$

This is done for all (s, a) . The complete algorithm for SARSA(λ) is shown here:

```

Initialize  $Q(s, a)$  arbitrarily and  $e(s, a) = 0$ , for all  $s, a$ 
Repeat for each episode:
  Initialize  $s, a$ 
  Repeat for each step of the episode:
    Take action  $a$ , observe reward  $r$  and next state  $s'$ 
    Choose  $a'$  from  $s'$  using policy derived from  $Q$  like  $\epsilon$ -greedy
     $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ 
     $e(s, a) \leftarrow e(s, a) + 1$ 
    For all  $s, a$ :
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ 
       $e(s, a) \leftarrow \gamma \lambda e(s, a)$ 
     $s \leftarrow s', a \leftarrow a'$ 
  until  $s$  is terminal

```

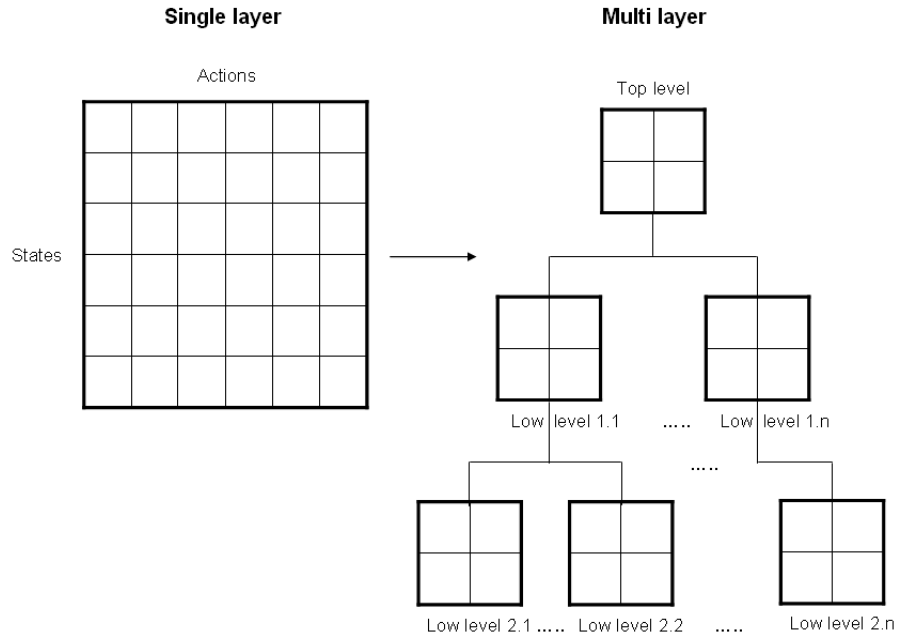



Figure 2.3: Structural transference from a single layer to multiple layers

This technique has been well proved and examined and can severely reduce the converge rate of the RL solution method. The computational need is larger, which can be observed fairly easy, but compared to the earlier converge, this eligibility trace update could be removed when an optimal solution has been reached. It is fairly easy to implement, if TD or SARSA is chosen for solution method, and should almost always be considered in this context.

HIERARCHICAL METHODS

A popular way to reduce the problems of the *Curse of Dimensionality* is to use a hierarchical method also called multi-level learning. This is a method which rely on decomposing a problem into smaller and more manageable parts. This section is written from the basis of [12]. The method involves multiple levels of decision making that together solves the whole problem. The task of designing this hierarchy is often left to the programmer himself, instead of being a common formula or technique, making it unlike many other limitation-solving methods. The issue of decomposing a problem into multiple layers through an algorithm is an area which currently have the attention of many researchers, since this would make the structural division significantly easier to execute.

The logical reason for splitting up a task into smaller tasks is not new in the world of programming, since the "Divide and Conquer" thought has been common practice for quite some time. Even in nature, the creation of multiple levels of decision making can be observed in ant hills, bee hives, lion dens, and other societies which involves the common goal of survival. Even in human societies the ability to delegate responsibility is seen as a common practice and necessity in governments, business life and social structure. So to employ this technique to Reinforcement Learning seems rational.

A common level structure is to have one top level and then several low levels, which themselves could have low levels as shown in figure 2.3. The different levels have different states and actions, making them independent in learning and convergence rate. It is possible for the lower levels to learn the optimal policy independent of the top level and other low level policies. The number of states in a single level is decreased, making the need for decisions lesser, since the states will not change as often. Different rewards can also be given at the different levels, making it easier to distinguish between different behaviors. It is also easier to hide unimportant or situational information for some levels, since the different levels only need the required information, and not include information which only applies to some specific states.

The number of actions can also be greater and more specific, since the complexity of the state space will not grow as explosively as before. An abstraction of action is now allowed, so the top level can give general orders which the low levels will act upon, either by including them as a state or just by selection.

The performance of this technique greatly depends on the AI designer or programmer, since it is his decision of division which makes the RL agent behave properly or not. If the designer makes too many low levels, the chain of command might be so long that the original order is lost or wrongly interpreted due to the growing chance of the ϵ parameter, and if too few low levels exist, then benefit of actually using a hierarchical structure is lost.

We will now examine some articles regarding the use of RL and other techniques in order to relate our work to existing work.

2.4 REINFORCEMENT LEARNING ARTICLES

In this section we will examine different articles, all with relevance regarding our project goal. We have four articles regarding Reinforcement Learning and one regarding Player Modeling. All of these articles are relatively new, and provide different approaches to old existing problems or issues regarding RL or Player Modeling. The articles can be found on the attached CD-ROM.

2.4.1 SUPERVISED LEARNING IN AN REINFORCEMENT LEARNING GAME

This article is concerned with two issues: the use of supervised learning in the process of Reinforcement Learning and the lack of previously learned data [7]. The article proposes the use of directed learning with the help from the player in order to reduce the number of episodes needed to converge toward an optimal solution, resulting in two different approaches. The first approach lets the RL make an extra episode after an episode, where it follows the advice given by the player during the first episode. since it was proposed by the player, the learning from this extra run is hopefully better. The second approach gives the RL an extra reward for choosing the action chosen by the player, making the RL agent prefer the player-chosen actions over all other actions. Both approaches can improve the RL to converge faster, given that the player provides sensible and optimal actions.

This technique is very hard to fit into a game like ours, since the AI in our game is

2.4. Reinforcement Learning Articles

playing against the human player instead of being controlled by it. The game would also contain a very long episode, making the recording of player actions and subsequent calculations complicated and cumbersome. The human given rewards in this game would also only function properly, if the decisions of the player were sound, which in our game are hard to determine before the end of the game.

All in all this article is of little use for our game, but in simpler games it could be used to improve the convergence speed.

2.4.2 CONCURRENT HIERARCHICAL REINFORCEMENT LEARNING

This article describes how to use threads to control a number of units which are carrying out different orders while working together toward a common goal [19]. In the small example the different units must learn how to collect resources as fast as possible to achieve a predefined amount to win the game. The rewards received are shared among all the units, which are selecting new actions constantly while updating their common action-values. A larger example is given where the goal is to defeat an enemy using the shortest amount of time, which is done by collecting resources, spending these on soldiers or gold collectors, and deciding to attack with a certain amount of units. The article does not provide any results of this example, but a brief look at the video reference in the article shows good results for this example.

The first example in this article is very crude, and one might ask what the reason for using RL for a pathfinding and anti-collision task might have been. If only to keep the example simple, the following example is much more relevant while still simple in structure. The first example uses the learning at a low level, while the next example uses it at a higher level, which is more similar to our project. A similar structure could have been exciting to try in our project, since it also involves highly abstracted actions. The structure and hierarchy in this article is very hard to imagine and understand, and compared to [12] the structure is nowhere similar, which makes it even more confusing when one example uses low-level actions while the following example uses high-level actions. The different uses of this proposal is changing from a unit to a commander point of view, making it span from not relevant to very relevant for our game.

The exact proposal of the article is difficult to understand and use, but the principles seems fine, and if examined at an earlier stage of the implementation of our game, could have been exciting to investigate.

2.4.3 REINFORCEMENT LEARNING IN GO

Using Reinforcement Learning in games is not an easy task, and this article highlights some issues that can be improved, if the game is viewed upon in some different light [23]. In the game of GO, one can look at different combinations of pieces instead of considering the entire board which reduces the number of states considerably and enhances the performance of the RL too. The authors go even further and combines the combinatory states with location independent states, reducing the state space even further and boosting the convergence rate even more.

The actual use of this article is hard to find, but one aspect to explore is the location independent states, which we are also using in our game. We have no indication of the positions of the headquarters, resources, or units in our game, similar to the game from

the article. This by all means also limits the complexity of the RL, since it does not have the capability of spotting potential choke points, which in advanced maps could determine the difference between a win and a loss. But you always have to balance the complexity of the state space, the calculation speed, and the actual gain of additional states.

The article is not frightfully interesting or relevant, but it still shows a nice design of RL in a game environment.

2.4.4 REINFORCEMENT LEARNING FOR BALANCING A GAME

This article is very much aimed at the gaming industry to convey a positive image of the use of Reinforcement Learning as technique for a different subject than traditional AI learning [4]. The article proposes the use of RL as a balancing technique, making the game adaptable to the skill of the player, without him feeling stupefied by his bad performance. The authors show in an example how this can be done by letting the RL agent choose suboptimal actions which matches the skill of the player instead of focusing on the sole purpose of improving its performance.

The article can be related to our project in the way that we are also using player modeling, but we are not using it to balance the game in favor of the player, but rather to be able to adapt to the strategy of the player and counter it. So where the article tries to make the game easier for the player by using suboptimal actions, we are offering the AI a variety of policies and rewards which it can use to optimize specific counter-strategies.

The article still raises an interesting proposal, but we do not find it directly relevant to our project.

2.4.5 PLAYER MODELING

Having some kind of player modeling in a computer game is very useful for the entertainment of the player [8]. This article highlights a very superficial structure to use in a computer game, when a need for player modeling is useful. The article suggests a game system which uses models of player types combined with player preferences to monitor the performance of the player, leading to an adaptation of the game to the individual player, which then are measured, evaluated and used to re-model the different player types.

This article suggests that having different player models to match the different kinds of players result in better player experiences with a game, since the player should feel challenged without being overwhelmed. Our game could utilize this to match different player strategies with certain counter-strategies which should be effective and provide the challenge, but without the ability to downscale the challenge proposed.

The article is very relevant to our game and games in general, when they exceed a certain amount of complexity.

These articles present different solutions to problems which could be encountered in a project like ours. The articles have provided different levels of inspiration to us and helped us move in a proper direction toward our project goal.

2.5 SUMMARY

In this chapter we have looked into the learning technique of Reinforcement Learning, and what it has to offer. Reinforcement Learning requires a certain amount of complexity of a problem for it to be reasonable to implement, and it can also be hard to balance regarding the values of different rewards. But the advantages of fairly low implementation complexity and high level of general use makes Reinforcement Learning interesting in a wide range of applications, especially considering the online solution methods Q-Learning and SARSA.

Next we looked at RL and other learning techniques and examined how they have and/or could have been used in different game genres. In simpler games the use of Fuzzy Logic and BDI is already widespread, because of the simplicity and low computational complexity. In more complex games Genetic Algorithms and Neural Networks have been used for decision making on a grand scale or solving of complex gaming elements. With a nice track record of the use of Machine Learning techniques in existing games, it was also nice to see the promising uses of RL in different gaming genres, where RTS and TBS are genres with the greatest use of this learning technique because of their high level of complexity.

To be able to give a proper review of RL from an implementational point of view, we looked at some of the limitations of RL, and found out that although a lot of these concerns time or space complexity, they are all manageable through certain kinds of techniques. Two promising techniques to reduce time and space complexity along with convergence rate were Eligibility Traces and Hierarchical methods. Eligibility Traces provides an extra layer which hold reward relevance information, speeding up the learning rate, and Hierarchical methods splits up the learning problem into smaller parts, reducing the space complexity by a large margin. Both techniques were used to found some of the proposed methods described in the following chapter.

Finally we investigated some existing articles which were relevant to our project in different ways, and discussed their relevance to our project. It was fairly hard to find articles which were directly correlated to our project, but a number of ideas were found and discussed. Most discussed games in a broad sense and some were hard to find a use for in a practical application. But the most interesting of the ideas provided a good foundation for us to work with in the following chapters.

Chapter 3

FRAMEWORK

This chapter describes the three extensions to RL that we propose in this project. These proposed techniques constitute a framework for extending known RL techniques. First we propose a way to divide standard RL into multiple hierarchical layers. Next we propose a player modeling method, called a *profiler* which should optimize the RL. Lastly we attempt to extend standard RL algorithms with an easy to implement extension, called *Backtracking*.

3.1 STRUCTURAL OVERVIEW

For the discussion in this section we define the following notation:

A given policy is denoted π , and the set of rewards used by an RL system is denoted \mathcal{R} .

A set containing a π and a \mathcal{R} is denoted π/\mathcal{R}

A typical reinforcement learning framework utilizes one fixed π/\mathcal{R} . Fixed in this context means that neither the π or \mathcal{R} is swapped with other π or \mathcal{R} . This means that all policy updates in every game session are performed in the same policy.

Having a single layered structure means that the policy file will contain the whole state space. Furthermore each eligible action for each state also has to be maintained in a structure. When the state space reach a given size, it may become infeasible to use this approach due to the memory requirements or the training time required to reach all appropriate states.

This also means that it will take longer time for the RL algorithms to converge to an optimal policy. The behavior of the agents can also be negatively affected by the standard framework because of the number of games required before the agent takes qualified decisions.

Therefore we propose a multi layered framework consisting of a top level RL system and several lower level RL systems. The performance of the multi layered framework is tested against a standard single layered RL in section 5.

First we extend the classical RL framework by splitting the RL system into a top level layer and a low level layer. The top level RL uses a swappable π/\mathcal{R} , which can be changed by a *profiler* which is described below. The lower level RL systems uses a

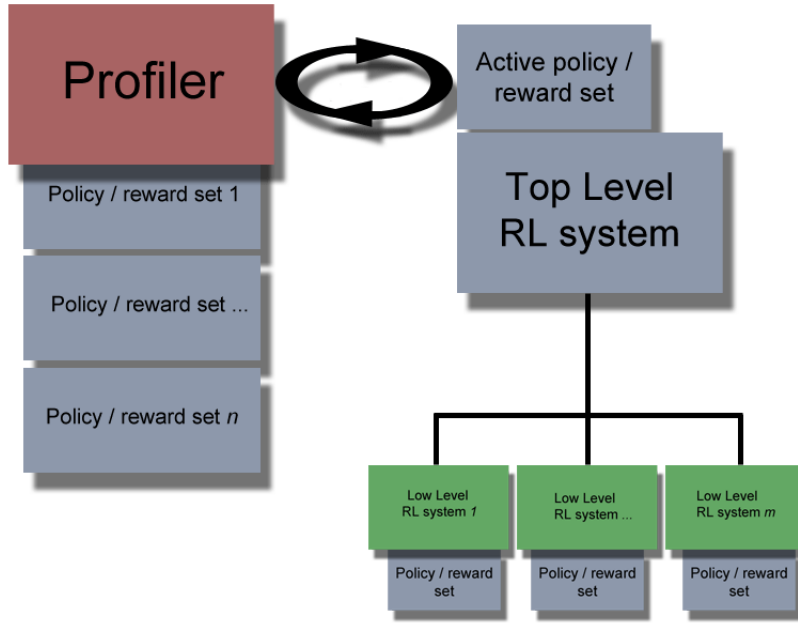


Figure 3.1: Multi layered Reinforcement Learning diagram that shows that the top level RL system uses a swappable π/\mathcal{R} , and has several lower RL systems attached.

fixed π/\mathcal{R} . Neither the reward set or the chosen policy are changed in the lower level. Figure 3.1 shows a graphical representation of this model.

To be able to decide when the active π/\mathcal{R} should be swapped in the top level RL system, a player modeling system, which from now on is called *profiler*, models (or profiles) the enemy to determine the behavior or strategy currently used. The *profiler* can for example determine that the enemy is playing defensive or aggressive. When the *profiler* determines that the behavior of the enemy has changed, it swaps out the currently used π/\mathcal{R} , used by the top level.

The last improvement we propose is an extension of the SARSA and Q-learning algorithms, so they update the action values seen multiple steps ago based on the reward given now. The general technique is called *Backtracking*.

The different RL algorithms can be implemented either as state values or action values. While both in theory are possible, there are a number of factors, which cause the action-value method to be far more applicable. When using state values, the optimal action must be found by looking one step ahead. This requires a model in which all possible states from one state can be found. The action-value method effectively caches results of all one-step-ahead searches. The expected optimal long-term return is provided as a value, which is immediately available for each state-action pair.

This means that at the cost of extra attributes required to store the action-values, the optimal action can be selected without having to know the environment's dynamics. In a game like Tank General where we do not know exactly where each state leads, the action-value method is far more applicable.

In the following sections we will describe the three elements of our proposed framework:

3.2. Multi Layer

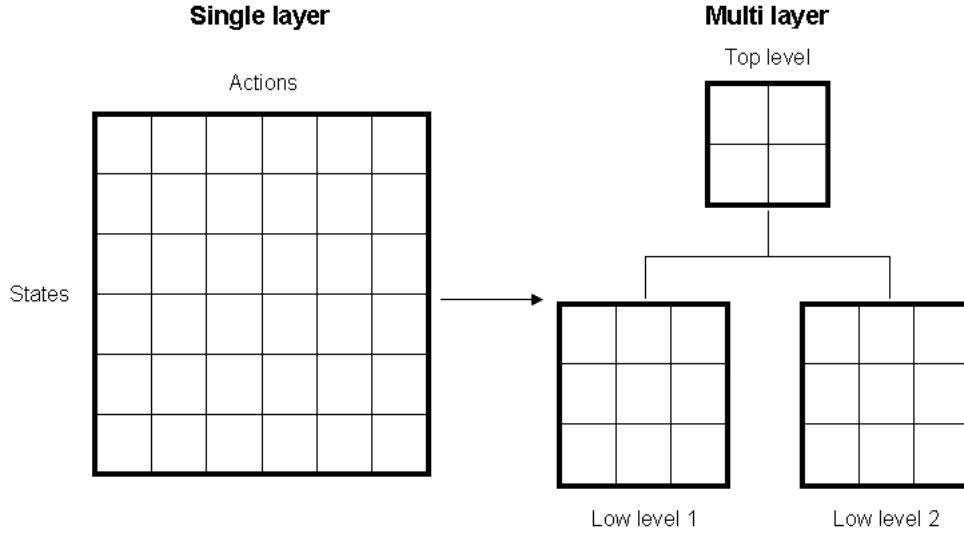


Figure 3.2: This figure shows how we modified the single layered structure into the multi layered structure.

multiple layers, player modeling and backtracking.

3.2 MULTI LAYER

When working with a single layered structure the state space quickly increases as state attributes and actions are added, which might lead to poor performance. That is why we choose to separate the different state attributes in several layers and the related actions as well.

We propose a framework that makes use of the divide and conquer principle and apply it to Reinforcement Learning. The framework is a conversion of a single layered structure into a multi layered structure and thereby decomposing the problem into smaller parts. The framework consist of a categorization of the layers, and the state attributes and actions are added to their related categories. The architecture of the structure resembles the military in the sense that the top levels have a wider area of responsibility and the actions have a high level of abstraction. As you move downwards in the hierarchy, the area of responsibility decreases and the actions becomes less abstract and more specific. Figure 3.2 shows how the single layered structure is converted into a multi layered structure.

The left side of the figure represents the state space of the single layered structure, where all the action-values for each state are indexed. The right side of the figure represents the state space of the multi layered structure. As it appears the single layer becomes a multi layer while retaining the same state attributes and actions and reducing the state space greatly. This is achieved by decomposing the structure from one big part into many smaller parts and thereby distributing the decision making into

smaller parts which together solves the whole problem, as mentioned on 23 in section "Hierarchical Methods".

The top level has the responsibility of taking an action in the low level 1 category or the low level 2 category. Afterwards the following low level will decide which specific action to take. To relate it to the military example, the commander can in this case either take an attack or defend action, which is represented by the two low levels. Depending on which action he takes a sub commander decides how that action is carried out in lower level.

The single layer in figure 3.2 has 36 total states, whereas the multi layer only has 22 total states. One element in our game that might suffer from the amount of state space is the time of convergence. If we played 100 games with an AI that utilizes the single layered structure and 100 games with an AI that utilizes the multi layered structure, it is seemingly obvious that the multi layered AI converges faster. The reason for this is that the multi layered AI only has 22 states to visit, whereas the single layered AI has 36 states, and therefore require more time to converge.

This small example only consists of very few states, but in a game like Tank General the state space could be very large, making it very hard to structure the state attributes and actions in an appropriate manner, minimizing the convergence time while retaining the amount of expressional complexity. Another advantage of reducing the state space and the convergence time is the number of games required for the AI to converge. Even though the simulation of the games have some kind of time multiplier fastening the simulations, we will save a lot of time during the test phase by the reduction of the state space.

The division of the layers is in theory not restricted to only two layers, which also is described in section 2.3.2. It depends on the size of the game, among other things, whether to partition the structure in two or more layers.

The idea of applying the divide and conquer principle to Reinforcement Learning is relatively new and there are certain issues that needs to be outlined. We will in the following text explain how the ϵ value can lead to poor performance, when converting a single layered structure into a multi layered structure.

Figure C.1 in appendix C illustrates how balancing the ϵ value can optimize the decision process of an AI. The ϵ value, which is a randomness factor in the process of taking an action, indicates the balance of the *exploit/explore* issue. As figure C.1 indicates, the ϵ -methods, which has an ϵ value of 0.1, spends a longer time exploring the actions than a method with an epsilon value of 0, but in the end yields a better result.

Imagine that a single layered structure, containing attributes and actions and the ϵ value is set to a relatively high, but reasonable percentage, is converted into a multi layered structure and the attributes and actions are distributed to their layers accordingly. The new structure now consists of smaller layers that each holds some of the attributes and actions which means that there are fewer states to be visited and the AI will converge faster. Due to the fewer states, the AI does not need to explore as much as the single layered structure and can decrease the ϵ value. When it has converged it knows what to do in the different states and will take the right actions. But this can be disrupted by an ϵ value that is not balanced. If it kept the ϵ value at the initial value, the chance of taking random actions will be too high, which might lead to a poor performance. That is why one should consider to lower the randomness factor when converting a single layered structure into a multi layered structure and thereby decreases the state space.

3.3. Player Modeling

Another way of dealing with the problem of taking suboptimal decisions is by using optimistic initial values, as we mention in section C.1.3 in appendix C. This can be achieved by combining optimistic initial values with a reduction of the ϵ value. Because there are fewer states to visit, it is sufficient to explore in the beginning of the game to visit the most of the states. When the optimistic initial values has settled, the AI has converged and the low ϵ value will not disrupt the process of taking an optimal action.

Dividing the framework into several layers provides us with the possibility of changing the strategy by changing one of the levels. In the next section we will describe how this can be archived by integrating a swappable system that changes the strategy according to the opponent's strategy.

3.3 PLAYER MODELING

The main idea about the *profiler* is that the RL system should be able to adapt better and react to the opponent. If the opponent follows a strategy over time, the *profiler* should react to this as soon as possibly and exploit its knowledge to apply an effective counter strategy.

One possibility is to use Bayesian Networks (BN) or Naïve Bayesian Networks (NBN) as classifiers [14]. By using observations in the game, the BN or NBN can be used to classify the player model. Given a set of variables $\{A_1, \dots, A_n\}$ called features, and a *class variable* C , the task is to determine the value of C . An advantage of using NBN is that the structure of the model is smaller, and therefore inference is much faster. All variables in an NBN have its class as its only parent. NBN assumes that the variables are independent given the class, which is rarely the case. However NBN have been shown to be very precise especially in classification where only the maximum posterior probability is used [14].

To classify the player model, we want to select the most likely classification C given the attribute values a_1, a_2, \dots, a_n . The result is:

$$V_{bn} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod P(a_i|v_j) \quad (3.1)$$

$P(a_i|v_j)$ is estimated using m-estimates [1]:

$$P(a_i|v_j) = \frac{n_c + mp}{n + m} \quad (3.2)$$

Where

n = the number of training examples for which $v = v_j$

n_c = number of training examples for which $v = v_j$ and $a = a_i$

p = a priori estimate for $P(a_i|v_j)$

m = user specified parameter. Higher values put more weight on the prior estimate.

Alternatively the maximum likelihood could be found just by using the formula:

$$P(A_i = a|V_{bn} = v) = \frac{N(A_i = a, V = v)}{N(A_i \neq ?, V = v)} \quad (3.3)$$

The training set could be constructed and classified by a human, by making a number of scenarios where the human chooses the value of the class variable in each scenario.

In a game like our created game Tank General, the attributes could for example be: *EnemyUnitsShootingAtHQ(Interval1,Interval2,...,Intervaln)*, *EnemyUnitsPatrollingOwnHQ(Interval1,Interval2,...,Intervaln)* and *NumberOwnedResources(Few, Some, Many, All)*. It may not always be clear how to choose which value the class variable should have, based on the variables. A scenario could contain values so that one could classify the enemy as aggressive, defensive and as a resource gatherer all at the same time. The correct value can depend on the specific state of the game in progress, who dominates the current game etc. This is why training examples should be classified by a human, if using this method.

Another approach is to classify the player model as a simple function of some internal game variables representing the different models. Given the class variable C which can be in one of n states and the internal game variables, we define n functions:

$$C_i : (GameVariables_i) \longrightarrow [0, 1] \quad (3.4)$$

Where $GameVariables_i$ are the internal game variables relevant for the function C_i to determine the model.

Each function C_i outputs a number between 0 and 1. For example to determine how aggressive the enemy is, the function could check how much damage the enemy has dealt to its opponent's headquarters and express this as a number between 0 and 1. To handle the situation where more functions returns the same values, all members of the class variable should be prioritized. If for example the enemy is determined to be aggressive with a value 0.5 and defensive also with value 0.5, and the aggressive is chosen to be of highest priority, the *profiler* would assume the enemy to be aggressive and react according to that.

Regardless of the method chosen to profile the enemy player, the result is the same: the enemy strategy determined by the *profiler* should be countered with the best counter strategy. From the perspective of the RL system, it means that the π/\mathcal{R} should be changed. If utilizing the multi layer approach, the π/\mathcal{R} could be swapped in any of the layers. In section 4.3 on page 44, details of how the player modeling is implemented in Tank General is described.

3.4 BACKTRACKING

Q-learning and SARSA updates its policy based on the information available one step back in time. This makes these algorithms far more efficient than Monte Carlo methods for our purpose, because they can learn while playing, and not only after an episode is finished. A disadvantage with standard Q-learning and SARSA algorithms is that when they are given a reward, the effect is only propagated one step back in time. This can be seen in the algorithms for SARSA and Q-learning on page 13. If the reward for example was given after five steps, it would require that the same path of actions were taken five times (five episodes), to propagate all the way back, so that a new episode could take advantage of it.

3.4. Backtracking

If instead a fraction of the given reward was distributed any number > 1 steps back, the values will be propagated all the way back faster. In the example from before, the advantage would be obvious after only one episode, because the value from the fifth step would then have propagated back to the start of the episode. Over many games this modification converges to the same values as the normal Temporal Difference methods, but it should converge faster, which is important in games.

To setup the theory for this modification of RL we call *Backtracking*, some definitions follow. Let $n \in \mathbb{N}$ be the number of steps, the *Backtracking* updates. Let *History* be a set containing states or state-action pairs from the last n steps. Every time an action is taken, the new state or state-action pair is added to the *History*. When the action value is updated, the old states or action-state pairs in the history receives a fraction of the newly received reward.

The states or state-action pairs in the *History* list are affected in a varying degree according to how many steps have been taken, since they were added to the list. This is achieved with a β parameter. For every step taken, the algorithm iterates over all states or state-action pairs in the *History* with a varying β parameter, which is added to the normal equation.

Below is shown an implementation of a modified Q-Learning algorithm with backtracking properties.

```

Initialize for all  $s \in S$  and  $a \in A$  the value  $Q(s)$  to a constant value
Set  $\beta$  to 1
Repeat for each episode:
    Initialize  $s$ 
    Repeat for each state in the episode, until  $s$  is terminal:
        Choose  $a$  for  $s$  using  $\pi$ 
        Execute action  $a$  and save reward  $r$  and next state  $s'$ 
        Add current state to History:
        Repeat for each state  $i$  in History:
             $\beta = 1/2^{\text{length}(\text{History})-i}$ 
             $Q(\text{History}[i]) = Q(\text{History}[i]) + \beta * (r + \gamma Q(s') - Q(s))$ 
        if  $\text{length}(\text{History}) == N$ 
            Remove first element from history
         $s = s'$ 

```

This differs from eligibility traces, which are described in section 2.3.2. The theoretical base of eligibility traces called *n-Step* [25] is a method which accumulates rewards for the n last steps, where n is the number of steps to backtrack. The i 'th state is then assigned this accumulated reward $i + n$ steps after. This means that each update for a state or state-action pairs is only performed once. The reward is delayed n steps, and this can really affect the online performance, if n is large enough. Our proposed method updates the n last states at every step. Instead of waiting n steps, the update is performed right from the start. At the very first step, the algorithm works like the standard TD algorithms.

3.5 SUMMARY

We have, in the previous sections, described how to convert a single layered structure into a multi layered. After that we explained how the player modeling system could be used to adapt the strategy to the playing style of the opponent. Lastly we described how a modification of the update process of the policy could improve the performance by propagating the reward n steps back instead of just one step back. We will now describe how these three elements can be used together in order to amplify the overall performance.

Of the three elements the multi layer framework and the player modeling system works very well together. The backtracking system can be applied to any framework that utilizes Reinforcement Learning and is not as dependent on the other two elements.

As we mentioned in the beginning of this chapter, the top level of the multi layer uses a swappable π/\mathcal{R} set. This provides us with the possibility of changing the strategy of the AI by having several sets where the rewards in each set are biased towards different behaviors. E.g. it would be a good idea to have an aggressive strategy when playing against a defensive opponent, or a defensive strategy when playing against an aggressive opponent.

Imagine a top level where actions like *PatrolHeadquarters*, *DefendHeadquarters*, *AttackEnemyHeadquarters* and *PatrolAreaAroundEnemyHeadquarters* existed. To bias the strategy towards a defensive one, the rewards for *PatrolHeadquarters* and *DefendHeadquarters* should be higher than *AttackEnemyHeadquarters* and *PatrolAreaAroundEnemyHeadquarters*. Whereas the rewards for *AttackEnemyHeadquarters* and *PatrolAreaAroundEnemyHeadquarters* should be higher than *PatrolHeadquarters* and *DefendHeadquarters*, for an aggressive strategy.

Having these different strategies enables the AI to adapt its strategy to counter the strategy of the opponent, which gives the AI a great advantage. But in order to change the strategy the AI needs a system that detects which strategy the opponent is currently using and that is where the player modeling system comes in handy.

The player modeling system profiles the opponent, meaning that it detects and analyzes every move the enemy makes in order to determine which strategy the opponent is currently using. When a certain strategy is detected, the profiler gives notice to the AI, which swaps the top level with a strategy that counters the strategy of the enemy. Imagine a scenario where the enemy attacks the headquarters with many units. The profiler will detect this by either detecting the hit points of the headquarters or the number of enemy units in sight of the headquarters. This indicates an aggressive action by the enemy and the profiler notice the AI of the attack. The AI can now change the strategy to a defensive one giving the units higher rewards for taking the actions *PatrolHeadquarters* and *DefendHeadquarters*. The units will now learn that being defensive will yield a high reward and they will therefore defend the headquarters.

Although this appears like a sensible co-operation between the multi layer framework and the player modeling system, there are disadvantages. A multi layer framework that does not swap the top level, thereby maintaining the strategy throughout the game, only has one set of π/\mathcal{R} to update. This means that there are only one policy that needs to converge. But when the framework has several π/\mathcal{R} sets, then all of these

3.5. Summary

needs to be converged in order for the AI to perform optimal. Therefore the AI will require a longer period for all of the sets to converge, which gives the opponent an advantage due to the AI's incapability of taking optimal decisions.

This concludes our proposed framework and we are now ready to show a practical example of how the framework can be implemented in an actual game.

Chapter 4

CASE STUDY

We will in this chapter turn the focus towards our game and how we have implemented our framework, which was described in the previous chapter. As mentioned before, our framework consisted of three elements, which we added or modified from the standard Reinforcement Learning approach, being:

- Multi Layer
- Player Modeling
- Backtracking

We will start by briefly introduce our game called Tank General. This will also include an examination of appropriate strategies and opportunities in the game which we need to consider before we implement our Reinforcement Learning AI for the game. Then we describe how we implemented the multi layered structure, followed by a description of the implementation of the player modeling system which determines the opponent's style of play, and the implementation of our proposed modification to the update procedure of the policy called backtracking. Lastly we will describe how we implemented the standard single layered Reinforcement Learning AI and we will finish the chapter with a discussion about how the proposed framework in general can be used in Real Time Strategy games.

4.1 TANK GENERAL ANALYSIS

"Tank General" is a Real Time Strategy game, where two armies fight each other on relatively small maps. The goal of the game is to destroy the headquarters of the enemy. There are five different units in the game, all with different strengths and weaknesses. During the game each army receives reinforcement points which can be used to build new units or to repair the headquarters. Two resources called War Factories are located on the map. The War Factories can be captured, which will give its owner more reinforcement points. A very comprehensive description of the game

containing detailed description of the game mechanics, the rules, the units, how to play etc. can be found in Appendix D.

Before we are able to implement our proposed Reinforcement Learning framework in Tank General, we need to analyze it in order to discover the most important strategies, choices and possibilities in the game. An important part of our vision for Tank General was to make a very simple game, with many interesting strategical choices for the player or computer AI to make. Choices which have an impact on the opponent and choices which the opponent has to react on. This analysis will help us decide how to implement the framework which we propose.

4.1.1 THE COMPOSITION OF UNITS

From the beginning of the game, each army consists of the same number and types of units meaning that both armies are identical. But as time passes by, each player can build new units from reinforcements. A big strategical decision is how and when to spend these reinforcements. One strategy could be to build an army which is fairly balanced consisting of all types of units. Another more advanced strategy that also might be more optimal would be to build units which counters the units of the enemy. But due to the fog of war, a player might not know the composition of the units in the enemy army. This strategy relies on information about the enemy which means it might be clever to build Recons for scouting purposes in order to get this information and then build the counter units.

4.1.2 MINOR OBJECTIVES

As mentioned in the introduction of this section the main objective of the game is to destroy the enemy headquarters. In order to achieve this, the AI has to consider achieving minor objectives which can make it easier to achieve the main objective. This could be to gain control of the resources in order to establish an economic advantage before pursuing the main objective. How much attention the AI has to give minor objectives might change according to the current game situation. If an aggressive AI has pushed the opponent back to a defensive position, it might be better to change style and go for the resources instead of keep attacking. The importance of minor and main objectives is a trade-off, the AI has to consider during the game.

4.1.3 ATTACKING

There are many different strategies when attacking, and the AI must be able to find out how to carry out a successful attack. One strategy can be to wait for a favorable moment and attack with all units at once in order to strike hard. This tactic requires that the AI keep track of the enemy units in order to strike at the right moment. A totally different strategy, where many small attacks are carried out and the attacking units sacrificed, might be a proper strategy, since little strokes fell great oaks. Even when the attack is broken up, it might have been successful if damage was dealt to the enemy headquarters. But the most important thing in this context is that the AI is

4.2. Multi Layer

able to coordinate the attack, so it does not keep attacking in vain with one unit.

4.1.4 DEFENDING

A strategy for defending is just as important as one for attacking. The AI have to consider if it should always have some units defending its headquarters or if this is unnecessary. An active defense might be a better choice against some players than a passive. One example could be to constantly attack the enemy with small groups of units keeping the enemy busy defending, meanwhile the AI is trying to capture the resources.

Another thing to mention is a precarious situation which can occur if the human player manage to break through the AI defense on a direct attack at the headquarters. If two human units are attacking the headquarters and no AI units are nearby, it might not be a good idea to rush out reinforcements one by one as they become available, since two units can quickly kill one unit. Instead it might be better to save up two or three reinforcements and then build the units at the same time making it easier to kill the attackers.

4.1.5 IDENTIFYING KEY AREAS

The level design might be a good thing for an AI to take into consideration when deciding how to move or place its units. Maybe there is a better place to defend than in the vicinity of the headquarters. If there are any bottlenecks on the map, that a player can take advantage of when defending either the headquarters or an important resource, the AI might be able to identify choke points and exploit them, because it is far easier to defend a narrow point than a wide area.

4.2 MULTI LAYER

Having considered and examined different strategic possibilities we are now ready to implement our Reinforcement Learning framework into our game. In this section we describe the process of selecting the state space attributes, the actions and the rewards for our multi layered Reinforcement Learning.

4.2.1 STATES AND ACTIONS

One of the first decisions to make when implementing a Reinforcement Learning algorithm is how to split the world up into a finite state space. This process is tightly bound to the process of selecting the actions since the state space can be seen as the foundation for the actions. The state space has to be big enough to contain relevant information on which sensible actions can be chosen. If the state space is too large, there are too many states to explore which results in a slow learning rate, but on the other hand if the state space is too small, there is no foundation for learning. All

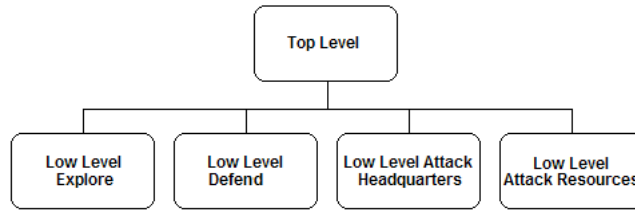


Figure 4.1: Multi Layered Reinforcement Learning in Tank General.

algorithms used in Tank General use the action-value method. For reasons stated in section 3.1 on page 29, this is the most appropriate choice.

In our framework we propose a top level which take decisions on a strategic level. From our analysis of the game we identified four strategically areas; defending, exploring, attacking resources and attacking the enemy headquarters. Therefore we found it reasonable to let our top level consist of these four superior actions and create four corresponding low level RL systems which each focus on one of these areas. It is the job of the top level to learn, in which situations defending, exploring etc. is favourable, while it is the low level RL systems' task to carry this out on a lower tactical level. Having decided this, the next step was to create a state space which can hold information to support these actions. This means that we need to figure out what information the top level needs to access in order to take a reasonable decision. All state space attributes on the top level must influence the belief of at least one of the actions. The following questions each influence the beliefs in at least one top level action and is therefore modeled as a top level state space attribute mentioned in brackets.

- Who is most powerful? - attack, defend (PowerDistribution).
- Are the resources discovered yet? - explore, attack resources (ResourcesExplored).
- Are the enemy headquarters discovered yet? - explore, attack headquarters (HeadquartersExplored).
- How many hit points does the friendly headquarters have? - defend (OwnHeadquartersHitPoints).
- How many hit points does the enemy headquarters have? - attack headquarters (EnemyHeadquartersHitPoint).
- Who has captured the resources? - defend, attack resources (ResourceDistribution).
- Does the enemy threaten either friendly resources or the headquarters? - defend (EnemyThreateningBuildings).
- Are any of the friendly units close to a unguarded not owned resource? - explore, attack resources (UnitCloseToUnguardedNotOwnedResource).
- How many units are defending the enemy headquarters? - attack headquarters (EnemyDefenders).

4.2. Multi Layer

These state space attributes each have two or three possible values. We use a method called function discretization which is described in section 2.3 in the subsection *Curse of dimensionality*, where intervals can be used instead of precise integers. As an example we narrow the hit points of the headquarters from precise integers to intervals of *high*, *medium* and *low* in the attributes called *OwnHeadquartersHitPoints* and *EnemyHeadquartersHitPoints*. The same approach is used to decide the state space attributes for the low level Reinforcement Learning systems. Appendix E contains, among other things, the state space attributes and actions for the top level and all low levels.

Each low level have between five and eight actions which support the strategy of the top level. Each action will give the units different commands. Some actions affect all units, and others only affects some of units. As an example the action *SendToHeadquarters.50* will take the 50% units which are closest the headquarters and command them to return to the headquarters and defend. These units will then defend the headquarters untill they get a new command. The action *ExploreRandomPosition* will take all *recons* and send them to random positions on the map. If there are no *recons* it will instead command one units, which is not an *artillery*, to explore random positions instead. Almost all actions have some kind of intelligent scripted behaviour like these examples which means that a relatively small amount of actions fit many different states.

4.2.2 REWARDS

Selecting the right rewards is vital for the Reinforcement Learning AI, since the rewards decide which actions are good in which situations. The behavior of the AI can be biased in the right direction by selecting the right rewards and afterwards assign the rewards with sensible numeric values. We start out with simple rewards, given when winning the game (*EnemyHeadquarterDestroyed*) and conquering resources (*ResourceCaptured*). These were later expanded to include many more sophisticated rewards, such as defending the headquarters (*EnemyUnitCloseToOwnHeadquartersKilled*), defending resources (*EnemyUnitCloseToEnemyHeadquartersKilled*), exploring (*ResourceExplored*, *EnemyHeadquartersExplored*) etc. The result of this was even more nuanced behaviors and action patterns.

This process of balancing the rewards often consists of a lot of trial and error. As an example we initially experienced that our Reinforcement Learning implementation was very carefully and almost never attacked. To solve this we created another reward which is given, when damage are dealt to the enemy headquarters (*DamageDealtToEnemyHeadquarters*), and a reward given when an enemy unit is killed close to its headquarters (*EnemyUnitCloseToEnemyHeadquartersKilled*). All rewards and their respective assigned values can also be found in Appendix E.

As mentioned previously we propose that each top level has its own set of rewards. In our implementation, we have three top levels (described in section 4.3), which have the same rewards, but the assigned numeric values are different depending on the purpose of the top level. An example is the reward called *ResourceLost* which is given when the enemy captures a resource owned by the other team. Table 4.1 shows the different numeric values for this reward for the three top levels.

The main task of the anti aggressive top level is to defend which include defending

Top level	Value for ResourceLost
Anti aggressive	-3
Anti defensive	0
Anti resource	-5

Table 4.1: Numeric values for the reward *ResourceLost* for the different top levels.

the resources. Therefore a negative reward is given when losing a resource to the opponent. The main task of the anti defensive AI is to attack the enemy headquarters which means that possession of resources is rather unimportant which explains why no negative rewards are given for losing a resource. On the other hand it is very important that the anti resource top level fights for the resources. If the anti resource top level loses a resource, a huge negative reward is given. We should mention that the specific size of the rewards are not of overly great importance, because their relative more than their absolute values are of the essence.

4.3 PLAYER MODELING

Where the multi layer structure was a modification of an already existing structure, player modeling is a system that we implemented from scratch and integrated into the overall structure. Although it is a completely new system, it works in close collaboration with the multi layer.

The purpose of the player modeling system, from now on called *profiler*, is to determine the style of play or strategy of the opponent in order to find the appropriate counter strategy. Hence the name profiler, as its job is to determine which profile that matches to the opponent's style of play.

Before we began the implementation of the Reinforcement Learning, we spend a lot of time creating scripted computer AI's. During this phase we did a comprehensive analysis in order to find out how to create a strong scripted AI. We even implemented an entire framework for creating scripted computer players. Using this framework made it very quickly to implement different scripted AI's and test them against each other. This framework is described in further details in Appendix B. During this process we discovered three important player profiles which we want to classify.

The three predefined profiles classified by the are:

- **Aggressive** This profile indicates that the opponent is attacking both the headquarters and resources a lot.
- **Defensive** This profile indicates that the opponent rarely attacks, but would rather stay home and defend either the headquarters or the resources.
- **Resource** This profile indicates that the opponent has prioritized the task of capturing the resources and defending them highly.

For each of these strategies we have created a counter strategy that the AI can switch

4.3. Player Modeling

between when it is necessary. The process of determining when to switch strategy is carried out by the profiler, and will be explained in further details later in this section. We created the profiler to optimize the performance of the AI in the aspects of adaptation and exploitation.

Remember that the top level of our structure takes the general decisions e.g. what type of actions to execute, whereas the low level takes more specific decisions e.g. how to carry out the action taken by the top level. This means that it is actually the top level that decides which kind of behavior the AI should have. So in order to change the behavior of the AI, it is sufficient to change the top level and not necessary to change the low levels.

The way we change the top level is to have several top levels, each biased towards the different behaviors. So when the *profiler* senses the need to change strategy, it simply switches the active top level with another top level that is able to counter the strategy of the opponent, and thereby increasing its survival odds.

The following text will explain in detail how the *profiler* determines which of the strategies is being used by the opponent, in order to change to a counter strategy. In section 3.3 on page 33 we proposed two different ways to create the *profiler*: one based on Bayesian Networks and a more simple approach based on functions of some internal game variables representing the different models. We chose to implement the simple function based approach. The reasons for this choice were to save time and because we estimated that in our case the chosen approach would perform just as well as the more advanced Bayesian Network approach. This choice saved us a lot of time which was used on implementing and balancing the learning algorithms.

AGGRESSIVE

To determine whether the opponent is using an aggressive strategy, the *profiler* keeps track of the damage dealt of its own headquarters and the units close to the headquarters. The threshold for changing to the anti aggressive strategy is set to 0.4. Then for each update in the game, the *profiler* calculates the level of *aggressiveness* of the enemy, and if this value exceeds the threshold, the AI will categorize the enemy as being aggressive, which triggers the activation of the anti aggressive strategy. The rewards for the anti aggressive strategy is biased to being defensive, and the AI will get high rewards for defending the headquarters.

Figure 4.2 illustrates the calculation of the aggressiveness value. To understand the calculation some expressions needs to be explained.

- **AverageRateOfFire:** The total damage of the 5 unit types divided by 5.
- **TimeWindow:** The interval that the *profiler* uses to check for the aggressiveness of the enemy is set to 4 seconds.
- **TotalHits:** The total hits dealt to the headquarters and the units nearby, in the interval of the TimeWindow.
- **NumberOfEnemyUnits:** The number of total enemy units.

$$\frac{AverageRateOfFire \cdot TotalHits}{TimeWindow \cdot NumberOfEnemyUnits}$$

Figure 4.2: Calculation of the aggressiveness value.

As it appears from the expression, the *AverageRateOfFire* and the *TimeWindow* are static, whereas *TotalHits* and the *NumberOfEnemyUnits* changes over time. The interesting part of the expression is the *TotalHits* and the *NumberOfEnemyUnits* which decides whether the AI should change its strategy to an anti aggressive strategy or not. This can be illustrated by the following two expressions:

$$\frac{2.3 \cdot 6}{4 \cdot 20} = 0.1725$$

$$\frac{2.3 \cdot 16}{4 \cdot 20} = 0.46$$

In the first expression the total number of hits is 6 and there are 20 total enemy units, which results in an aggressiveness value of 0.1725 not exceeding the threshold. In the second expression the total number of hits is 16 hits and there are 20 total enemy units, which results in an aggressiveness value of 0.46, exceeding the threshold and therefore triggers the activation of the anti aggressive strategy.

DEFENSIVE

The process of determining whether the enemy is using a defensive strategy is less complicated than determining if the enemy is using an aggressive strategy. One thing they have in common is a threshold, which indicates what strategy the enemy is leaning towards.

The way the *profiler* determines if the enemy is being defensive is to scan the area around the enemy headquarters for enemy units. If the number of units surrounding the headquarters exceeds 70% of the total unit count, then the enemy is using a defensive strategy. The 70% is the threshold for being in a defensive mode.

If the *profiler* detects that the enemy is using a defensive strategy, it would trigger the activation of the *Anti defensive* strategy where the rewards are biased towards a more offensive/aggressive strategy.

We ran several tests with varying values of the threshold parameter and concluded after the balancing phase that 70% was a fitting threshold.

RESOURCE

The last strategy that the *profiler* is able to detect is the strategy of an opponent which prioritizes the resources on the map very highly. This strategy is known in the RTS genre as booming, and it refers to capturing and retaining the resources in order to expand the economics.

The threshold for changing the strategy to the anti resource is set to 0.4. The meaning of anti resource is that the AI will prioritize the resources higher. This is achieved by

4.4. Backtracking

increasing the value of some rewards e.g. capturing the resources, killing enemies near the resources and having own units close to the resources. The *profiler* can calculate how much the opponent desires the resources, and if this desire exceeds the threshold, then the AI will activate the anti resource strategy. In order for the *profiler* to calculate this desire it needs 3 information.

- **TimeResource1IsEnemy**: indicates the total time that resource 1 has been under the enemy's control.
- **TimeResource2IsEnemy**: indicates the total time that resource 2 has been under the enemy's control.
- **TimeResource1CouldHaveBeenEnemy**: the total time of the game so far, where resource 1 has not been under enemy's control.
- **TimeResource2CouldHaveBeenEnemy**: the total time of the game so far, where resource 2 has not been under enemy's control.

The *profiler* then uses this information and insert it into the following expression to calculate the desire:

$$\begin{aligned} res1 &= \frac{TimeResource1IsEnemy}{TimeResource1CouldHaveBeenEnemy} \\ res2 &= \frac{TimeResource2IsEnemy}{TimeResource2CouldHaveBeenEnemy} \\ desire &= \frac{res1+res2}{2} \end{aligned}$$

So the longer the enemy controls the resources, the higher desire will be for the Reinforcement Learning AI to attack the resources and try to capture them. When the desire exceeds the threshold, the Reinforcement Learning AI will switch to a counter resource strategy.

When our multi layered Reinforcement Learning AI knows what kind of player it is up against, it can very quickly adapt to that style of play by changing the top level and the top level rewards to a top level which counter that specific playing style.

4.4 BACKTRACKING

As we discussed in the proposed framework in section 3.4 on page 34, SARSA and Q-learning only updates its policy based on the information available one step back in time. In order to decrease the time of learning even more, we therefore proposed a new way to update the policy file where several steps back in time are updated. Our actual implementation of the backtracking update function is equal to the algorithm described on page 35 in section 3.4.

Using both the previous mentioned *profiler* and this new backtracking update function it is our hope to make our Reinforcement Learning algorithm learn even faster. Our comprehensive tests and results are discussed in chapter refResults: results on page 57.

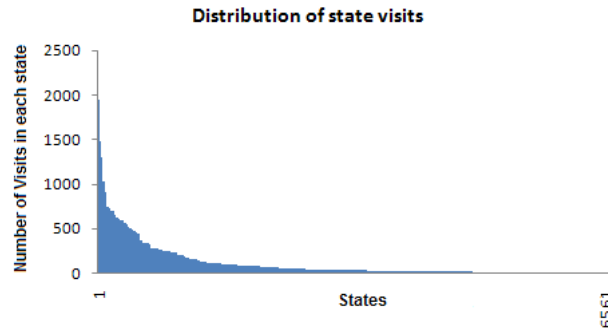


Figure 4.3: A Graph showing how many times each state is visited during 500 games in a top level policy file.

4.5 STANDARD SINGLE LAYERED REINFORCEMENT LEARNING

In this section we describe how we implemented the standard single layered Reinforcement Learning AI, which consists of one big policy containing all states and action values. Originally we wanted our single layered Reinforcement Learning AI to consist of the same state space attributes and actions as the multi layered. But this turned out to be impossible since the state space would be too big as the following expression clearly shows. The multi layered Reinforcement Learning consist of 20 unique state space attributes distributed to several RL systems (some attributes are used in more RL systems). If these state space attributes were assembled in the same RL system, we would end up with:

$$3 \cdot 3 \cdot 2 \cdot 3 \cdot 3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 \cdot 4 \cdot 2 \cdot 3 \cdot 2 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 2 \cdot 2 = 181.398.528 \text{ states}$$

First of all, learning would be very slow, and secondly dealing with that amount of states in a file would require a lot of memory. To solve this, we decided to reduce the number of states. But in order to still have a decent single layered AI, we had to remove the less important states meaning the states which are visited less times. Therefore we made some tests with our multi layered Reinforcement Learning to discover which kind of states are visited the most. Looking at the policy file for a top level which had played 500 games we found out that in average each state has been visited 5.16 times but, as figure 4.3 shows, some states are more frequently visited than other states.

We decided to keep 14 of the 20 state space attributes and let each attribute have 2 values. Taking this approach results in only $2^{14} = 16384$ states which is acceptable. The single layered Reinforcement Learning AI has the same actions as its multi layered counter part. Since the actions are dependent on the state space attributes, we removed those state space attributes, which our tests showed contained the least amount of information. This was a difficult process, because we had to identify, which state space attributes have different values and are among the most visited states. To comprehend this task, we developed a small application which can extract information from the policy files. The application is called *Policy.exe* and a screen shot can be seen on figure 4.4.

4.6. General Application

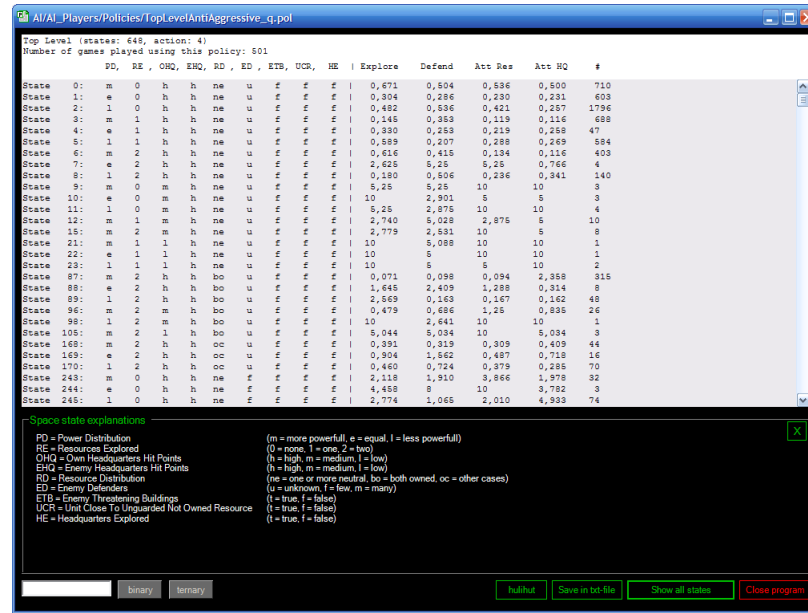


Figure 4.4: Screen shot of our program we developed to study our policy-files.

The rewards of the single layered Reinforcement Learning is very similar to the rewards of the multi layered. Further details about state space attributes, actions and rewards regarding the single layered Reinforcement Learning AI can be found in Appendix E on page 105.

The single layered Reinforcement Learning AI described in this section is a part of the tests in Chapter 5 Results section 5.4 page 64 where the multi layered approach is compared with the single layered.

4.6 GENERAL APPLICATION

In the following section we describe our thoughts and considerations when applying our proposed framework in RTS games. We discuss the process one have to go through, and the choices and questions which arise when designing a multi layered Reinforcement Learning AI using player modeling. Implementing Reinforcement Learning into an RTS game successfully is a difficult task and this is also the case when using our framework. Reading this section will give some advices about how to approach the task.

4.6.1 MULTI LAYER

The first important and critical task is to carry out a comprehensive analysis of the game in which you want to design a Reinforcement Learning AI for. You have to analyze all aspects of the game including different strategies and choices, the player faces during the game. One way to discover successful strategies could be to play the game in different ways against other players or implement simple scripted AI's and measure their performance against each other and against human players.

Having completed the preface analysis you are ready to design the Reinforcement Learning elements. The first task to consider is the overall structure, since other parts, like the player modeling and elements like state space and actions, are dependent on the overall structure. Depending on the complexity of the game you should consider how many layers are necessary, and how many sub RL systems each layer should consist of. It is necessary to consider how to map sub problems into small sub RL system. One way is to put highly abstracted actions in the top of the hierarchy and more specific actions in the lower levels. The important issue is for each RL system to have a clear problem to solve, whether it represent an abstract problem like a strategy which needs to be carried out, a more concrete task like gathering resources, control of a specific unit or a group of units or something totally different. In a multi layered Reinforcement Learning AI, the top level could select a strategy which was then carried out by a lower level. One could also consider another structure where the top level sometimes takes a concrete action, affecting the units directly, while other times activating a lower level to carry out a task.

Having the overall structure in place you are ready to further specify each RL system. The first task is to identify actions and defining a state space which supports the chosen actions. From the comprehensive analysis during the preface you should have a good idea about which actions might be good for solving each sub problem. What you do not need to worry about is the order of these actions in different situations. It is the task of the RL system to figure this out, but it is your job to give the RL system the possibility to solve this by provide it with reasonable actions. Since the state space form the basis of which actions are taken, it should contain all relevant information. The state space consists of several state space attributes containing information about the game environment. Depending on the complexity of the game you should consider if it is necessary to use function discretization for state space attributes to reduce the number of states.

To check whether your state space is adequate or not, consider each action one by one. For each action you should have at least one state in which it would be reasonable to take this action. If you are not able to create a proper mapping between states and actions by assigning different values to your state space attributes, you should reconsider your actions or state space attributes.

Rewards are very important, since they basically decide what the Reinforcement Learning should learn. If the game is very simple, it might be sufficient to give only one reward when the terminal (winning) state is reached. When dealing with longer and more complicated games, typically characterized by having a huge state space, you should consider to give more rewards in order to guide the Reinforcement Learning algorithm in the proper direction. An example of this could be to give rewards when sub-goals are achieved in the game. You should bear in mind that negative rewards can be used to punish the Reinforcement Learning algorithm, and in some games this kind of punishment could provide a way to avoid undesirable states or events. When assigning numeric values to the rewards, you should have in mind their relative values, since the values only makes sense when you look at the interrelationship between the values of the rewards.

Having designed the overall structure, the sub RL systems and its element, you are now ready to consider how to incorporate player modeling in the Reinforcement Learning AI.

4.6. General Application

4.6.2 PLAYER MODELING

We will in this section describe which areas to consider when implementing a player modeling system in our multi layered framework. These are:

- Player types and counter strategies
- Player evaluation
- AI behavior changing

PLAYER TYPES AND COUNTER STRATEGIES

One of the most important things when playing RTS games is how to approach the game. It is questions like; when should you attack the opponent, in which situations should you explore the map, how many of the total units should defend the resource sites etc. Many of these questions can be answered by determining which type of player you are up against. It is important to have detailed knowledge about each player types and their motivations, in order to take advantage of their weaknesses and strengths, which can be used to create effective counter strategies. One of the first steps when implementing a player modeling system is to identify the common player types in the game.

Some of the most common player types are *aggressive*, *defensive* and *boomer* [10]. A boomer is a player type that relies on capturing the resources with the purpose of expanding the technology and aim at building an advanced army. Identifying these player types is not always an easy task as it depends on the genre of the RTS game. Though these are very general and it can therefore be a good idea to further sub categorize them in order to fully tailor a counter strategy for each specific sub category. In fast-paced and action filled RTS games the aggressive player could be sub categorized into two player types; *A rusher* or a combination of an *aggressive/boomer*. The strategy of the rusher is to build a small army very fast in order to destroy the enemy early in the game, whereas the aggressive/boomer builds a larger army, which is not necessarily an advanced army, to attack the enemy.

Knowing each player type in detail including the weaknesses and strengths are important when creating counter strategies. For each player type there should exist a strategy that is tailored exactly to counter that specific player type. For you to achieve this, you must exploit all the knowledge you has about that player type. E.g. when dealing with a defensive opponent, the counter strategy should be modeled in such a way that the defensiveness of the opponent becomes an advantage for the player.

When the player types and their corresponding counter strategies has been clarified, the next step of the player modeling system is the procedure of determining the playing style of the opponent.

PLAYER EVALUATION

The process of determining the opponent's strategy is the core functionality of the player modeling system. This is where the system profiles the opponent and decides which type of player it is up against.

An important thing you have to consider is the time interval between each profiling of the opponent. If the time interval is too short, the system would change strategy too often, resulting in a very confusing and incoherent behavior. On the other hand, if the time interval is too long, it could be too late to change the strategy. Balancing the profiling interval can be a difficult task and it also depends on the type of RTS game you are implementing.

One approach of determining the strategy of the opponent is to detect the categories of actions the enemy has taken over time. If the majority of the actions have aggressive characteristics, e.g. *attackHeadquarters* or *attackResource*, then the profiler could change to an anti-aggressive strategy.

Another approach is to initiate threshold values for each of the player types. Then for every time a threshold is exceeded the profiler would know which strategy the opponent is using and is able to activate a counter strategy.

A third approach is to monitor the action-values in the policy in order to determine whether the current strategy is appropriate for countering the strategy of the opponent. If the action-values is decreasing, over a time period, then the profiler would know that the current strategy is unsuitable to counter the opponent's current strategy.

Even though these procedures are different, they all have in common that they are able to determine the player type of the opponent. As mentioned in the previous section each player type should have a 1:1 mapping to a counter strategy that is able to help the RL AI gain a strategic advantage. When dealing with a single layered framework, the entire policy is switched in order to change the strategy. However this is not the case in a multi layered framework.

AI BEHAVIOR CHANGING

One of the purposes of dividing the framework in multiple layers is to decompose a problem into smaller parts where each part solves a bit of the problem. In order for this framework to cooperate well with the player modeling system, it is a good idea to have a clear chain of command and area of responsibility.

One way of achieving this is to start with a layer that has a wide area of responsibility and thereby decides the overall behavior of an AI. In this case the behavior of an AI corresponds to the strategy currently used. The area of responsibility then narrows as you add new layers to the framework. This provides you with a quick lookup in the framework to identify the layers that needs to be changed according to the player type of the opponent.

The way the player modeling system changes the strategy can be to change one of the layers, which corresponds to changing the behavior in that specific layer. An intuitive approach when switching to a counter strategy is to change the layers which has the widest area of responsibility as it is these layers that determines the overall behavior.

4.6. General Application

Imagine a framework where general actions like *Attack*, *Defend* and *Explore* existed in the top layer, and the lower layers consisted of specific ways/actions of executing the top layer actions. If the player used an aggressive strategy the rewards for the attack actions would be higher than defend and explore actions, to encourage the AI to attack more. If the player was under attack the counter strategy would be to encourage the player to be more defensive. This can be achieved by switching the top layer with another strategy that receives higher rewards for defending, and thereby countering the attack.

Another approach of switching the strategy is instead of switching the overall behavior then only small adjustments in the strategy is switched. Imagine an aggressive strategy which successfully captures the resources on the map, but the attacks on the enemy headquarters is carried out without any success. Then changing the overall behavior would be a bad idea, but changing the way the attacks on the headquarters is carried out would be enough.

It is a question of balancing the level of micromanagement vs. macromanagement in the switchment of the layers, to the type of RTS game that is being implemented. It could be an advantage to have a player modeling system that micromanaged in games with many different styles of play as one could mix different sub strategies in different layers, in order to create an overall superior strategy. Whereas, in games with restricted playing styles, it could be overkill to mix strategies in the layers. Then the player modeling system should be restricted to only switch the top layers.

The player modeling system is used to aid the RL AI with various counter strategies with rewards that are biased towards different behaviors. These rewards are updated using solution methods which we will describe in the next section.

4.6.3 SOLUTION METHOD

The choice of solution method to use in our framework is a difficult one. To start off simple, SARSA should be implemented, since using the next step instead of a maximum action-value like in Q-Learning is initially easier. After this, Q-Learning should be implemented and tests to determine which method to use should be carried out. It can almost never be predicted which method provides the best result, so testing is needed. Since the end policy of the two policies are identical, the convergence rate must be the parameter to be evaluated. A simple method to evaluate this could be Standard Deviation mentioned in section 5.1.2, which can be used to show which of the two solution methods converge the fastest. The test should be conducted against several different opponents and on different maps to get a clear picture of the end result.

If the game suffers from poor convergence rate, several step can be taken to alleviate this. The techniques of Eligibility Traces mentioned in section 2.3.2 or *n-step* mentioned in section 3.4 are both solid methods to reduce the time needed to achieve the best possible policy.

Being the simplest of the two, *n-step* requires the implementation of a history of the previous states and rewards, which it uses to calculate the reward backwards in time. How far to look back, i.e. determine the *n* parameter, depends greatly on the structure

of the game. If long periods of states and actions are common or if the state space is large, then the n parameter could potentially also be large, although using a large number for n might cause problems. Also note that the higher the n , the more calculations are needed for each reward, so use large numbers with caution. Since the implementation of n -step should be dynamical, it should be fairly easy to vary this parameter to find a suitable value.

Eligibility Traces is independent of the number of states looking back, since it uses a separate table of eligibility to determine which and how much reward is given. In an environment with large state spaces this table would be quite larger, but since most of the calculations taking place are quite simple, the overall complexity should not pose too big a problem. Unlike the n -step which grows with the number of backtracked step, the table here remains constant in size, making the technique expensive for short amounts of backtracking, but cheaper for larger amounts. Eligibility Traces also contains the traditional parameters to be tweaked, but also introduces the λ parameter, which can be tweaked to mimic both Monte Carlo and TD. The λ determines how far the rewards are to be traced, and can be balanced to suit the needs of the game.

A final comment on the solution method is more of an implementation issue. When dealing with multiple policies, it is very important to consider the possible solutions to the problem of missing action-value updates. If a policy all of the sudden is replaced with another policy, it is very important to take care of the learning, since this could otherwise cause lost data, less learning and ultimately a poor convergence rate.

One immediate solution to this problem could be to just forget the previous states and actions when the switch is made and clear all recorded data. This might seem crude and harsh, but the important thing is to wipe the history to prevent the updating of incorrect states and actions.

Another solution is to just include the rewards given at the present time and use a default next state value for the update. A third is to make sure the switch of policies only occur after an update has happened, preventing any lost learning.

4.7 SUMMARY

We introduced our case study game called Tank General, which is an RTS game utilizing our proposed framework. The different elements in Tank General are described. Then we described how the layers in the multi layer structure was divided, and how the environment was split up into a finite state space, so the RL could be implemented. Tank General is divided into a strategic top level and a tactical level. We also describe how the rewards was determined for the different anti-strategies.

Next a description of how the *profiler* is implemented in Tank General is given. Three functions are implemented: one that determines when the enemy plays aggressive, one that determines defensive strategies and the last one that triggers when the enemy focuses on the resources.

A description of the technique of *Backtracking* and our simple single layered RL implementation was carried out, which are described in more details in following chapters.

4.7. Summary

The section General Application gives some considerations and guidelines to take into account when implementing an RTS using our proposed framework. It gives some guidelines about designing the multiple layers, in an RTS game, and what to consider when implementing the player modeling. At last some general notes on which solution method to use are given.

Chapter 5

RESULTS

In this chapter we will present the results from our tests. In the first section we describe the test setup including the methods we use to compare our tests in order to get meaningful results. After this we go through each test describing the purpose and the results of the test. We also explain the reason for the results of the tests. In the end of the chapter we will summarize our most important results.

5.1 TEST SETUP

During the test phase we have done a comprehensive amount of tests which generated a lot of data. In this section we show how we use this data to generate meaningful graphs so it is easy to interpret the tests. But first we need to describe how a test is performed and what data it yields.

A test is basically a series of games where two AI's play against each other. Typically it is an AI which learn using Reinforcement Learning against a static scripted AI which always plays in a consistent way. Normally a game takes between 9 and 25 minutes depending on the players, so running a lot of tests can take very long time. To deal with this problem we implemented a time multiplier in the game which makes it possible to run the game 20 times faster than normally. Each test consists of 500 games since this is enough to get a meaningful result with a clear tendency. Simulating 500 games takes approximately 4 to 12 hours on one computer, but fortunately we had seven computers at our disposal so time did not turn out to be a problem during testing.

As we have already described in chapter 2.1.3 on page 11, there are many parameters to balance when using SARSA and Q-learning as solution methods for Reinforcement Learning. In order to have consistent tests we ran every test with the same values. We will briefly summarize the parameters again. The stepsize parameter α decides whether to focus on recent rewards or on earlier achieved rewards. The parameter ϵ decides how often an explorational random action is chosen instead of choosing the best action exploiting what is known to be best. The discount factor γ emphasizes the next action-value. These parameters must be set to values in the range between 0 and 1. Besides this we also need to find decent values for every reward. The values for the rewards can be found in Appendix F.

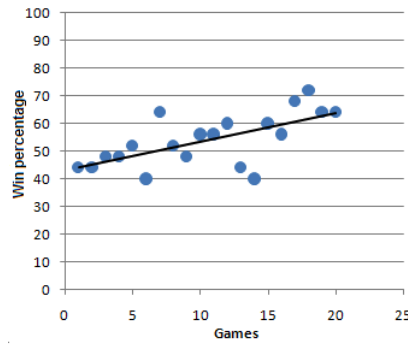


Figure 5.1: Example of a performance graph.

Finding the right values for the parameters can be very tedious and require many initial tests. But since we have used Reinforcement Learning in other projects, we have learned several guidelines to follow and that helped us a lot finding appropriate values. It is beyond the scope of this report to describe this long process, instead we will briefly summarize what we discovered.

The stepsize parameter was set to 1 because we were working in a non-stationary environment, where a high and constant stepsize value is appropriate to use. We chose 0.1 for the epsilon parameter since a balance between exploitation and exploration had to be found, and 10% random actions seemed like a good choice. The discount factor was set to 0.5 because early tries with the value 1.0 provided values too large for the program to handle, and the value of 0.5 provided fine results. All test games were played on the same map which is called SimpleAndHuge.level. This map consist of a huge island and was chosen because there are no obstacles on it implicating it is a fair map for both players to play on and because the pathfinder therefore will not take too much CPU time.

Each test simulation of 500 games yields a file with a complete history of each game containing information about who won the game and how long the game lasted. If a Reinforcement Learning computer AI participated, a policy file will also be generated for each RL system. Policy files and how to interpret policy files are described further in Appendix A. Now we will show how we turn this data into meaningful graphs.

5.1.1 PERFORMANCE

We wanted to be able to measure how much a Reinforcement Learning AI improves over time. In order to show improvement over time we came up with a graph which we call a performance graph. An example of this graph can be seen in figure 5.1.

As mentioned each test consist of 500 games. In order to draw this graph we split up these games chronologically in 20 clusters of 25 games. Then for each cluster we calculate the percentage of the games which was won by the Reinforcement Learning AI. Each dot on figure 5.1 represents 25 games where the Y-axis shows the winning percentage. But since it is not easy to compare graphs consisting of dots, we wanted to convert the dots into a linear line. This can be done by drawing the corresponding tendency line. The tendency lines are calculated using linear regression, which tries to

5.1. Test Setup

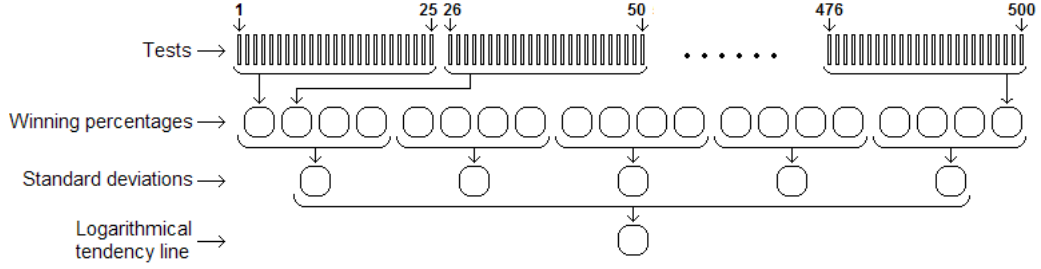


Figure 5.2: Process of calculating a convergence graph.

fit the given data with a linear function, while considering the least squares criterion [11]. The performance graphs in this chapter will only show the tendency lines, however all results behind these tendency lines can be found in Appendix F and on the attached CD-ROM.

5.1.2 RELIABILITY

In the tests we also need to show the reliability in terms of how quick a Reinforcement Learning AI converges. In order to show this we use a graph where the y-axis is the standard deviation and the x-axis is the number of games played. An example of this graph is shown in figure 5.4 on page 61 in the next section. In the following we describe how such a graph is calculated. The process is illustrated in figure 5.2.

As previously mentioned we chronologically split every test consisting of 500 games into clusters of 25 games each. Then we calculate the winning percentage of each cluster of 25 games. For example winning 20 of 25 games gives a winning percentage of 80%. Then we calculate the standard deviation for each four winning percentages. Standard deviation is a measurement of dispersion and is denoted σ . Standard deviation measures the spread of data about the mean, measured in the same units as the data. The formula for calculating the standard deviation is shown below [28].

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

N is the number of samples taken. For each value x_i , the difference $x_i - \bar{x}$ between x_i and the average value \bar{x} is calculated. The difference is squared and then the average of the squared differences is calculated. Finally the square root of σ^2 is calculated resulting in the standard deviation σ .

The more the standard deviation values descend, the more the algorithm is close to convergence. When the standard deviation is zero, the algorithm has converged in a solution. The tendency line is calculated as described in the previous section but using logarithmic regression instead of linear regression.

5.1.3 WHAT IS LEARNED

Besides measuring performance and reliability we also want to get an idea of what the Reinforcement Learning AI's have actually learned. This can be done by looking at the action-values in the policy files. Since there are many thousand states, we want to select a few critical states from certain policies in order to analyze what the Reinforcement Learning AI's has learned. This is done in section 5.7 "Examples of Learning".

We are now ready to present and analyze the results from our tests. Notice that each test consist of several subtests. The result of these subtest can be found in Appendix F. Each of these subtests have an unique number which we will refer to in the following sections when analyzing the results.

5.2 TEST 1: SARSA VERSUS Q-LEARNING

The initial test is very simple. The purpose is to find out which solution method performs best in our framework; SARSA or Q-Learning. In order to test this we first let a Multi Layered Reinforcement Learning AI based on SARSA, from now on called *Multi Layered 3-Top Levels (SARSA)*, play against 3 scripted AI's and after this we let a Multi Layered Reinforcement Learning AI based on Q-Learning, from now on called *Multi Layered 3-Top Levels (Q-Learning)* play against the same 3 scripted AI's. *Multi Layered 3-Top Levels (SARSA)* and *Multi Layered 3-Top Levels (Q-Learning)* both use a profiler to adapt to the opponent as described in chapter 3.3 on page 33, and that is why we call them 3-Top, e.g. the profiler selects one of three top levels based on how the opponent act.

The reason we do not let *Multi Layered 3-Top Levels (SARSA)* and *Multi Layered 3-Top Levels (Q-Learning)* play directly against each other is that we would not be able to see, if the reason that one of them won was that it played good or the opponent played badly. Testing in a setup where only one part learns also makes it much easier to see what is actually learned.

The three scripted AI's we use is an aggressive AI which attacks the enemy a lot, from now on called *Aggressive*, a very cautious and defensive AI from now on called *Defensive*, and an AI which has focus on capturing the resources, from now on called *Resource*. These scripted AI's and the framework, we developed in which they are scripted, is described in further details in Appendix B. These three AI's each follow a different strategy which are probably the three most common strategies in RTS games and especially in our game Tank General. As mentioned earlier in chapter 4.3 on page 44 this is the same three strategies which our profiler classifies.

Figure 5.3 shows how *Multi Layered 3-Top Levels (SARSA)* and *Multi Layered 3-Top Levels (Q-Learning)* perform against the three scripted AI's. As mentioned in chapter 2.1.3 SARSA and Q-learning are very similar, the only difference is how the action-values are updated. When updating the action-value, Q-learning always uses the best action in the following state, while SARSA waits until a second action is chosen and then it uses this action-value in the update function. Our results shows that versus *Defensive* and *Resource*, Q-learning is slightly better than SARSA. Figure 5.4 shows

5.2. Test 1: SARSA versus Q-Learning

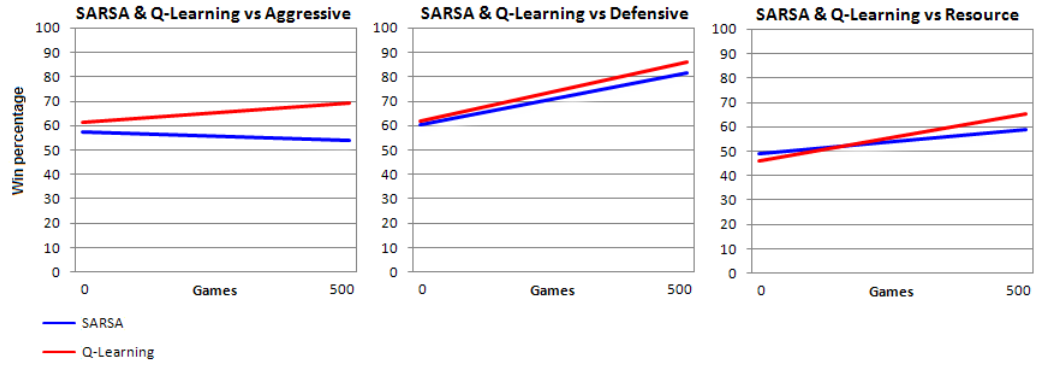


Figure 5.3: Test 1: SARSA versus Q-Learning. Comparison of subtests 1.1, 1.2, 1.4, 1.5, 1.7 and 1.8 (see appendix F).

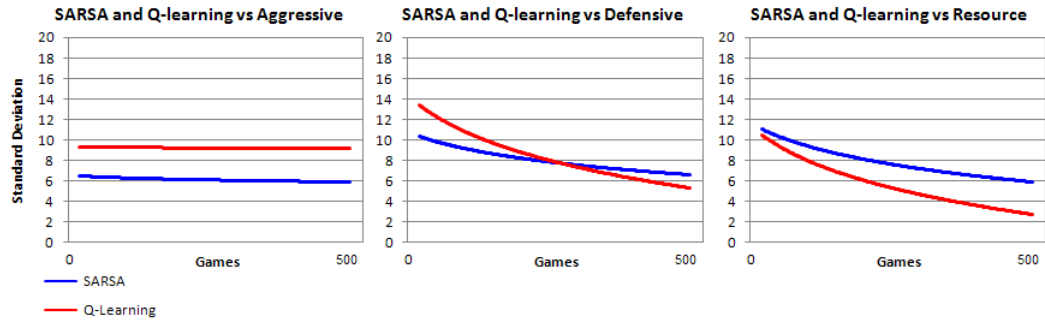


Figure 5.4: Reliability graphs for respectively SARSA and Q-learning.

that against *Defensive* and *Resource* Q-learning in fact does converge faster which conform to the theory and this is probably also the reason for the better performance. Looking at figure 5.3 against *Aggressive* the SARSA graph surprisingly has a decreasing tendency showing that SARSA does not play better over time. The reason for this is most likely that playing against *Aggressive* involves a lot of randomness. First of all the games against *Aggressive* (average 9 minutes) is much shorter than against *Resource* (average 25 minutes) which is shown in figure 5.5. *Aggressive* will attack its enemy as soon as it finds the enemy headquarters which explains why the games are quite short. Our tests have shown that if *Aggressive* quickly finds the enemy headquarters it will win most of the time, and if it does not, it will lose because the opponent will have more units. What decides how fast *Aggressive* finds the enemy headquarters is the location of its own headquarters and the enemy headquarters. Since the headquarters are placed randomly in each game, this often decides who will win and that again can be the explanation of why the performance graph is descending.

After Test 1 we conclude as expected that in our game Q-learning performs slightly better than SARSA. In the following tests we therefore use Q-learning as the solution method.

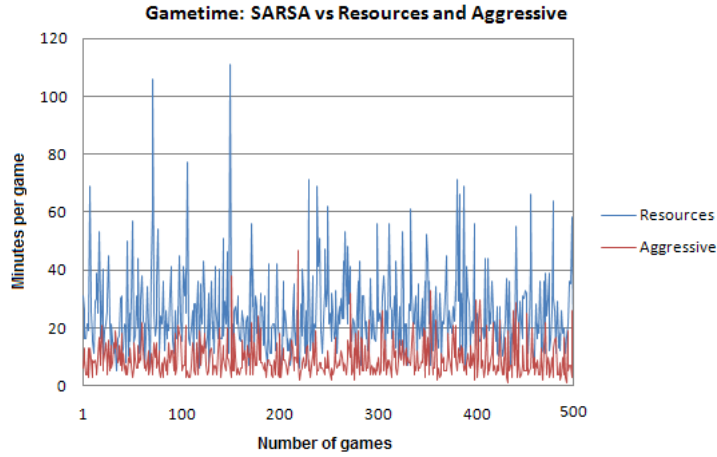


Figure 5.5: Comparison of gametime when *Multi Layered 3-Top Levels (SARSA)* plays against *Aggressive* and *Resources*.

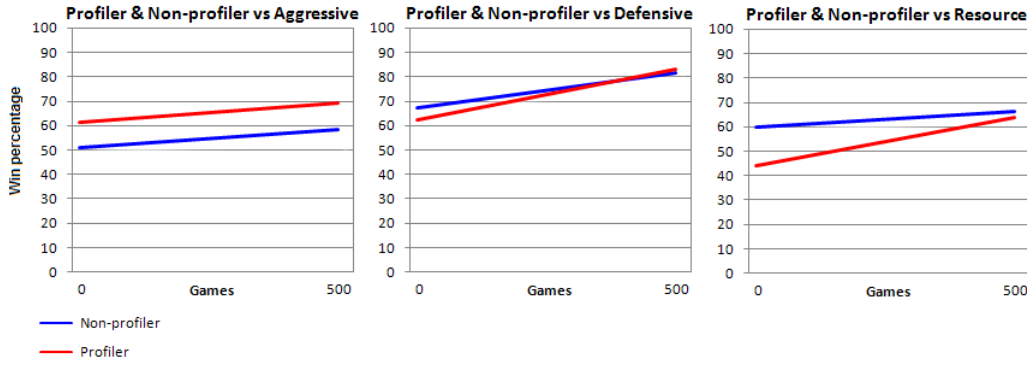


Figure 5.6: Profiling versus non-profiling. Comparison of subtests 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6.

5.3 TEST 2: PROFILING VERSUS NON-PROFILING

The purpose of the second test is to test whether profiling the opponent improves the performance of the RL AI or not. Like in the previous test, the Reinforcement Learning AI's plays against the same three scripted AI's as mentioned in the previous section. Since the previous test included profiling, we do not have to run the tests with *Multi Layered 3-Top Levels (Q-Learning)*, because we already have those results. The Reinforcement Learning AI without profiling is called *Multi Layered 1-Top Levels (Q-Learning)*, since it only has one top level which will never be changed.

The results are shown in figure 5.6. The Reinforcement Learning AI using the profiler performs clearly better against *Aggressive* and slightly better against *Defensive* while the Reinforcement Learning AI without profiling is slightly better against *Resource*.

We believe that the reason for the clearly better results against *Aggressive* (figure 5.6) is that these games are very changeable. Games starts out peacefully until *Aggressive* finds the headquarters of its opponent. From this point onwards it will send wave

5.3. Test 2: Profiling versus Non-profiling

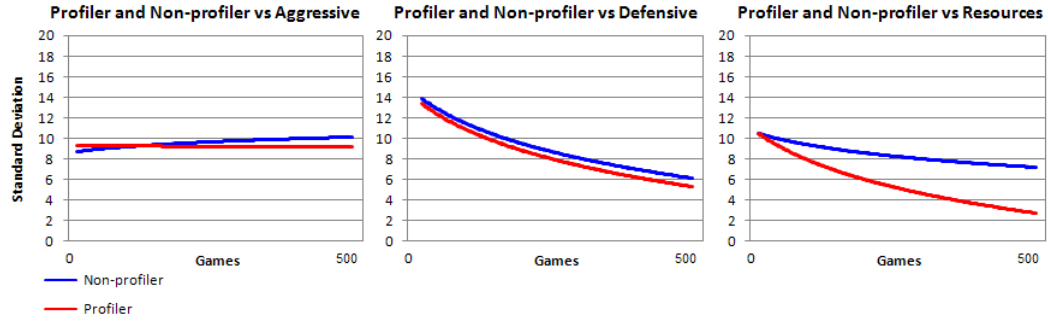


Figure 5.7: Reliability graph for respectively profiling and non-profiling.

after wave of attacks, meaning the opponent will have the experience of being under attack, not being under attack, being under attack, not being under attack etc. The Reinforcement Learning AI which uses a profiler also uses different policies when it is under attack and when it is not. This means that the Reinforcement Learning AI with profiler can adapt to the situation faster resulting in better performance.

Looking at the graphs we can see that the performance graphs against *Defensive* (figure 5.6) are most steep which is not surprising since *Defensive* is the most passive opponent. When playing against a passive opponent like *Defensive*, the Reinforcement Learning AI's have time to explore which actions are best without being punished immediately when taking suboptimal decisions. This both count for profiling and non-profiling and because of this, the profiling does not have as big an advantage as against *Aggressive*.

The last graph against *Resource* (figure 5.6) is interesting because non-profiling seems better, but actually this result was expected. After 500 games the performance of the Reinforcement Learning AI's using and not using profiling is almost equal but looking at the tendency it seems like the Reinforcement Learning AI using profiling will catch up with non-profiling Reinforcement Learning. Games against *Resource* are also very changeable since *Resource* focus on capturing the resources and defending but also occasionally attack its opponent. We believe that the reason, why the profiling AI does not perform well in the beginning, is that it has three top levels to converge which requires many games while non-profiling only has one top level to converge which is much faster. But in the long run the Reinforcement Learning AI, which has three top levels and uses a profiler, will learn at least the same as the Reinforcement Learning AI with no profiler and only one top level.

Figure 5.7 shows that the two Reinforcement Learning AI's reliability graphs against respectively *Aggressive* and *defensive* are very similar, while against *Resource* the Reinforcement Learning using the profiler converge faster than its counterpart without the profiler.

Overall it seems that profiling the opponent results in better performance and that is why we in the following tests will continue to use profiling.

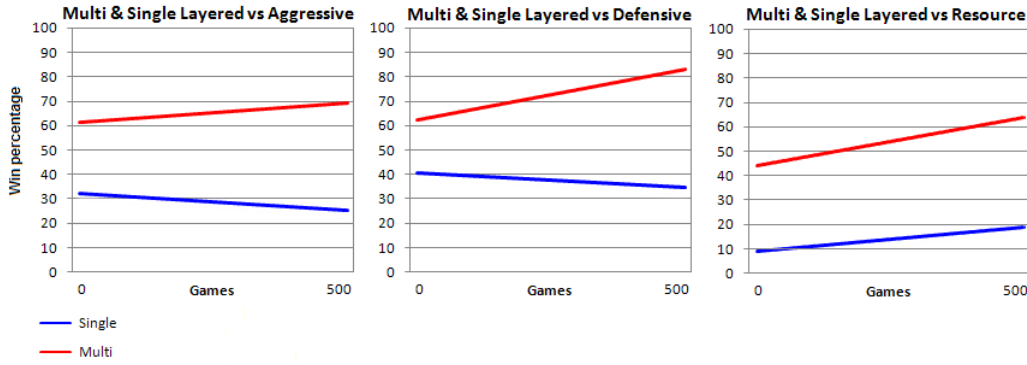


Figure 5.8: Single layered versus multi layered approach. Comparison of subtests 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6.

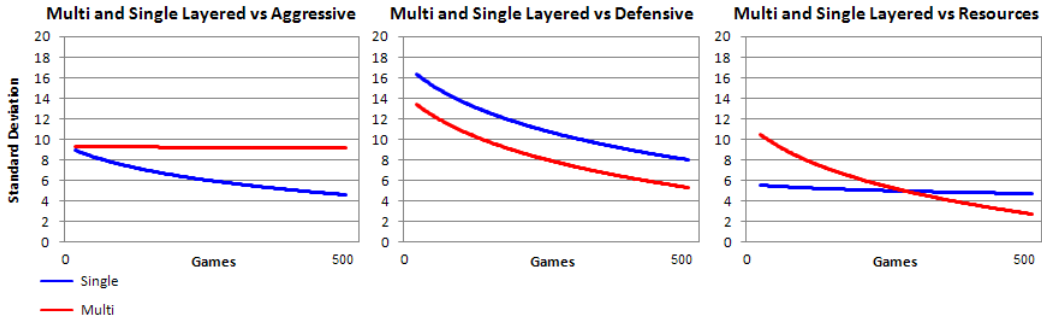


Figure 5.9: Reliability graph for respectively multi layered and single layered.

5.4 TEST 3: SINGLE LAYERED VERSUS MULTI LAYERED

In this test we want to find out if a multi layered Reinforcement Learning AI using Q-learning as solution method and using a profiler (called *Multi Layered 3-Top Levels (Q-Learning)*) performs better than the traditional Reinforcement Learning implementation with only one big policy, but also using Q-learning as solution method. In the following we call this traditional Reinforcement Learning *Single Layered (Q-Learning)*.

We originally wanted the *Single Layered (Q-Learning)* to consist of the same actions and same state attributes as its multi layered counterpart so they could compete on the same conditions. But unfortunately without the layers the state space for the *Single Layered (Q-Learning)* turned out to be too big and we had to reduce it as described in section 4.5 page 48.

As the results of test 3 shows on figure 5.8 the *Single Layered (Q-Learning)* performs very poorly compared to *Multi Layered 3-Top Levels (Q-Learning)* and this is a direct consequence of the big reduction of the state space attributes. Even though *Single Layered (Q-Learning)* have access to the same actions as *Multi Layered 3-Top Levels (Q-Learning)* the reduced state space available simply does not contain enough information for it to take reasonable actions. figure 5.8 shows that against *Aggressive* and *Defensive*

5.5. Test 4: Backtracking versus Non-backtracking

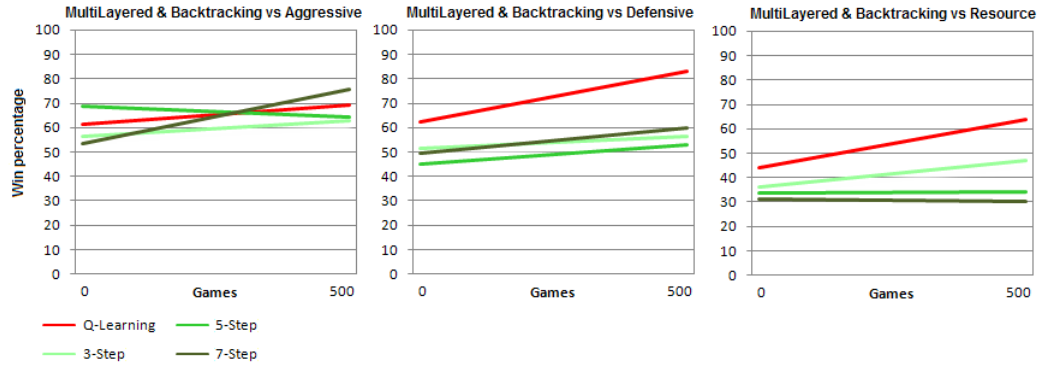


Figure 5.10: Comparison of the different backtracking algorithms.

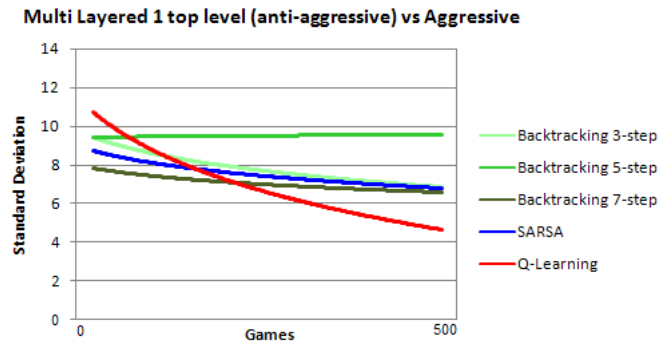


Figure 5.11: Convergence properties of different algorithms.

it does not even improve its performance over time. The tendency graph only increases against *Resource* but we believe that this might be due to statistical dispersion. Figure 5.9 shows that the single layered approach does converge but looking at figure 5.8 we can also conclude that it does converge into a solution which does not perform better which we believe is because of the reduced state space. Showing that the multi layered Reinforcement Learning AI performs better than a single layered does not mean that it also learns faster. But looking at figure 5.8 and figure 5.9 again it is clear that when the multi layered Reinforcement Learning AI improves its performance, the standard deviation simultaneously decreases which indicates that the algorithm has learned to perform better.

5.5 TEST 4: BACKTRACKING VERSUS NON-BACKTRACKING

This test investigates whether our proposed backtracking algorithm improves the AI in the game. As it can be seen in figure 5.10 and 5.11, the backtracking algorithms do not show an improvement over the non-backtracking algorithms. They do not converge as fast as Q-learning or SARSA. Furthermore it seems that the more steps they backtrack the convergence speed decreases. The explanation can either be that the theory for the

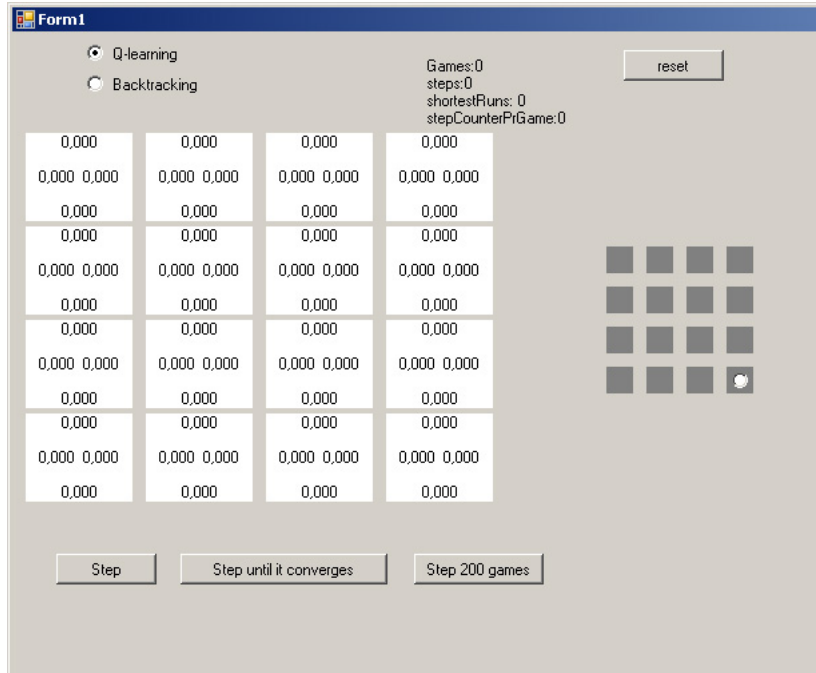


Figure 5.12: Tool developed to check different algorithms in a small world.

backtracking modification, is not working, or that something in the implementation in Tank General is wrong.

To investigate the cause, we have made a Reinforcement Learning tool in which we can test different algorithms in a small environment. With this tool it is easy to check how different algorithms performs, because the simulation is so simple, and can be computed very fast. The tool (shown in figure 5.12) have a grid, to the right, and a white dot in the lower right corner. The learning task for the RL is to get the white dot to the upper left corner square in fewest steps. Every step is given a negative reward of -1. The upper left corner triggers a reward of 0. When the RL has learned to take the shortest path a given number of times in a row, we say that it has converged to an optimal solution. Implementing both the Q-learning and the backtracking algorithm, we can now check if the backtracking shows any improvement over the Q-learning in a small and controlled environment. The first task was to confirm that implementations of the backtracking algorithm was done properly, by setting the number of steps to backtrack to 1. This means that it should perform and operate exactly as Q-learning. By checking the average number of steps over many simulations we confirmed the algorithms yield the same result.

Two tests were performed. In both tests the Q-learning algorithm was tested against the backtrack algorithm. Every simulation ran until the optimal route was learned and taken 3 times in a row. For example in a 4x4 world, the shortest route is $(4 - 1) * 2 = 6$ steps long. When the goal in this 4x4 world has been reached in 6 steps, 3 times in a row, the simulation is finished. The number of steps taken in these three shortest routes is subtracted from the total number of steps to give only the steps used to learn the route.

5.5. Test 4: Backtracking versus Non-backtracking

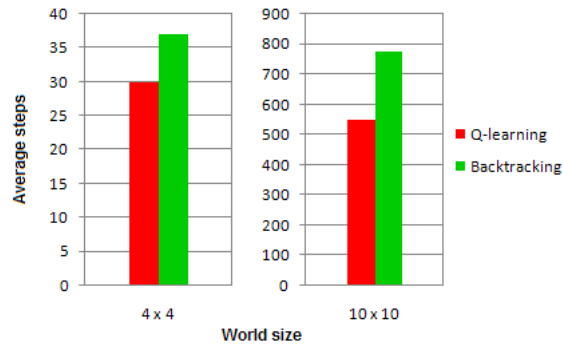


Figure 5.13: Q-learning versus Backtracking implemented with state-values, with the number of backtracking steps = 3.

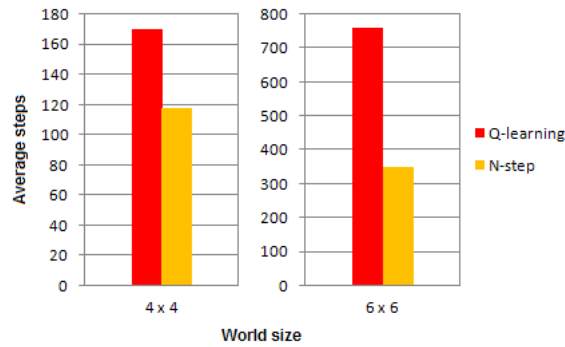


Figure 5.14: Q-learning vs. *n-Step* implemented with action-values, with the number of backtracking steps = 4.

The first test is a small 4x4 world, with the number of backtracking steps set to 3. In the second test the world is 10x10 and the number of backtracking steps is 8. Both simulations were run until the average number of steps it takes to converge has stabilized. The average is calculated simply by summing the number of steps from each simulation, and dividing by the number of times it was run.

In figure 5.13 it can be seen that Q-learning is using fewer steps to learn the optimal path. Where Q-learning is using about 30 steps in average, backtracking is using 37 steps in the small 4x4 world. In the larger 10x10 world Q-learning is also better. This tendency appears in all combinations of size and number of backtracks we have experienced with. So we can conclude that the backtracking theory is not working very well. To explain why the proposed backtracking method has failed, we wanted to make sure that this result do not apply to the *n-Step* as well. Due to this we implemented the *n-Step* method, to the tool as well.

The result is seen in figure 5.14 and shows that the *n-Step* performed much better than the normal Q-learning algorithm. One thing to note is that in figure 5.14 both the Q-learning and the eligibility algorithms are implemented with the action-value method, whereas our test for backtracking versus Q-learning is implemented using state-values. The state-values is easier to implement, but the theory for the eligibility makes an action-value implementation more appropriate. This is the reason for the different average value for Q-learning in the two tests. With the action-values the number of



Figure 5.15: The white dot is stuck in the red zone.

possible action-states to explore is much larger than in the state-value environment. In the action-value implementation a square has different values depending on the direction from where it entered, where the state-value only holds one value for each square.

This shows that while our backtracking theory resembles the *n-Step*, it actually decreases the performance compared to the standard Q-learning. By investigating the step by step values in our tool, we can see that, by updating several states back, based on the reward given, leads to problems. In some of the simulations carried out with the state-based backtracking algorithm, we noticed that after some explorative steps, the white dot got stuck in the starting area, surrounded by a "wall" of lower values. The effect is that it spends lots of steps lowering the values around it, until it can go toward the actual goal again. This is the primary cause of the lower performance.

Figure 5.15 shows an example of the problem. The white dot is caught in the red zone in the lower right part of the square. It can only move inside this red zone until these state-values have received enough negative reward to become lower than the values in the green zone surrounding the red zone.

We concluded that our backtracking theory was unsuccessful but in general the use of backtracking techniques is worth investigating.

5.6 TEST 5: FRAMEWORK VERSUS STANDARD

When we designed our test cases, this test was meant to be the final showdown, where what we call the standard Reinforcement Learning compete against the multi layered Reinforcement Learning approach based on our proposed framework, in a direct match RL vs. RL. When designing the tests we intended a fair match but since we had to reduce the state space attributes for the standard Reinforcement Learning AI the multi layered Reinforcement Learning had a clear advantage. Looking at the previous results from test 3, the result of this test shown in figure 5.16 is not very surprising. *Multi Layered 3-Top Level (Q-Learning)* is the clear winner with 355 victories of 500 games (71%).

5.7. Examples of Learning

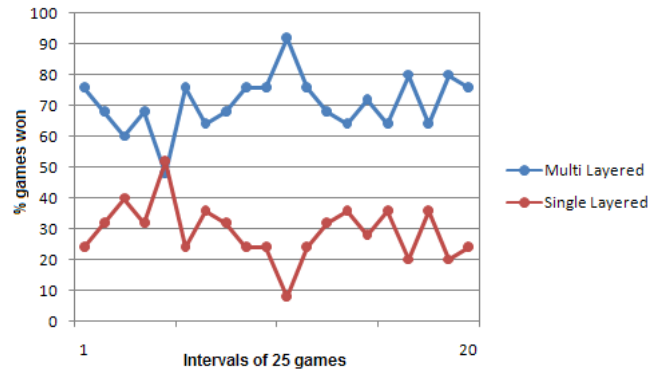


Figure 5.16: Direct fight between *Multi Layered 3-Top Level (Q-Learning)* and *Single Layered (Q-Learning)*.

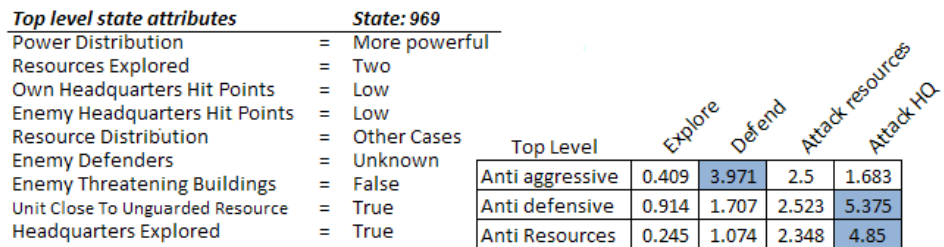


Figure 5.17: Example of action-values in top level policy files.

5.7 EXAMPLES OF LEARNING

In this section we will take a look at some of the policy files which were learned during the test phase in order to see what was actually learned after 500 games.

The first example is from multi layered Reinforcement Learning AI using Q-learning and profiling against the scripted AI called *Resource*. In figure 5.17 we compare the tree top levels in state 969. The left side of the figure shows the values for the state attributes. In this state the Reinforcement Learning AI has more units (*PowerDistribution = more powerful*), both headquarters has low hit points. Also notice that *ResourceDistribution = other cases* means that in this state at least one resource is not owned by the Reinforcement Learning AI and that the Reinforcement Learning AI has a unit close to an unguarded resource which it does not own. The right side of figure 5.17 shows the actual action-values in the tree top levels which each has different rewards according to the strategy they counter. The anti aggressive top level, which counter an aggressive opponent, has learned that it is best to defend its own headquarters, probably because it has few hit points left. Because the enemy headquarters is also low on hit points the anti defensive, which counter a defensive opponent, on the other hand has learned that the best action is to attack the enemy headquarters instead of defending. The last top level which has focus on capturing the resources has also learned that the best action is to attack the enemy headquarters and the second best action is to capture the unguarded resource.

<i>Low level defend state attributes</i>		<i>State: 1495</i>	<i>State: 1134</i>	<i>State: 738</i>
Power Distribution At Own Headquarters	=	Less powerful	More powerful	More powerful
Total Power Distribution at Own HQ	=	More powerful	More powerful	More powerful
Power Distribution At Resource 1	=	Less powerful	More powerful	Less powerful
Power Distribution At Resource 2	=	Less powerful	More powerful	More powerful
Own Headquarters Hit Points	=	High	Low	High
Reinforcements Available	=	False	True	False
Resource Distribution	=	Other Cases	Both owned	Both owned

	Build unit	Repair headquarters	Send 25% to HQ	Send 50% to HQ	Send 100% to HQ	Send 25% to resource	Send 50% to resource	Send 100% to resource
State 1495	NA	NA	2.5	3.083	4.95	3.083	2.85	3.333
State 1134	5	5.7	5	5.25	4.391	2.546	5.25	5.25
State 738	NA	NA	0.910	0.968	0.985	1.013	2.372	2.194

Figure 5.18: Example of action-values in a low level defend policy file.

The next example is taken from the low level defend policy file and in this example we compare tree different states. The values of the state attributes and there respective action-values are shown in figure 5.18.

State 1495 is a state where the Reinforcement Learning AI must be under attack since *PowerDistributionAtOwnHeadquarters* = *Less Powerful*. Notice that *TotalPowerDistributionAtOwnHeadquarters* = *more Powerful* means that all the units owned by the Reinforcement Learning AI (including units which is not close to its headquarters) are more powerful than the enemy units which currently are close to the headquarters. Looking at the action-value for this state we can see that the Reinforcement Learning AI has learned that the best action would be to send all its units back to its headquarters which makes perfect sense since the headquarters is under attack. We should also mention that the actions *Build unit* and *Repair headquarters* are not available because there are no reinforcements available in this state (*ReinforcementsAvailable* = *False*).

State 1134 is a state where the Reinforcement Learning AI has everything under control. It has the most units and it also possess both resources. The headquarters has taken some damage (*OwnHeadquartersHitPoints* = *Low*) but fortunately one or more reinforcements are available. Looking at the action-values we can see that the best action learned for this state is to use the available reinforcement points to repair the headquarters.

In state 738 the Reinforcement Learning AI is more powerful while it also possess both resources. Unfortunately resource 1 is under attack since the value of the state attribute *PowerDistributionAtResource1* = *Less powerful*. Looking at the corresponding action-value the best action is to send either 50% or 100% of the units to resource 1 in order to defend it.

5.8 SUMMARY

In the following section we will summarize the most important results regarding our proposed framework.

5.8. Summary

The results clearly show that using the *multi layered* approach was a big advantage. When dealing with a lot of state space attributes, as it is the case in our game, a multi layered approach will reduce the state space considerable. This reduction is an advantage because many state space attributes normally results in an enormous number of states which again results in a very slow learning rate because of the many states which need to be visited.

The tests have shown that there are both pros and cons regarding *profiling*. It might not always be an advantage to use profiling since learning can take more time because there are more policies which need to be filled. Dependent on the opponent another clear disadvantage can be the case where the profiler is not able to make a specific classification. However in our tests we did not encounter this problem since our game has a limited number of good strategies which our profiler was designed to profile against. In our game, profiling performed better in most cases and the reason was the ability of the learning algorithm to quickly change behavior and react when the opponent change behavior.

Generally our proposed *backtracking* method did not generate the results we hoped for. After we discovered that the backtracking did not improve the AI in our game, we created a tool to highlight the causes of the poor performance. We concluded that it was statistically sufficient to summarize the history based on the previous state, and trying to update more steps back seemed to mess with the updated values. We implemented the *n-step* algorithm to identify whether it was superior to Q-learning which the theory states that it should be. It appeared that this was the case, so we concluded that our proposed backtracking algorithm did not give the improvement we expected it to provide.

Chapter 6

CONCLUSION

The goal of this project was to investigate the feasibility of monitoring human opponents' strategy, and then counter it by swapping predefined sets of policies and rewards, where these are updated using Reinforcement Learning.

We proposed three extensions of standard RL to improve the Reinforcement Learning structure. We proposed a simple and effective *multi layer structure*, which together with our player modeling technique called *profiler*, makes it easy to swap the current set of rewards and policy currently used by the Reinforcement Learning AI. We also proposed a modification to the update function of RL algorithms which we call *backtracking*, which updates several steps back, instead of one step as *TD(0)* algorithms do.

These proposed extensions constitute a framework, which can be applied to any RTS game. To verify that our proposals do in fact work, in a practical setting, we have applied the proposed framework to the RTS game *Tank General*, which mainly was developed in the preparation semester. This provided us with a solid foundation to test the framework in.

Several types of tests were carried out to check the performance of the framework. Some tests showed the performance of the RL algorithms, others the reliability from which we could see how fast the algorithms converged.

First the proposed *profiler* was tested. Three different kinds of scripted AI's played against an RL AI with and without a profiler. This test showed that using the profiler gave an advantage over the RL AI without a profiler.

To test the proposed *multi layer structure*, several tests were performed. We let a standard single layered RL AI play against the three scripted AI's. Then the RL AI, which uses a multi layered structure, played against the same scripted AI's. The multi layered RL AI beat the standard single layered RL AI, but that result was not entirely caused by our multi layered structure. The single layered AI is crippled, because of the necessary reduction of states. This means that the RL AI's do not play on even conditions. The test showed one of the strengths of our proposed multi layered structure which is the ability to maintain a proper level of performance with a reduced state space.

To test the proposed *backtracking* algorithm we tested standard Q-learning against a 3-step, 5-step and 7-step version of our backtracking algorithm. The test showed that

Q-learning converged faster, and also won more games against the three scripted AI's compared to the backtracking versions. The backtracking algorithm did not improve the performance of the RL AI in the test. To learn why, we created an RL tool to highlight the causes of the poor performance. After testing several RL algorithms in the tool including the *n-step* algorithm which resembles the backtracking, we concluded that it was statistically sufficient to summarize the history based on the previous state. But we confirmed in a practical setting that an implementation of the *n-step* actually does improve the *TD* methods.

While the backtracking do not give a performance boost, the multi layered structure and the profiler performs very well in an RTS game. Together they reduces the state space drastically, while reducing the time it takes for the RL algorithms to converge. Therefore the conclusion of this thesis regarding our project goal is that it definitely is feasible to use our proposed framework in an RTS game.

Bibliography Listing

- [1] GIWEBB & ASSOCIATES 1999-2005. M-estimates.
<http://www.rulequest.com/MOestimate.html>, 2005.
- [2] Kresten Toftgaard Andersen, Jens Peter Berth, Dennis Dahl Christensen, Jimmy Marcus Larsen, and Quoc Dung Tran. *Murder of Crows - Emergence and Learning*. AAU, 2007.
- [3] Kresten Toftgaard Andersen, Dennis Dahl Christensen, and Quoc Dung Tran. *Tank General - Reinforcement Learning in an RTS game*. AAU, 2008.
- [4] Gustavo Andrade, Geber Ramalho, Hugo Santana, and Vincent Corruble. Automatic computer game balancing: A reinforcement learning approach, 2005.
- [5] Christian Baekkelund. *A Brief Comparison of Machine Learning Methods*. AI Game Programming Wisdom 3, 2006.
- [6] Staffan Björk and Jussi Holopainen. Games and design patterns. *The Game Design Reader: A Rules of Play Anthology*, 2006.
- [7] Yngi Björnsson, Vignir Hafsteinsson, Ársæll Jóhannsson, and Einar Jónsson. Efficient use of reinforcement learning in a computer game, 2004.
- [8] Darryl Charles and Michaela Black. Dynamic player modeling: A framework for player-centered digital games.
- [9] Blizzard Entertainment. Warcraft. <http://www.blizzard.com>, 1994.
- [10] Lindsay Fleay. Rts basics. http://www.rakrent.com/rtsc/rtsc_basics.htm, 2007.
- [11] Inc. GraphPad Software. A complete guide to nonlinear regression.
http://www.curvefit.com/linear_regression.htm, 1999.
- [12] Bernhard Hengst. Discovering hierarchy in reinforcement learning, 2003.
- [13] Jørn Holm and Jens Dalgaard Nielsen. Genetic programming - applied to a real time game domain. 2002.
- [14] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Verlag, 2007.
- [15] Raymond Kurzweil. The law of accelerating returns. 2001.

- [16] John E. Laird. *Bridging the Gab between Developers and Research*. (*Game Developer Magazine August 2000*). Game Developer Magazine (August 2000), 2000.
- [17] John E. Laird and Michael van Lent. Machine learning for computer games. 2005.
- [18] Michael Littman. Pomdp information page, 1997.
- [19] Bhaskara Marthi, Stuart Russel, David Latham, and Carlos Guestrin. Concurrent hierarchical reinforcement learning, 2005.
- [20] Sid Meier. Civilization series.
http://en.wikipedia.org/wiki/Civilization_%28series%29, 1991.
- [21] Amit J. Patel. A-Star comparison.
<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>, 2007.
- [22] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002.
- [23] David Silver, Richard Summon, and Martin Müller. Reinforcement learning of local shape in the game of go, 2007.
- [24] Lionhead Studios. Black & White. <http://www.lionhead.com/bw/index.html>, 2001.
- [25] Richard S. Sutton and Andrew G. Barton. *Reinforcement Learning - An introduction*. 2002.
- [26] Valve. Counter-Strike. <http://www.valvesoftware.com/>, 1999.
- [27] Wikipedia. Supervised learning.
http://en.wikipedia.org/wiki/Supervised_learning, 2007.
- [28] Wikipedia. Standard deviation.
http://en.wikipedia.org/wiki/Standard_deviation, 2008.
- [29] Florentin Woergoetter and Bernd Porr. Reinforcement learning.
http://www.scholarpedia.org/article/Reinforcement_learning, 2006.
- [30] Steven Woodcock. *Game AI: The State of the Industry*. Game Developer Magazine (August 2000), 2000.
- [31] Steven M. Woodcock. Games making interesting use of artificial intelligence techniques. <http://www.gameai.com/games.html#CIV>, 2008.
- [32] Steven M. Woodcock. Games making interesting use of artificial intelligence techniques. <http://www.gameai.com/games.html#CREATURES>, 2008.

Chapter 7

APPENDICES

Appendix A

EXAMPLE OF A POLICY FILE

Figure A.1 shows a *low level attack resource* policy file learned after *Multi Layered 3-Top Levels (SARSA)* has played 500 games against the scripted AI *Resource*. We have chosen the *low level attack resource* policy file because it is the smallest policy file with only 5 state space attributes and 5 actions. In comparison a *top level* policy file for example contains 9 state space attributes and 4 actions and a *low level defend* policy file contains 7 state space attributes and 8 actions.

In the following we explain what we can read from the policy shown in figure A.1. Each line represents a state. Each state is represented by a number and an explanation of what is going on in this state. This can be seen to the left of the policy file. To the right we can see the action-values. Since we use optimistic initial values, all action values are 10 from the beginning. That is the actions which have never been taken. We should also mention that the rightmost column shows how many times each state has occurred.

Looking at the policy file we can see that the AI has learned the following in state 80. When it has no reinforcements available ($RA = f$) and it does not have any units close to resource 1 ($PD1 = u$) and it has more units close to resource 2 than the enemy ($PD2 = m$) and resource 1 is not owned by the enemy ($R1E = f$) and resource 2 is owned by the enemy ($R2E = t$) the best action is to capture the resource with one unit ($SingleAtt1 = 4,454$) which logically makes sense.

A. Example of a policy file

Low Level Attack Resource (states 31, actions: 5)
Number of games played using this policy: 500

	RA	PD1	PD2	R1E	R2E		Build	SendSuff1	SendSuff2	SingleAtt1	SingleAtt2	#
State 0:	f	u	u	f	f		10	0,386	0,253	0,016	0,492	1369
State 34:	f	l	u	t	f		10	0,005	0,001	0,031	0,031	195
State 35:	t	l	u	t	f		0,020	1,703	10	0,326	10	31
State 36:	f	m	u	t	f		10	2,246	0,974	-0,03	0,009	255
State 37:	t	m	u	t	f		-0,24	-0,98	10	0,525	10	26
State 38:	f	le	u	t	f		10	0,012	1,499	0,081	2,384	178
State 39:	t	le	u	t	f		0,001	0,000	10	0,001	0,080	65
State 72:	f	u	l	f	t		10	0,000	0,358	8,248	0,001	207
State 73:	t	u	l	f	t		0,009	10	0,306	10	2,531	51
State 80:	f	u	m	f	t		10	0,000	1,265	-0,32	4,454	231
State 81:	t	u	m	f	t		0,050	0,084	0,012	0,023	3,101	64
State 88:	f	u	le	f	t		10	0,015	-0,89	0,281	0,153	212
State 89:	t	u	le	f	t		0,816	10	-0,22	10	-0,51	69
State 106:	f	l	l	t	t		10	0,002	2,146	0,000	0,002	1700
State 107:	t	l	l	t	t		2,292	0,005	0,824	0,762	1,260	719
State 108:	f	m	l	t	t		10	3,007	0,000	0,049	0,002	1297
State 109:	t	m	l	t	t		-0,02	2,430	0,022	1,162	0,412	755
State 110:	f	le	l	t	t		10	0,005	-0,66	-0,12	0,513	1397
State 111:	t	le	l	t	t		0,000	-0,89	0,007	-0,00	0,169	704
State 114:	f	l	m	t	t		10	1,436	3,685	1,510	0,140	1882
State 115:	t	l	m	t	t		0,181	2,871	0,049	0,443	0,131	967
State 116:	f	m	m	t	t		10	0,061	0,136	0,020	2,207	2612
State 117:	t	m	m	t	t		0,010	-0,24	0,003	0,812	3,000	993
State 118:	f	le	m	t	t		10	0,002	0,032	0,000	-0,92	2738
State 119:	t	le	m	t	t		0,000	7,298	2,190	0,003	0,094	1203
State 122:	f	l	le	t	t		10	0,000	0,376	0,000	0,001	1246
State 123:	t	l	le	t	t		9,155	0,001	0,002	9,155	0,190	606
State 124:	f	m	le	t	t		10	0,124	0,000	3,003	0,000	1493
State 125:	t	m	le	t	t		0,000	-0,20	5,568	0,250	0,169	900
State 126:	f	le	le	t	t		10	-0,24	1,758	0,001	-0,02	2655
State 127:	t	le	le	t	t		0,000	-0,98	-0,11	0,295	-0,12	1509

Space state explanation:

RA = Reinforcements Available
PD1 = Power Distribution at resource 1
PD2 = Power Distribution at resource 2
R1E = Resource 1 owned by Enemy
R2E = Resource 2 owned by Enemy

Values:

(t, f)
(u, m, l, le)
(u, m, l, le)
(t, f)
(t, f)

Value explanation:

t = true
f = false
u = unknown
m = more powerfull, the player have more units nearby the resource than the enemy
l = less powerfull, the player have less units nearby the resource than the enemy
le = lesser powerfull, the enemy has at least 2 times as many units nearby the resource than the player

Action explanation:

Build = Build new unit.
SendSuff1 = Send sufficient units to attack resource 1
SendSuff2 = Send sufficient units to attack resource 2
SingleAtt1 = Send single unit to attack resource 1
SingleAtt2 = Send single unit to attack resource 2
= The number of times the AI has been in this state

Figure A.1: A low level attack resource policy file.

Appendix B

AI SCRIPTING FRAMEWORK

We will in this appendix describe a framework, which we created with the purpose of simplifying the task of scripting AI's. Furthermore we wanted a framework which restricted the AI's on acquiring information on other teams or altering critical data, leading to a potential game crash.

In order to fulfill these requirements we had three keywords in mind during the implementation of our scripting AI framework.

- Consistency
- Anti-cheat
- Fairness

These will be described in the following subsections.

B.1 CONSISTENCY

One of the primary reasons for our choice of spending time on this framework was to ensure that the game would not suffer from data inconsistency because of the AI's. The consequences of this problem increased the risk of crashing the game due to altering of critical data, usage of incorrect data etc. In order to avoid this unacceptable error we decided to create a framework for scripting the AI's.

Figure B.1 illustrates the structure of our scripting framework.

The class *Game1* initializes everything in the game, which includes all AI related classes. One of them is the abstract class *ComputerAI* from which all AI classes inherits. From *ComputerAI* two types of AI's can be created, either the class *ScriptedAI* or the class *AbstractRL*. The latter is further branched in the classes *StandardRL* and *MultilayeredRL*, but due to the unimportance of these classes in this section, these will not be described further. The class *ScriptedAI* represents the actual scripted AI classes

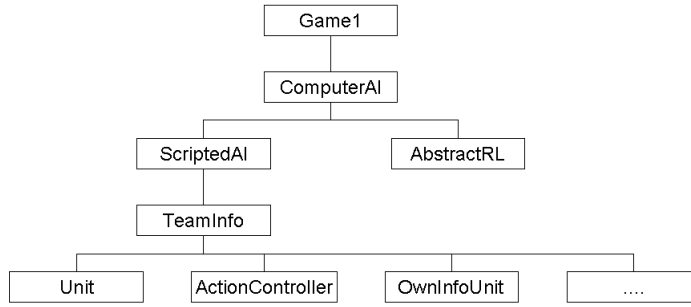


Figure B.1: A framework that simplifies the task of scripting an AI.

implemented by us. We have several AI's, that we created for test purposes, all behaving differently but they have in common that they inherit the same abstract class, namely *ComputerAI*. The class *ComputerAI* has an object called *TeamInfo*, which we created to restrict the AI developers from critical game data. An example of what critical game data might be is the value holding how many available units that one team is able to build. If this value was free to alter, it could be altered to an invalid value, causing the game to crash when the value would be used. The class *TeamInfo* pretty much says it all. It is basically a container class, which contains all the relevant information on a team, that a developer needs when implementing an AI. Some examples of what this class contains are:

- **team:** this variable might be the most important variable, as this is a unique identifier on this team. When a developer needs to find out whether a resource is owned by this team, then this variable is being checked on.
- **OwnInfoUnit:** before this object can be described, another object has to be described and that is the *Unit* object. The unit object is a class containing all the information on a single unit, for example health, position, speed, range, type and a lot more. But only a portion of this information is relevant for the developer and more importantly, in some cases the developers do not even have permission to alter these variables, but only read them. Therefore we decided to create the *OwnInfoUnit* class, which accomplished the restrictions, and provided the developer with only the necessary information to implement an AI. For example the health of a unit can be altered if the variable was taken from the *unit* class. But the developer does not have access to the unit class anymore but only the *OwnInfoUnit* class. This class ensures that permissions to alter critical game data is being respected. So the developer would access the health of the unit through the *OwnInfoUnit* class, which is a read-only variable, thereby respecting the critical game data. We also created the counterpart for enemy units and called it *EnemyInfoUnit*.
- **unitList:** the last thing we will describe is a list containing all the units of a team, called *unitList*. At the beginning of the programming phase we had a list containing all the units. This turned out to be a bad idea, as it violated the concept of this framework. And since we have a *TeamInfo* object for each team, it was natural to divide the old unit list into two lists and insert these into the *TeamInfo* objects. By doing this, we prevented the developers from accessing

B.2. Anti-cheat

information on the other team's units, and denied them access to critical game data.

There are a lot more information in the *TeamInfo* class. But we only highlighted the most important ones just to clarify that one of the purposes of this class is to prevent developers to accidentally alter critical game data that might cause the game to crash.

B.2 ANTI-CHEAT

As it appears from figure B.1, the class *TeamInfo* has relations with classes like *Unit*, *ActionController* etc. We mentioned before, that the *Unit* class contains all the information for a single unit. This information was carefully selected according to the level of permission, and wrapped into the *OwnInfoUnit* class. The reason for this was to deny developers from accessing critical game data and accidentally make the game crash. But this is not the only reason for why we chose this architecture.

Before we implemented the *TeamInfo* and *OwnInfoUnit* classes, almost all unit handling during runtime was dealt by a class called *GameLogic*. This class had access to information regarding units, buildings, resources, map details etc. And when we would script an AI we would access a lot of these information through the *GameLogic* class. That meant that developers implementing an AI for one team could get hold of information on the other team and visa versa. Due to this problem, a developer that was scripting an AI for team A could easily obtain detailed information of team B through the *GameLogic* class: information like the location on the enemy headquarters, location on the resources or how many units the enemy had, their locations and health etc. That would give the "cheating" developer an enormous advantage, making team A somewhat near invincible, because he would have all the information.

We knew we would run a lot of tests to analyze whether our multi layered AI, using Reinforcement Learning, could play against scripted AI's and end up with a positive learning curve. In order to optimize the validity of these tests we had to ensure that the scripted AI's that we would set up against our Reinforcement Learning AI did not cheat or have any advantage at all. Otherwise we would risk analyzing tests that were "incorrect" due to this injustice.

B.3 FAIRNESS

The last subject we had in mind has been indirectly described and explained in the previous two subjects, *consistency* and *anti-cheat*. We wanted all AI's to have the same starting conditions and during runtime, all AI should have the same possibilities to improve their strategic position in the game. This could be achieved by capturing the resources to be able to build units faster than the opponent. We have been able to achieve this level of fairness through the *TeamInfo* class, which provides the AI's with exactly the same methods and information.

The scripted AI's and our Reinforcement Learning AI have access to the same methods

and information through the *TeamInfo* class, but the management of units is different and this can give the scripted AI an advantage. The units of the Reinforcement Learning AI is handled through the *ActionController* class, which is a class that contains predefined methods representing actions for handling units. Some of these actions are:

- **ExploreRandomPosition:** when taking this action, all the recons will explore a random unexplored position on the map. If no recons exist, then a random unit will be exploring an unexplored position.
- **SendToHeadquarters_50:** when taking this action, 50% of the total units will move back to the headquarters and defend it. We have similar actions with 25% and 100%, doing exactly the same thing.
- **CoordinatedAttack_100:** when taking this action, all units will meet at a specific point near the enemy headquarters. When all units have arrived at the meeting point, they will rush the enemy headquarters.

These are some of the available actions in the *ActionController* class, which can be used by all AI's. This class was implemented with the purpose of handling units at a commander level and not unit level, which was required by our Reinforcement Learning AI in order to avoid micromanagement. Furthermore it simplifies the handling of units, for example if an AI wanted to defend the headquarters with all its units, it could simply take the *SendToHeadquarters_100* action instead of manually give the same action to all its units.

B.4 TEST AI'S

The previous section described our scripting framework and the reasons why we chose to spend time on it. In this section we will describe three actual AI's that we scripted, with the purpose of testing how our Reinforcement Learning AI performs against the scripted ones.

We decided on three test AI's that we would set up to play against our Reinforcement Learning AI. In real time strategy games the winning conditions are often to destroy all your enemies, protect something or someone for a given amount of time or harvest a certain amount of a specific resource. From these winning conditions we are able to extract three different styles of play, which are:

- Aggressive
- Defensive
- Resource

The reasons we chose these three strategies were because we wanted to be able to test and measure our Reinforcement Learning AI against different playing styles and because they are traditional strategies which have been proved successful in other RTS games. These strategies also turned out to be very successful in Tank General.

B.4.1 AGGRESSIVE

A popular strategy, seen in games like Warcraft [9], called *rushing* is a very aggressive style of play. This strategy often takes place at the beginning of the game, where the enemy defense is weak. One of the purposes of this strategy is to quickly eliminate the enemy, trying to win the game fast, and forcing the enemy to stay home defending the base. The aggressive AI attacks the enemy headquarters as soon as it is discovered. This attack then continues until one team loses its headquarters. When the aggressive AI has not discovered the enemy headquarters, it explores the map and captures resources when any in sight. With this behavior we ensure that our Reinforcement Learning AI will be attacked quite frequently, leaving a lot of defending tasks.

B.4.2 DEFENSIVE

The defensive AI has as a primary task to defend its headquarters. Even though the main focus is defending the headquarters the defensive AI also explores the map with recon and captures resources when any in sight. Every 5th minute it gathers a small group of units and attacks the enemy headquarters. In order to win, the Reinforcement Learning AI has to build a strong army in order to break through the enemy defense.

B.4.3 RESOURCE

The resource AI has as a primary task of protecting its headquarters. But it also prioritizes the tasks of capturing and protecting the resources very high. This moves the combat area to the resources instead of only the headquarters. In order to win against the resource AI, the Reinforcement Learning AI has to win the fight for the resources or maybe win with an early attack before the resource AI gets the advantage of capturing the resources.

Appendix C

REINFORCEMENT LEARNING

In this appendix we will investigate the learning approach called Reinforcement Learning. RL is a promising approach to the learning algorithms, as it is somewhat easy to design and program. The hard part is to adjust the algorithm to fit the game, but this will be discussed in detail later. This section is written using [25] and is a copy from our previous report [3]. An introductory chapter, a list of the elements of RL, and the description of Temporal Difference is already mentioned in chapter 2.1, and will therefore not be mentioned here.

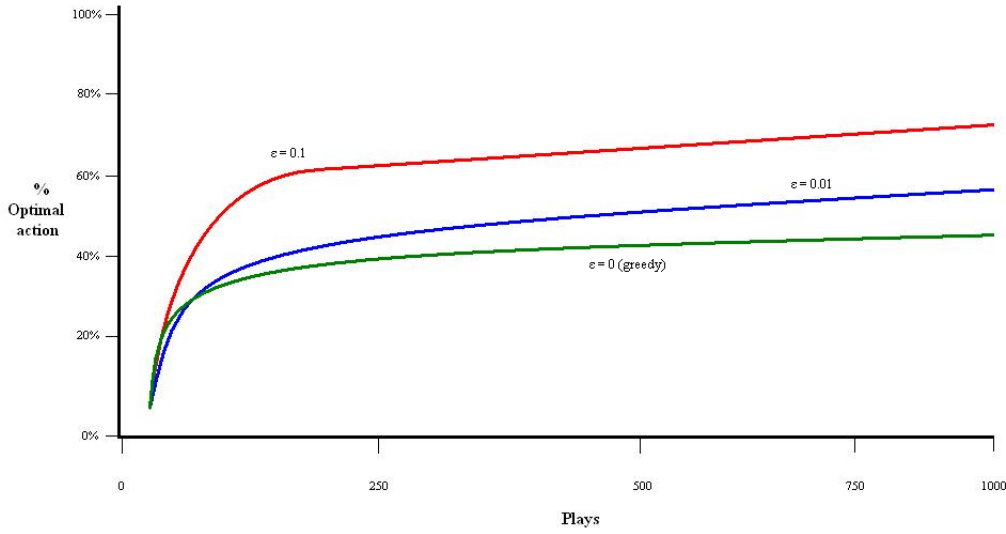
C.1 ACTION-VALUE METHOD

Action-value is a simple method to predict the values of actions, and to use the predicted values to take decisions regarding which action should be taken in a specific state. First we define the correct value for an action a as $Q^*(a)$, and the expected value at the t 'th game $Q_t(a)$. The correct value of an action is the average expected reward when taking the action and using the *sample-average method*. A straightforward method to predict this is to calculate the average of the received rewards, when the action previously have been taken. In other words, if the action a has been taken k_a times before in the t 'th game, where the rewards r_1, r_2, \dots, r_{k_a} were given, the value of the action will be predicted to:

$$Q_t(a) = \frac{r_1 + r_2 + \dots + r_{k_a}}{k_a}$$

If $k_a = 0$, $Q_t(a)$ is defined to a standard value like 0. When $k_a \rightarrow \infty$, $Q_t(a)$ converges toward $Q^*(a)$. This is called the sample-average method to predict action-values, since each prediction is a simple average of the previously obtained values. This is just one way to predict values, and not necessarily the best. Next we will show how these action-values are used to choose an action in a state.

The simplest method to choose an action would be to choose the action with the greatest action-value, e.g. choosing the greedy action a at game t where $Q_t(a) = \max_a Q_t(a)$. This method will always exploit the known data to maximize the immediate reward. It does not consider choosing other actions yielding predictively lower reward. A simple alternative would be to be greedy most of the time, but sometimes with probability


 Figure C.1: Graph of ϵ -greedy methods.

ϵ choose a random action. This approach is called ϵ -method which satisfies the exploration part of RL in a simple and straightforward way.

As seen on the graphs in figure C.1, the ϵ -greedy method yields a much better result than a purely greedy method. The greedy method improves a lot faster but is caught in a suboptimal action with a low reward. The ϵ -greedy method spends longer time improving themselves but yields a better result over time, because it explores actions giving a low immediate reward but a higher reward over time.

C.1.1 INCREMENTAL IMPLEMENTATION

The action-value methods looked upon all use the average of the future rewards to calculate the action-values. This requires saving all previously experienced rewards from each action and state. To repeat the formula, the value of an action a in game t is $Q_t(a) = \frac{r_1 + r_2 + \dots + r_{k_a}}{k_a}$, where $r_1 + r_2 + \dots + r_{k_a}$ are previous rewards. This calculation is cumbersome when dealing with late-game actions, and it will also take up quite a lot of memory. In a game, this is not acceptable

Fortunately this is not necessary, and it is quite easy to make a formula of lesser time- and space-complexity, while yielding the same result. If Q_k is the average of the previous rewards for the action a the first k times, and r_{k+1} is the amount of reward the $k + 1$ 'th game, the formula can be rewritten like this:

$$\begin{aligned}
 Q_{k+1} &= \frac{1}{k+1} \sum_{i=0}^{k+1} r_i \\
 &= \frac{1}{k+1} (r_{k+1} + \sum_{i=0}^k r_i) \\
 &= \frac{1}{k+1} (r_{k+1} + kQ_k + Q_k - Q_k)
 \end{aligned}$$

C.1. Action-Value Method

$$\begin{aligned} &= \frac{1}{k+1}(r_{k+1} + (k+1)Q_k - Q_k) \\ &= Q_k + \frac{1}{k+1}(r_{k+1} - Q_k) \end{aligned}$$

This formula is true for all k requiring only Q_k and k to be saved, and the bottom calculation to be executed. The general formula for this is:

$$\text{NewEstimate} = \text{OldEstimate} + \text{StepSize}(\text{Target} - \text{OldEstimate})$$

The expression $(\text{Target} - \text{OldEstimate})$ is an error of the estimate. This is reduced by taking a step toward the "target", which is the reward of the action in the $k+1$ step. This only updates the value if the new reward is different from the reward at the step k resulting in less calculations too.

C.1.2 EXPONENTIAL, RECENCY-WEIGHTED AVERAGE

The sample average method is not the only method to calculate action values. Another method called *exponential, recency-weighted average* is focused on the stepsize parameter. Instead of weighting the rewards equally, this method puts more weight on the recent values and less on the old ones. If we look at this formula: $Q_k + \frac{1}{k+1}(r_{k+1} - Q_k)$ from the sample-average method, we can see that $\frac{1}{k+1}$ is the stepsize here. Let us replace $\frac{1}{k+1}$ with α , where α is the constant stepsize parameter $0 < \alpha \leq 1$. Now we are ready to rewrite the formula:

$$\begin{aligned} Q_k &= Q_{k-1} + \alpha(r_k - Q_{k-1}) \\ &= \alpha r_k + (1 - \alpha)Q_{k-1} \\ &= \alpha r_k + (1 - \alpha)\alpha r_{k-1} + (1 - \alpha)^2 Q_{k-2} \\ &= \alpha r_k + (1 - \alpha)\alpha r_{k-1} + (1 - \alpha)^2 \alpha r_{k-2} + \dots (1 - \alpha)^{k-1} \alpha r_1 + (1 - \alpha)^k Q_0 \end{aligned}$$

This is called a weighted average, because the weight $(1 - \alpha)^{k-i}$, which is multiplied onto the reward, is larger for recent values and smaller for older values. This only holds true if α is a constant, and not a calculated value like $\frac{1}{k+1}$ in the sample-average method. By keeping α constant, the action values are capable of changing if the environment is non-stationary.

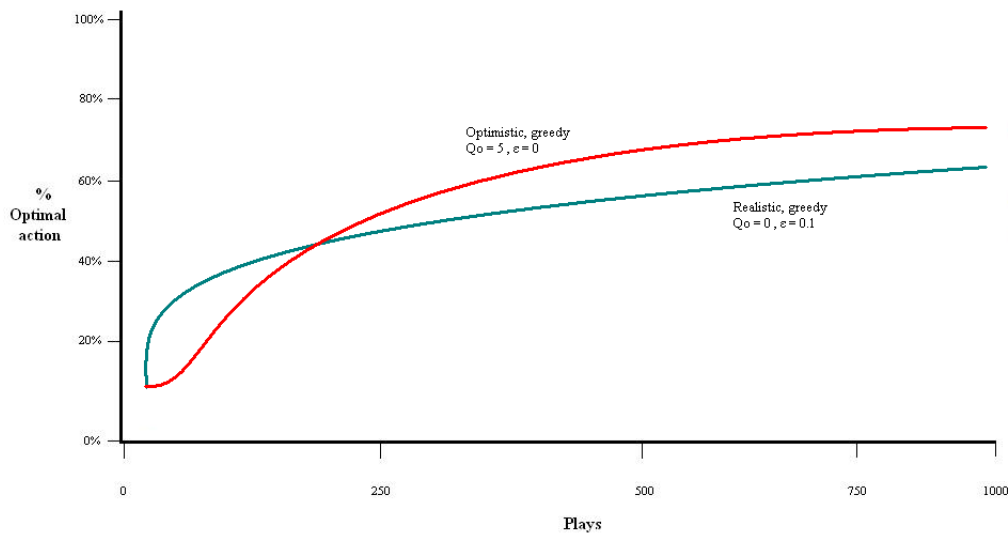


Figure C.2: Graph showing the influence of optimistic initial values to the degree of exploration and end result.

C.1.3 OPTIMISTIC INITIAL VALUES

All previously mentioned action-value methods use predefined initial values for the actions. This value is of course overwritten when all actions have been tried once in a specific state. This value can be initialized to 0, but if a more appropriate value is applied, the initial learning might be improved significantly.

The initial value makes sure that the learning process is very eager to explore all values before settling on the rational action. If the initial value is e.g. 5, and the reward given varies between 0 and 1, the initial value would be very optimistic and encourage the action-value method to explore all possibilities before the best action is found.

This technique is called *optimistic initial value* and secures a large amount of initial exploration. Initially the performance will be poor due to the deliberately chosen "bad" action, but over a long period of time this method will yield a better result than e.g. the ϵ -greedy method, which is shown on figure C.2.

C.2 SOLUTION METHODS

In this section we will examine some of the most used solution methods to the RL problem. We will list their functionality and their strengths and weaknesses.

C.2.1 DYNAMIC PROGRAMMING

The name Dynamic Programming refers to a collection of algorithms capable of solving the RL problem, and all require perfect models of the environment as a Markov Decision Process¹. DP is the oldest solution to RL and is today only treated as a curiosity

¹http://en.wikipedia.org/wiki/Markov_decision_process

C.2. Solution Methods

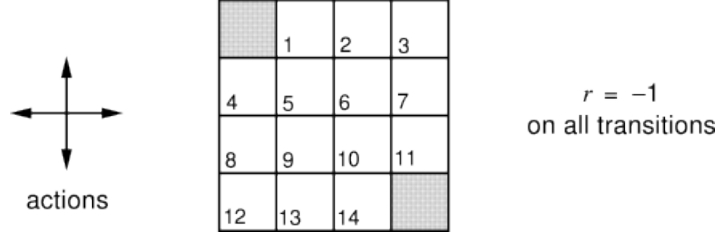


Figure C.3: Figure of grid world, actions, and reward.

because of its great computational requirements.

DP uses the iterative policy evaluation shown below:

```

Input  $\pi$ , the policy which is to be evaluated
Initialize  $V(s) = 0$  for all  $s \in S$ 
Repeat until  $\Delta < \phi$  (a small positive number)
     $\Delta = 0$ 
    for each  $s \in S$ 
         $v = V(s)$ 
         $V(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')]$ 
         $\Delta = \max(\Delta, |v - V(s)|)$ 

```

The output of this will be the optimal policy.

First all values for the states in the policy π will be initialized to 0, so all states are equal.

Then the loop is started. The loop continues until the Δ value is less than the ϕ value. The ϕ value is a predefined value which represents the margin of difference between the previous value of a state and the new value.

For each pass of the loop, the Δ value is set to 0, and all the states are passed. First the initial value of the state s is saved in the value v , so it can be used later. The new value of s is calculated summing for all actions and next states ($\sum_a, \sum_{s'}$) by multiplying the probability of selecting an action a in the state s ($\pi(s, a)$) with the probability of entering state s' when being in state s and selecting a ($P_{ss'}^a$) and the sum of the reward of executing action a in state s resulting in state s' ($R_{ss'}^a$) and the value of state s' multiplied with a discount factor γ ($\gamma V(s')$).

Then the largest value of Δ and the difference between v and $V(s)$ is saved in the Δ value. This value will contain the largest margin of change happening during an iteration of the inner loop.

This can easier be seen in an example. Imagine a 4x4 grid world like in figure C.3, where the top left corner and bottom right are terminal states, the states are $S = 1, 2, \dots, 14$, and the four possible actions are $A = \{up, down, left, right\}$. All rewards are -1 and discount factor γ is 1. Some probabilities are $P_{5,6}^{right} = 1$, $P_{5,10}^{right} = 0$, and $P_{7,7}^{right} = 1$. The values and policy by running the iterative policy evaluation can be seen in figure C.4.

As it can be seen, the DP solution requires several passes to convey toward an optimal solution, and with a larger number of states and actions, the number of passes increases greatly. The complexity of a single pass is n^n where n is the number of states. This

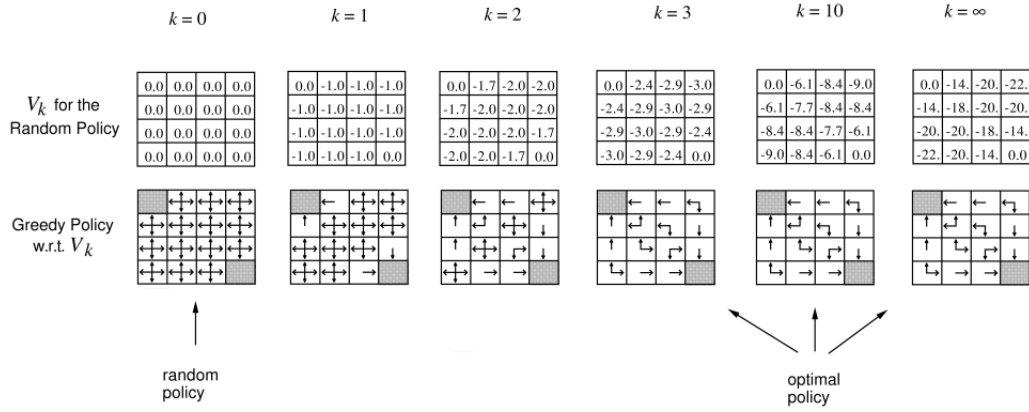


Figure C.4: k is the number of passes of the loop. As seen, the optimal policy is reached by 3 passes of the loop.

makes it very heavy computationally and useless in real-time environments like ours.

C.2.2 MONTE CARLO METHODS

The Monte Carlo methods differs from DP by requiring no model of the environment. The MC methods only uses experience to calculate the optimal policy. By experience we mean sequences of states, actions and rewards gained. MC can also learn from simulated interaction of previously recorded samples. This feature makes it very interesting. MC solves the Reinforcement Learning problem by evaluating the policy after an episode, being a sequence of states, actions, and rewards from a start state to a terminal state. An algorithm which ensures optimal policy through exploring starting values can be seen here:

```

Initialize for all  $s \in S$  and  $a \in A$ :
     $Q(s, a) = \text{predefined value}$ 
     $\pi(s) = \text{predefined action}$ 
     $Returns(s, a) = \text{empty list}$ 

Repeat forever:
    Get episode from using  $\pi$ 
    For each pair  $s, a$  in episode
         $R = \text{reward from doing action } a \text{ in state } s$ 
        Append  $R$  to  $Returns(s, a)$ 
         $Q(s, a) = \text{average}(Returns(s, a))$ 
    For each  $s$  in episode:
         $\pi(s) = \max_a(Q(s, a))$ 
    
```

This algorithm starts out with initializing all values and actions, along with a clean reward list. Then the infinite loop starts.

First an episode is retrieved from using the policy π on the current problem. Now the policy is ready to be improved.

Then each pair of actions and states (s, a) in that episode is examined. The reward

C.2. Solution Methods

that followed the action a in state s is placed in the R variable, which is appended to the list of returns from that specific state-action pair ($Returns(s, a)$), and finally the value of the state-action pair ($Q(s, a)$) is calculated by averaging the reward given ($average>Returns(s, a)$)).

Finally the policy of choosing which action in a state (π) is updated by choosing the action in a state-action pair with the highest value ($max_a(Q(s, a))$).

In our grid world example, an episode could look like (14, left, -1), (13, left, -1), (12, left, -1), (12, up, -1), (8, up, -1), (4, up, 0) with pairs of state, action and reward given. When evaluating this the policy would find out that the best thing to do in state 4 was the action *up*. Armed with this new information, another pass is commenced and so on.

This shows the disadvantage of MC: they do not bootstrap, i.e. they do not carry back the value of the adjacent states. This is fine in some games like Blackjack, where a state is defined by the combined value of your cards (Section 5.3 in [25]). Another disadvantage is that it is necessary to end an episode before any improvements can be calculated. This makes MC very poor at improving long, episodic problems.

Their big advantage is of course that they learn from experience and not from a model like DP.

Appendix D

THE GAME

D.1 GAME DESCRIPTION

In this appendix we will describe the game we have created which is called "Tank General". This game was implemented during the preparation phase. The goal of this section is to give a brief overview of the game mechanics and the main features in the game. After this appendix, the reader should be able to understand the basic rules of the game. Screenshots from the game can be found in Appendix G.

D.1.1 GAME DESIGN

SETTING

The setting of the game is a battlefield where two teams, which we will refer to as the blue army and the red army respectively, has to fight each other on a relatively small map. The two armies consist of five different types of units which will be explained in details later. We decided to let the units be different kinds of armoured vehicles in a World War II setting. The World War II setting we chose had no impact on the game play, and we could have chosen a medieval setting with the units consisting of knights and pikemen ect., without having to change the basics of our game significantly.

At the start of the game both armies are placed at different and almost random positions. There will always be a certain amount of space between the armies meaning the armies will never start next to each other. Each team starts with a main building called the headquarters and a small number of units. Both armies start with identical units which means that the armies are equal from the start of the game.

WINNING CONDITIONS

The goal of the game is to destroy the headquarters of the enemy. This is normally done by avoiding or beating the defending units of the enemy army to get free access to the enemy headquarters which can then be destroyed by the attacking units. The

game is lost if ones headquarters are destroyed.

UNITS

There are five different units in the game. Each unit has strengths and weaknesses which the players should consider when playing and deciding on a strategy. Each unit has seven attributes which make the units different. The attributes are described below:

- **Line of sight** (LoS). This attribute decides how far away the unit can see.
- **Range**. The range of a unit decides how far the bullets can reach.
- **Speed**. The speed of a unit decides how fast the unit can move.
- **Area of effect** (AoE). The area of effect decides whether the bullets affect multiple targets within a specified area.
- **Hit points** (HP). The hit points of a unit decides how much damage the unit can sustain.
- **Damage**. This attribute decides how much damage a unit deals to other units when hit by a bullet from this unit.
- **Rate of fire** (RoF). The rate of fire of a unit decides how fast the unit can shoot.

Having defined the attributes that makes the units different, we are now ready to describe the units in the game. There are five different types of units in the game which are:

- Tank.
- Anti Tank.
- Mechanized Infantry.
- Recon.
- Artillery.

The units Tank, Mechanized Infantry and Anti Tank are quite similar. They all have average line of sight, range, speed, hit points, damage, rate of fire and no area of effect. What makes them special is their mutual bonus strength against each other based on the Paper-Rock-Scissors pattern [6]. The Tank deals double damage to the Mechanized Infantry which deals double damage to the Anti Tank, and the Anti Tank closes the circle by dealing double damage to the Tank.

The Recon unit is fast and has a huge line of sight which makes it ideal as a scout. The disadvantages of this unit are a short range of its weapon and a low number of hit points.

The last unit is the Artillery. This unit is very slow and its number of hit points is below average. The advantages of the Artillery are a very long range, and the fact of a bonus which means it deals double damage against headquarters. The shells from an

D.1. Game Description

Artillery unit has an area of effect on other units meaning that an Artillery unit can hit several enemy units standing close to each other.

The attributes of each unit are summarized in the table below. * indicates the just mentioned bonuses. The attribute value px is pixels.

	LoS	Range	Speed	AoE	HP	Damage	RoF
Tank	200 px	150 px	x 1	no	100	10*	x 1
Mechanized Infantry	200 px	150 px	x 1	no	100	10*	x 1
Anti Tank	200 px	150 px	x 1	no	100	10*	x 1
Recon	275 px	90 px	x 1.5	no	80	7	x 1
Artillery	200 px	275 px	x 0.5	yes	80	10*	x 1.5

Table D.1: Units and there corresponding attributes.

We should also point out that there is no friendly fire in the game meaning that units on the same team cannot hit each other. Instead the bullets passes through friendly units.

In order to decrease the amount of micromanagement the player and the AI has to deal with, we have implemented several features to help the players to take care of trivial tasks. These features will be described in the following.

Fire at will

All units fire at will which means that enemy units within range and line of sight will automatically be targeted and shot at, unless the unit receive a counter order from the player or AI. The priorities when shooting are as follows:

1. Targets appointed by the player.
2. Enemy units (since they can shoot back).
3. Enemy headquarters.

Path finding

All units use pathfinding in order to find the shortest path between two points. This is a great help when large groups of units are moving in terrain with many obstacles. The pathfinding is implemented using A* algorithm [21].

REINFORCEMENTS AND RESOURCES

Every half minute the players have the opportunity to reinforce their army by building a new unit of their choice. New units are placed next to the headquarters. The players are not forced to build a new unit every half minute. It is possible to save reinforcements until later by for example waiting two minutes and then immediately build four units. When reinforcements are spent, new units are created instantly. Reinforcements can also be used to repair the headquarters for 100 hit points.

Two resources called *war factories* are placed almost randomly on the map. The algorithm placing the resources will always try to place both resources within the same

distance to both headquarters, meaning that no army will have an unjust advantage because of the location of the resources. Initially both war factories are neutral meaning they do not belong to any team. These resources can be captured by an army by having one or more units standing close to the resource while no enemy units are present. After standing beside the war factory for thirty seconds the war factory will then belong to the army of that unit. The war factory will change color and stay under control of that army even though no units are present.

It is possible to lose a war factory to the enemy or capture a war factory which belongs to the enemy. The procedure is similar to capturing a neutral war factory. If a unit is in the process of taking over a war factory belonging to the opponent army by standing close to it, it is possible to defend the war factory by bringing one or more units close to it. The "timer" for taking over the war factory is then reset, and the attacking unit has to fight the defending units in order to capture the war factory.

The war factories are important resources in the game. Each war factory an army possesses reduces the time between reinforcements by 10 seconds. Therefore if an army possesses two resources that army can receive reinforcements every 10th seconds compared to the 30 seconds in which the opponent army receives its reinforcements.

FOG OF WAR

In order to increase the strategical element of surprise, the map is initially covered by the fog of war. This means that both armies can make strategic manoeuvres that the enemy cannot track unless they have scouting units in the same area. This also means that the players need to reconnoitre in order to reveal the map. There are three levels of fog of war:

1. Areas which are fully observable because the area currently is in the line of sight of one or more units or buildings.
2. Areas of the map which have previously been revealed but are currently not in line of sight of any units. The terrain and buildings will be observable but it is not possible to see enemy units.
3. Areas not yet discovered cannot be seen at all and will just appear black on the map.

OTHER FEATURES

In the following we will describe a few features of the game which does not affect the game play much but are still worth to mention.

3d sound

By taking advantage of the sound library in XNA we implemented 3d sound which means that during combat it is possible to hear where on the map the combat is taking place. This also means that the sound volume will decrease or increase depending on how far away the camera is.

D.2. Game Manual

Level-editor

In order to make level creation quick and easy we implemented a level-editor. The level-editor helped us save a lot of time during implementation and testing because we could easily make appropriate levels when testing different game mechanics like A*, battles etc. The level-editor is further described in our previous report "Tank General" [3].

Customizable ini-file

We decided to load unit-attributes and many game attributes from an ini-file. This makes it very easy to test different attribute values when balancing the game, eliminating the need to recompiling the game every time.

D.1.2 SUMMARY

In this section we presented our game in order to give the reader a brief overview of the rules and the basic concepts of the game. In the next section we will describe in detail how to play the game regarding controlling the units and the camera, spending reinforcements etc.

D.2 GAME MANUAL

In this section we go into details with controls, hot keys etc., and it is the goal of this section to teach the player how to play the game.

D.2.1 CONTROLS

In this section we will teach the reader how to control the units with movement orders, grouping, camera and so on. We have implemented the controls following the standards of traditional RTS games. In this section we define a click as a left click on the mouse. A key press on the keyboard will be defined as $[KEY]$ where key is a specific key for instance the SHIFT-key. $[SHIFT] + [A]$ means holding down the SHIFT-key and then pressing A.

SELECTION

In order to move one or more units the player must first select the units, which can be done in several ways. Selected units will carry out movement and attack orders given by the player. We should mention that selected units are shown with a slightly lighter color than non selected units.

- A unit can be selected by clicking the unit.
- More units can be selected by dragging a selection box around the units.
- To unselect all selected units, just click on an empty spot on the map.
- To unselect only one unit, hold down $[SHIFT]$ and click the unit.

- To add one non-selected unit to a group of selected units, hold down `[SHIFT]` and then clicking the unit.
- Non-selected units can be added to a group of selected units by holding down `[SHIFT]` and then drag a selection box around the units you want to add.
- Double-clicking on a unit will select all units of the same type on the screen.
- Double-clicking on a unit while holding down `[SHIFT]` will add all units of the same type to the group of selected units.
- By pressing `[CTRL] + [A]` it is possible to select all units on the screen.
- By pressing `[CTRL] + [SHIFT] + [A]` all units on the screen will be added to the group of already selected units.

NUMERIC GROUPS

A group of units can be tied together in something called a *numeric group*. The idea is to make it easier for the player to handle several groups of units.

- Units can be allocated in one or more numeric groups by selecting the units and pressing `[CTRL] + [1 - 5]` where `[1 - 5]` is one of the numeric keys from 1 to 5. A small number in the lower right corner of the unit will show which numeric group it is allocated to.
- All units in a numeric group can be selected by pressing `[1 - 5]`.
- Two numeric groups can be merged into a new numeric group. To merge group 1 and 2 in group 2, press `[1]` followed by `[SHIFT] + [2]` followed by `[CTRL] + [2]`. First all units in numeric group 1 are selected. Then all units in numeric group 2 are added to the selected units. Last all units are saved into numeric group 2.
- To deallocate all units from a group, simply press `[CTRL] + [1 - 5]` without having selected any units.

MOVEMENT

This section will deal with movement of units and how to order units to attack the enemy. Having selected units, either by selecting them directly or by using numeric groups, they are now ready to receive orders. Orders are giving by using the right mouse button.

- Right clicking on the map will order all selected units to move to that location. The units will automatically follow the shortest path. If you click on a point which the selected units does not have access to, the mouse cursor will show a different cursor indicating this.
- If one or more units are ordered to go to a specific location on the map and they are intercepted by enemy units, they will stop and fight the enemy units. If the units survives the battle, they will continue to move to their destination.
- Units can be ordered to attack enemy units or the enemy headquarters by right clicking on the target. The cursor will show a cross hair when pointing on a valid target. If one or more of the selected units are out of range of their target they will automatically approach their target in order to attack. If the enemy flee, the

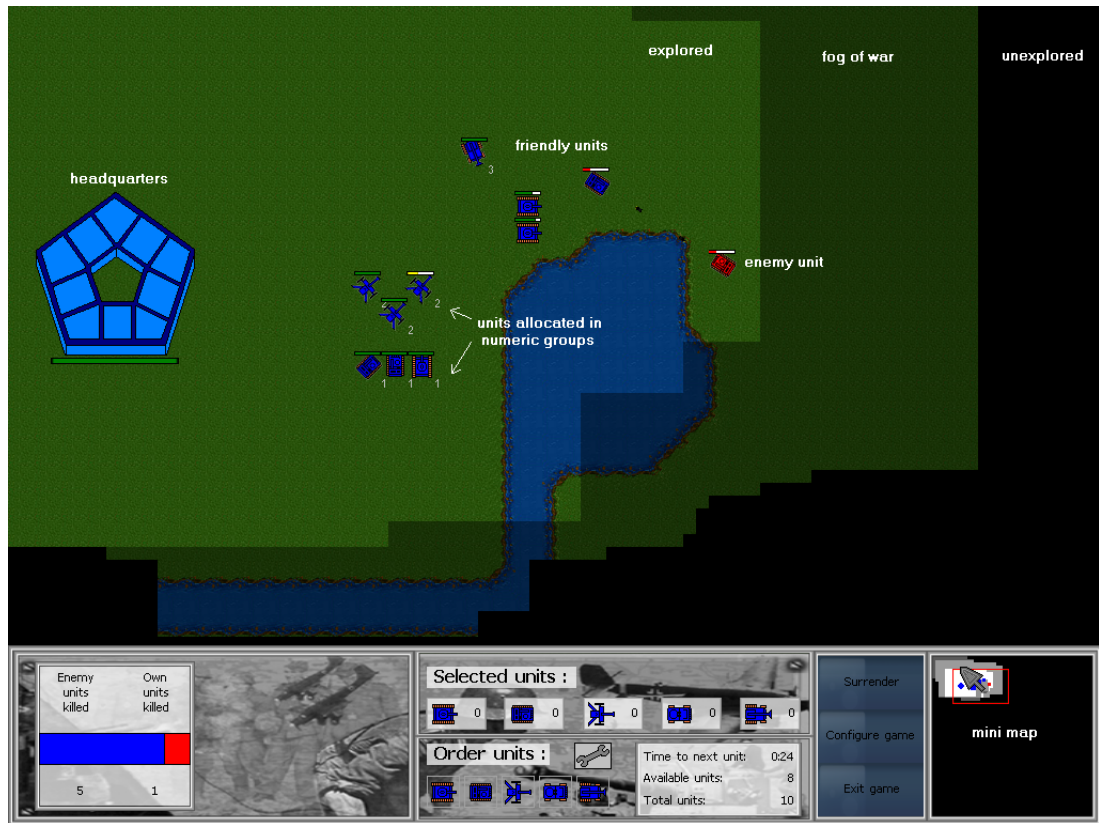


Figure D.1: Screenshot showing the in-game screen.

units will follow their target as long as the target is in sight of a friendly unit. If the target unit disappear from the line of sight, the unit will stop and await new orders.

- It is possible to give orders by right clicking the mini map as well. This is an easy way to command units to go to an area far away.

CAMERA

This section explains how to move the camera on the map.

- The camera can be moved around the map by either using the four arrow keys, by clicking on the mini map or by moving the mouse cursor to the edge of the screen.
- The camera can be locked to follow a unit by pressing $[CTRL] + [L]$. The camera will be unlocked by pressing $[CTRL] + [L]$ a second time.
- If a numeric group has been made, the camera will jump to that group by double pressing $[1 - 5]$ where $[1 - 5]$ is the number key for the numeric group.
- It is possible to zoom and in and out using either the scroll-wheel on the mouse or by pressing $[Z]$.
- If two AIs are playing against each other a special overview map can be displayed by pressing $[B]$.

D.2.2 GUI - IN-GAME SCREEN

Figure D.1 shows a screenshot from the in-game screen with some explanatory comments. In this section we will describe the Graphical User Interface (GUI) on this screen.

STATUS

The lower left corner is the status GUI. When no units are selected a status bar will indicate how many units each side has killed during the game. If a unit is selected, a picture of the unit and some information about the unit will be shown here.

SELECTED UNITS

To the right of the status GUI, the selected units are shown. This GUI always shows how many units of each type that are currently selected. This is indicated by a picture of each unit and a number showing how many units of this type are selected.

ORDER UNITS

It is possible to order new units from the GUI below the selected units GUI. A timer shows how many seconds are left, until the next reinforcement will be available. It is also possible to see how many reinforcements are available. Each time the timer reaches zero, an extra reinforcement will become available. The units can be ordered from the five unit-buttons to the right. To order a unit just press the corresponding unit-button. If no reinforcement are currently available, it is still possible to click on one of the buttons. When doing this the button will get a green frame indicating that the next time reinforcement is available, a unit of the ordered type is built. When the new unit has arrived, the green frame will disappear. Two clicks on a unit-button will give the button a yellow frame indicating that this unit will be built, each time a new unit is available. This can be canceled by clicking one more time at the button which will remove the yellow frame.

REPAIR HEADQUARTERS

If reinforcements are available it is possible to repair the headquarters for 100 hit points by clicking the "spanner" instead of building new units.

BUTTONS

Apart from unit-buttons used to order units from reinforcement, three other buttons are available from the in-game screen. If the human player realizes that defeat is inevitable, the player can surrender the game by pressing the *Surrender* button. This will end the game and bring the player to the achievement screen which will be described later.



Figure D.2: Screenshot showing mini map.

There is also a *Settings* button. Lastly there is an *Exit game* button from where the game can be x.

D.2.3 GUI - MINI MAP

The mini map is located in the lower right corner of the screen. The mini map shows information about the entire map. A screenshot of the mini map can be seen on figure D.2. The red rectangle indicates the current area shown by the camera. Blue and red dots are friendly and enemy units respectively. The headquarters are shown as small circles. White areas are areas which is currently in line of sight of some friendly units. Grey areas are areas which has been explored but are covered by the fog of war. When an enemy unit attacks a human unit outside the current view of the camera, it is shown on the mini map as a red flashing circle indicating the position of the attack.

D.2.4 GUI - ACHIEVEMENT SCREEN

When a game has ended, the achievement screen will be shown. This screen, shown in Figure D.3, shows various information and statistics from the game. The bottom section displays a timeline on which major battles are plotted as a picture of two crossed swords. A small picture of headquarters indicates that this headquarters was attacked. A small picture of a red or blue war factory indicates that a war factory was captured at this time. A headquarters with a mushroom cloud means that the headquarters was destroyed.

D.3 HOT KEYS

The hot keys available in-game are shown in table A.1.

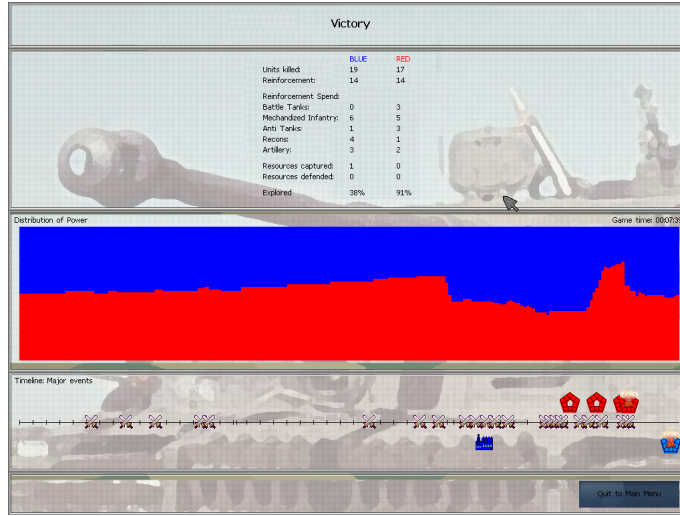


Figure D.3: Screenshot showing the achievement screen.

Hot keys	Functionality
[CTRL] + [1 – 5]	The selected units are saved in battle group [1 – 5].
[1 – 5]	Selects the battle group [1 – 5].
[1 – 5] x 2	Camera jumps to battle group [1 – 5].
[SHIFT] + [1 – 5]	The battle group [1 – 5] is added to the selected units.
[SHIFT] + (left click)	The clicked unit is added to the selected units.
[SHIFT] + (selection box) (double click)	The units in the selection box are added to the selected units. All units on the screen of the same type as the clicked units are selected.
[E]	If one unit is selected, all units on the screen of same type are selected.
[CTRL] + [A]	All units on the screen are selected.
[CTRL] + [SHIFT] + A	All units on the screen are added to the selected units.
[CTRL] + [L]	The camera is locked/unlocked on the selected unit.
[F]	Toggles fog of war on or off.
[P]	Pauses or unpauses the game.
[SPACE]	Toggles between full screen and window mode.
[Z] or (mouse-wheel)	Zoom in and out.
[B]	Toggle overview-map (Only available in AI vs AI games).
[O]	Toggle profiler info (Only available in AI vs AI games).
[R]	Toggle RL-info (Only available in AI vs AI games).
[V]	Switches side (Only available in AI vs AI games).
[ESC]	Exits and closes the game.

Table D.2: In-game hot keys.

Appendix E

REINFORCEMENT LEARNING ATTRIBUTES IN TANK GENERAL

State Space Attributes - Multi Layered

Top Level

Attribute name	Values
PowerDistribution	[More powerful] [Equal] [Weaker].
ResourcesExplored	[None] [One] [Two].
HeadquartersExplored	[Enemy HQ known] [Enemy HQ unknown].
OwnHeadquartersHitPoints	[High] [Medium] [Low].
EnemyHeadquartersHitPoints	[High] [Medium] [Low].
ResourceDistribution	[One or more neutral] [Both owned] [Other].
EnemyThreateningBuildings	[True] [False].
UnitCloseToUnguardedResource	[True] [False].
EnemyDefenders	[Unknown] [Few] [Many].

Low Level Explore

Attribute name	Values
HeadquartersExplored	[Enemy HQ known] [Enemy HQ unknown].
ResourcesExplored	[None] [One] [Two].
NumberOfRecons	[0] [1-2] [3-4] [More than 4].
UnitCloseToUnguardedResource	[True] [False].
ReinforcementAvailable	[True] [False].

Low Level Defend

Attribute name	Values
PowerDistributionAtOwnHQ	[More powerful] [Less] [Lesser].
TotalPowerDistributionAtOwnHQ	[More powerful] [Less powerful].
PowerDistributionAtResource1	[More powerful] [Less] [Lesser].
PowerDistributionAtResource2	[More powerful] [Less] [Lesser].
OwnHeadquartersHitPoints	[High] [Medium] [Low].
ReinforcementAvailable	[True] [False].
ResourceDistribution	[One or more neutral] [Both owned] [Other].

Low Level Attack Headquarters

Attribute name	Values
ReinforcementAvailable	[True] [False].
PowerDistribution	[More powerful] [Equal] [Weaker].
EnemyHeadquartersHitPoints	[High] [Medium] [Low].
ResourceDistribution	[One or more neutral] [Both owned] [Other].
EnemyDefenders	[Unknown] [Few] [Many].
NumberOfArtillery	[0-3] [4-6] [More than 6].
PowerDistributionAtEnemyHQ	[More powerful] [Less] [Lesser].

Low Level Attack Resource

Attribute name	Values
ReinforcementAvailable	[True] [False].
PowerDistributionAtResource1	[More powerful] [Less] [Lesser].
PowerDistributionAtResource2	[More powerful] [Less] [Lesser].
Resource1OwnedByEnemy	[True] [False].
Resource2OwnedByEnemy	[True] [False].

State Space Attributes - Single Layered

Attribute name	Values
HeadquartersExplored	[Enemy HQ known] [Enemy HQ unknown].
OwnHeadquartersHitPoints	[High] [Medium] [Low].
EnemyHeadquartersHitPoints	[High] [Medium] [Low].
NumberOfRecons	[0] [1-2] [3-4] [More than 4].
PowerDistributionAtOwnHQ	[More powerful] [Less] [Lesser].
TotalPowerDistributionAtOwnHQ	[More powerful] [Less powerful].
PowerDistributionAtResource1	[More powerful] [Less] [Lesser].
PowerDistributionAtResource2	[More powerful] [Less] [Lesser].
ReinforcementAvailable	[True] [False].
PowerDistributionAtEnemyHQ	[More powerful] [Less] [Lesser].
Resource1OwnedByEnemy	[True] [False].
Resource2OwnedByEnemy	[True] [False].
ResourcesUnknown	[True] [False].

Actions – Multi Layered

Top Level

Action name
Explore
Defend
AttackHeadquarters
AttackResource

Low Level Explore

Action name
BuildExploreUnit
ExploreRandomPosition
ExploreSpiral
ExploreRandomDirection
ExploreInfluenceMap
CaptureResource

Low Level Defend

Action name
BuildDefendUnit
SendToHeadquarters_25
SendToHeadquarters_50
SendToHeadquarters_100
SendToResource_25
SendToResource_50
SendToResource_100
RepairBase

Low Level Attack Headquarters

Action name
BuildAttackUnit
AllOutAttack
CoordinatedAttack_100
CalculatedAttack
FeintAttack

Low Level Attack Resource

Action name
BuildAttackUnit
SingleUnitAttackResource1
SingleUnitAttackResource2
SendSufficientGroupToAttackResource1
SendSufficientGroupToAttackResource2

Actions – Single Layered

Action name
BuildExploreUnit
ExploreRandomPosition
ExploreSpiral
ExploreRandomDirection
ExploreInfluenceMap
CaptureResource
BuildDefendUnit
SendToHeadquarters_25
SendToHeadquarters_50
SendToHeadquarters_100
SendToResource_25
SendToResource_50
SendToResource_100
RepairBase
BuildAttackUnit
AllOutAttack
CoordinatedAttack_100
CalculatedAttack
FeintAttack
BuildAttackUnit
SingleUnitAttackResource1
SingleUnitAttackResource2
SendSufficientGroupToAttackResource1
SendSufficientGroupToAttackResource2

Rewards – Multi Layered

Top Level – Anti Aggressive

Reward name	Values
EnemyHeadquarterDestroyed	5
DamageDealtToEnemyHeadquarters	0
OwnHeadquarterDestroyed	-5
DamageDealtToOwnHeadquarters	-1
EnemyUnitCloseToOwnHeadquartersKilled	1
OwnUnitCloseToEnemyHeadquartersKilled	0
EnemyUnitCloseToEnemyHeadquartersKilled	0
EnemyUnitCloseToResourceKilled	1
OwnUnitCloseToResource	0
OwnUnitCloseToHeadquarters	0.25
ResourceCaptured	3
ResourceLost	-3
ResourceExplored	2
EnemyHeadquartersExplored	3
ResourceExploredAgain	1
EnemyHeadquartersExploredAgain	1

Top Level – Anti Defensive

Reward name	Values
EnemyHeadquarterDestroyed	10
DamageDealtToEnemyHeadquarters	3
OwnHeadquarterDestroyed	0
DamageDealtToOwnHeadquarters	0
EnemyUnitCloseToOwnHeadquartersKilled	0
OwnUnitCloseToEnemyHeadquartersKilled	0
EnemyUnitCloseToEnemyHeadquartersKilled	1
EnemyUnitCloseToResourceKilled	0
OwnUnitCloseToResource	0
OwnUnitCloseToHeadquarters	0
ResourceCaptured	3
ResourceLost	0
ResourceExplored	3
EnemyHeadquartersExplored	3
ResourceExploredAgain	0
EnemyHeadquartersExploredAgain	0

Top Level – Anti Resources

Reward name	Values
EnemyHeadquarterDestroyed	5
DamageDealtToEnemyHeadquarters	0
OwnHeadquarterDestroyed	-5
DamageDealtToOwnHeadquarters	-0.25
EnemyUnitCloseToOwnHeadquartersKilled	-1
OwnUnitCloseToEnemyHeadquartersKilled	1
EnemyUnitCloseToEnemyHeadquartersKilled	0
EnemyUnitCloseToResourceKilled	3
OwnUnitCloseToResource	2
OwnUnitCloseToHeadquarters	0.25
ResourceCaptured	5
ResourceLost	-5
ResourceExplored	3
EnemyHeadquartersExplored	2
ResourceExploredAgain	2
EnemyHeadquartersExploredAgain	1

Low Level Explore

Reward name	Values
OwnHeadquartersDestroyed	-5
EnemyUnitCloseToResourceKilled	1
OwnUnitCloseToResource	0.25
ResourceCaptured	7
ResourceLost	-5
ResourceExplored	5
EnemyHeadquartersExplored	6
ResourceExploredAgain	1
EnemyHeadquartersExploredAgain	1

Low Level Defend

Reward name	Values
DamageDealtToOwnHeadquarters	-1
OwnHeadquartersDestroyed	-5
OwnUnitCloseToHeadquarters	0.25
EnemyUnitCloseToOwnHeadquartersKilled	-1
Repair	5
ResourceUnderAttack	-3
EnemyUnitCloseToOwnResourceKilled	1
OwnUnitCloseToOwnResource	0.25

Low Level Attack Headquarters

Reward name	Values
EnemyHeadquarterDestroyed	10
DamageDealtToEnemyHeadquarters	2

Low Level Resources

Reward name	Values
ResourceCaptured	3
EnemyUnitCloseToEnemyResourceKilled	1
OwnUnitCloseToEnemyResourceKilled	-1

Rewards – Single Layered

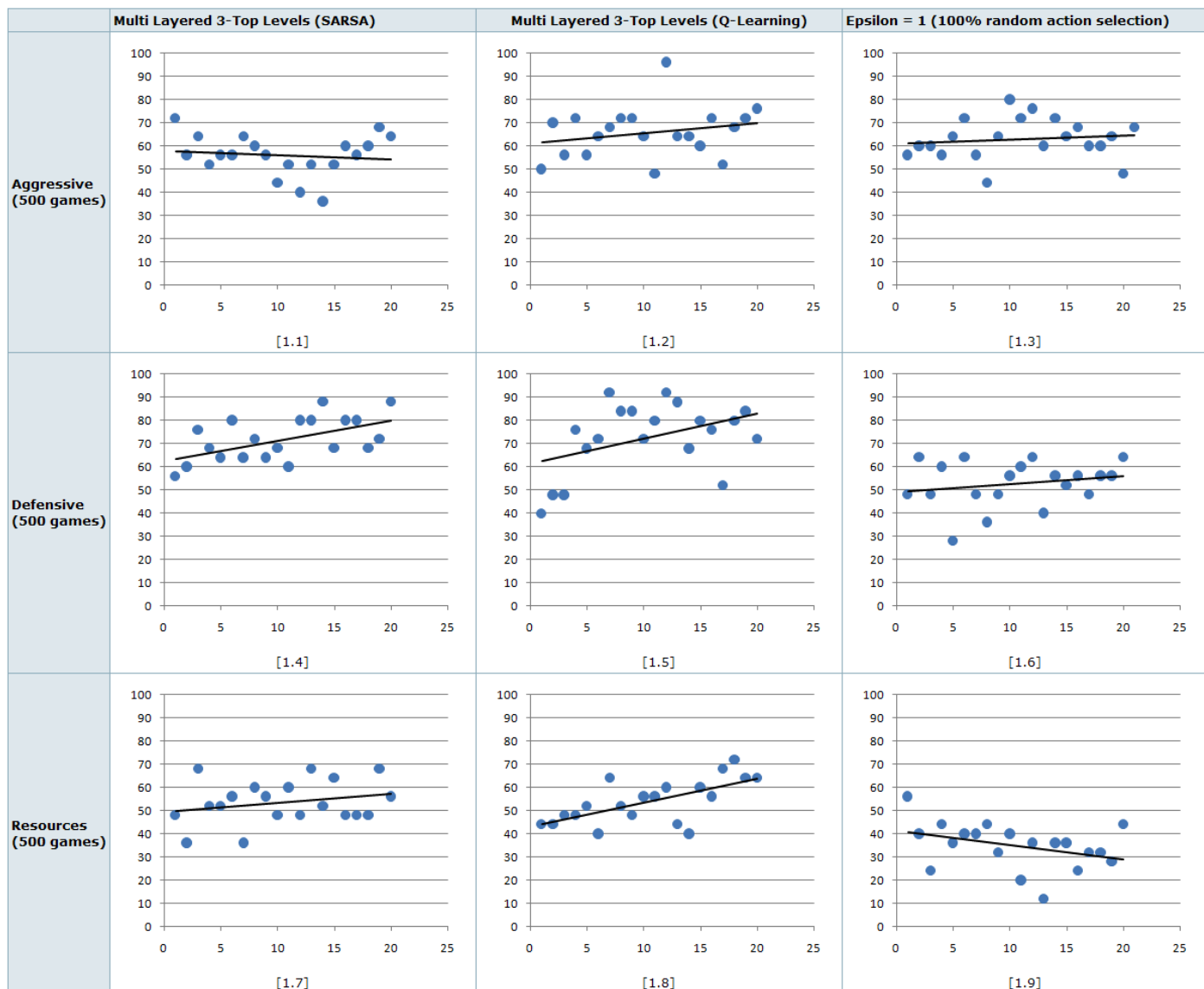
Reward name	Values
EnemyHeadquarterDestroyed	10
DamageDealtToEnemyHeadquarters	1
OwnHeadquarterDestroyed	-10
DamageDealtToOwnHeadquarters	-0.25
EnemyUnitCloseToOwnHeadquartersKilled	1
OwnUnitCloseToEnemyHeadquartersKilled	-1
EnemyUnitCloseToEnemyHeadquartersKilled	1
EnemyUnitCloseToResourceKilled	1
OwnUnitCloseToResource	0.25
OwnUnitCloseToHeadquarters	0.5
ResourceCaptured	5
ResourceLost	-5
ResourceExplored	5
EnemyHeadquartersExplored	5
ResourceExploredAgain	1
EnemyHeadquartersExploredAgain	1
EnemyUnitCloseToEnemyResourceKilled	1
OwnUnitCloseToEnemyResourceKilled	-1
Repair	5
ResourceUnderAttack	-2

Appendix F

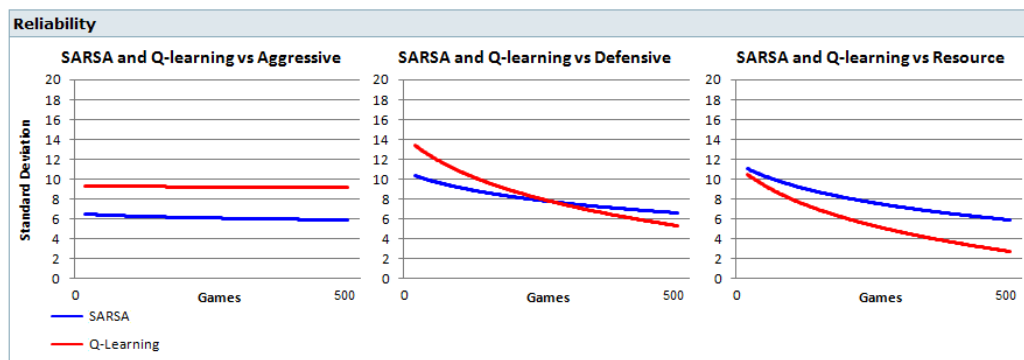
TESTS

APPENDIX

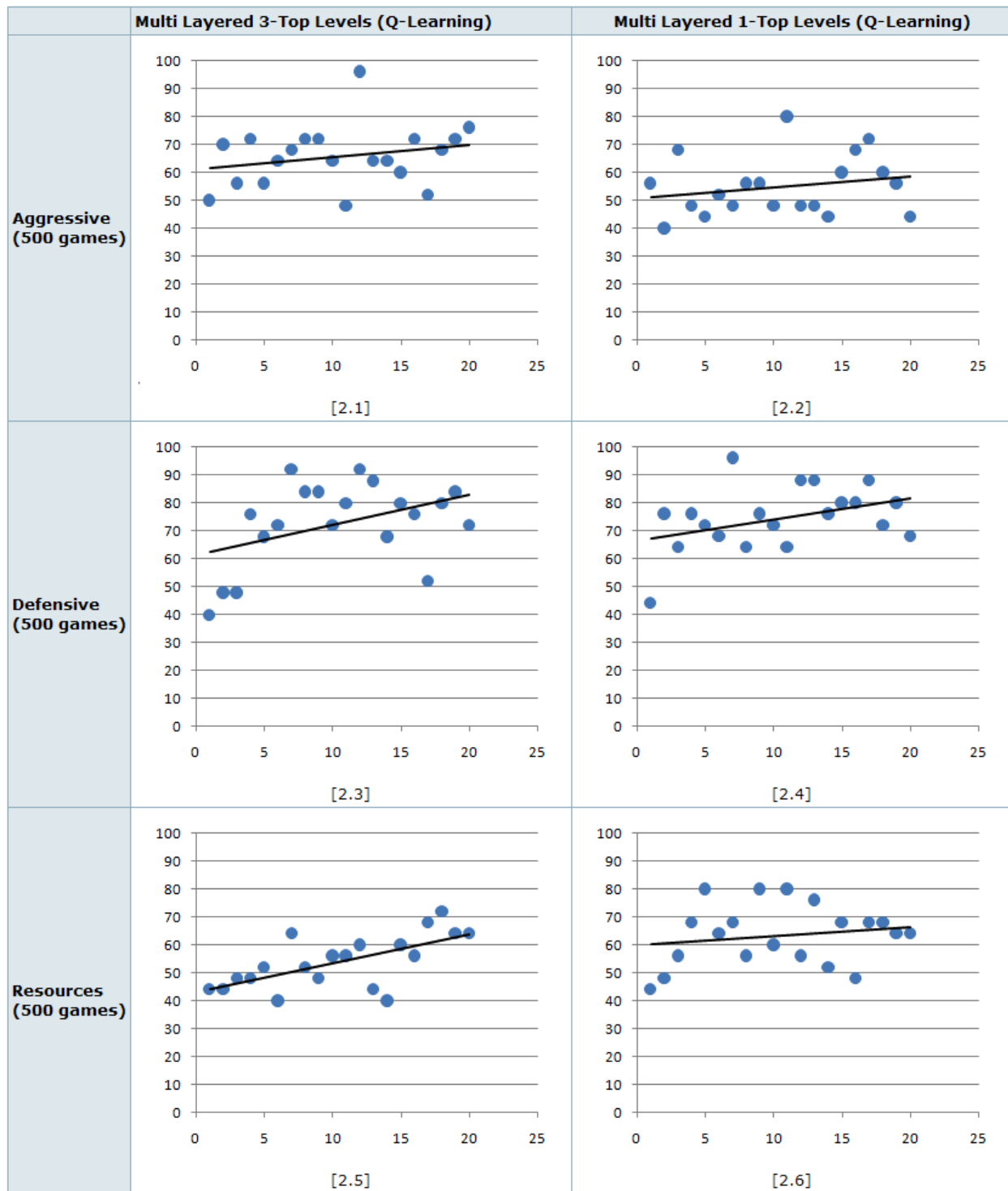
Test 1



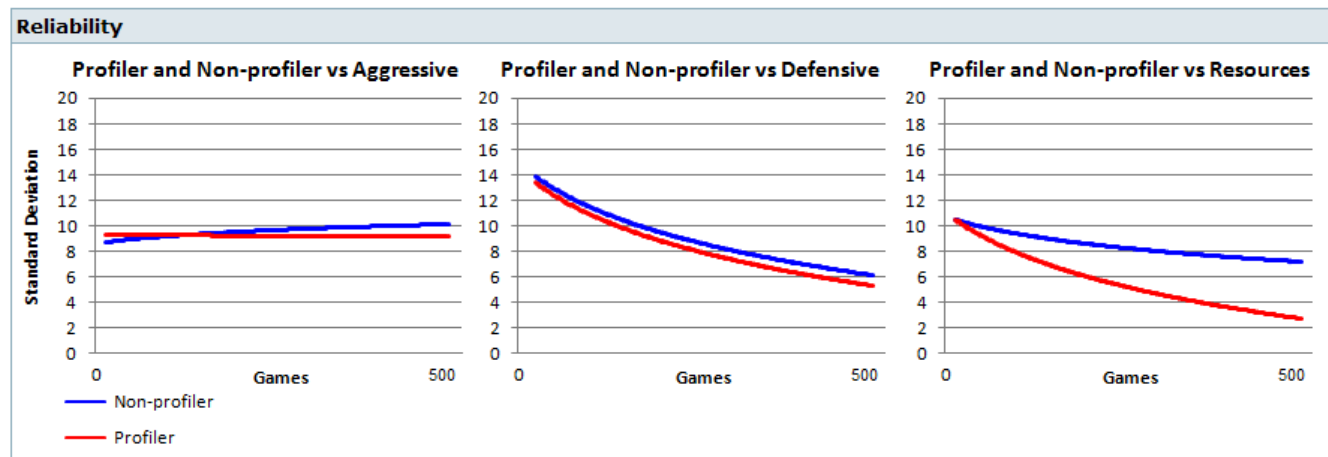
SARSA vs Q-Learning



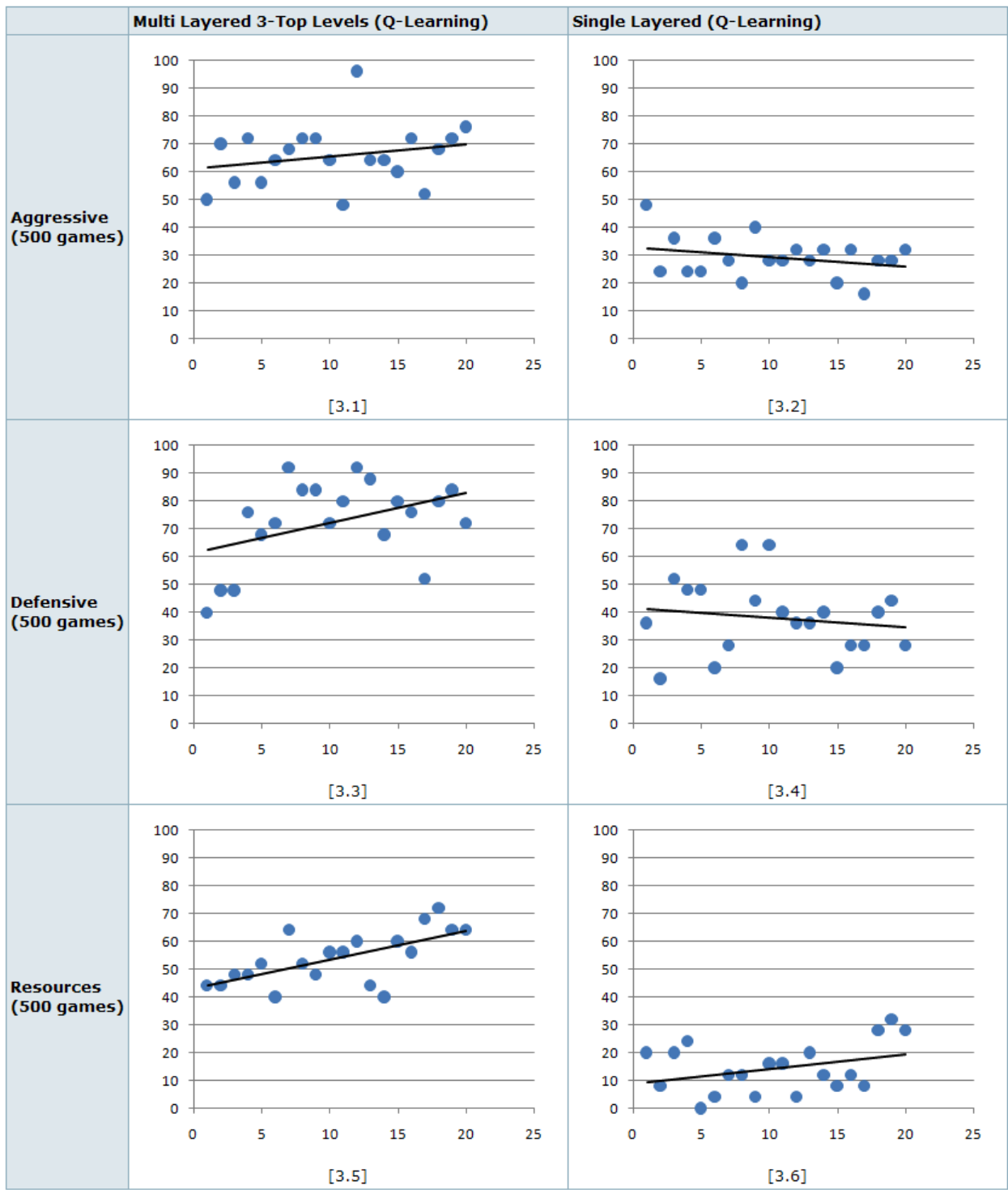
Test 2



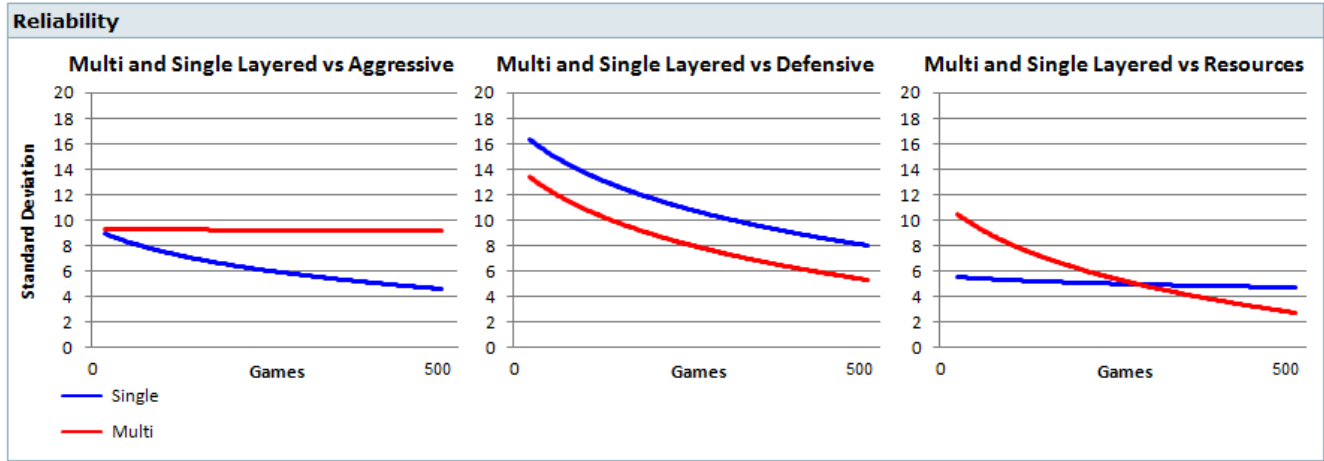
Profiling vs Non profiling



Test 3



Single vs Multi Layered

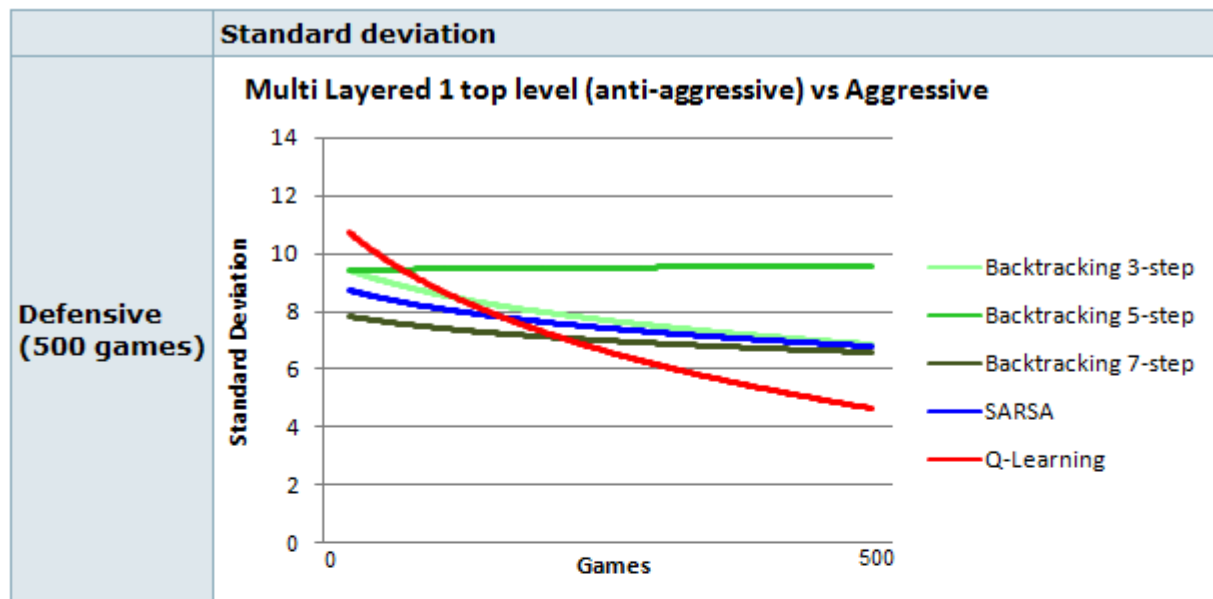


Test 4

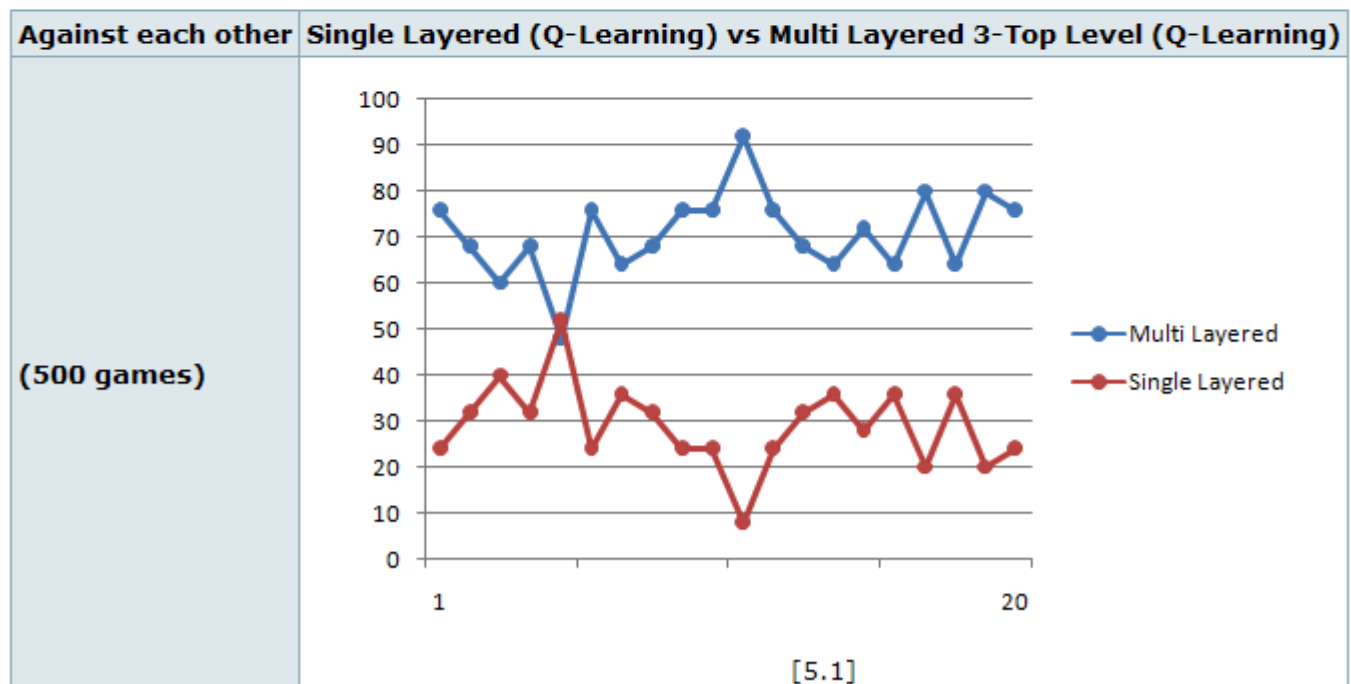
Backtracking vs Non backtracking

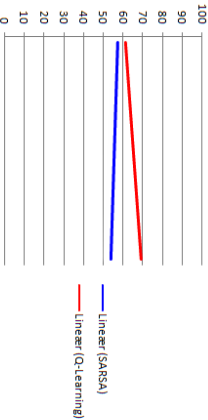
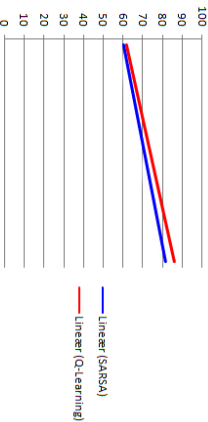
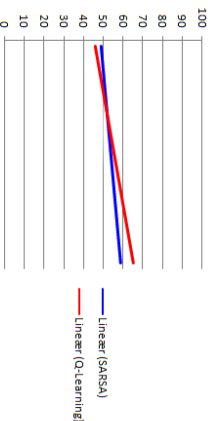
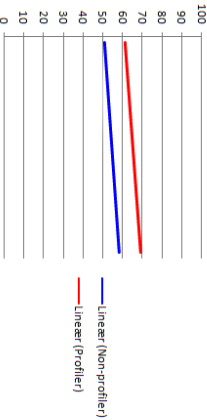
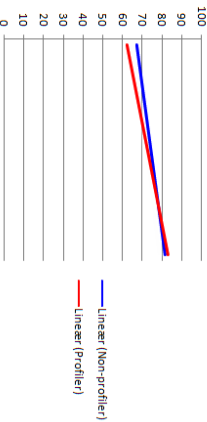
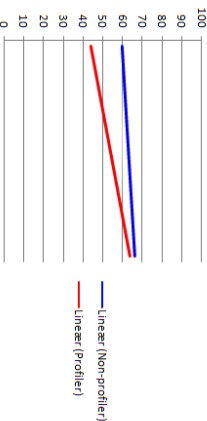
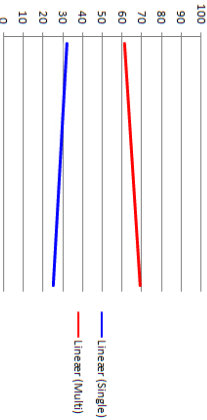
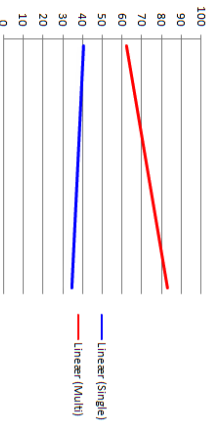
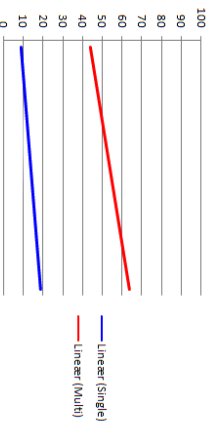
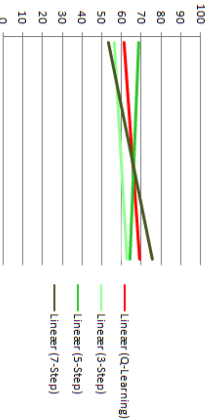
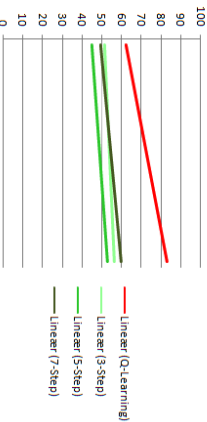
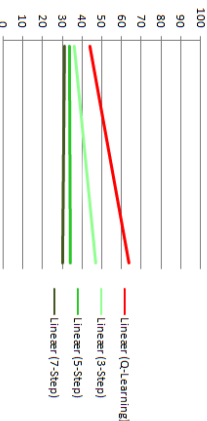
	Multi Layered 3-Top Levels (Q-Learning)	Multi Layered 3-Top Levels Backtracking (3 steps)	Multi Layered 3-Top Levels Backtracking (5 steps)	Multi Layered 3-Top Levels Backtracking (7 steps)
Aggressive (500 games)				
Defensive (500 games)				
Resources (500 games)				

Test 4



Test 5



Test 1	<div>Aggressive [1.1], [1.2]</div> <div>SARSA & Q-Learning vs Aggressive</div> <div>Performance</div>  <div>Games</div>	<div>Defensive [1.4], [1.5]</div> <div>SARSA & Q-Learning vs Defensive</div> <div>Performance</div>  <div>Games</div>	<div>Resources [1.7], [1.8]</div> <div>SARSA & Q-Learning vs Resource</div> <div>Performance</div>  <div>Games</div>
	<div>Aggressive [2.1], [2.2]</div> <div>Profiler & Non-profiler vs Aggressive</div> <div>Performance</div>  <div>Games</div>	<div>Defensive [2.3], [2.4]</div> <div>Profiler & Non-profiler vs Defensive</div> <div>Performance</div>  <div>Games</div>	<div>Resources [2.5], [2.6]</div> <div>Profiler & Non-profiler vs Resource</div> <div>Performance</div>  <div>Games</div>
	<div>Aggressive [3.1], [3.2]</div> <div>Multi Layered & Single Layered vs Aggressive</div> <div>Performance</div>  <div>Games</div>	<div>Defensive [3.3], [3.4]</div> <div>Multi Layered & Single Layered vs Defensive</div> <div>Performance</div>  <div>Games</div>	<div>Resources [3.5], [3.6]</div> <div>Multi Layered & Single Layered vs Resource</div> <div>Performance</div>  <div>Games</div>
	<div>Aggressive [4.1], [4.2], [4.3], [4.4]</div> <div>Multi Layered & Backtracking vs Aggressive</div> <div>Performance</div>  <div>Games</div>	<div>Defensive [4.5], [4.6], [4.7], [4.8]</div> <div>Multi Layered & Backtracking vs Defensive</div> <div>Performance</div>  <div>Games</div>	<div>Resources [4.9], [4.10], [4.11], [4.12]</div> <div>Multi Layered & Backtracking vs Resources</div> <div>Performance</div>  <div>Games</div>

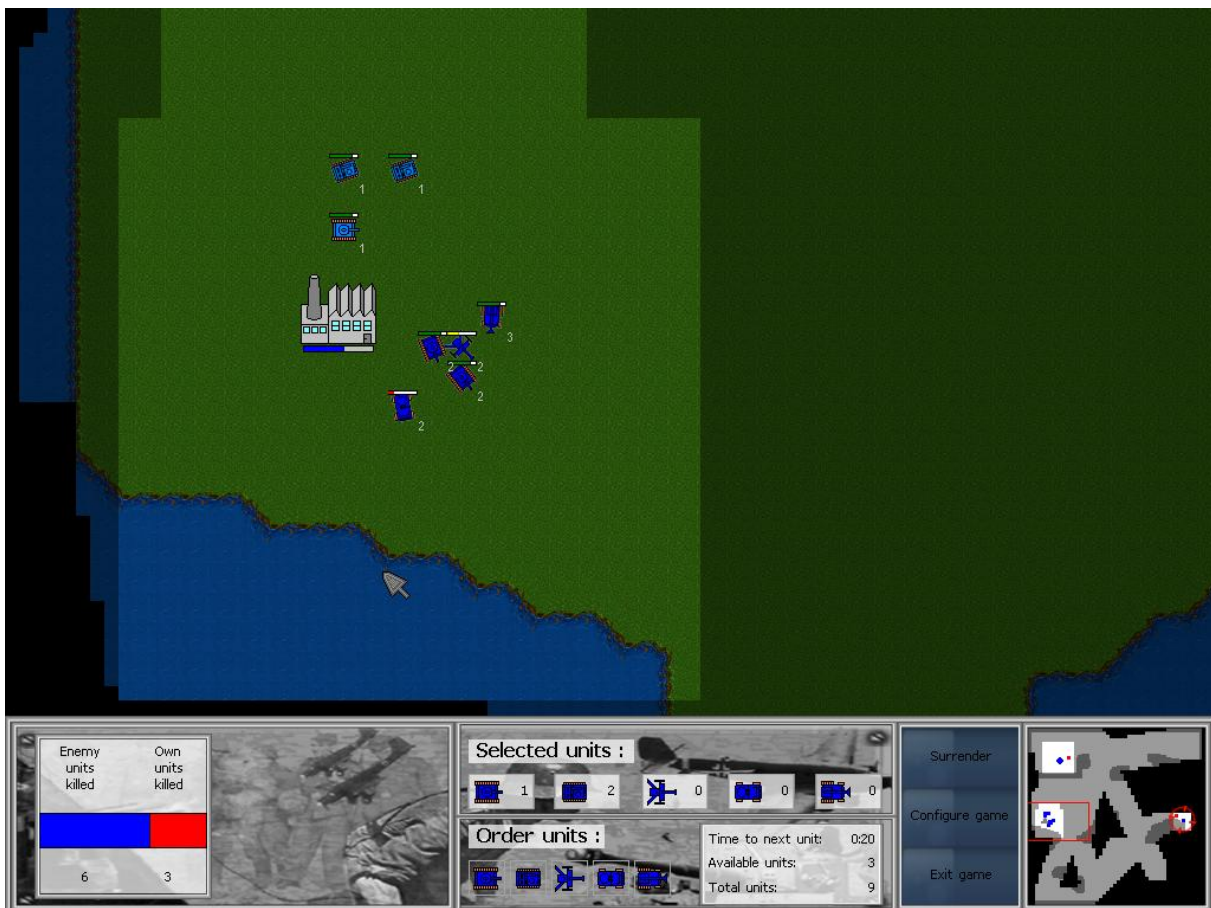
Appendix G

SCREEN SHOTS

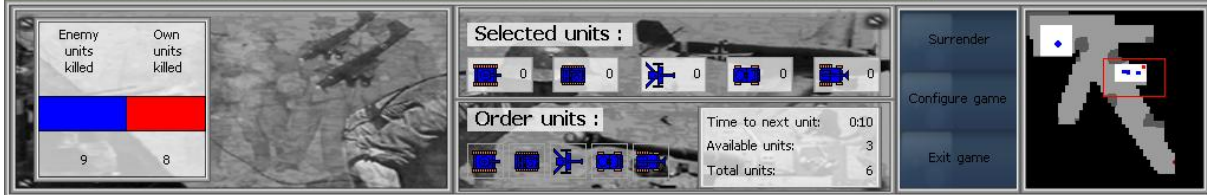
Appendix Screenshots.



Screenshot 1.



Screenshot 2.



Screenshot 3.



Screenshot 4.



Screenshot 5.

Current simulation info:
Blue: AI_Aggressive
Red: AI_MultiLayeredRL

HEADQUARTERS:
Blue: 2000 hp (100%)
Red: 1960 hp (99%)

Power distribution:

	Blue	Red
Kills	9	26

	Blue	Red
Total	2	24
Tank	0	5
Anti Tank	0	5
Mech. Inf.	0	5
Artillery	2	8
Recon	0	1

	Blue	Red
Total	17	24
Spend	17	22
Tank	0	4
Anti Tank	0	4
Mech. Inf.	0	4
Artillery	17	10
Recon	0	0

	Blue	Red
Captured	1	2
Defended	0	0

EXPLORATION:
Blue: 66%
Red: 66%

Game time: 00:07:18
FPS: 2

SIMULATION INFO:
Simulation 1 of 500

TIME:
Total: 00:00:31
Average: 00:00:31
Est. Remaining: 04:18:20

GAME TIME:
Total: 00:07:18
Average: 00:07:14
Est. Total Time: 60:16:40

Time multiplier: 20

SCORE:
Blue wins: 0
Red wins: 0
Draw: 0

Average ms per update
Blue AI: 0
Red AI: 0,0720823798627

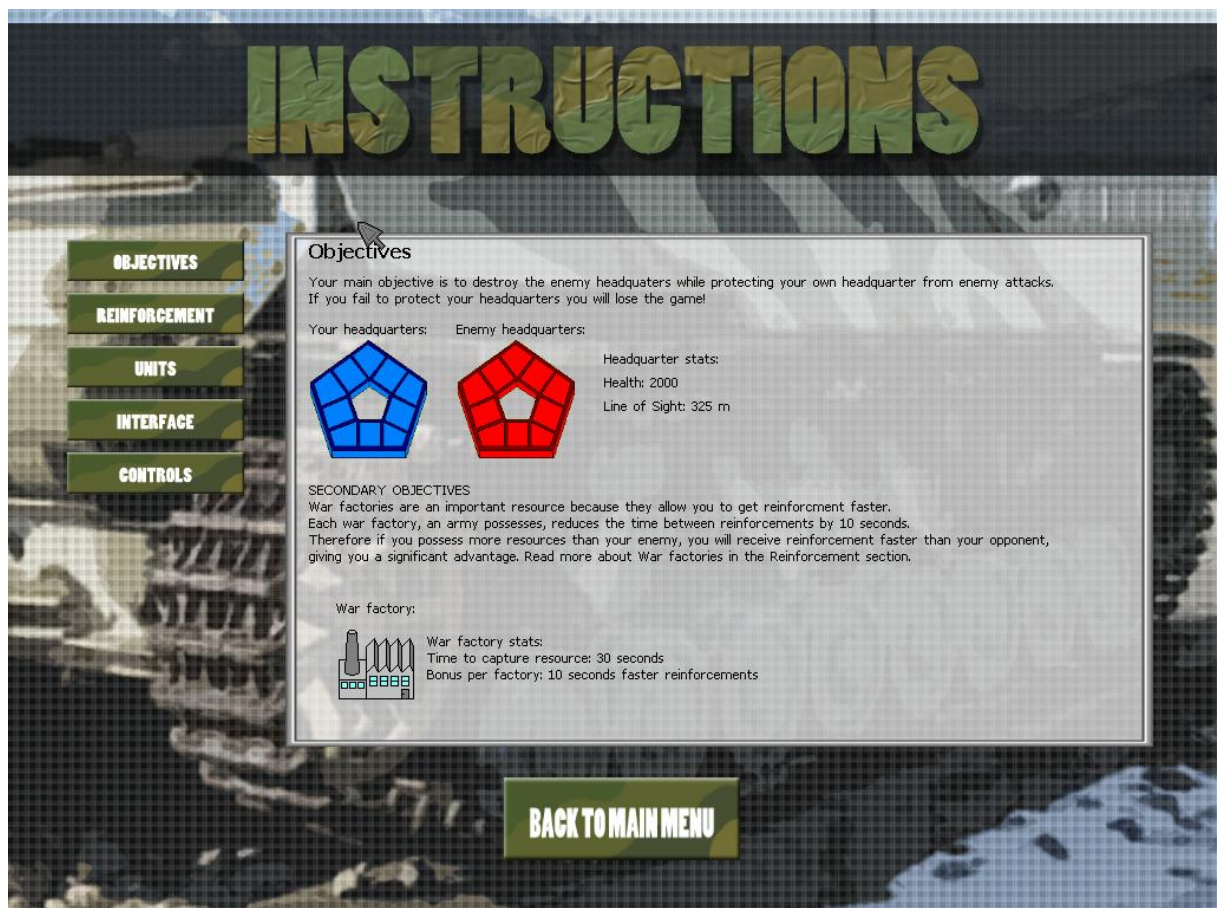
Highest ms in an update
Blue AI: 0
Red AI: 16

Simulation history:

Screenshot 6.



Screenshot 7.



Screenshot 8.

Resumé

This report documents our research and results during our master thesis in which we propose a framework for successfully implementing Reinforcement Learning in a Real Time Strategy game.

The goal of this thesis is: "Investigation of the feasibility of monitoring human opponents' strategy in an RTS game to counter it by predefined rewards and policies, where policies are updated using Reinforcement Learning."

We propose three new extensions to the exiting theory of Reinforcement Learning. The first one is a *multi layered* approach, which splits the state space into a hierarchical structure to bring down the number of states. In standard Reinforcement Learning the structure is single layered, meaning that the policy file will contain the whole state space. Furthermore each eligible action for each state also has to be maintained in a data structure. When the state space reaches a given size, it may become infeasible to use this approach due to the memory requirements or the training time required to reach all appropriate states. This also means that it will take longer time for the RL algorithms to converge to an optimal policy. The behavior of the agents can also be negatively affected by the standard framework because of the number of games required before the agent takes qualified decisions. Therefore we propose a *multi layered* framework consisting of a top level RL system and several lower level RL systems.

The second proposal is called the *profiler*, which through monitoring the enemy, determines its currently used strategy and apply an appropriate counter strategy by swapping the active policy and reward set. This should make the RL AI better to adapt to different enemy strategies.

The last proposal is an extension to Q-learning and SARSA algorithms, we call *Backtracking*. Q-learning and SARSA updates their policies based on the information availably one step back. If instead the last n steps are updated based on the newly received reward, the values will be propagated back faster. It is our belief that the algorithm will converge faster than Q-learning and SARSA.

These three proposals constitute a framework which can be applied to any RTS game. To prove that the framework improves the Reinforcement Learning AI, we have developed a full RTS game called Tank General, in which the proposed extensions are tested. Tank General is a full working RTS game, with a world war two theme, where two teams battle each other, while capturing resources to accelerate their income, to get the upper hand.

The *profiler*, *multi layer* and *backtracking* extensions are tested by letting different RL AI's which uses combinations of these extensions, play a large number of matches against three different scripted AI's.

The results clearly show that using the *multi layered* approach is a big advantage. When dealing with a lot of state space attributes, as it is the case in our game, a *multi layered* approach will reduce the state space considerable. This reduction is an advantage because many state space attributes normally results in an enormous number of states which again results in a very slow learning rate because of the many states which need to be visited.

It also shows that there are pros and cons regarding *profiling*. It might not always be an advantage to use *profiling* since learning can take more time because there are more policies which need to be filled. Dependent on the opponent another clear disadvantage can be the case where the profiler is not able to make a specific classification. However in our tests we did not encounter this problem since our game has a limited number of good strategies which our profiler was designed to profile against. In our game, profiling performed better in most cases and the reason was the ability of the learning algorithm to quickly change behavior and react when the opponent changes behavior.

Generally our proposed *backtracking* method did not generate the results we hoped for. After we discovered that the backtracking did not improve the AI in our game, we created a tool to highlight the causes of the poor performance. With this tool it is easy to check how different algorithms performs, because the simulation is so simple, and can be computed very fast. Using this tool we conclude that it is statistically sufficient to summarize the history based on the previous state, and therefore do not receive further improvement by updating more steps back. To further investigate if this result also applied to the theoretically base of *Eligibility Traces* called *n-step* we implemented this extension into Q-learning. Fortunately *n-step* showed much improvement compared to standard Q-learning.

While the backtracking do not give a performance boost, the multi layered structure and the profiler performs very well in an RTS game. Together they reduce the state space drastically, while reducing the time it takes for the RL algorithms to converge. Therefore the conclusion of this thesis regarding our project goal is that it definitely is feasible to use our proposed framework in an RTS game.