Technical University of Crete
Department of Electronic and Computer Engineering

Reinforcement Learning in Real Time Strategy Games
Case Study on the Free Software Game Glest

Vlasios K. Dimitriadis

Committee
Assistant Professor Michail G. Lagoudakis
(Supervisor)

Assistant Professor Katerina Mania

Assistant Professor Nikolaos Vlassis
(Department of Production Engineering and Management)

Chania, September 2009

Πολυτεχνείο Κρήτης
Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Ενισχυτική Μάθηση σε Παιχνίδια Στρατηγικής Πραγματικού Χρόνου Εφαρμογή στο Παιχνίδι Ελεύθερου Λογισμικού Glest

Βλάσιος Κ. Δημητριάδης

Εξεταστική επιτροπή
Επίκουρος καθηγητής Μιχαήλ Γ. Λαγουδάκης
(Επιβλέπων)

Επίκουρος καθηγήτρια Κατερίνα Μανιά

Επίκουρος καθηγητής Νικόλαος Βλάσσης
(Τμήμα Μηχανικών Παραγωγής και Διοίκησης)

Χανιά, Σεπτέμβρης 2009

This work is dedicated to all the people that stood by me and supported me all those years!!!

- *My committee.*
- *My family.*
- *My best friends Nikos, Kostas, Konstantinos, and Vassilis.*
- *My good friends Stavroula, Sevi, Maria, and Dimitra*
- *My patient girl, Maria.*

Abstract

With the advent of personal computers and computer gaming, Real Time Strategy (RTS) games have vastly gained ground in the last few years, not only among gamers, but also within the field of Artificial Intelligence (AI), since they represent a challenging domain for autonomous agents. Unfortunately, the performance of AI agents in RTS games is still lagging behind, perhaps due to the fact that most games are commercial and thus companies give more emphasis to graphics than AI player development. Open-Source RTS games on the other hand motivate the development of AI agents by providing source code and documentation that allows users to code their own players. This thesis investigates the design of Reinforcement Learning (RL) autonomous agents who learn to play RTS games through interaction with the game itself. In particular, we have used the well-known SARSA and LSPI algorithms to learn how to choose among different high-level strategies in the popular free RST game Glest with the purpose of winning against the embedded AI opponent. Our learning agent was able to defeat the opponent after a few games in the intermediate level (Duel scenario), but could not defeat the opponent at the hardest level (Tough Duel scenario) where the opponent is cheating against our player. In any case, our study indicates that RTS fans can elevate their gameplay experience to a totally new and challenging level, namely that of playing by designing and programming autonomous players for their favorite games.

Περίληψη

Με την πρόοδο στον χώρο των ηλεκτρονικών υπολογιστών, τα παιχνίδια στρατηγικής πραγματικού χρόνου (real-time strategy games) έχουν κερδίσει πολύ έδαφος, όχι μόνο μεταξύ των φανατικών του είδους, αλλά επίσης και στο χώρο της Τεχνητής Νοημοσύνης, καθώς αντιπροσωπεύουν ένα προκλητικό πεδίο για αυτόνομους πράκτορες. Δυστυχώς, η απόδοση τέτοιων αυτόνομων παικτών σε παιχνίδια στρατηγικής πραγματικού χρόνου, φαίνεται ακόμη να υστερεί, καθώς στην πλειοψηφία τους τέτοια παιχνίδια είναι εμπορικά και οι εταιρείες δίνουν έμφαση στα γραφικά παρά στην ανάπτυξη αυτόνομων παικτών. Η παρούσα διπλωματική εργασία εξετάζει τη σχεδίαση αυτόνομων πρακτόρων ενισχυτικής μάθησης (reinforcement learning) που μαθαίνουν να παίζουν παιχνίδια στρατηγικής πραγματικού χρόνου μέσα από αλληλεπίδραση με το ίδιο το παιχνίδι. Συγκεκριμένα, χρησιμοποιήσαμε τους διαδεδομένους αλγορίθμους SARSA και LSPI για να μάθουμε πώς να επιλέγουμε μεταξύ διαφορετικών στρατηγικών υψηλού επιπέδου στο δημοφιλές παιχνίδι ελεύθερου λογισμικού Glest με στόχο να νικήσουμε τον ενσωματωμένο αυτόνομο αντίπαλο. Ο πράκτοράς μας έμαθε να κερδίζει τον αντίπαλο μετά από λίγα παιχνίδια στο μεσαίο επίπεδο (σενάριο Duel), αλλά δεν κατάφερε να επιβληθεί πάνω του στο δυσκολότερο επίπεδο (σενάριο Tough Duel), όπου ο αντίπαλος κλέβει απέναντι στον πράκτορά μας. Σε κάθε περίπτωση, η μελέτη μας δείχνει ότι οι οπαδοί τέτοιων παιχνιδιών μπορούν να ανεβάσουν την εμπειρία παιχνιδιού σε ένα άλλο τελείως νέο επίπεδο, μεγαλύτερης πρόκλησης, όπου οι χρήστες θα παίζουν σχεδιάζοντας και προγραμματίζοντας αυτόνομους παίκτες για τα αγαπημένα τους παιχνίδια.

Acknowledgements

During the last one and a half year or so, I have come down to what messing up with science means. It really turned out to be much harder than I expected. But I guess, it was exactly this that made me want to elaborate on machine learning and robotics. It remains to see how well I will put up with what comes next.

I would like to take some time here and thank the people who made the completion of my work possible, despite the difficulties I have come across in my personal life and student course.

I would like to thank all my friends who were there for me when I needed them. With those people I have shared the most creative, funny, and happy moments during my stay in Chania.

Of course, it goes without saying that great help was provided by my supervisor, professor Lagoudakis.

Most importantly, I would like to thank my family, my parents, and my sister for all the patience they have shown all these years, their help, support, and love that made everything possible.

Finally, I would like to thank my girlfriend Maria, the person who stood there by my side the whole time through, even during my worst moments, especially during the last few months. I owe her a lot.

September 2009,

Vlasis Dimitriadis

Contents

Chapter

List of figures

Figure **Page**

Chapter 1

Introduction

Real Time Strategy (RTS) games are games in which several players choose races and struggle against enemy factions by harvesting resources scattered over a terrain, producing buildings and units, and fighting one another in order to set up economies, improve their technological skill and level, and win battles, until their enemies are extinct. The better balance you get among economy, technology, and army, the more chances you have to win. Of course, the way you manage your units in gathering, fighting, and building, as well as path finding, play a major role in player performance [6, 7, 8, 15, 23, 24, 29]. Nowadays, most RTS games (or maybe we should say games in general) are commercial, though free and open software solutions are beginning to gain ground in that field too.

1.1 Artificial Intelligence in RTS games

Despite the great technological progress that has been achieved in the fields of computer science till today and the fact that RTS games have been around for more than ten years already, the current performance of Artificial Intelligence (AI) agents in RTS games is considered to be quite poor.

One good reason why AI performance in RTS games is lagging behind is that the vast majority of these games are commercial games. This means that the development of these games follows the rules of market. The main market

rule is maximizing profit. For this reason, companies prefer to put emphasis on graphics than on artificial intelligence in the game. The game industry, unfortunately, dictates that more graphics hardware is sold (as games have greater demands) and more sales of games are made, if the games look more realistic. It has been shown that only about 15% of the CPU time and memory is currently allocated for AI tasks in commercial games. On the other hand, such phenomena do not occur so often in open and free software games and there is an obvious tendency for that to change also in commercial games as graphics hardware is getting faster and memory is getting cheaper.

Another thing that we should note here, is that because AI has not evolved as much as it could have, in most games hard level of game play means cheating. Cheating might refer to things like AI player's unristricted access to information that it should not have (such as enemies' or resources' map positions on unexplored areas), faster gathering of resources, or cheaper units' production.

Another reason for the comparatively poor performance of AI agents in RTS games, is closed-source games. In the past, hardly any company used to give an Application Programming Interface (API) for their game, something that has also changed in the last few years, under the pressure of open and free software games. Users do not only want to be consumers, but also creative developers during their gameplay experience. Programming, even if it is just scripting, has a lot to offer to artificial intelligence.

Last, but not least, RTS games contain a great deal of information in the worlds they describe: environment and map objects, different factions with different kinds of units, buildings, weapons, skills, micro and fast-paced actions constitute only a part of the whole picture. Players are also faced with imperfect information, i.e. partial observability of the game state. For instance, the location of enemy forces is initially unknown. It is up to the players to scout to gather intelligence, and act accordingly based on their available information [6, 7, 8, 23, 24].

All this means, that RTS games might be among the most demanding and complex games and RTS users have to put up with a great deal of information concerning the environment, the state, and the actions (something that adds up to intense processing), in quest for a better strategy.

1.2 Motivation

As we have already pointed out, the level of play of RTS AI players in general is weak. We have reffered to cheating and how often it appears in the artificial intelligence part of RTS games. Usually in most real-time strategy games, there is a standard or parametric way for the AI to play. This can only be clever, if it has been tested and seen that it works fine in most cases. But, there is still a problem.

A human player has to play only a few games before he/she finds out how to beat the AI player. A few games are enough for the human to understand how the computer plays and find a way to "trick" the AI player and win. But what would have happened if the AI player did the same thing or almost the same thing, i.e. learn from the human player in a few games how to play more efficiently? This is exactly what human players usually want: more and more challenging games with sophisticated opponents who cannot be beaten easily by simple tricks [6, 7, 8, 15, 23, 24, 29].

Adopting to human player's behaviour and making AI agents more challenging was one of the reasons that drove us to this work. Another reason was the fact that modern methods for constructing intelligent agents, such as reinforcement learning, have been applied widely to many other fields, such as robotics and even board games, but hardly ever on real-time strategy (RTS) games. This has been a problem under research (without any great results we could say) only in the last few years. What could be more challenging than that?

So, we grew up playing such games and now there is a chance for us to do what we have always wanted: try to find some way to make games more clever and show people that we can also be creative in games. Instead of mouse clicks and fast button presses, now we have some kind of alternative way to beat computers. Simple, but clever, programming and scripting shows the future in gaming. Not just players-consumers, but players-programmers. How much would this have to offer in Artificial Intelligence and computer science really? We believe a lot and this is only the beginning!

1.3 Our contribution

In our project, we applied two well-known reinforcement learning algorithms (SARSA and LSPI) on the Free RTS game Glest. In the first scenario (middle difficulty level), we managed to develop a controller much better than the hand-coded one used by the game's AI. In the second scenario (harder difficulty level) however, it was impossible to beat the default AI agent, as he was cheating while we were only using the standard tactics available to human players. Still, we managed to improve our AI agent's behaviour. One might think that, if we were playing with the same cheating tactics as our opponent, we would probably have achieved to beat him. But, it seems that is not the case! In both scenarios our AI agent learned how to mix three different tactics in order to find the best strategy possible instead of commiting to one of them. The improvement achieved in our player's behaviour using reinforcement learning was the most important outcome. It really indicates that, what we were saying about players-programmers and the contribution all this would have to computer science is not an utopic dream, but it could be the future.

1.4 Thesis overview

This thesis is organized as follows: In Chapter 2 we will provide the necessary theoretical background, which includes a quick introduction to Reinforcement Learning and a brief description of the algorithms we used (Qlearning, SARSA and LSPI, in particular). In Chapter 3 we provide a more detailed problem statement, including a description of the game Glest, and an overview of the related work in the area. In Chapter 4 we describe our approach on applying reinforcement learning methods for learning good strategies. In Chapter 5 we present the most important aspects of the implementation of our approach. In Chapter 6 we report the experimental results of our work, and finally in Chapter 7 we discuss the significance of our results, suggest future work directions, and draw some conclusions.

Chapter 2

Background

2.1 Sequential Decision Making

Machine learning [2, 5, 16] is a scientific discipline concerned with the design and development of algorithms that allow computers to learn based on data, coming from sensors or stored in databases. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on these patterns. Hence, machine learning is closely related to fields such as statistics, probability theory, data mining, pattern recognition, artificial intelligence, adaptive control, and theoretical computer science.

2.1.1 Terminology

Some terms [1, 5] we must be familiar with in machine learning and that will help us understand the various ways of approaching our problem are described here in brief, while others that need to be further analyzed follow in the next subsections.

Supervised Learning is a kind of machine learning where the learning algorithm is provided with a set of inputs along with the corresponding correct outputs, and learning involves the algorithm comparing its current actual output with the correct or target outputs, so that it knows what its error is,

and modify things accordingly.

Reinforcement Learning is the area of machine learning in which an agent learns from interaction with its environment, rather than from a knowledgeable teacher that specifies the action the agent should take in any given state.

State can be viewed as a summary of the past history of the system, that determines its future evolution. It includes all the information on which action decisions are based.

Action refers to any of the possible actions an agent can take in a given state of our system.

Reward is a scalar value which represents the degree to which a state or action is desirable, considering only the current step. Reward functions can be used to specify a wide range of planning goals (e.g. by penalizing every non-goal state, an agent can be guided towards learning the fastest route to a goal state).

Episodes are the subsequences into which the agent-environment interaction breaks naturally, such as plays of a game, trips through a maze, or any sort of repeated interactions. Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states.

Policy is the decision-making function (control strategy) of the agent, which represents a mapping from states to actions.

Value Function is a mapping from states or from state-action pairs to real numbers, where the value of a state represents the long-term reward achieved

starting from that state(or state-action), and executing a particular policy. The key distinguishing feature of RL methods is that they learn policies indirectly, by instead learning value functions. RL methods can be constrasted with direct optimization methods, such as genetic algorithms (GA), which attempt to search the policy space directly.

Model-based algorithms compute value functions using a model of the system dynamics. Adaptive Real-time Dynamic Programming (ARTDP) is a well-known example of a model-based algorithm.

Model-free algorithms directly learn a value function without requiring knowledge of the consequences of taking actions. Q-learning is the best known example of a model-free algorithm.

Temporal Difference (TD) algorithms refer to a class of learning methods, based on the idea of comparing temporally successive predictions. Possibly the single most fundamental idea in all of reinforcement learning.

Monte Carlo methods refer to a class of methods for learning value functions, which estimate the value of a state by running many trials starting from that state and then average the total rewards received on those trials.

Stochastic approximation Most Reinforcement Learning algorithms can be viewed as stochastic approximations of exact Dynamic Programming algorithms, where instead of complete sweeps over the state space, only selected states are backed up (which are sampled according to the underlying probabilistic model). A rich theory of stochastic approximation (e.g Robbins Munro) can be brought to understand the theoretical convergence of RL methods.

Learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn

anything, while a factor of 1 would make the agent consider only the most recent information.

2.1.2 Markov Decision Processes

A Markov Decision Process (MDP) [2, 5, 26, 28] is a probabilistic model of a sequential decision problem, where states can be perceived exactly, and the current state and action selected determine a probability distribution on future states. Essentially, the outcome of applying an action to a state depends only on the current action and state and not on preceding actions or states.

An MDP can be described as a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where:

- $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is the (finite) state space of the process. The state s is a description of the status of the process at a given time.
- $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is the (finite) action space of the process. The set of actions are the possible choices an agent has at a particular time.
- $\mathcal{P}$ is a Markovian transition model, where $\mathcal{P}(s, a, s')$ is the probability of making a transition to state s' when taking action a in state s. A Markovian transition model, means that the probability of making a transition to state s' when taking action a in state s depends only on s and a and not on the history of the process.
- $\mathcal{R}$ is the reward function (scalar real number) of the process. It is Markovian as well and $\mathcal{R}(s, a)$ represents the immediate reward for taking action a in state s at the current step.

An MDP is often augmented to include γ and $\mathcal{D}$ as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{D})$, where:

- $\gamma \in (0, 1]$ is the discount factor. When $\gamma = 1$ a reward retains its full value independently of when it is received. As γ becomes smaller, the importance of rewards in the future is diminished exponentially by γ^t.

- $\mathcal{D}$ is the initial state distribution. It describes the probability that each state in $\mathcal{S}$ will be the initial state. On some problems most states have a zero probability, while few states (possibly only one) are candidates for being an initial state.

The optimization objective in an MDP is to find a way of choosing actions (policy) that yields the maximization (or minimization, depending on the problem) of the expected total discounted reward, which is defined as:

$$\pi* = \arg \max_{\pi \in \Pi(A)} E_{s \sim \mathcal{D}; a_t \sim \pi; s_t \sim \mathcal{P}; r_t \sim \mathcal{R}} \left(\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right)$$

2.1.3 Policies

A policy [2, 5, 26, 28] π is a mapping from states to actions. It defines the response (which may be deterministic or stochastic) of an agent in the environment for any state and it is sufficient to completely determine its behavior. In that sense, $\pi(s)$ is the action chosen by the agent following policy π.

An optimal policy π^* also known as an "undominated optimal policy" is a policy that yields the highest expected utility. That is, it maximizes the expected total discounted reward under all conditions (over the entire state space). For every MDP there is at least one such policy although it may not be unique (multiple policies can be undominated; hence yielding equal expected total discounted reward through different actions).

The state-action value function $\mathcal{Q}_\pi(s, a)$ for a policy π is defined over all possible combinations of states and actions and indicates the expected, discounted, total reward when taking action a in state s and following policy π thereafter:

$$\mathcal{Q}_\pi(s, a) = E_{a_t \sim \pi; s_t \sim \mathcal{P}; r_t \sim \mathcal{R}} \left(\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right)$$

2.2 Reinforcement Learning

Reinforcement Learning is learning in an environment by interaction [20, 26, 28, 30]. It is usually assumed that the agent knows nothing about how the environment works (has no model of the underlying MDP) or what the results of its actions are. Typically, the environment is stochastic, yielding different outcomes for the same situation. In contrast to supervised learning there is no teacher to provide examples of correct or bad behavior.

The goal of an agent in such a setting is to learn from the consequences of its actions, in order to maximize the total reward over time. Rewards are in most cases discounted, in the sense that a reward early in time is "more valuable" than a reward later. This is done mainly in order to give an incentive to the agent to move quickly towards a solution, rather than wasting time in areas of the state space where there is no large negative reward.

Two related problems fall within Reinforcement Learning: Prediction and Control. In prediction problems, the goal is to learn to predict the total reward for a given policy, whereas in control the agent tries to maximize the total reward by finding a good policy. These two problems are often seen together, when a prediction algorithm evaluates a policy and a control algorithm subsequently tries to improve it.

The learning setting is what characterizes the problem as a Reinforcement Learning problem. Any method that can successfully reach a solution is considered as a Reinforcement Learning method. This means that very diverse algorithms coming from different backgrounds can be used; and that is indeed the case. Most of the approaches can be distinguished into Model-Based learning and Model-Free learning. In Model-Based learning, the agent uses its experiences in order to learn a model of the process and then find a good decision policy through planning. Model-Free learning on the other hand tries to learn a

policy directly without the help of a model. Both approaches have their strengths and drawbacks (guarantee of convergence, speed of convergence, ability to plan ahead, use of resources).

2.2.1 Q-Learning

Q-learning [12] is a model-free, off-policy, control algorithm that can be used in an online or off-line setting. It uses samples of the form (s, a, r, s') to estimate the state-action value function of an optimal policy. The simple temporal difference update equation for Q-learning is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a))$$

Essentially, it is an incremental version of dynamic programming techniques, which imposes limited computational demands at every step, with the drawback that it usually takes a large number of steps to converge. It learns an action-value representation, allowing the agent using it to act optimally (under certain conditions) in domains satisfying the Markov property.

Q-learning is an off-policy method; samples can be drawn from D in any arbitrary manner, using any exploration policy. One common exploration policy is the $1-\epsilon$ stochastic policy, where the exploration rate ϵ gradually decreases over time. Under certain conditions (infinitely many samples for each state-action pair and appropriately decreasing learning rate), Q-learning is guaranteed to converge to Q^* with probability one. Once a good estimate of Q^* has been learned, a good policy (the optimal policy, in the limit) can be retrieved as the greedy policy over the learned state-action value function, an operation that needs no model information. Figure 2.1 shows the Q-learning algorithm.

Q-learning at its simplest uses tables to store data. As expected, tabular Q-learning works only for small problems. For larger problems, it can be modified to learn an approximate value function. In this case, it makes use of an approx-

Q-Learning
Input: $D, \gamma, Q_0, \alpha_0, \sigma, \pi$. Learns $\hat{Q}^*$ from samples
// D: Source of samples (s, a, r, s')
// γ: Discount factor
// Q_0: Initial value function
// α_0: Initial learning rate
// σ: Learning rate schedule
// π: Exploration policy
$\tilde{Q} \leftarrow Q_0; \alpha \leftarrow \alpha_0; t \leftarrow 0$
for each (s, a, r, s') in D **do**
$\tilde{Q}(s,a) \leftarrow \tilde{Q}(s,a) + \alpha\Big(r + \gamma \max_{a' \in A} \tilde{Q}(s', a') - \tilde{Q}(s,a)\Big)$
$\alpha \leftarrow \sigma(\alpha, \alpha_0, t)$
$t \leftarrow t + 1$
end for
return $\tilde{Q}$

Figure 2.1: The Q-Learning algorithm

imation architecture, which needs to be differentiable with respect to the free parameters for a well-defined gradient. Unfortunately, Q-learning with approximation looses its convergence properties mostly due to the non-linearity of the maximization operation in the update rule.

2.2.2 SARSA

SARSA (State-Action-Reward-State-Action) [3, 28] is a model-free, on-policy algorithm for learning policies in uknown Markov Decision Processes. It was introduced in the technical note "Online Q-Learning using Connectionist Systems" by Rummery & Niranjan (1994) where the alternative name SARSA was only mentioned as a footnote.

This name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent s_t, the action the agent chooses α_t ,the reward r_{t+1} the agent gets for choosing this action, the state s_{t+1} that the agent will now be in after taking that action, and finally the next action α_{t+1}

the agent will choose in its new state. Taking every letter in the quintuple (s_t , α_t , r_{t+1} , s_{t+1} , α_{t+1}) yields the word SARSA. The temporal difference update equation for SARSA is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

A SARSA agent interacts with the environment and updates its own policy based on actions taken, known as an on-policy learning algorithm. As expressed above, the Q value for a state-action is updated by an error, adjusted by the learning rate α. Q values represent the expected reward received in the next time step for taking action α in state s, plus the discounted future reward received from the next state-action observation. SARSA was proposed as an alternative to the existing temporal difference technique, Watkin's Q-learning, which updates the policy based on the maximum reward of available actions. The difference is that SARSA learns the Q values associated with taking the policy it follows itself, while Watkin's Q-learning learns the Q values associated with taking the exploitation policy while following an exploration/exploitation policy.

One of the key ideas in SARSA is that the policy that is being evaluated is not stationary, but changes slowly from a highly exploratory policy towards a greedy policy. As a result, SARSA tracks the value function of a policy that is slowly improving because of its greediness. In a sense, SARSA implements some sort of policy iteration, where policy evaluation and policy improvement are interleaved. With an appropriately decreasing schedule for the learning rate (i.e. $1 - \epsilon$), an appropriate greediness schedule for the policy under evaluation, and sufficiently many samples, the function $\tilde{Q}^{\pi}$ learned by SARSA converges to the optimal value function Q^*. The optimal policy π^* can then be easily retrieved as the greedy policy over Q^*. Figure 2.2 shows the SARSA algorithm.

This tabular formulation of SARSA works only for small problems. For prob-

SARSA
Initialize $Q(s, a)$ arbitrarily
  repeat {for each episode}
    Initialize s
    Choose a from s using policy derived from Q (e.g. ϵ-greedy)
    repeat {for each step of episode}
      Take action a, observe r, s'
      Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
      $Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma Q(s',a') - Q(s,a)\Big)$
      $s \leftarrow s'; \alpha \leftarrow a';$
    until s is terminal
  until last episode

Figure 2.2: The SARSA algorithm

lems with a large state space, it can be modified to learn an approximate value function. The main requirement is that the approximation architecture is differentiable with respect to the free parameters for a well-defined gradient. Let $\hat{Q}^\pi$ with parameters w^π be a good approximation to the state-action value function Q^π of policy π. SARSA, in this case, learns a set of parameters $\tilde{w}$, so that the learned approximate value function $\hat{Q}$ (with parameters $\tilde{w}$) is close to $\tilde{Q}^\pi$. Given any sample (s, α, r, s'), the SARSA update equation for the parameters $\tilde{w}$ of the architecture, in this case, is:

$$\tilde{w}^{(t+1)} = \tilde{w}^{(t)} + a\left(\nabla_w \hat{Q}\right)\left(r + \gamma\hat{Q}(s', \pi(s'); \tilde{w}^{(t)}) - \hat{Q}(s, a; \tilde{w}^{(t)})\right)$$

Using a linear architecture as the approximation architecture,

$$\hat{Q}(s, a; w) = \sum_{i=1}^{k} \phi_i(s, a) w_i = \phi(s, a)^\intercal w,$$

the SARSA update equation becomes:

$$\tilde{w}^{(t+1)} = \tilde{w}^{(t)} + a\phi(s, a)\left(r + \gamma\phi\left(s', \pi(s')\right)^\intercal \tilde{w}^{(t)} - \phi(s, a)^\intercal \tilde{w}^{(t)}\right)$$

In this case, SARSA tries to approximate stochastically the least-squares fixed point approximation of Q^π. However, because of the non-stationary nature of

LSPI
Learns a policy from samples.

Input: samples D, basis ϕ, discount factor γ, error ϵ
Output: weights w of learned value function
Initialize $w' \leftarrow \mathbf{0}$
repeat
  $w \leftarrow w'$, $\mathbf{A} \leftarrow \mathbf{0}$, $b \leftarrow \mathbf{0}$
  for each (s, a, r, s') in D **do**
    $a' = \arg\max_{a'' \in \mathcal{A}} w^\top \phi(s', a'')$
    $\mathbf{A} \leftarrow \mathbf{A} + \phi(s, a)\Big(\phi(s, a) - \gamma\phi(s', a')\Big)^\top$
    $b \leftarrow b + \phi(s, a)r$
  end for
  $w' \leftarrow \mathbf{A}^{-1}b$
until $(\|w - w'\| < \epsilon)$

Figure 2.3: The LSPI algorithm

policy π, SARSA is not guaranteed to converge. In practice a well-tuned schedule for the learning rate can help alleviate this problem.

2.2.3 Least-Squares Policy Iteration

Least-Squares Policy Iteration (LSPI) [25] is a relatively new, model-free, approximate policy iteration algorithm for control. It is an off-line, off-policy, batch training method that exhibits good sample efficiency and offers stability of approximation. LSPI has met great success in the last few years, applied in domains with continuous or discrete states and discrete actions. LSPI iteratively learns the weighted least-squares fixed-point approximation of the state-action value function of a sequence of improving policies π, by solving the $(k \times k)$ linear system

$$Aw^\pi = b$$

where w^π are the weights of k linearly independent basis functions. Once all weights have been calculated, each action-state pair is mapped to a Q value,

as such:

$$Q(s,a) = \sum_{i=1}^{k} w_i \phi_i(s,a)$$

and the greedy action choice in each state s is given by

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} Q(s,a)$$

LSPI iterates over policies (in fact, weights) until there is no change.

We have to note here that no learning rate is needed in LSPI method in contradiction to other methods, and the order of presentation of samples does not affect in any way the learning proccess. Figure 2.3 summarizes the LSPI algorithm.

Chapter 3

Problem Statement

3.1 A few things about Glest

Figure 3.1: Glest game

Glest [17, 18] is a free 3D real-time strategy game (Figure 3.1), where you control the armies of two different factions: Tech, which is mainly composed of warriors and mechanical devices, and Magic, that consists of mages and summoned creatures in the battlefield. Glest is not just a game, but also an engine to create strategy games based on XML and a set of tools.

3.1.1 TechTree

Though you can adapt the game to whatever you like, there are the following default races in Glest:

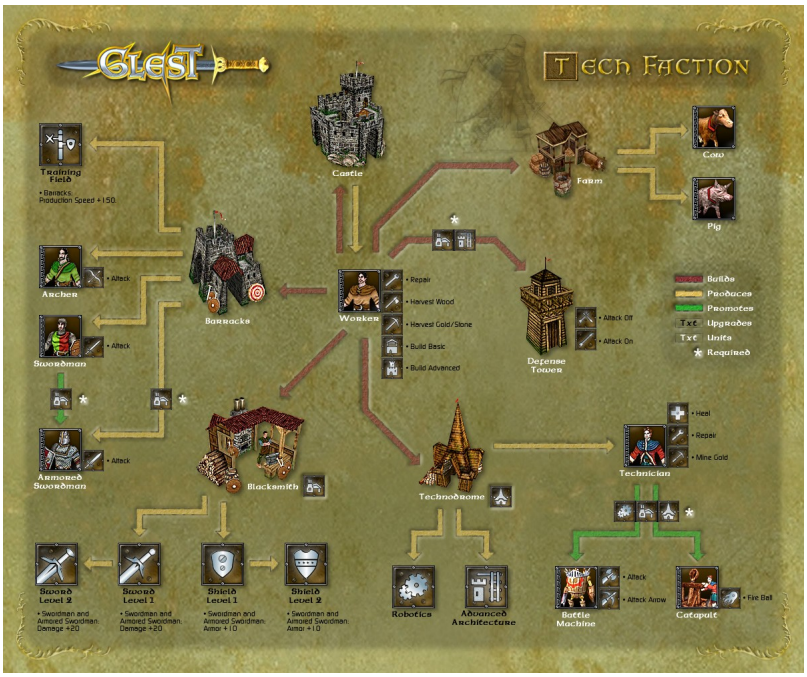

Figure 3.2: Tech Faction

Tech Faction: Uses traditional human warriors and medieval mechanical devices to fight. Strong in hand to hand combat, plays more like other strategy games, but with a few twists. Ideal for new players to get used to the game, but also deep and challenging for experienced players (Figure 3.2).

Magic faction: Targeted towards more advanced players. Most of their units are morphed or summoned from others, and it lacks the hand-to-hand combat strength of the Tech faction. However, it features more versatile and powerful ranged attackers, which if used properly will match or even exceed the power of their technician counterparts (Figure 3.3).

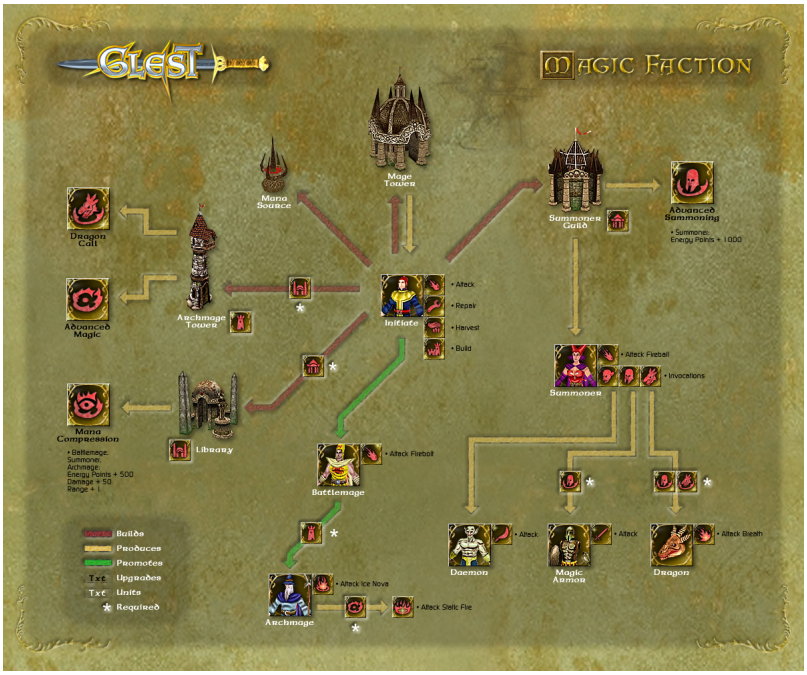

Figure 3.3: Magic Faction

3.1.2 Engine

In Glest modifying everything is quite simple, thanks to the way it is constructed. Some of the main features of the game's engine follow.

XML definitions: Every single unit, building, upgrade, faction, resource, and all their properties and commands are defined in XML files. Just by creating or modifying the right folder structure and a bunch of XML files, the game can be completely modded and entire new factions or tech trees can be created.

G3D models: Glest uses its own 3D format for unit models and animations. A plugin to export from 3DSMax is provided, but tools for Blender are also available.

Map editor: A simple and easy to use map editor is also available to the public.

3.1.3 Technology

The technology used in Glest bears in mind various aspects concerning gaming flexibility and user friendliness. Technology used accomplishes portability, platform independence, single and multiplayer mode, and high-quality entertainment. A brief description of the technical part of the game follows:

Standard C++: All the code is standard C++ and has been written with portability in mind, so it can be compiled using either GCC or VS.NET.

OpenGL: Glest uses the cross platform OpenGL API to render 3D graphics. It uses the fixed pipeline, so only version 1.3 of OpenGL and a few extensions are required.

DirectSound/OpenAL: The game can play static and streaming sounds using ogg and wav files.

Multiplayer: Online game is possible over LAN/Internet with up to 4 players simultaneously.

A.I.: Human players can test their skills against challenging AI controlled opponents. Glest uses the A* pathfinding algorithm and other well known AI techniques.

3.1.4 Modding and features

We should note here that Glest is a Free Software (cross-platform) game, fully modifiable, and also supported by many Glest users. Glest's fans contribute to the game, both aesthetically and technically. Many of them have offered tilesets[1], maps, translations, tools (including the Blender exporter), and many extra

The Glest team has received a few awards, such as the Sony PlayStation ArtFutura (2004), Mundos Digitales (2005), Ytanium Game of the Year (2004) and others.

[1] Tilesets are the collection of graphics that are used to display the Glest world

features. We should not forget the map editor, the g3d viewer, the model exporter for 3DSMax, sample units, tutorials, and other Glest tools that make such contributions and Glest development possible.

Glest's structure and toolkits available offer extra useful features to the game concerning:

- Modding
- Modelling (3DSMax, Blender)
- Reference (XML, Folders)
- Map making
- Scenario making (lua)[2]
- Mods/Maps for Glest
- Compile the source
- Glest Advanced Engine[3]

3.2 Glest Rules

Glest is a Free Software Real-Time Strategy Game that takes place in an environment that could be compared to that of the pre-renaissance Europe. Players build a "town center" type building, collect resources, and build war units. In Glest, the two teams are humans and mages. Humans include workers, soldiers and archers. The mages have a small variety of different wizards and witches, as well as a big tree golem with extremely slow movement but great stamina.

[2] In newer Glest versions (3.2.0 beta and newer) lua scripting gives opens a new possibility of scripted scenarios.

[3] Glest Advanced Engine (GAE) is an advanced mod involving the modification of the Glest engine. At this time, tentative plans still exist to integrate GAE back into Glest once it has achieved a portion of it's goals, has integrated all latest Glest features (such as lua scripting and all features currently in Glest 3.2 beta) and has stabilized.

Figure 3.4: Glest in action

There are also various types of buildings and constructions, some of them used as weapons. In Glest someone can play as a warrior using the weapons and fight, or play as a magician and use spells and magic forces.

A great feature in the game is that you can train your own units on each other. Killing your own (cheap) units, can transform your simple warriors to elite soldiers. Another great feature in the game is the way that some of the soldiers can build other units, such as siege towers, or even better: late-medieval death-robots. Magical side though is a bit tedious and relies a lot on energy to fight. This is one of the reasons why the technologists' archers will win pretty much every time. On the other hand, magicians' attacks are powerful and their buildings look pretty nice.

What you have to do in short in Glest, is to make your own army, make archers and weapons, learn new spells, group your people and battle your adversaries. Despite the differences in the two factions, the final goal for both sides is the same. In order to win, you have to extinct all enemy units. Finish the fight, or die in shame!

3.3 Facing the problem

Almost every Real Time Strategy game uses a standard or parametric way of playing. This means that the AI of such games is usually not very intelligent, and, if someone finds a certain way to beat this AI, playing the same way every time would always lead to human player's victory. This does not sound very challenging for a player, and becomes quite boring after a few games. So, the problem we come up against is to develop a controller that could adapt to any player's behaviour and improve gradually through a few games.

3.4 Related research in RTS games

Games always played a major role in the development of Artificial Intelligence. But the sort of games that were usually picked up, were simple games, mostly board games without hidden information or uncertain environments. But what happens with those games that are not fully observable or action explorable?

In the last few years, through the rapid development and evolution of RTS games, many researchers got involved with that field too, trying to apply Artificial Intelligence in those games. As a matter of fact, what was tried first, was the design of opponents who would be able to face any player dynamically, using learning, not just another static strategy. Some researchers used reinforcement learning, while others chose different AI techniques and machine learning algorithms, in order to solve "the problem". Some of them adopted a multi-agent approach to the problem, trying to make the various agents of their player coordinate appropriately in order to achieve goals, while others had a more abstract, but easier to handle approach, treating their own player as a single agent. Guestrin, Koller, Gearhart, and Kanodia (2003) [10] applied relational Markov Decision Process models for specific RTS game cases, such as three-on-three combats. Cheng and Thawonmas (2004) [13] proposed a case-

based plan recognition approach for assisting RTS players, but only for low-level management tasks. Spronck et al. (2004) [27] and Ponsen and Spronck (2004) [21] implemented a reinforcement learning (RL) technique appropriate for video games, called dynamic scripting. Though they got quite good learning performance on the challenging task of winning, there was a severe disadvantage in this approach. Dynamic scripting requires a considerably reduced state and action space to be able to adapt sufficiently fast. Ponsen and Spronck (2004) manually designed game tactics (stored in knowledge bases) by evolving high-quality domain knowledge in the domain of RTS games with an offline evolutionary algorithm [22]. Although the approach was able to generate strategies with surprisingly high fitness values, it needed many experimental games to test the large number of possible full strategies [15].

Aha, Molineaux, and Ponsen (2005) [14] built on the work of Ponsen and Spronck (2004) by using their evolved tactics and a case-based reasoning technique which decides which of those tactics are more appropriate in each case, given the current state and opponent. This appoach's greatest benefit in comparison to Dynamic Scripting is that it is able to cope well with different opponents, even with random consecutive opponents [15, 22].

Marthi, Russell, and Latham (2005) [4] applied hierarchical reinforcement learning in a limited RTS domain. This approach used reinforcement learning augmented with prior knowledge about the high-level structure of behaviour, constraining the possibilities of the learning agent and thus greatly reducing the search space [15]. The action space consisted of partial programs, essentially high-level preprogrammed behaviors with a number of choice points that were learned using Q-learning [22].

Ponsen, Muñoz-Avila, Spronck and Aha (2006) [22] introduced the Evolutionary State-based Tactics Generator (ESTG), which uses an evolutionary algorithm to generate tactics automatically. The method in contrast to many previous

methods focuses on the highly complex learning task of winning complete RTS games and not only specific restrained scenarios. Experimental results showed that ESTG improved dynamic scripting's performance in a real-time strategy game, concluding that high-quality domain knowledge can be automatically generated for strong adaptive game AI opponents. The greatest benefit is that ESTG considerably reduces the time and effort needed to create adaptive AI agents.

Eric Kok, Frank Dignum, and Joost Westra (2008) [15] proposed a multi-agent approach, in which each agent could autonomously reason on goals and world belief, using Dynamic Scripting and Monte Carlo methods. To better cope with opponents that switch strategies, implicit and explicit adaptation was tried out. This kind of adaptive learning showed that incorporating opponent statistics into the learning process of a Monte Carlo agent gives the best learning results, as agents could learn quickly and consistently how to outperform the scripted fixed and strategy-switching players.

Guestrin, Lagoudakis, and Parr [11] proposed coordinated reinforcement learning for cooperative agents. In this approach, agents make co-ordinated decisions and share information to achieve a principled learning strategy. This method successfully incorporated the cooperative action selection mechanism derived by Guestrin et al. [9] into the reinforcement learning framework to allow for structured communication between agents, each of which has only partial access to the state description. Three instances of Coordinated RL were presented: Q-learning, LSPI, and policy search. The initial empirical results demonstrated that reinforcement learning methods could learn good policies using the type of linear architecture required for cooperative action selection and that, using LSPI, could lead to reliably learned policies that were comparable to the best policies achieved by other methods and close to the theoretical optimal achievable in the test domains.

Finally, we should refer to Michael Chung, Michael Buro, and Jonathan Scha-

effer (2005) [24] , who described a plan selection algorithm - MCPlan - which was based on Monte Carlo sampling, simulations, and replanning. Applied to simple CTF (capture the flag) scenarios MCPlan showed promising initial results.

Chapter 4

Approaching the problem

In our approach, we tried to formulate our problem in such a way that would not require keeping all the detailed information contained in our game. For this reason we used an abstraction method, in order to describe states and actions of the game. We did not consider states and actions at a primitive level. Instead, we gathered states and actions and grouped them, in order to compand action and state spaces, keeping only the important information for processing. We used two well-known reinforcement learning methods, such as SARSA and LSPI, in our approach, though we also experimented with Q-learning at the beginning. Those initial Q-learning experiments lead us to certain conclusions, remapping of the initial plan, and parameters' initial values for subsequent experimentation.

4.1 Basics

4.1.1 States description

Each state in our problem is described by the values of four variables[1]: kills, units, resources, and timer.

Kills refers to the number of kills that the player has achieved.

Units is a variable that counts the units the player has produced.

[1] All four state variables are discrete.

Resources refers to the resources the player has harvested.

Timer is a counter, counting game cycles (game time units).

We assumed that those four variables are enough to describe the whole state without neglecting any important information. That was our state abstraction, as information concerning the map environment or our position would need a lot of space and it is something that has to do with the path-finding processes, which are anyway included and companded in abstract actions.

There is not a definite upper bound for any of the variables' values, as each bound value depends on the total duration of the game and the actions taken. All variables are non-negative and the timer value loops at the value of 2400. In any case, due to the large size of the state space tabular approaches are inapplicable. Therefore, we have to rely on approximation techniques.

Figure 4.1: Magic vs Magic

4.1.2 Actions structure

Game actions are structured in several levels. The first level consists of many really primitive low-level actions, such as modifications of basic variables that describe low-level micro states. A simple example of this would be an action

that checks whether a single worker is task-free or not and sends him to a specific location, if he is task free. In the second level of actions, grouped primitive actions form a more abstract task or rule. There are thirteen different rules in our game with the following names, discreptive of what each rule does: WorkerHarvest, RefreshHarvester, ScoutPatrol, ReturnBase, MassiveAttack, AddTasks, ProduceResourceProducer, BuildOneFarm, Produce, Build, Upgrade, Expand, Repair. The third and more abstract level of actions is more like a strategy-tactic level, consisting of grouped second level actions-rules. The grouping has been defined with respect to the timer of the game. Each second level action-rule is associated with a certain time interval. Whether a certain rule will be picked up and applied by the tactic, depends on the modulo of the timer of the game with the rule's interval. If this modulo is zero, then the rule is selected and it is applied in the current time step. In other words, the rule time interval defines the frequency of occurence for each rule. This check is performed for all thirteen rules at each time step, therefore any number between 0 and 13 rules could be executed at once. Due to the fixed rule time intervals the whole strategy eventually loops over time. The original AI agent of the game consists of such a tactic with specific rule interval values.

Figure 4.2: Tech vs Tech

All this lead us to form the three third-level actions-tactics we used. We wanted a fair play to the original AI agent and for that reason we picked up a similar logic in our strategies. We did not want to play at different speed (faster or slower), as this would be in a way "unfair" to the original player. So our way of playing, used three similar tactics. The first one was exactly the same as the original. The second and the third were similar, but with the timer shifted by certain intervals (800 and 1600 timer units) on the modulo calculation. That means that actions in the second and the third tactics were tried earlier (or much earlier) than in the first tactic. Without knowing whether an action applied earlier in time would do us good or bad, what we wanted as outcome of our experiments, was to find out a specific strategy of mixing all three tactics, in a way that would give us a better result compared to the original way of playing.

4.1.3 Reward function

In reinforcement learning we have to define a reward function. In Glest the winner is the player with the highest score, therefore we decided that the reward given at each discrete step of the game-interface cycle, should be derived from the way the game score evolves. So what we did was to take the score calculation function and adopt it to deliver as reward the increase in score from one step to the next. In particular, the reward for the transition to a new state from an old state is given by[2]:

$$reward = d(kills) + d(units) + d(resources)$$

[2]The weights used in the calculation of the reward are exactly those used by Glest to calculate the score.

Where:

$$
\begin{aligned}
d(kills) &= \Big[(kills^{(our)}_{<newstate>} - kills^{(our)}_{<oldstate>}) \\
&\quad -(kills^{(enemy)}_{<newstate>} - kills^{(enemy)}_{<oldstate>})\Big] \times 100 \\
d(units) &= \Big[(units\_produced^{(our)}_{<newstate>} - units\_produced^{(our)}_{<oldstate>}) \\
&\quad -(units\_produced^{(enemy)}_{<newstate>} - units\_produced^{(enemy)}_{<oldstate>})\Big] \times 50 \\
d(resources) &= \Big[(resources^{(our)}_{<newstate>} - resources^{(our)}_{<oldstate>}) \\
&\quad -(resources^{(enemy)}_{<newstate>} - resources^{(enemy)}_{<oldstate>})\Big] \div 10
\end{aligned}
$$

4.2 Learning

The main learning procedure consists of the adaptation of the SARSA and LSPI algorithms to our problem formulation, and the use of those two methods on our design. We need to say here, that the opponent was fixed to the original high-level tactic in all experiments.

4.2.1 Basis Functions

In both methods of reinforcement learning we finally applied (SARSA and LSPI) we had to find an appropriate set of basis functions to use. At first we tried 15 basis functions consisting of the constant term (1), the four simple state variable terms, the six cross terms, and the four square powers of the simple state variable terms, but finally we ended up using only 11 basis functions, leaving out the squared terms as this proved to work much better for our case. We must note here that we normalized the values of all basis functions to [0,1] to eliminate magnitude differences. We did that by using the proper normalizer[3] in each case (finding the maximum values of each simple term and their cross terms in a proportionate round of dummy-sampling experiments).

[3]Each one of the max values in the following equations stands for the proper normalizer in each case

Therefore, the final set of basis functions was the following:

$$
\begin{aligned}
bf1 &= 1 \\
bf2 &= resources / \max resources \\
bf3 &= units / \max units \\
bf4 &= kills / \max kills \\
bf5 &= timer / \max timer \\
bf6 &= \frac{resources \times units}{\max resources \times \max units}
\end{aligned}
$$

$$
\begin{aligned}
bf7 &= \frac{units \times kills}{\max units \times \max kills} \\
bf8 &= \frac{kills \times timer}{\max kills \times \max timer} \\
bf9 &= \frac{timer \times units}{\max timer \times \max units} \\
bf10 &= \frac{resources \times timer}{\max resources \times \max timer} \\
bf11 &= \frac{resources \times kills}{\max resources \times \max kills}
\end{aligned}
$$

This block of 11 state-dependent basis functions was repeated separately for each of the 3 actions. Therefore, the total number of weights in our approximation was 33, however only one block of 11 basis functions was active at any time, the one corresponding to the current action.

4.2.2 SARSA

SARSA is an online policy algorithm. That means that the policy is updated from game to game, and therefore learning takes place while the game is being played.

What we did in our algorithm while applying SARSA, step by step was:

Figure 4.3: Advanced Magic

- Formulation of tactics (timer dependent)
- Selection of tactic (randomly picked or by Q-value maximization criterion for state)
- Application of the selected tactic.
- Calculation of reward(score differences).
- Application of the SARSA update rule.

These steps were repeatedly applied in each game cycle till the end of the game.

We must also note here that in terms of exploration and exploitation we used a sigmoid function for exploration discounting while exploitation rose. Here, follows the expression of the sigmoid function we used to determine the criterion (ϵ), according to which we pick up the best (most highly valued) tactic of the three (if random generator gives us a number smaller than $1 - epsilon$) or just a tactic in random (otherwise):

$$\epsilon = \frac{1}{1 + e^{\frac{\delta \times (gNo - tNg)}{2}}},$$

where $\delta = 0.1$, gNo is the game number, and tNg is the total number of games.

4.2.3 LSPI

LSPI on the other hand is an off-line policy algorithm. That means that we used a collection of samples (both randomly and from SARSA's collection) we got from a few games that we played before using LSPI to extract the learned policy. We used the Phoenix framework and Jason Pazis' work [19] in order to develop the system we needed by writing the proper environment, domain, experiment and approximator classes for Glest.

Figure 4.4: Tech in Duel scenario

What we did in LSPI was to take the sample set we mentioned before and begin iterating in an attempt to converge to a good policy. Epsilon's (convergence criterion) value was set to 10^{-6}, while the limit of iterations was set to 30 in order to ensure convergence, though it turned out that this was not needed as LSPI usually converged in 5 or 6 iterations.

Chapter 5

Implementation

5.1 Speaking Glest's language

The Glest game is mainly written in C++. This is the main reason why we also implemented most of our work in C++. Besides, we needed a language quite compact, object-oriented, and surely a language that would allow us to manage memory as best as possible. Writing our code in C++ would satisfy these requirements and would also allow us to incorporate our work into the Glest's system.

It took us quite a long time to study and understand Glest's source code and we had to modify many files and classes in order to achieve what we wanted. But, it was worth it. Besides, learning how to "speak" the "language" of Glest, the whole procedure made us love it even more!

Finally, we must say that we wrote many tools and scripts concerning external work in python and shell script, taking advantage of each language at what is best.

5.2 Implementation Technology

For Glest compilation we used Glest tools and mainly jam (which makes use of g++). We also used the netbeans IDE for the development of source code

and gdb for the debbuging process. The operating system we used was Linux (Slackware at the initial phase of the project and Ubuntu during the last few months), though it could have even been Windows, as everything concerning Glest is cross-platform (from dependency-libraries to Glest tools). As for plots, we used gnuplot and pygnuplot through clever python and shell scripting. For the LSPI part, we also used the Phoenix framework as we have already mentioned and the whole "Glest experiment" took place on a 2.26Ghz dual core processor - 4GB memory laptop, almost totally dedicated to science, at least for the last few months.

Of course, it goes without saying that great care and work has been put into our project, giving high priority to performance, memory management and compact, nicely-indented source code.

5.3 Interventions in the Structure of Glest

Glest's code is structured in various levels (directories) which correspond to different functionalities of the whole game. The AI level for example is concerned with the AI functions of the game, while the menu level is related to option trees, and the sound level describes the sound container and renderer. Levels consist of classes (in different files), each one implementing an individual part of the level. Client and Server Interface classes for example, implement the respective interface parts of the network level.

The main modifications that we performed in Glest's structure were mostly related to the AI level. In the ai class, a member function was added in order to be able to play policy testing games when needed, as well as an extra module for the human interface's behaviour update. In the ai interface class, we also added an additional class function which decides whether to call our ai update or play a policy testing game, depending on the value of a specific AI interface

class variable.

Some additional modifications were made in the Game class of the Game level in order to avoid message boxes and increase the speed at which the game plays. A few more changes were also made in the update controller of the Game class, so that the human interface plays according to our ai update module.

Bypassing of menus, scenarios selections and game iterations were all automated by modifying the main menu class's constructor in the menu level.

Modifications were also made in the afforementioned classes' constructors for weights', states', and parameters' initialisations. Changes in those classes' destructors were made in order to store data values needed for subsequent games, and destroy redundant data structures after the end of each game.

Chapter 6

Results

6.1 Experimental Methodology

The reinforcement algorithms we chose were both applied in two scenarios (Duel and Tough Duel), in order to see how they behave in different environments and levels of game difficulty. Both for SARSA and LSPI we played test games after every 10 learning games, assuming that progress of learning would be notifiable after such a number of games.

Figure 6.1: Magic vs Tech

The final learned policy in each case was a mixture of the three tactics. The percentage of time each tactic was used in a full game changed from game to game during learning, till convergence to a certain mixed-strategy was achieved.

In order to calculate the performance in each game, we initially calculated the difference between our score and the enemy's score on the first game where only the first-default tactic was used by our player. Then we found the difference between our score and enemy score on the current game where the current learned policy was used by our agent. The final performance was calculated by substracting the two score differences and dividing with the first score difference to get the percentage of performance increase. Finally, we must also mention here that the values of the various parameters, such as the discount factor (0.999), the number of learning games (200), the delta (0.1), the learning rate (0.1), were determined by taking into account typical values of these parameters in other kinds of reinforcement learning problems, as well as empirical information.

6.2 Performance Evaluation

In this chapter, we show the results of the SARSA and LSPI algorithms applied on the Glest problem. We compare the two algorithms' performance, in terms of relative improvement (negative improvement values when difference declines). We also show the percentage of each tactic's contribution to a full game, i.e. to the final performance of our algorithms in each game.

We check results for two different scenaria: Duel and Tough Duel - different environments and difficulty levels. We have to note, that in the second scenario the basic tactics of our player are not changed. However, the opponent's way of playing is not the same as in the previous scenario. As a matter of fact, the opponent player is cheating in this level of difficulty (hard and Cpu-Ultra control), the hardest level on a one-on-one Glest scenario. Harvesters gathering resources much faster, units having greater stamina, and unit producing being much "cheaper" than usual are only some of the new "features" in our enemy's

behaviour. But our intension in this project was not to prove that we always win no matter what goes on, just by applying our method. Our point was to show that applying our method on Glest could always lead to a better strategy for our player than the initial hand-coded AI strategy.

6.3 Duel scenario

Duel is a medium difficulty level scenario and the simplest of the two cases we studied. In Figures 6.2, 6.3, and 6.4 we can see the scores achieved using SARSA and LSPI in the 20 test games played. Recall that each policy (testing) game is played after every 10 games of learning.

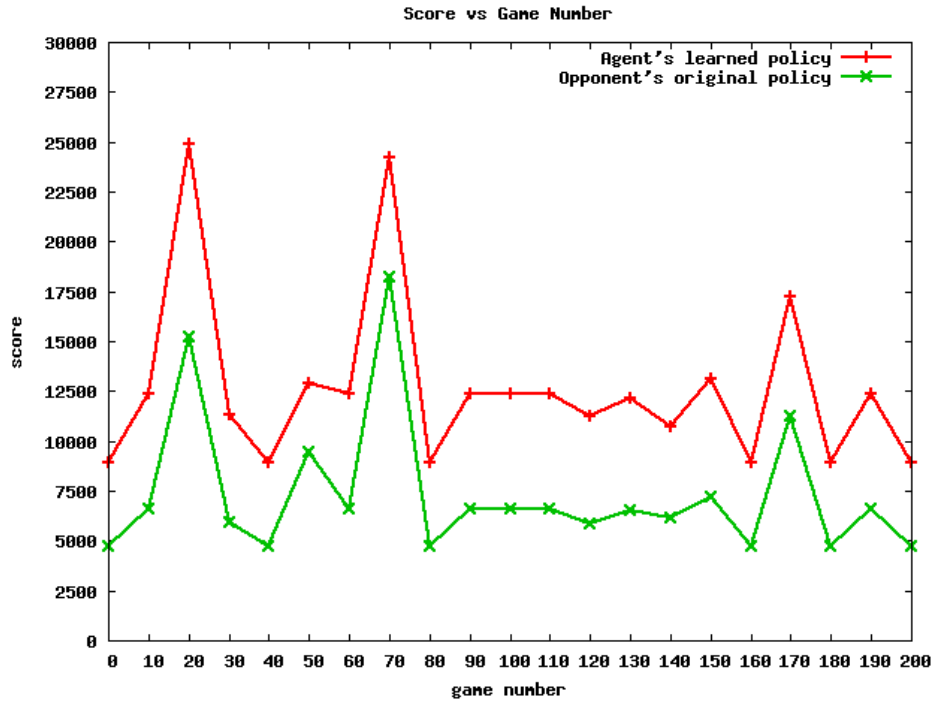

Figure 6.2: SARSA: policy scores on the Duel scenario.

For LSPI algorithm we have results for two kinds of experiments. One case where we used the same games' samples that were extracted and used in SARSA and one with the games' samples were extracted using a totally ran-

dom tactic selection at each game cycle.

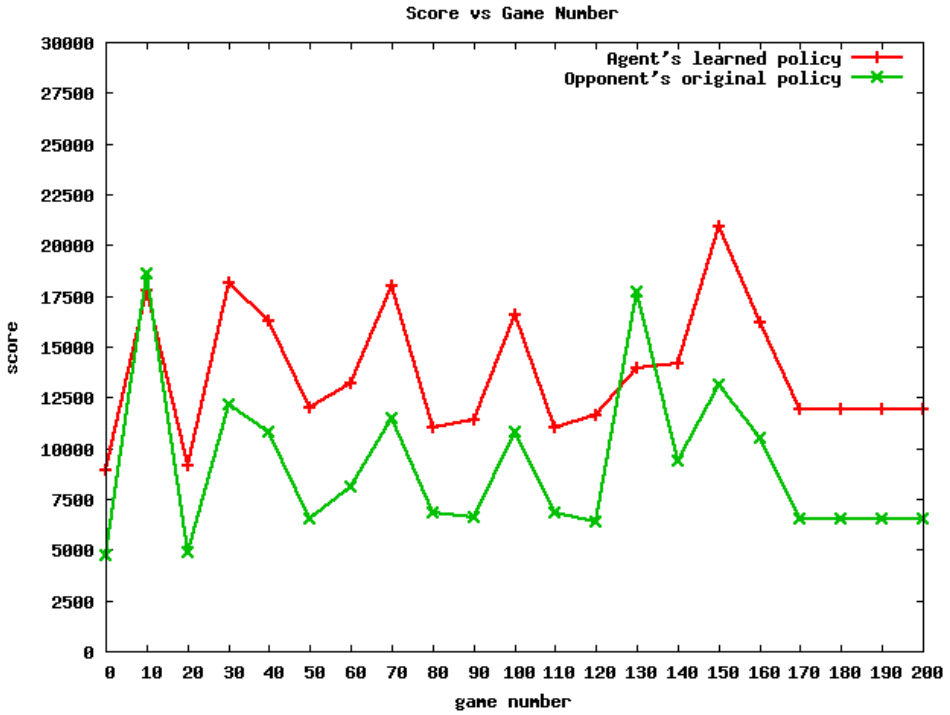

Figure 6.3: LSPI: policy scores on the Duel scenario (SARSA's game samples).

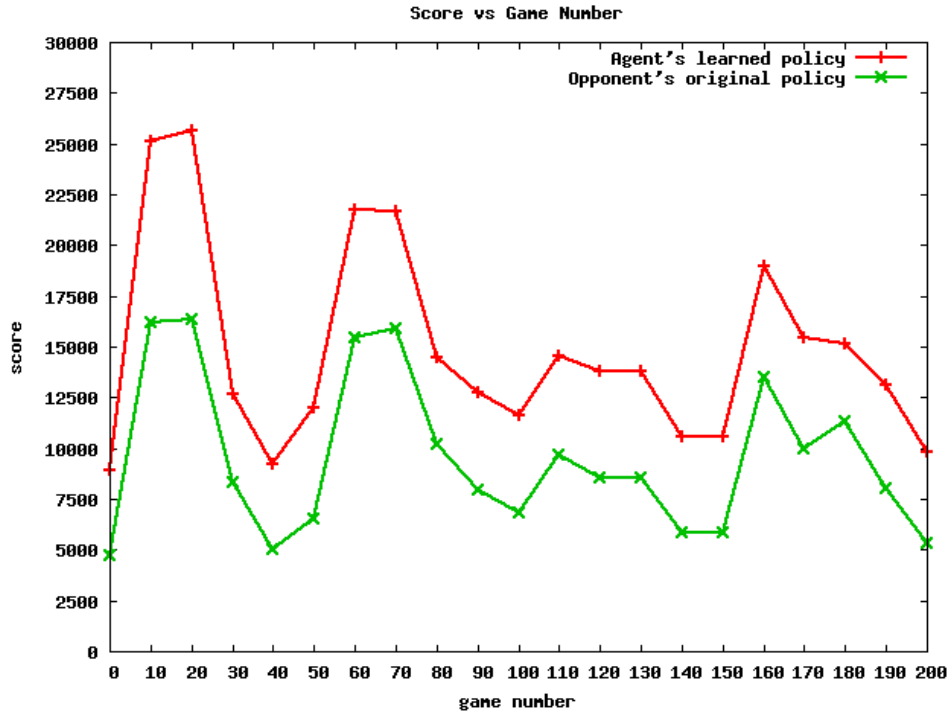

Figure 6.4: LSPI: policy scores on the Duel scenario (random game samples).

In Figures 6.5, 6.6, and 6.7 we see the improvement or decline of relative performance with respect to the default Glest tactic.

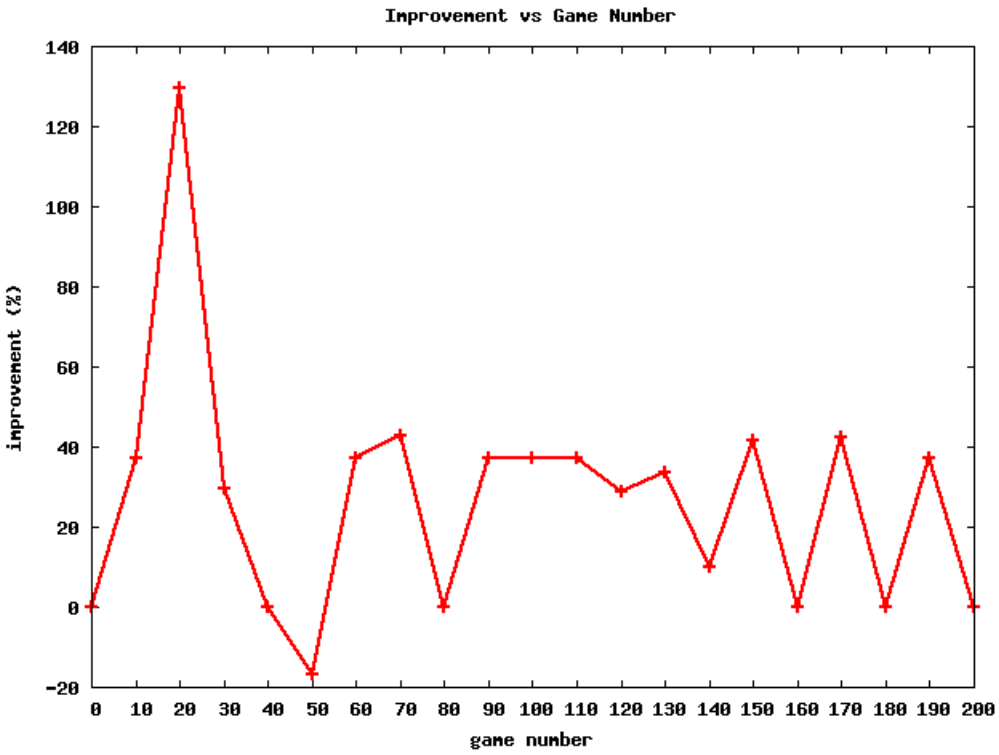

Figure 6.5: SARSA: improvement on the Duel scenario.

Figure 6.6: LSPI: improvement on the Duel scenario (SARSA's game samples).

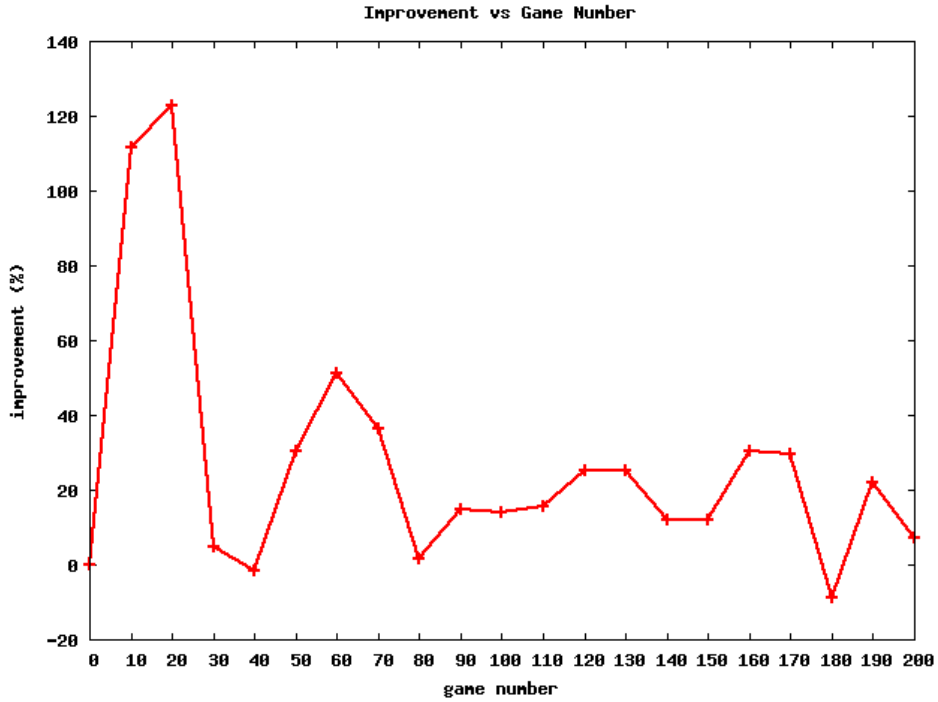

Figure 6.7: LSPI: improvement on the Duel scenario (random game samples).

Finally, for the Duel scenario Figures 6.8, 6.9, and 6.10 show for each test game, the percentage of each tactic's contribution to action choices throughout a full test game.

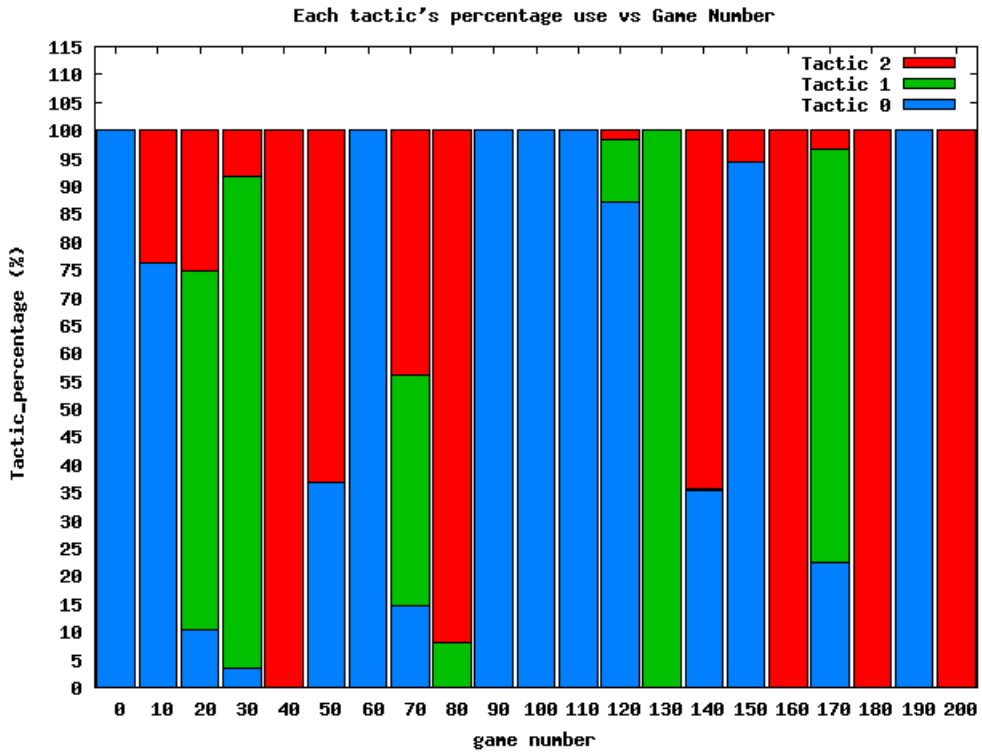

Figure 6.8: SARSA: tactics' contribution to a game, on the Duel scenario.

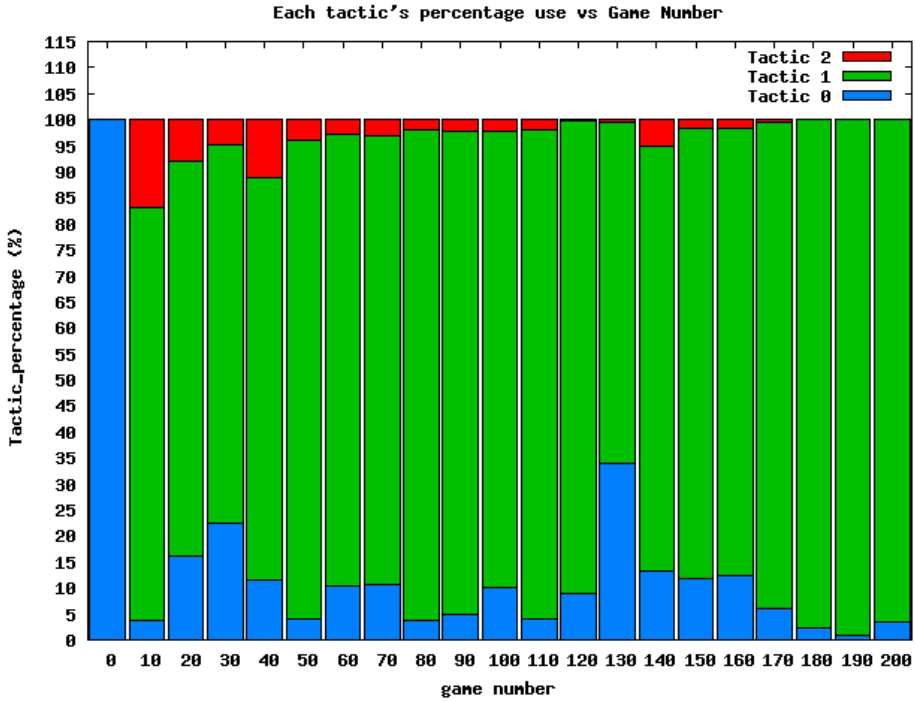

Figure 6.9: LSPI (SARSA's game samples): tactics' contribution to a game, on the Duel scenario.

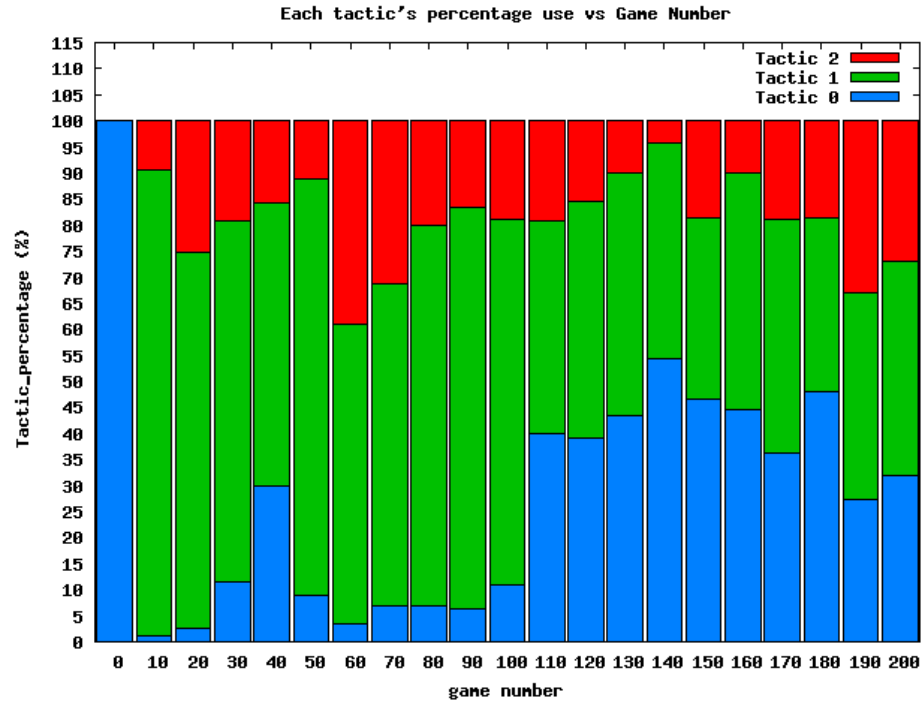

Figure 6.10: LSPI (random game samples): tactics' contribution to a game, on the Duel scenario.

6.4 Tough Duel scenario

Tough Duel is a high difficulty level scenario. As a matter of fact Tough Duel's difficulty level is the hardest of all, on a one-on-one player Glest game. In this subsection, we show the results we got from Tough Duel's scenario experiments. As we have already said, it is totally natural to loose all games we played in this scenario, as the opponent cheated and we were not. We played fairly, with exactly the same basic tactics used in Duel scenario (even though a different combination of them was learned).

In Figures 6.11, 6.12 and 6.13 we can see the scores achieved using SARSA and LSPI in the 20 test games played. Recall that each test game is played after every 10 games of learning.

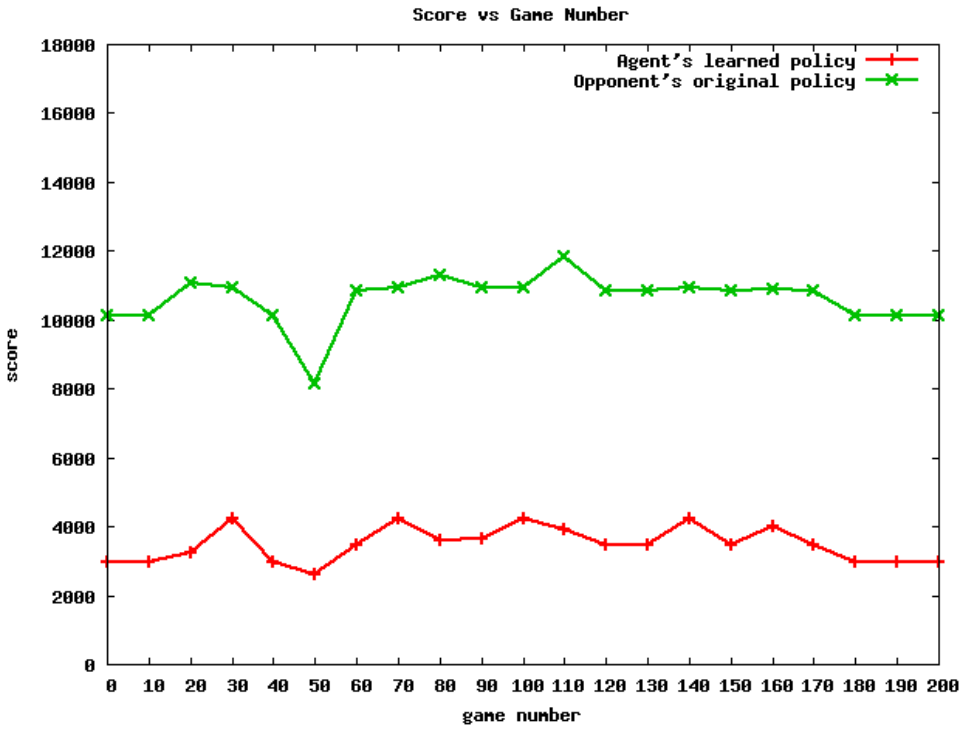

Figure 6.11: SARSA: policy scores on the Tough Duel scenario.

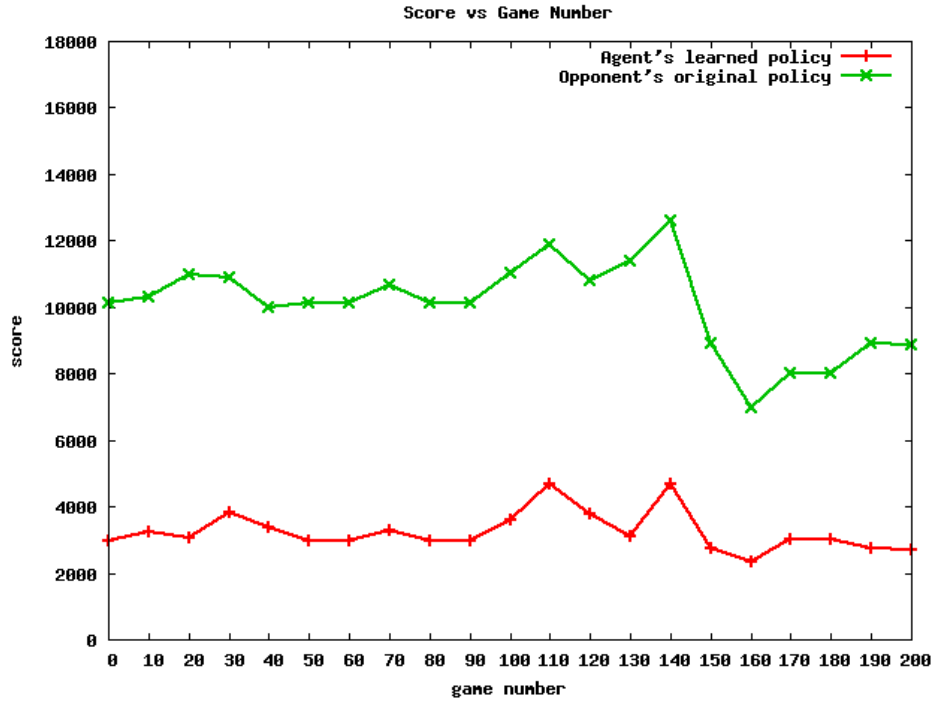

Figure 6.12: LSPI: policy scores on the Tough Duel scenario (SARSA's game samples).

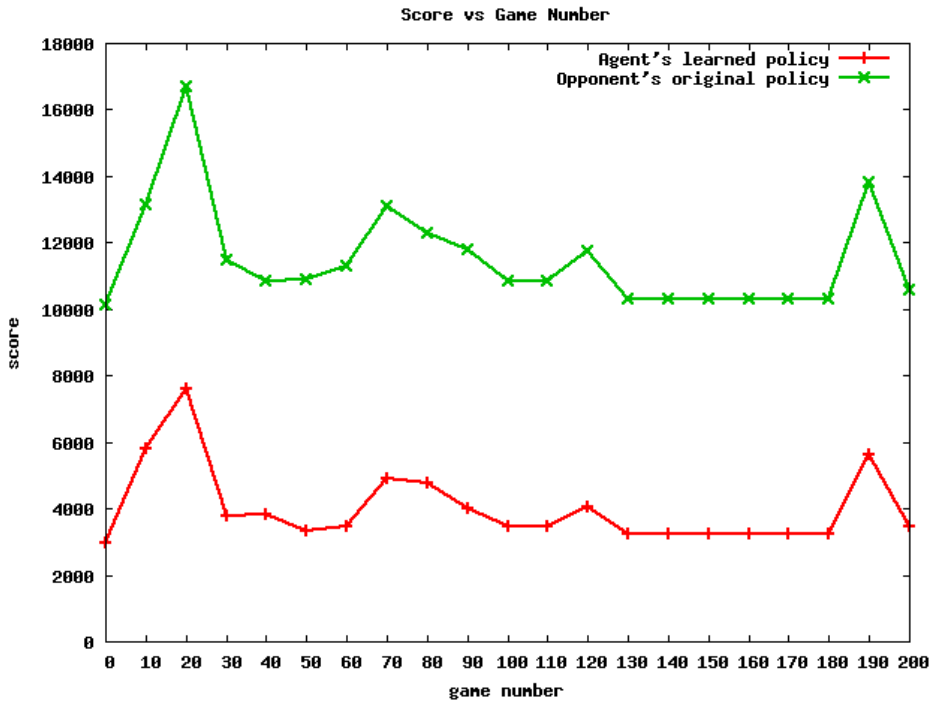

Figure 6.13: LSPI: policy scores on the Tough Duel scenario (random game samples).

In Figures 6.14, 6.15 and 6.16, we can see, as before (but for Tough Duel scenario this time), the improvement or decline of the relative performance over the default Glest tactic.

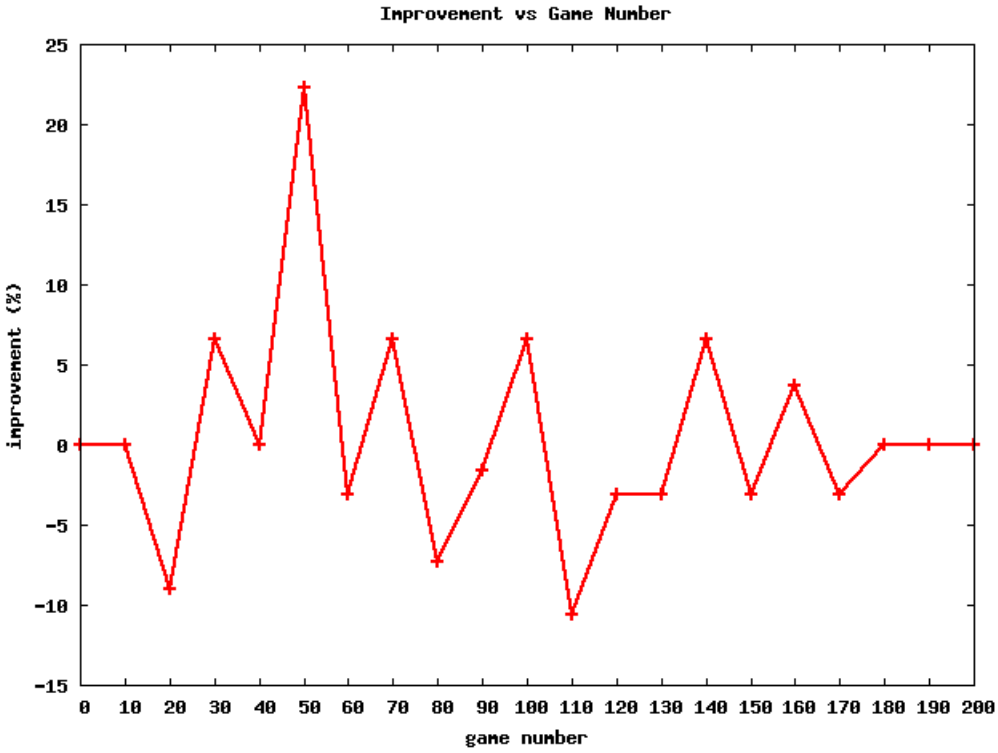

Figure 6.14: SARSA: improvement on the Tough Duel scenario.

Figure 6.15: LSPI: improvement on the Tough Duel scenario (SARSA's game samples).

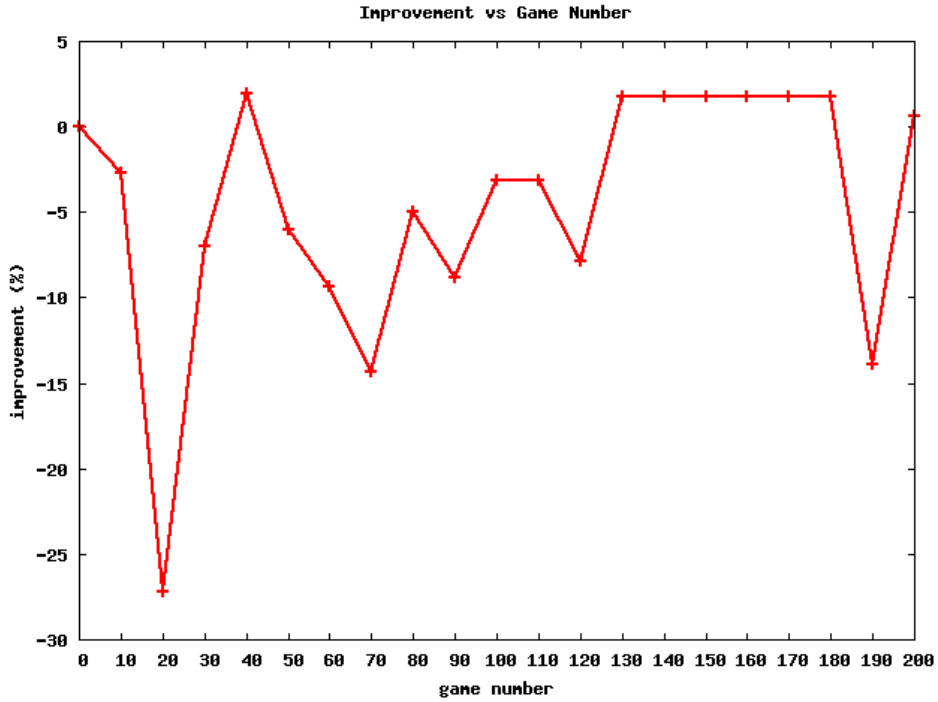

Figure 6.16: LSPI: improvement on the Tough Duel scenario (random game samples).

Finally for Tough Duel scenario, Figures 6.17, 6.18, and 6.19) show for each policy game, the percentage of each tactic's contribution, to action choices throughout a full test game.

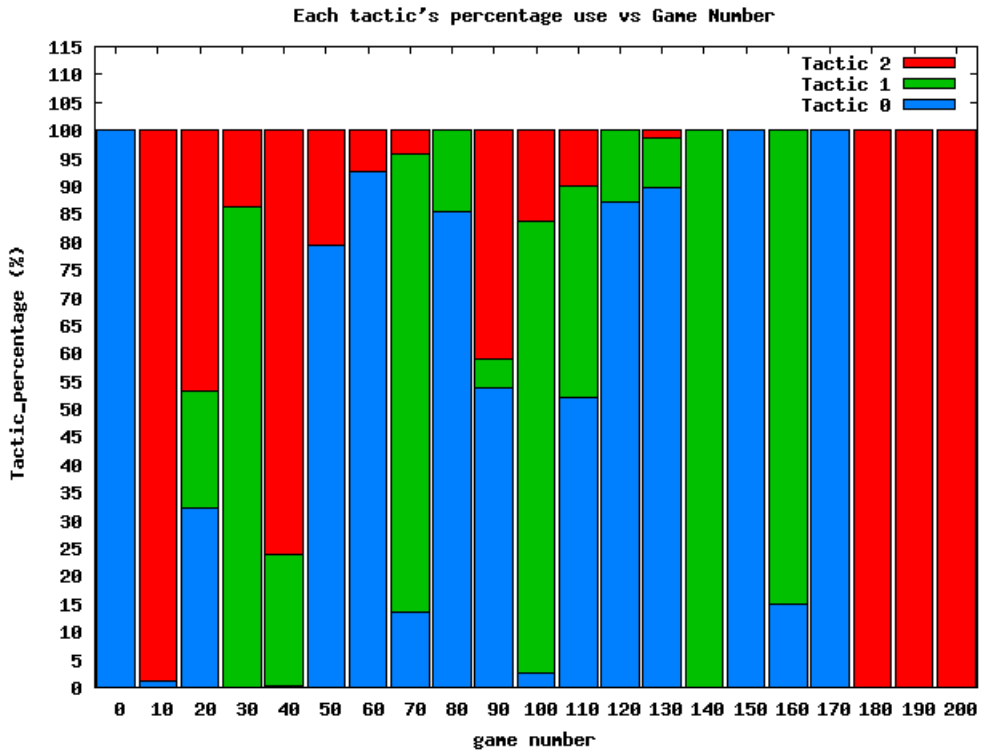

Figure 6.17: SARSA: tactics' contribution to a game on the Tough Duel scenario.

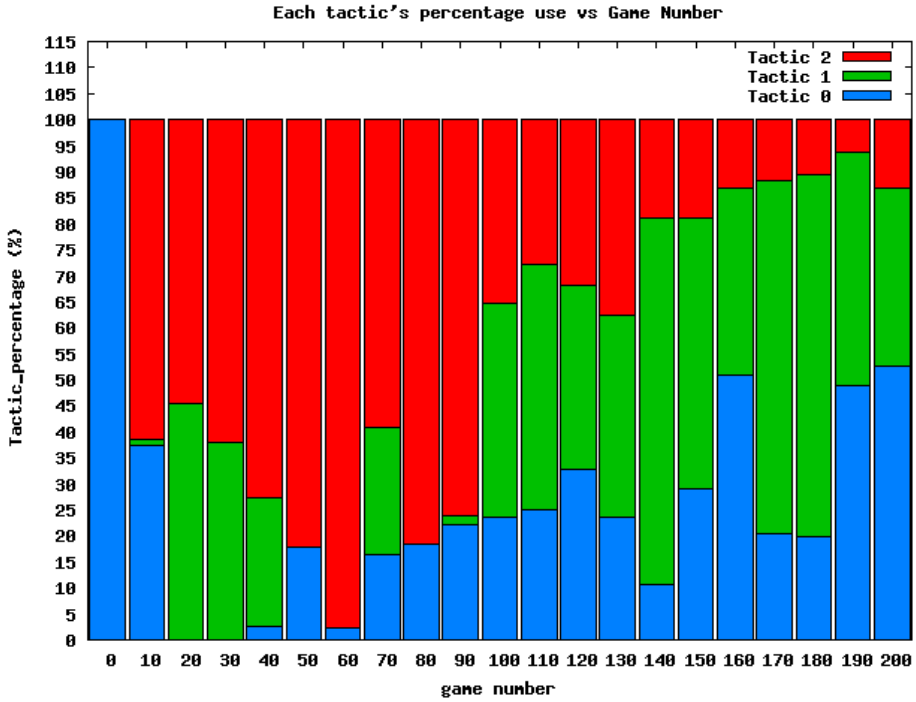

Figure 6.18: LSPI (SARSA's game samples): tactics' contribution to a game, on the Tough Duel scenario.

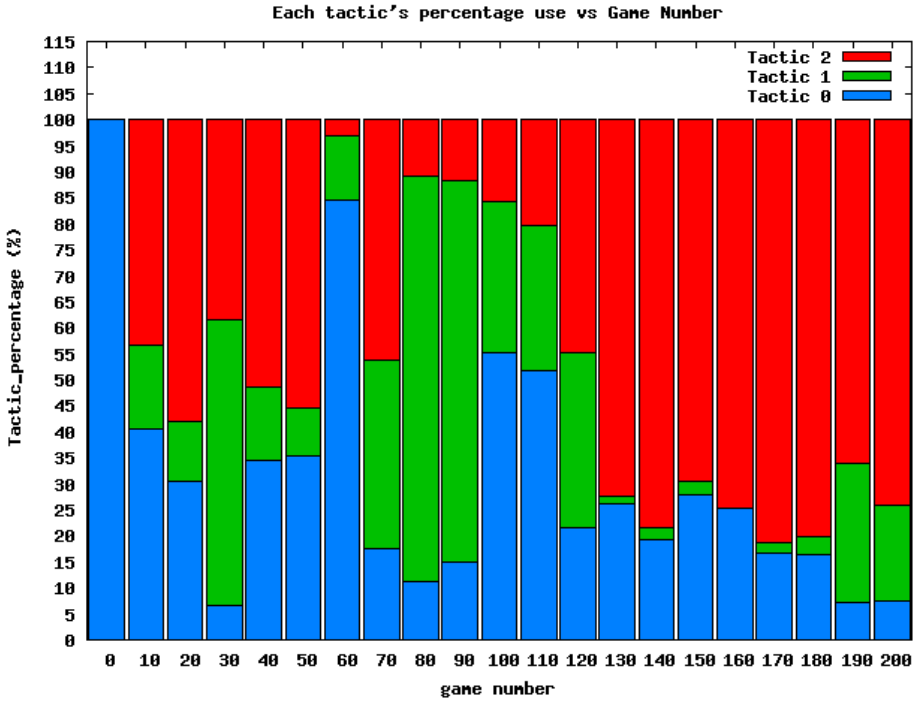

Figure 6.19: LSPI (random game samples): tactics' contribution to a game, on the Tough Duel scenario.

6.5 Discussion

It is easily seen from the results, that in all cases (both SARSA and LSPI algorithms) there were a few test games, where the behaviour of the learning player was improved significantly. That could possibly be a first indication that reinforcement learning is applicable to RTS games too, and even in an abstract-macroscopic way of approaching the problem.

As we see in the Duel scenario's results, the peak performance of SARSA (129.49%), is a little better than LSPI's when using random samples (122.77%). We also see that both SARSA's and LSPI's (with random samples) peak is achieved after 20 learning games (or at the 2nd test game).

LSPI with SARSA's samples, on the other hand, achieves its best score improvement (85.60%) after 150 learning games (or at the 15th test game). Its first positive score improvement is achieved after 20 learning games (2nd test game), whereas SARSA's and LSPI's with random samples first positive score improvement is achieved after 10 learning games (1st test game).

In the Tough Duel scenario's results, we see that SARSA's best score improvement (22.30%), is much better than LSPI's with random samples (1.91%). The peak performance of LSPI with random samples though, is achieved a little sooner (after 40 learning games, i.e. at the 4th test game) than SARSA's (after 50 learning games, i.e. at the 5th test game).

Finally, we see that LSPI with SARSA's samples is better than both SARSA and LSPI with random samples. Its best score improvement is 35.71% achieved after 160 learning games (16th test game) and its first positive score improvement is achieved after 10 learning games (1st test game), whereas SARSA's first positive score improvement is achieved after 30 learning games (3rd test game) and LSPI's with random samples after 40 learning games (4th test game).

Unfortunately, we did not observe convergence to a stable good policy over

an increasing number of learning games. This phenomenon could be due to the fact that a lot of state information (geographic deployment, opponent status) is hidden and therefore the Markovian assumption might be violated.

Chapter 7

Conclusions

7.1 Unrolling results' secrets

Experimental results always show things about the nature of a problem. Sometimes they come to assure a theoritical hypothesis, whereas other times they give us indications about various things we should improve in our theoritical model. Sometimes they help us understand whether a theoritical model or algorithm is applicaple to a specific problem or not. In our experiments the results we got, almost proved what we expected.

From what we saw in the Duel scenario, SARSA achieves the best improvement of all three methods used. LSPI's peak performance, when using random samples, is quite close to SARSA's though. Both SARSA and LSPI using random samples reach their peak a lot sooner than LSPI with SARSA samples in this scenario. In the Tough Duel scenario, we saw that LPSI with SARSA's samples has a much better behaviour (in terms of performance) than the other two methods. All these could possibly mean that LSPI with the apropriate samples could give us generally good results for Glest in all cases.

In both scenarios' results, we also saw that mixed tactics which make use of all three tactics during a game can get us higher scores than the default or any other tactic alone. This is a general rule in RTS games. Finding a good balance among different tactics is almost always the key to victory.

In our project we have proved that reinforcement learning is applicable to RTS games without having to approach the problem in a microscopic way. So, we have fulfilled our first target and that is a reward in RL terms for us to continue in that direction, till the next subgoal.

7.2 Future work

This project was just one small step towards the study of reinforcement learning in RTS games. There is a lot to be done in the future, such as the use of other RL algorithms for learning or the use of other basis functions (such as RBFs or Tile Coding) in both SARSA and LSPI. We can also carry out similar experiments but with various tactics, totally different in reasoning from the ones we tried. For example, tactics which have been formed by choosing rule time intervals in a random way, might be quite interesting to use.

Further study of supervised learning[1] or use of reinforcement learning methods for dynamic adaptation to opponent's behaviour could also be our next challenging field to explore. Another good idea for further research, as online (or multiplayer) gaming is a great feature provided by most RTS games (Glest included), might be reinforcement learning with samples collected from different opponents through the network or online adaptation to network opponents using reinforcement learning methods.

[1] A great feature we have added in system's functionality is the capability given to users to intervene and manually select actions while a reinforcement learning method is used, in order to improve system behaviour or exploration further.

References

[1] Glossary of Terminology in Reinforcement Learning. http://www-all.cs.umass.edu/rlr/terms.html.

[2] Machine Learning. http://en.wikipedia.org/wiki/Machine_learning.

[3] SARSA. http://en.wikipedia.org/wiki/SARSA.

[4] Bhaskara Marthi, Stuart Russell, and David Latham. Writing stratagus-playing agents in concurrent alisp. In *Proceedings of Workshop on Reasoning, Representation and Learning in Computer Games (IJCAI-05)*, pages 67--71, 2005.

[5] Bill Wilson. The Machine Learning Dictionary for COMP9414. http://www.cse.unsw.edu.au/~billw/mldict.html.

[6] Michael Buro. ORTS: A hack-free RTS game environment. In *Proceedings of the 3rd International Conference on Computers and Games*, pages 156--161, 2002.

[7] Michael Buro. Real-time strategy games: A new AI research challenge. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 485--486, 2003.

[8] Michael Buro and Timothy Furtak. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS-04)*, pages 63--70, 2004.

[9] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *Proceedings of the 14th Neural Information Processing Systems (NIPS-14)*, pages 1523--1530, 2001.

[10] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational MDPs . In *Proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1003--1010, 2003.

[11] Carlos Guestrin, Michail Lagoudakis, and Ronald Parr. Coordinated reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML-02)*, pages 227--234, 2002.

[12] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[13] Danny Cheng and Ruck Thawonmas. Case-based plan recognition for real-time strategy games. In *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON-04)*, pages 36--40, 2004.

[14] David Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR-05)*, pages 5--20, 2005.

[15] Eric Kok, Frank Dignum, and Joost Westra. Adaptive reinforcement learning agents in real-time strategy games. Master's thesis, Utrecht University, 2008.

[16] Ethem Alpaydın. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, Cambridge, Massachusetts, 2004.

[17] Glest Team. Documentation wiki for Glest. `http://glest.wikia.com/`.

[18] Glest Team. Glest's official site. `http://www.glest.org`.

[19] Jason Pazis. Phoenix: An object oriented framework for systematic reinforcement learning experiments. 2009.

[20] Leslie Pack Kaelbling, Michael Littman, and Andrew Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237--285, 1996.

[21] Marc Ponsen and Pieter Spronck. Improving adaptive game AI with evolutionary learning. In *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04)*, pages 389--396, 2004.

[22] Marc Ponsen, Héctor Muñoz-Avila, Pieter Spronck, and David W. Aha. Automatically Generating Game Tactics via Evolutionary Learning. *AI Magazine*, 27:75--84, 2006.

[23] Michael Buro. Call for AI Research in RTS Games. In *Proceedings of the Challenges in Game Artificial Intelligence workshop*, pages 139--141, 2004.

[24] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte Carlo Planning in RTS Games. *IEEE Symposium on Computational Intelligence and Games*, pages 117--124, 2005.

[25] Michail G. Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107--1149, 2003.

[26] Jason Pazis and Michail Lagoudakis. Learning continuous-action control policies. In *Proceedings of the 2009 IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 169--176, 2009.

[27] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent AI with dynamic scripting. In *International Journal of Intelligent Games and Simulation*, pages 45--53, 2004.

[28] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

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

[30] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.