DEVELOPING GAME AI FOR THE REAL-TIME STRATEGY GAME STARCRAFT

Research Report

Spring 2011

Aleksandar Micić, Davíð Arnarsson, Vignir Jónsson

Computer Science B.Sc.

Leiðbeinandi: Yngvi Björnsson

Prófdómari: Henning Úlfarsson

T-619-LOKA

Tölvunarfræðideild

ABSTRACT

Research into intelligent agents designed to play Real-Time Strategy (RTS) games has been limited compared to the amount of research that has gone into designing agents for playing board games such as chess. This is largely due to the fact that developing an agent for RTS is more complicated.

There has been some positive development in the last couple of years though with the introduction of the Brood War API, which allows researchers to hook their AI into the popular RTS game StarCraft. StarCraft provides an interesting platform for AI RTS research due to its large variety of units and balance between playable factions. We designed an AI agent capable of making informed decisions in StarCraft. The agent uses Reinforcement Learning to improve its performance over time. Empirical evaluation of the agent shows this type of an approach to be viable in RTS games.

Útdráttur

Lítið hefur verið um rannsóknir á hönnun gervigreindra forrita sem að spila Rauntíma herkænsku (RTH) leiki í samanburði við það magn rannsókna sem hafa farið fram á hönnun forrita sem að spila borðspil svo sem skák. Þetta er að stórum hluta vegna þess hve flókið það er að hanna gervigreind forrit fyrir þessa rauntíma leiki.

Á undanförnum árum hefur þó orðið jákvð þróun, þökk sé hinum svo kallaða Brood War API, sem að leyfir rannsóknarmönnum að tengja forritin sín við hinn sívinsæla RTH leik StarCraft. StarCraft er góður grunnur fyrir RTH rannsóknir á sviði gervigreindar vegna mikils úrvals peða(e. Units) og jafnvægis milli liða.
Við þróuðum gervigreint forrit sem að getur tekið ákvarðanir byggðar á umhverfi sínu í rauntímaherkænskuleikjum. Forritið notar Reinforcement Learning til að betrumbæta hegðun sína eftir því sem að fleiri leikir eru spilaðir. Tilraunir með forritið sýna að sú aðferð er raunhæfur kostur í þróun gervigreindar fyrir RTH leiki.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

1.1 Real-Time Strategy Games

Real-Time Strategy (RTS) is a genre of computer games where the player is in control of certain, usually military, assets (solders, vehicles, structures) which the player can manipulate in order to achieve victory over the enemies.

RTS has been one of the most popular video game genres since the early 90s; entering the mainstream with the release of the groundbreaking Dune II (released in 1992) which defined the genre by introducing many of the basic components which almost every modern RTS game includes.

In a typical RTS game, it is possible to create additional units and structures during the course of a game. Creating additional assets requires the player to spend resources, usually in the form of some monetary currency or a precious material. These resources can be acquired during the course of a game as well, most commonly by creating units which can harvest precious materials. In most RTS games there are other ways of accumulating resources, like controlling special points on the game map or by controlling structures devoted to this (e.g. an oil rig).

1.2 The Problem

Over the years, game research has primarily been focused on turn-based board games like chess and backgammon. Most of these turn-based games are also deterministic. In chess, determinism guarantees that if a player *A* decides to move his rook from square *h8* to *h2*, assuming the move is legal, he can be completely sure that the rook will end up in *h2*. RTS games, on the other hand, are not turn-based but rather, as the name implies, real-time. What this means is that there are no turns and all players act simultaneously, and as such if a player issues a command there are no guarantees that the command will be executed in a predictable manner. There are also no restrictions in number of commands a player can issue, so best human RTS player learn to issue commands as fast as possible. This is one edge that computer RTS players have over their human counterparts, since computers are much faster than humans.

Another thing adding to the complexity of RTS games is that the state space is enormous. Chess, as an example, is estimated to have around 10^{43} to 10^{47} different legal positions on an 8x8 board (Tromp, 2011). RTS games are much more complex. As

Michael Buro, associate professor at the Department of Computing Science University of Alberta, so adequately puts it:

„*To get a feeling for the vast complexity of RTS games, imagine to play chess on a 512x512 board with hundreds of slow simultaneously moving pieces, player views restricted to small areas around their own pieces, and the ability to gather resources and create new material*". (Buro M. , 2004)

It is clear that writing an AI for RTS games is no easy task, due to uncertainty and complexity these games can reach. Our goal is to see if using reinforcement learning is an efficient way to learn under the arduous circumstances RTS games provide for the AI, and doing so, see if StarCraft is a viable test bed for future RTS AI research.

1.3 AI IN COMMERCIAL RTS GAMES

Until recently, only the developers of RTS games were concerned with AI for RTS games. Since most of the games are developed with entertainment in mind, the AI for these games is not designed to be the best possible, but just to present a challenge for the human player. It wasn't uncommon for the developers to allow their intelligent agents to cheat and ignore the restrictions imposed to human players. For example, one very common feature of RTS games is called the fog of war.

This so called fog usually covers the parts of the map the player has not yet explored, see darkened are to the right on Figure 1-1, hiding them from sight. Human players, therefore only have information on those areas where they have their units located, and in order for them to find and defeat their enemies they would, first either have to send scouts to find them or march their forces around hoping to stumble across the foe. To simplify things for the game developer, the AI usually has this information available to it at all times; that is to say, the AI can see through the fog of war and is not impeded by it, while the player has to deal with it. Sadly, a typical RTS AI does little with this information other than to use it to point its forces in the right direction.

Most standard in game AIs do not get their edge from analyzing their opponent and exploiting their weaknesses, they simply have more information than a human player does and can multitask things such as building order and unit training better than your average human mind. We are not aware of any AI in commercial RTS games which adapts to the way the human player is playing the game. Because of this static nature these agents become homogenous, predictable and therefore vulnerable to certain

strategies used against them. Moreover, they very quickly become easy to beat for a skillful player and playing against them loses appeal; the game becomes boring.

FIGURE 1-1: FOG OF WAR, THE AREAS OF THE MAP WHICH ARE DARK HAVE NOT YET BEEN EXPLORED.

1.4 THE PROJECT

It has been more than a decade since Deep Blue, a computer chess player, defeated the undisputed chess world champion, Garry Kasparov, in a six-game match[1]. Before this milestone in AI there was a lot of skepticism and few believed that computers would ever be able to beat human world champions in chess. Nowadays, best computer player are so superior that there is no point in matching humans versus computers when it comes to chess. This is also true for most turn-based games, with a clear exception of the game Go[2].

Computer RTS players are still in their infancy, and as such are still dominated by best human players. Because of their real-time nature and the fact that human players are still better, researching AI for RTS games is exciting as we will need to find new approaches and apply new techniques the beat human players.

This will most likely further the field of AI, but it has at least one more important application; to make better RTS games. If the agent could learn new behaviors while

[1] http://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov
[2] http://en.wikipedia.org/wiki/Computer_Go

playing against a human player it would make for a much tougher opponent, a greater challenge for the human player, and therefore more interesting games. Many of the best RTS players do not even bother playing against a computer opponent, but if it were able to learn then things could change dramatically.

In this project we are focusing on one particular aspect of RTS games, and that is controlling the units or micro-managing them, in combat specifically. Micro-managing combat units in an RTS games mostly involves knowing when to fight, whom to target and when to retreat. A well designed AI ought to be able to out-maneuver your typical human player, as it is much better at multitasking. The AI could be managing its base while fighting on several fronts, while the human player will most likely focus on one aspect or battle at a time, sifting through his units, telling them to attack the units he deems to be the biggest threat, or just relying on the very basic in game combat AI, which usually much comes down to shooting at whatever is closest to you. An artificial agent however, with information regarding the health and damage output of every unit on the battlefield ought to be able to come up with a better tactic or series of commands.

2. BACKGROUND

2. 1 STARCRAFT

A Real-Time Strategy computer game called StarCraft was released March 31st 1998, by Blizzard Entertainment and is the game we used for this project. StarCraft garnered much critical acclaim upon release and has become a benchmark for a good RTS game in the video game industry. StarCraft is by many considered the next big step in RTS evolution, introducing the notion of having several different races, all of which have completely unique units.

2.1.1 RACES

The makers of StarCraft had done something similar before with one of their earlier RTS games, WarCraft, but there every unit had an exact counterpart on the other race. Each race comes from a different world and has different characteristics and weaponry.

FIGURE 2-1: PROTOSS ZEALOT

FIGURE 2-2: TERRAN MARINE

FIGURE 2-3: ZERG ZERGLING

There are three playable races:

- The Terran
    - Humanity, with standard human military equipment such as tanks, jets and infantry.
    - Units are fairly cheap to produce and are of average strength.

- The Protoss
    - Elf-like aliens, with futuristic units with plasma weapons and shields, spaceships and magical warriors.
    - Units very expensive to produce but are very strong individually.

- The Zerg
    - Beastly aliens, seemingly feral creatures, resembling James Cameron's aliens. All claws, teeth and barbs.
    - The units are very cheap to produce and the Zerg rely on quantity over quality.

So each race is quite unique, with its own units and buildings.

2.1.2 Unit Types

There are basically three different kinds of units each team has: worker, combat and support units.

- Workers gather resources in the form of crystals and gas, for which they can then build buildings from where you can train more units and purchase upgrades.

- Combat units come in all shapes and sizes, from infantry to large spaceships. Most combat units have a single attack which can target flying and/or ground units, and is either a melee attack or a ranged attack.

- Support units are units such as medics and drop ships, which support the combat units without actually harming the enemy, by for example ferrying them over obstacle or restoring their health.

2.1.3 Success

What made StarCraft great is how well its developer managed to balance the three factions for competitive play, by having no one race noticeably better than any of the others under any circumstance. This was not an approach taken by other RTS games before StarCraft, where one faction would perhaps rule the sky while another ruled the seas, like for example in Command & Conquer[3]. This caused the game to become massively popular among online PC gamers, becoming the most played RTS game in history and the unofficial national sport of South Korea, with three television stations dedicated entirely to coverage on what's going on in the world of StarCraft.

2.2 Learning in Artificial Intelligence

An intelligent agent is an autonomous entity which observes and acts upon an environment and directs its activity towards achieving goals. An agent perceives its environment through a set of sensors. Based on the input from these sensors the agent decides which action to take next.

2.2.1 Machine Learning

The way an agent chooses its actions can be predefined (explicitly stated in an algorithm) or the agent can learn to choose the best available action in the given state of the environment. A branch of Artificial Intelligence (AI) called Machine Learning is specifically concerned with designing and developing agents that can learn.

Machine learning algorithms can be categorized as supervised or unsupervised. The former requires expert knowledge, whereas in the latter one the agent learns on its own by acting in the environment. We use one such method called reinforcement learning.

[3] http://en.wikipedia.org/wiki/Command_%26_Conquer

2.2.2 Reinforcement Learning

2.2.2.1 Introduction

Reinforcement learning is learning what to do, that is, map situations to actions (Sutton & Barto, 1998). So, while the agent is acting upon the environment it gets rewarded for its actions with a numerical reward which it tries to maximize. The agent must be rewarded for doing well, and punished for doing poorly. For example, the agent may be given a reward of +1 for winning a game, 0 for drawing and -1 for losing a game.

The agent should have no control over the rewards; if it does then it might find ways to rewards itself without achieving its goal. The environment is responsible for generating rewards.

Ideally, the agent should be rewarded only after achieving its ultimate goal (e.g. winning a game) or punished to failing to achieve it (e.g. losing a game) and not for achieving sub-goals. This is because those are the things we want the agent to achieve. If we reward it for sub-goals we risk introducing a bias to the way the agent acts, and therefore, the agent might find ways to achieve those sub-goals while failing to achieve its ultimate goal.

In reinforcement learning the agent learns by trial and error. The agent must try different actions before it can learn which one is the best and it must choose the best actions in order to accomplish its goal. That is, the agent must explore and exploit the environment.

Exploration and exploitation are at odds with each other, and therefore they need to be balanced. One popular way of doing this is called ε-greedy policy. A policy, usually represented with π, describes the probability of some action a being taken given a state s. A greedy policy is a policy in which the action with estimated highest value is always chosen, that is, $\pi(s, a) = 1.0$ if a has the highest estimated value of all actions.

To introduce exploration, ε-greedy policy introduces randomization. The ε number represents the probability of the agent choosing a random action. So, if $\varepsilon = 0.1$ the agent will choose a greedy action most of the time, and one in every ten actions will be random.

2.2.2.2 The Markov Property and Markov Decision Process

If a state representation successfully retains all important information for future decisions is said to be Markov, or to have the Markov property. So, if the state contains all the information that is necessary for making the best possible decision, that is, the state did not forget any important information that happened before, that means that this state has the Markov property. A configuration of pieces in a checkers game would be an example of a Markov state.

This property is very important in reinforcement learning because all of the developed theory relies on it.

A formulation of a reinforcement learning task which state representation satisfies the Markov property is called a *Markov decision process*, or MDP.

Other than state representation MDP also consists of a collection of *transition probabilities* and *expected rewards*. Transition probability is a value which states, given a state s and an action a, the probability that the agent will end up in the state s'.

$$\mathcal{P}_{ss'}^{a} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

Similarly, expected reward is a numerical reward the agent can expect, given a state s, an action a and the next state s'.

$$\mathcal{R}_{ss'}^{a} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

This is a common way to model a reinforcement learning problem and many reinforcement learning algorithms require it.

2.2.2.3 Learning Algorithms

When using reinforcement learning we learn either the value of states or state-action pairs. A state-action pair is, as the name suggest, a 2-tuple of some state s and some action a. If the agent had a choice of four different action from some state s, there would be four state-action pairs for that state. The value is the future accumulated reward (possibly discounted) for being in a state ($V(s)$), or for taking an action in a state ($Q(s, a)$).

If the MDP is fully specified, that is all the transition probabilities and all the rewards are known, we can solve the reinforcement learning problem using systematic methods, such as *Dynamic Programming* (DP).

Most of the time, however, these values are not known. To solve the reinforcement learning problem then, we can use sampling methods such as, *Monte Carlo* (MC) and *Temporal-difference* (TD) methods. Here we observe the agent acting in the environment, and average the returns to get the values we need.

For our project we chose to use TD, in particular, the Sarsa algorithm. The update formula for Sarsa is as follows:

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

3. Agent Architecture

One common way to implement an intelligent agent capable of playing a full RTS game is to split the agent into managers. Each manager would be given a task of managing a major aspect of the game, for example creating buildings and units, gather resources, scouting, fighting, etc.

The order in which important structures are built during a game is a big part of the overall strategy and very important for the final outcome of the game. One manager solely devoted for this purpose would not be out of place in an RTS AI, because RTS games are very dynamic and new information can influence the build order significantly. Another example would be a manager which task is to gather information about the game. If the agent is not allowed access to the perfect information then it must do what expert human player do and send out scouts to gather information (McCoy & Mateas, 2008).

For the purpose of our project, which focuses on micro-managing units during combat, we have identified two of such managers. A manager which controls the units, which we dubbed Unit Manager, and a manager which will hold and generate information about the game to help our agent make informed decisions, which we called Information Manager.

3.1 Information Manager

One thing in particular that we did not have access to but we thought to be important was the path that units take while moving. This is what is generally referred to as pathfinding and is an integral part of every RTS game. We also needed some way to abstract the information about important areas on the battlefield, and to accomplish that we used the concept of heatmaps.

FIGURE 3-1: PATHFINDING VISUALIZED, THE BLUE LINE SHOWS THE SHORTEST AVAILABLE PATH BETWEEN ITS LEFT END AND ITS RIGHT END.

3.1.1 Pathfinding

Pathfinding means finding the shortest path, that is, the path of least cost between two points. Pathfinding is essential to any RTS game, including StarCraft. However, we do not have access to the paths the in-game pathfinding generates it finds, and therefore we decided to implement a form of pathfinding ourselves. One of the most popular pathfinding algorithms is A* (read as A-star), and it is the route we chose to take with our pathfinding implementation.

A path found using our pathfinding implementation will not necessarily be the same as the one found by the in-game pathfinding, and therefore the unit might not follow our

path. None the less, this information can still be used to many purposes, including to estimate the time it will take the unit to get to a certain location. Such information is valuable when making decisions in combat scenarios. An example path generated by our pathfinding algorithm is shown in Figure 3-1.

3.1.2 HEATMAP

We needed a good way to abstract information about the military strength of large armies by map regions. We use heatmaps for that. A heatmap is a visual representation of data. A color is associated with some value range, and that color is displayed on the screen instead of the number. It makes for a very good visualization tool.

FIGURE 3-2: HEATMAP VISUALIZED, COLORED REGIONS ON THE MAP SHOW THE AMOUNT OF FIREPOWER THAT CAN BE DELIVERED TO THAT SPOT. COOLER COLORS MEAN LESS FIREPOWER AND HOTTER COLORS MEAN MORE FIREPOWER.

The way we used this concept is not, necessarily, for visualization, but rather to give values to squares on a game map. Those values can be anything; they can represent the firepower that can be delivered to any point on the map (calculated as damage per frame), which is depicted in Figure 3-2 or they can represent the proximity of a unit to

other friendly units. This concept is also used for our pathfinding implementation, except that this time values show if the square if occupied or not.

3.2 Unit Manager

The Unit Manager is the other module in our agent architecture and its purpose is to handle units' individual low-level decisions and actions. Part of this is micro-managing units while they are in combat.

The unit manager, which architecture is depicted in Figure 3-3, is the core of our project and is constructed with the idea that each individual unit will have an intelligent agent controlling its low-level behavior.

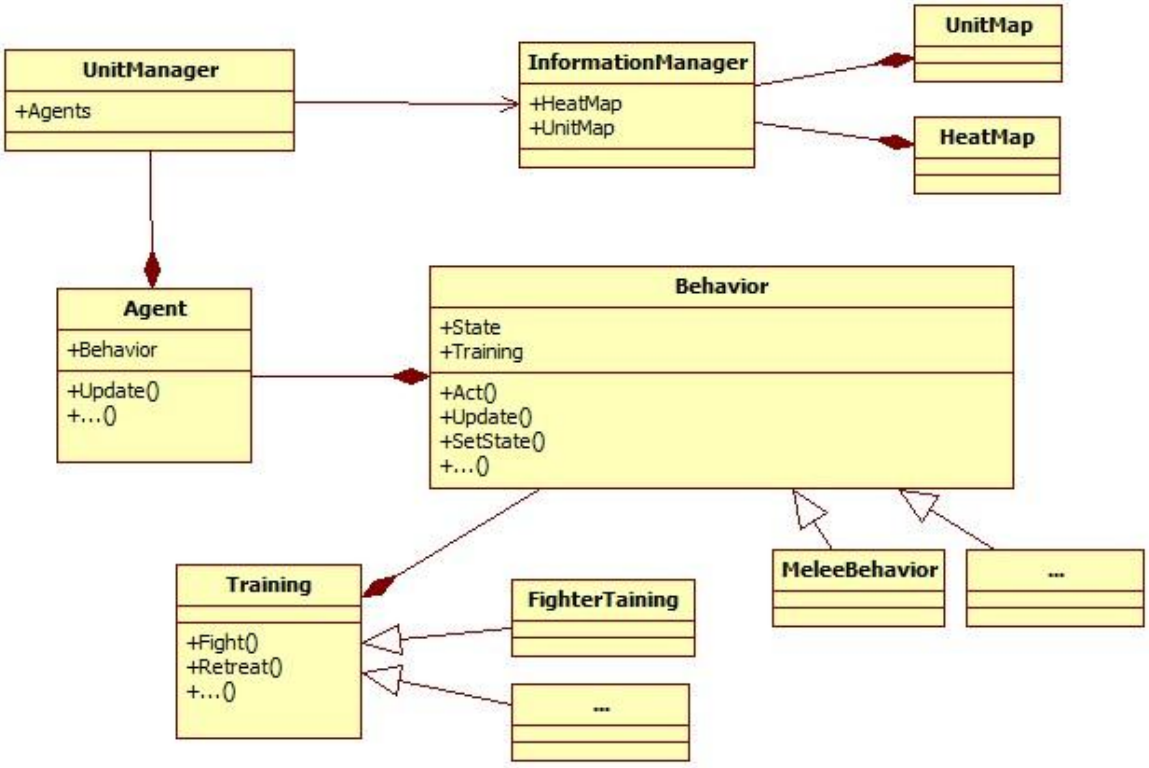

FIGURE 3-3 UNIT MANAGER ARCHITECTURE, UML DIAGRAM SHOWING THE MOST IMPORTANT PARTS OF THE UNIT MANAGER ARCHITECTURE.

The agents are modeled using *Final-State Machines* (FSMs) with transitions between every state. This FSM is contained in Behavior and the FSM we used for our experiments has four states:

- Fight
- Retreat

- Regroup
- Idle

New Behaviors with different FSMs can be created to suit the needs of individual units as they can be very different. Here the units are acting individually and do not have a collective behavior because that functionally should be delegated to another module as the unit manager is only concerned with low-level behavior of units.

Once a unit is in a particular state it calls upon its Training to execute what it's "trained" to do when it happens to be in that state. So, Training contains scripted behavior for each state in the FSM, and it too can be extended to be tailored to individual units or a group of units.

Scripted behavior stays the same all the time, and the scripts we used were rather simple. Fight involves attacking the most injured enemy within a certain radius from the unit. When in the retreat state the unit would move in the opposite direction to its enemies. When in Regroup the unit would more towards its closest friendly unit and when in idle the unit first sits and does nothing.

To manage the details of state transitions, we decided that we want use reinforcement learning to learn when to transition between states. Although we kept the behavior of individual states scripted, although reinforcement learning could be applied there as well, although that is outside the scope of this work.

4. Learning Method

To utilize the learning we need to look at the environment to see in which state the agent is in now, calculate the reward and apply the update formula. This needs to be done on each update step.

In order to test this concept of applying reinforcement learning to micro-management in RTS games we decided to create simple scenarios and test those first. We started with the simplest scenario we could think of and increased the complexity afterwards.

These reinforcement learning tasks are modeled as partially specified Markov Decision Processes (MDPs), where only the state and the rewards are known.

4.1 Scenario #1: Dragoon vs. Zealot

Is this scenario we pitted a dragoon unit, which was controlled by our agent, against a zealot unit, which was controlled by the AI found in the game.

Dragoons and zealots are the two core units of the Protoss race. The dragoon is their basic ranged combat unit, and the zealot is their basic melee unit. Each unit's health is measured in points, appropriately named health points, and what both units will be attempting to do is bring its enemies health to zero by attacking it.

This scenario is an exercise in what is called kiting. Since the dragoon is a ranged unit and the zealot is a melee unit, the range of effectiveness of the dragoon is much greater than that of the zealot. If the dragoon used his superior range to his advantage and kept moving while still attacking the zealot and not taking damage himself, then the zealot is said to be kited by the dragoon.

Should the dragoon remain stationary throughout the entire fight, the zealot will win. Should the dragoon stay mobile and keep his distance from the zealot, he has a chance of winning. Attacking, however, is not instant and a unit must remain stationary for some time while the attack is being performed. In this case, the dragoon's attack, shooting from the cannon on its back, takes much longer to perform than the zealot's attack, swinging his plasma blades. The dragoon therefore needs to figure out that he needs to stop just long enough to get one shot off and then keep running, before the zealot catches up and hits him.

Each attack also has a so called cooldown time, the least amount of time that must pass between attacks. An attack is said to be off cooldown if it is ready to be used and on cooldown otherwise.

We came up with a very simple scenario at first which only told the agent whether his weapon was on cooldown or not, whether the enemy unit was within range and whether he was within the enemies range. After seeing that it worked and that it was in fact learning, we expanded on it as follows:

- **Distance** d between the units which measured in pixels is represented with five discrete values:
  - $d > 140$
  - $100 \leq d \leq 140$
  - $80 \leq d < 100$
  - $35 \leq d < 80$
  - $35 > d$

- The **health points** h for each units was also a part of the state representation, and it was represented with six discrete values:
  - $h > 60$
  - $40 < h \leq 60$
  - $30 < h \leq 40$
  - $20 < h \leq 30$
  - $10 < h \leq 20$
  - $h \leq 10$

- The dragoon's weapon **cooldown** is described with two discrete values:
  - **On**: The dragoon is unable to attack.
  - **Off**: The dragoon is able to attack.

- An indicator on whether the Zealot is inside the dragoon's attacking range:
  - **No:** The zealot is out of range.
  - **Yes**: The zealot is within range

- And an indicator whether the dragoon is inside the zealot's attacking range:
  - **No:** The dragoon is outside the range.
  - **Yes:** The dragoon is inside the range.

- The dragoon also had two actions available to him:
  - Attack
  - Retreat

The agent was rewarded based on its and the enemy units health difference on each update step. So if the agent's health had decreased since the last frame he got a negative reward equal to the amount of health lost, likewise if the enemy unit had less health the agent would get a positive reward equal to the amount of health it had lost, if both sustained an equal amount of damage he would receive a reward of 0.

Keep in mind that these actions are transitions in the agents FSM and that once the agent is in a particular state it follows a scripted mechanism. The attacking mechanism was simple, just fire at the zealot whenever possible. The retreat action uses the positions of the dragoon and the zealot to create a vector. That vector is used as a direction for the dragoon to run into to evade the zealot. This combination of states and actions yielded a total of 12 state-action pairs.

Adding the distance allowed the agent to figure out whether there were times when he might get more shots off before he needed to start retreating again. The actions remained the same, either attack or retreat. This adds up to a total of 2,880 state-action pairs. In the MDP the method of rewards stayed the same as it was in the first scenario. This formulating strikes a delicate balance between trying to minimize the number of state-action pairs to learn, and retaining the information relevant for informed decisions.

4.2 Scenario #2: Four Dragoons vs. Four Dragoons

Unlike the previous scenario, this scenario is a multi-agent one, with each team controlling four dragoons units. All four dragoons making decisions based on a shared representation of the environment. It should be noted that all agents used the same state-action values, so each agent benefitted from the experience of other units as well as its own.

We felt that some modification of the state representation was needed for multiple agents. A way was needed for the agents to know how many enemy units are nearby, as well as friendly units. For this we used a pair of heatmaps. We also kept track of how many enemy units are targeting each unit. We still kept track of the agent's health and his current target's health, as well as whether his weapon is on cooldown. We also added a third action, regroup, which has the agent move to its nearest friendly unit. The final state representation was as following:

- The **health points** h for each units was also a part of the state representation, and it was represented with six discrete values:
    - $h > 80$
    - $55 < h \leq 80$
    - $35 < h \leq 55$
    - $22 < h \leq 35$
    - $10 < h \leq 22$
    - $h \leq 10$

- The unit's weapon **cooldown** is described with two discrete values:
    - **On**: The unit is unable to attack.
    - **Off**: The unit is able to attack.

- The number of enemy units targeting the agent's unit was represented with five discrete values:
    - None
    - One
    - Two
    - Three
    - Four or more

- The unit's enemy heat value, collected using a heatmap, tells the agent how many enemy units have it within range of their attack.
    - None
    - One
    - Two
    - Three
    - Four or more

- The unit's friendly heat value, collected using a heatmap, tells the agent how many friendly units are near it.
    - None
    - One
    - Two
    - Three
    - Four or more

This representation resulted in a total 13,824 state-action pairs in the MDP. This scenario is considerably more complicated than the previous ones, and as such it takes a lot longer to learn the best actions.

5. Empirical Evaluation

During our testing of both scenarios the agents were following an ε-greedy policy with $\varepsilon = 0.05$, and the values of state-action pairs were initialized to 0. In the first, Dragoon vs. Zealot, scenario we played 50 games on the same policy and after each game we recoded the rewards and the outcome of the game. After the 50^{th} game the values of the state-actions pairs to were reset to 0. This process was repeated 82 times in order to get some conclusive results.

Figure 5-1 shows a plot with the average rewards the agents managed to accumulate for each game over those 82 iterations. As can be seen, the linear trendline which is drawn in black is inclining, which shows that the agent managed to accumulate higher reward in the later games as it got more experienced. Win ratio plot of the same test is shown on Figure 5-2, and as it shows the agent also managed to improve its win ratio with more experience. The agent's performance goes from winning 40% of the games to close to 60%.

We have also played hundreds of games without resetting the state-action values which has shown that the agent win percentage tapers off at a 72%. For comparison, in this scenario, if the dragoon was controlled by the in-game AI it would never manage to achieve a single victory, and in that sense our agent managed to outperform the default AI found in StarCraft by far.

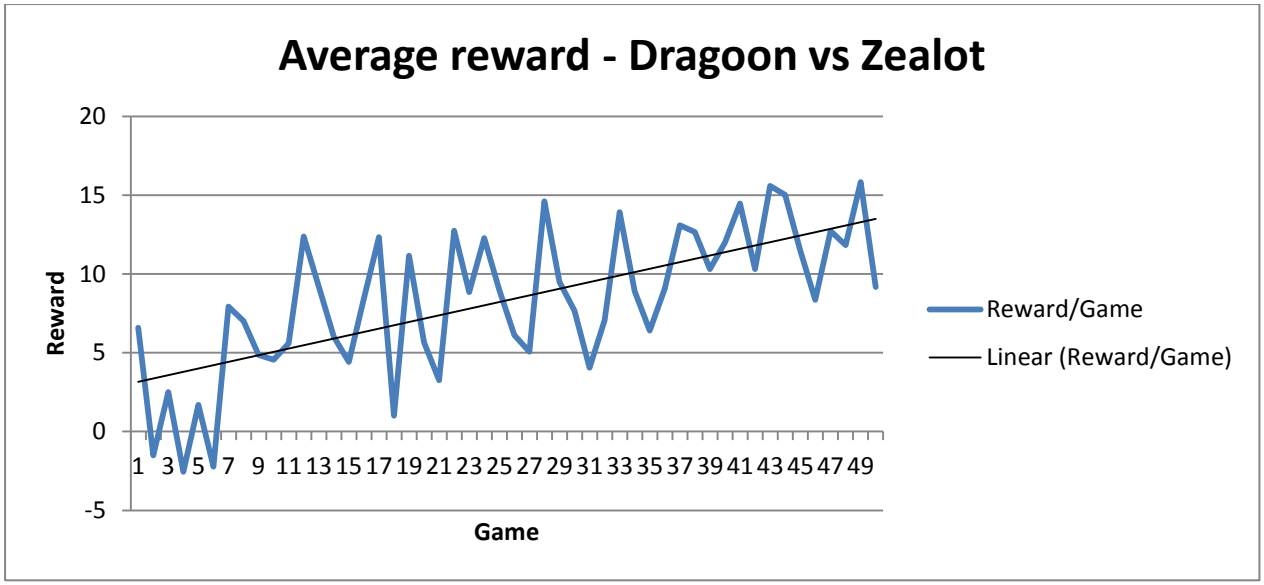

FIGURE 5-1: AVERAGE REWARD GIVEN TO THE AGENT FROM PLAYING 82 ITERATIONS OF 50 GAMES IN THE DRAGOON VS. ZEALOT SCENARIO.

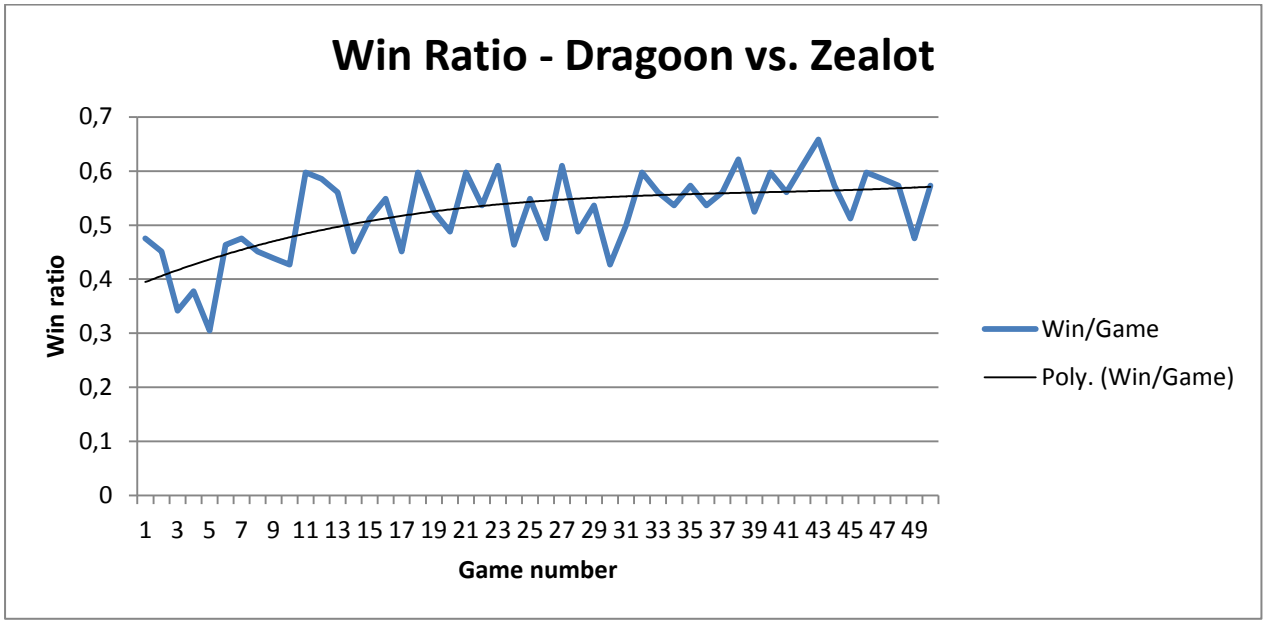

FIGURE 5-2: AVERAGE WIN RATIO FROM PLAYING 82 ITERATIONS OF 50 GAMES IN THE DRAGOON VS. ZEALOT SCENARIO.

Testing of the second scenario was conducted in a similar manner as the first one. This time, however, because this scenario is considerably more complex than the previous one we needed to play 100 games to show any trends. The number of iterations was also smaller, because of time constraints imposed by the large number of games in each iteration; this time 15.

The results of this test are shown in a plot in Figure 5-3. The polynomial trendline, drawn in black, shows that win ratio of is dropping for a certain number of games

before it starts to pick up again. A similar trend can be seen on Figure 5-2, which leads us to believe that we need to play many more games before seeing conclusive results, in this test domain. That is because the underlying MDP is much larger.

Similar to the first scenario, we played hundreds of games without resetting the state-action values our agent managed to achieve a 32% winning percentage. Learning the true values of state-action pairs would take many games because of the large state-space and demonstrating this is not feasible with the computational resources which were available to us. However, improvements in the agent's behavior were noticeable.

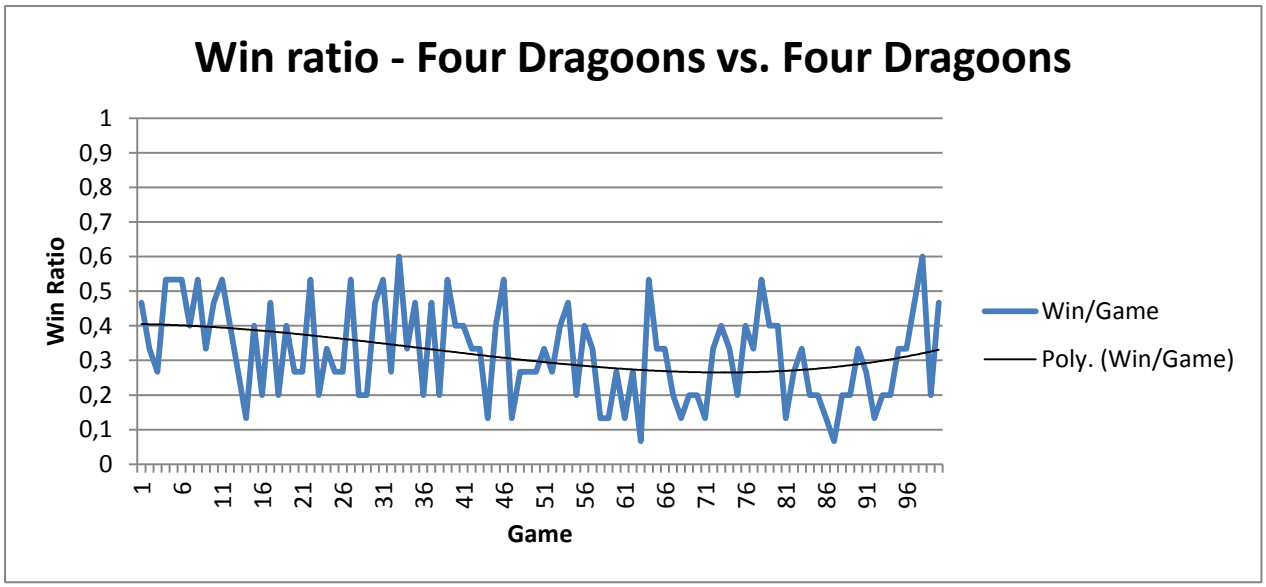

FIGURE 5-3: AVERAGE WIN RATIO FROM PLAYING 15 ITERATIONS OF 100 GAMES IN FOUR DRAGOONS VS. FOUR DRAGOONS SCENARIO.

6. RELATED WORK

Since most of the RTS AI developed is proprietary software and therefore off limits to researchers, a good platform for developing and testing methods and algorithms for use in RTS game was missing. In the last few years, such tools have been made available and with the development of *Open Real-Time Strategy*, or ORTS (Buro M.), researchers finally had a tool in which they have full control of every aspect of an RTS game.

BWAPI (Brood War API), which stands for Brood War Application Programming Interface, is another example of such a tool and is what we used for this project. BWAPI

enables researchers to hook their own custom AI onto StarCraft, and effectively use StarCraft as a test bed for RTS AI research.

Thanks to BWAPI a competition was started in 2010 by Artificial Intelligence for Interactive Digital Entertainment (AIIDE), named AIIDE StarCraft: Brood War AI Competition, as an initiative to get people work on AI for RTS gams. The response has been something and several research groups compete in the last year.

The competition is split into four categories, each more complicated than the last; the first two focusing on combat, the second two a full game with base building. This project was inspired by the first two categories which mostly involves the micro-management of combat units.

The winner of the first two tournaments of the last year's completion was with FreScBot[4]. It is an intelligent agent modeled using multi-agent finite-state machines similar to our agent, but it did not use any learning techniques.

Another competitor of the last year's competition, called Overmind (Huang, 2011), was created by a group of PhD students at the Computer Science department of the University of California, Berkeley. It was engineered to compete in the competition's most advanced tournament. It won decisive victories over all its opponents, winning the tournament itself in the end.

Overmind is a complex, multi-layered agent, with different AI techniques approaching different aspects of the RTS AI problem. The combat element of Overmind was focused on a tactic called Mass-Mutalisk, in which the main strategy is to build many of the Zerg flying units called Mutalisk. When grouped up, Mutalisks can do considerable damage and have good maneuverability, but are notoriously difficult to micro-manage by humans. Overmind exceled at this task, dealing high, concentrated damage and keeping the Mutalisks alive with great effectiveness.

To accomplish this, Overmind utilized potential fields, a concept similar to our heatmap, and action prediction mechanisms. The potential fields are used to describe a number of different elements of a combat scenario. Using the action prediction mechanism, Overmind can predict the outcome of an action taken, making sure that the best

[4] http://bgweber.com/starcraft-ai-competition-results

predicted action is taken at any given moment. Overmind uses this information to make an informed decision about what the next action should be.

There have also been a few theses written on the subject of applying learning to RTS game intelligent agents, but most of them concerned with high-level decision, such as build order (Walther, 2006) (Rørmark, 2009).

7. Conclusion and Future Work

Based on the experiments and the results, we believe that reinforcement learning can be applied to micro-management in RTS games. The results show that applying RL to simple scenarios is clearly feasible, although it gets harder the more the complexity increases.

We only experimented with learning when to transition between the states in a unit's FMS, but it doesn't have to end there. Reinforcement learning techniques can be applied for many more things the intelligent agent has to decide. The first thing that comes to mind is the scripted behavior we currently use when the agent is in a particular state.

For example, if the agent is in the fight state there are many things it needs to consider when choosing a target. Is shooting an enemy with the least health which in range, or is it better to move and attack the enemy with the least health overall? Reinforcement learning can learn that. Same with the retreating state, the agent can learn to change its retreating behavior based on, for example, the proximity to a wall. This was, however, outside the scope of this project.

However, this technique is not yet ready for commercial application. The learning would take a very, very long time and it would need to be done offline, so any noticeable learning during a game would be out of question. Still, the results are promising and with further research who knows what can be achieved.

REFERENCES

Brood War API. (n.d.). Retrieved from http://code.google.com/p/bwapi/

Buro, M. (2004). *Call for AI Research in RTS Games.* Retrieved 5 11, 2011, from skatgame.net: http://skatgame.net/mburo/ps/RTS-AAAI04.pdf

Buro, M. (n.d.). *ORTS*. Retrieved from http://skatgame.net/mburo/orts/orts.html

Huang, H. (2011, January 19). *Skynet meets the Swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition*. Retrieved from ars techinca: http://arstechnica.com/gaming/news/2011/01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-competition.ars/3

Intelligent Agent. (n.d.). Retrieved from Wikipedia.org: http://en.wikipedia.org/wiki/Intelligent_agent

Machine Learning. (n.d.). Retrieved from Wikipedia.org, Machine Learning: http://en.wikipedia.org/wiki/Machine_learning

McCoy, J., & Mateas, M. (2008). *An Integrated Agent for Playing Real-Time Strategy Games.* AAAI.

Rørmark, R. (2009). *Thanatos - a learning RTS Game AI.* Master Thesis, University of Oslo.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning - An Introduction.* MIT Press.

Tromp, J. (2011, May 9). *Chess.* Retrieved May 12, 2011, from John's Chess Playground: http://homepages.cwi.nl/~tromp/chess/chess.html

Walther, A. (2006). *AI for real-time strategy games.* Master Thesis, IT-University of Copenhagen.

APPENDIX A: TESTING ENVIRONMENT

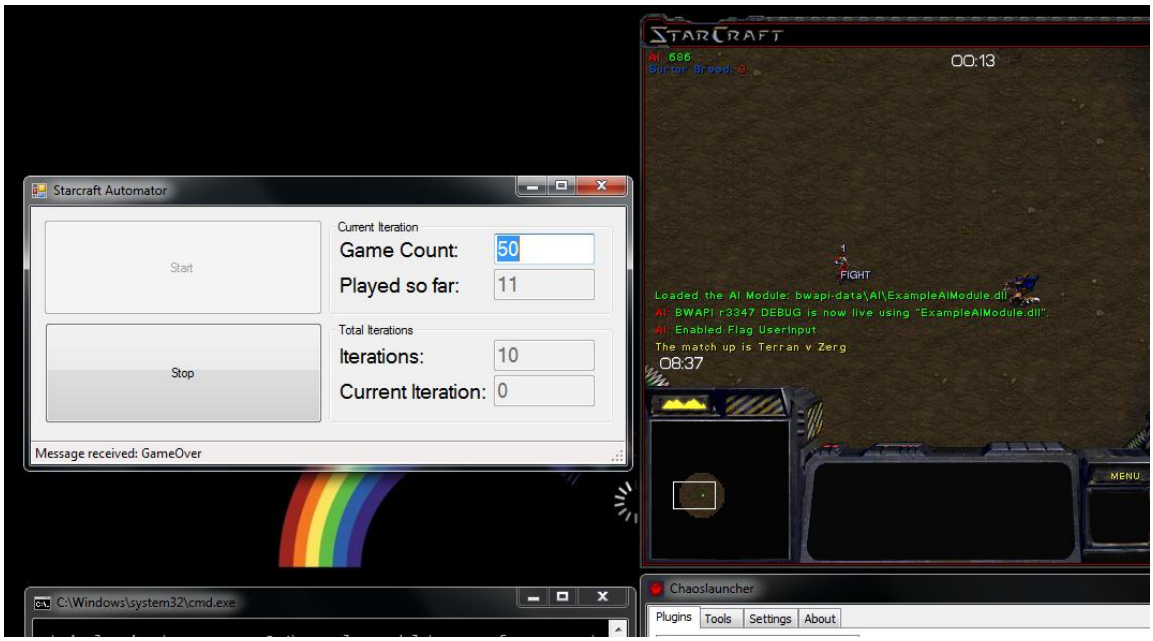

FIGURE A-0-1: STARCRAFT AUTOMATOR IN ACTION.

In order to make testing as easy as possible, a special tool was written. The tool, aptly named StarCraft Automator, handled automation and control over the StarCraft client, as well as handling various logs that the bot outputs during its run. StarCraft Automator connects via a socket on port 1337 to the StarCraft client. Each time a StarCraft match is over, the StarCraft client sends a single „GameOver“ packet to the automator, signaling the end of the match. BWAPI itself restarts the match and automates StarCraft's user interface. Every *n* games played, the tool deletes a file that describes the policy the agent is following, effectively resetting the agent's learning process. Whilst doing this, the tool stores the average reward given per episode.

APPENDIX B: SYSTEM REQUIREMENTS

The software requirements are as follows:

- StarCraft: Brood War, version 1.16.1
- BWAPI, version 3.5.4[5]
- The Visual Studio 2008 C++ Toolkit[6]
- Visual Studio 2010 (our IDE of choice)
- Visual C++ 2008 Feature Pack Release[7]
- Chaoslauncher[8]

RUNNING THE AGENT

After building it, you have to copy the resulting DLL file (called ExampleAIModule.dll) from the project debug folder into the StarCraft/bwapi-data/ai folder. In addition, you have to make sure that the *ai_dll_dbg* line in the StarCraft/bwapi-data/bwapi.ini file is set to ExampleAIModule.dll. Then you start Chaoslauncher/Chaoslauncher.exe and tick the BWAPI – Debug mode setting, and click start to start StarCraft with the agent loaded. In StarCraft you have to select Single Player – Expansion – Custom Game – Select a map from the BWAPI folder, and choose *Use Map Settings*. After doing so, you should be good to go.

[5] http://code.google.com/p/bwapi/downloads/detail?name=BWAPI_3.5.4.zip&can=2&q=
[6] Installs the C++ features detailed in the TR1 specification draft.
[7] http://www.microsoft.com/downloads/en/details.aspx?FamilyId=D466226B-8DAB-445F-A7B4-448B326C48E7&displaylang=en
[8] http://code.google.com/p/bwapi/wiki/Injection