Vulture Combat

Potential Fields, Reinforcement Learning and Bayesian Networks
Applied to Starcraft Broodwar

Student Report
Department of Computer Science
5. Semester
Subject: Build Intelligent Non-Player Characters (NPCs) for F
Address: Selma Lagerløfts Vej 300
Postcode: 9220 Aalborg East
Telefon: 99 40 99 40
Website: `http://cs.aau.dk`

Title:
Vulture Combat - Potential Fields, Reinforcement Learning and Bayesian Networks Applied to Starcraft Broodwar.

Theme:
Machine Intelligence

Projectperiod:
Fall 2011

Projectgroup:
d503e11

Supervisor:
Paolo Viappiani

Project Members:
Anuhi Ruiz Rivera
Dan Duus Thøisen
Daniel Heidemann
Kristian Pilegaard Jensen
Michael Louis Church
Thomas Birch Mogensen

Synopsis:

This report will focus on Bayesian networks, potential fields, and reinforcement learning applied to a real time strategy game called Starcraft Broodwar.

Number of copies: 3

Number of pages: 75

Appendices: 0

Completed: 19-12-2011

----------------------------------	----------------------------------	----------------------------------
Anuhi Ruiz Rivera	Dan Duus Thøisen	Daniel Heidemann
----------------------------------	----------------------------------	----------------------------------
Kristian Pilegaard Jensen	Michael Louis Church	Thomas Birch Mogensen

Contents

Contents **VII**

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Overview . . . 2

2 Analysis **3**
- 2.1 Basic Game Rules . . . 3
- 2.2 Important Techniques . . . 4
- 2.3 Terran Tactics . . . 6
- 2.4 Bot Analysis . . . 7
- 2.5 Unit Analysis . . . 7

3 Design **9**
- 3.1 Bot Managers . . . 9
- 3.2 Potential Fields . . . 10
- 3.3 Functions for the Potential Fields . . . 13
- 3.4 Agent Learning . . . 15
- 3.5 Application of Generalization of Q-Learning . . . 20
- 3.6 Bayesian Networks and Decision Trees . . . 22
- 3.7 Designing Bayesian Networks for Prediction . . . 23
- 3.8 Design Summary . . . 27

4 Implementation **29**
- 4.1 Class BayesianNetwork . . . 29
- 4.2 Managers . . . 30
- 4.3 Implementing Potential Fields . . . 34
- 4.4 $\hat{Q}$-Learning . . . 40

5 Tests **45**
- 5.1 Test Without Reinforcement Learning and Potential Fields . . . 45
- 5.2 Test Using the Potential Fields Without Reinforcement Learning . . . 45
- 5.3 Comparing α and γ Values . . . 46
- 5.4 Convergence Analysis . . . 51
- 5.5 Spawn Prediction Test . . . 52
- 5.6 Build Order Prediction Test . . . 53

6 Conclusion **55**

7 Future work **57**
- 7.1 Future Work on Potential Fields . . . 57
- 7.2 Future Work on Reinforcement Learning . . . 57
- 7.3 Future Work on Prediction Networks . . . 58

8 Appendices **59**
8.1 Picture of a Normal Game 59
8.2 A9G2 60
8.3 A6G4 62
8.4 A4G6 64
8.5 A2G9 66
8.6 Winning Streak 70
8.7 Reinforcement Learning Field Summary 72

Bibliography **75**

Chapter 1

Introduction

In this project we look into the Real Time Strategy game Starcraft Broodwar from the Machine Intelligence perspective. Starcraft Broodwar is interesting in this field for several reasons: it requires as much strategy as chess, it has the unknown aspect of poker, and it is as unpredictable as scrabble. These three factors make it an ideal game for Machine Intelligence.

Another advantage of Starcraft Broodwar is that it is easy to learn but difficult to master. The basics of the game are very simple and can be understood in a matter 30 minutes, but it can take years of intensive training to become a true professional.

Using a computer to play Starcraft Broodwar is interesting since a computer exceeds humans in several areas. It will be able to do many calculations during a game, and be able to control the mouse and keyboard very rapidly. Specially since these mouse and keyboard movements open the door for some interesting opportunities. A human can only control a limited amount of units or buildings at a time, but a computer is completely able to control them simultaneously.

Machine Intelligence is interesting in relation to Starcraft Broodwar because we would be able to make our bot really good at just specific fragments of the game. These could be:

- Teach it to use just the right strategy at the right time.
- Predict what the other players are during.
- Teach it to control single units in just the right way.

All the above parts make Starcraft Broodwar an interesting game to consider. We have a lot of different opportunity areas to focus on for the remainder of this report.

1.1 Purpose

The purpose of this project is to create a computer program that is able to play a full game of Starcraft Broodwar. The computer should be able to perform tasks that would be impossible for a human being. It should be able to learn through trials from its mistakes and improve by using machine intelligence theory.

1.2 Problem Statement

Our goals for this project are to:

- Create an intelligent bot for the game Starcraft Broodwar

- Apply Machine Intelligence theory in the modelling of the bot
- Make a bot that can improve by playing Starcraft Broodwar

1.3 Overview

We will now give a brief overview of the different chapters of the report.

Chapter 2 - Analysis The analysis begins with an introduction to the basic rules for playing a game of Starcraft Broodwar. Then we go on to explain some of the most common Terran tactics. We also talk about the importance of information and the balance between macro and micro. Then we take a look at what Machine Intelligence can be used to make a bot and look at which aspects of the game can benefit the most from Machine Intelligence. The final thing will be an analysis of the different units available for Terran and their strengths.

Chapter 3 - Design In the design chapter we begin by describing the different criteria we have for our bot. Then we go on to explain how we designed the different managers we use for various tasks like building and scouting. Then we give an overview of what potential fields are and how we use them for movement. Afterwards we describe different kinds of agent learning and how we can use it to improve potential fields. At the end we talk about Bayesian networks and decision trees to make predictions and decisions.

Chapter 4 - Implementation In this chapter we begin by looking at how we implemented Bayesian networks. Then we talk about the implementation of the managers and explain the most important parts of the code. We then look at potential fields and the differences between the design and implementation. At the end we look at how we implemented Q-Learning.

Chapter 5 - Tests First we look at how the built in bot does against itself. Then we look at how our potential fields do without any learning against the built in bot. Then we look at how different alpha and gamma values effect the way our bot learns and how this affect its performance. After this we test the different Bayesian networks and how they do at predicting the enemy's spawn, predicting the enemy's build order, and the enemies threat level.

Chapter 6 - Conclusion In this chapter we conclude on this project.

Chapter 7 - Future Work This chapter focus on different actions we could take to improve our work if we had more time.

Chapter 2

Analysis

In this chapter we will define the basic rules of the game and the techniques a player can use to help him achieve victory. This will help us determine in which ways we can apply Machine Intelligence theory so the bot can win a game of Starcraft Broodwar.

2.1 Basic Game Rules

Starcraft Broodwar is a complex game in which a player battles his army against his opponent's army. There are many strategies involved in doing this, but the game can be summarized in some basic concepts.

Before the match starts the player chooses one of three different races: Protoss, Terran, or Zerg. Even though all of these races are unique with different strategies and units, the basic techniques of the game remain the same no matter which one you choose.

The first rule of the game is that if a player loses all of his buildings then he loses. This means that in order to win against an opponent you must kill his last building before he kills your last building. The player has to worry about this threat throughout the entire match.

At the beginning of the game you start out with one building that can only train the most basic worker unit which can collect resources. Resources are used to buy buildings, units, and upgrades. It is important to collect as much resources as possible so you can buy more buildings, units, and upgrades than your opponent. This can be a difficult task. It is not enough just to get more resources than your opponent. You need to utilize your resources. Resources that are not being used for anything are wasted.

In general you want to have a high income but keep a low amount of resources in reserve. If you have extra resources you need to either buy more buildings, train more units, or buy upgrades for your army.

In order to destroy your opponent's buildings you must have an army (or at least some units). In the beginning of the game you have almost no army and you want to make your army stronger as the game progresses. It is not good enough to just have an army, but you must have a better army than your opponent. The optimal balance in army size is the exact force that is needed to keep you alive and force the enemy back, if an attack occurs.

Generally there is a trade off between your army and your economy. The larger the army you have the less you are spending on growing your economy (and vice versa). If you spend too much on your army but not enough on economy you may become too far behind later in the game to win. On the other hand if you spend all of your resources on your economy the enemy army could come and wipe you out. A picture from the beginning of a normal game can be seen in figure F8-1 in section 8.1 in the appendix.

2.2 Important Techniques

In this section we will introduce the elements of Starcraft Broodwar which have to be taken into account in order to win a match. This will give an understanding of Starcraft Broodwar. We will take a look at build orders, information gathering, macromanagement, and micromanagement.

2.2.1 Build Orders

A build order is an early game strategy describing the order in which certain buildings and units should be produced see reference [1]. The purpose of a build order is to optimize the player's economy and to control the timing of units which can give the player advantages at certain points in the game.

The build order depends on several factors like the specific map played, and the race the opponent has chosen. Early game scouting(see below) helps determine the opponent's build order and so you can counter their strategy.

A build order is most important at the beginning of the game and relies a lot on timing. The player must choose a spot to build a certain building, ensure a worker is there to do it, and make sure the player has the required resources. A good player knows the exact time to do these kinds of actions. He will make worker reach the construction site at the exact time he gets enough minerals to actually build the building.

Later in the the game the player must adapt his strategy depending on the information received about the opponent.

2.2.2 Information Gathering

Every strategy and decision made throughout the game depends on the pieces of information known in the current moment. From the most basic decisions like where to build, what to build, and which units to build; to wider decisions like how to defend, when to attack, or how to counter attack. Everything depends on the known information. The decision making process begins and grows with the input information. How the bot obtains pieces of information is just as important as how it process them. You do not want to waste too many resources finding out information otherwise you will waste units.

Map Knowledge

During the game many actions have to rely on the map that the players are playing on. Movements, starting point, tactics, and general strategy depend on how much data there is about the map. For example the resources you need for building and expanding through the game are normally distributed equally between the players, and the players are normally placed at opposite ends of the maps. Using the general map information and the information obtained in the moment the game starts (map, size, exact location, opposite race), a player will decide on his build order and starting actions.

Scouting

Reasons for exploring the map are not just to find out about the geographical structure but to obtain information on the actions of the opposing player. The player's actions will depend on the enemy's position, buildings, units, tactics, and strategy. The processing of information obtained from scouting is important and delicate since it is obtaining small pieces of information that structure a larger network of unknown information. Also, this other source of information is more volatile and temporal than the previous considerations because it is only obtained for the limited amount of time the unit stays in a certain place and to the areas the buildings limit the fog of war (the area on the map that the players is unable to see).

2.2.3 Macromanagement

Macromanagement often just called Macro is a key concept in Starcraft Broodwar. Macro is the ability to use all ones resources and the ability to expand at the right times to keep a healthy economy. The most difficult part of macro is to keep up production of units and to keep down resources while attacking. If players focus on economy and do not attack each other, the game is considered a macro game. This defensive tactic where a player stays in his base and builds up a big army is called turtling. Macro heavy games where players are turtling, lead to long games with massive battles.

Queuing

Queuing is a part of macromanagent where a player is queuing units in the same production facility. If a player has good macro he should not queue up units. Every time you queue a unit the resources are withdrawn instantly before the unit begins its production time. This means that optimally you only want to have one unit queued up at any given time for each production facility. This is also the case with workers. Normally the player wants to keep building workers throughout the entire match in order the have the best economy. In some cases it can be an advantage to stop building workers and focus on spending resources on something else like a building.

2.2.4 Micromanagement

Micromanagement also known as micro requires a player to control their units, so that you can get the most out of them.

For a human player this can be very difficult as it requires a lot of concentration and can hurt an inexperienced players macro. An advantage for a computer player since it does not have to balance its attention between micromanagement and macromanagement.

One way to do micro is to keep your units alive as long as possible. If your units stay alive after a battle then you do not have to replace as many units as your opponent. Players can move injured units out of battle and then back in. The reason for this is that the enemy will then start attacking another one of your units, so your injured unit can enter back in the battle and continue doing damage. A unit will do the same amount of damage no matter how much health it has but will do no damage if it is dead.

Another form of micro is making several of your units attack one unit at a time. This action is called *focus firing.* A computer can do this very well. Computers can perform calculations to find how many units it takes to kill another unit in one shot. This is useful so that too many units do not waste their shots trying to kill one unit when they can be damaging other units.

In order to micro units effectively you must be able to do a lot of different actions at one time. This is generally measured by a unit called APM. A computer can have an extremely high APM making it easy for it to micro its units effectively.

2.2.5 Meta-Game

Being unpredictable can win you the game when playing in tournaments and normal matches. If your opponent is adapting to your play style you can throw him off by doing something different or playing strange. In tournaments psychology is a big factor. Professional players can make their opponent do a certain strategy because they know how their opponent will react to certain actions. This feature will be hard or nearly impossible for a bot to learn because the bot won't be able to identify the opponent's play style in depth because the human can do a creative new build that the bot won't know and can't react to. If it can not identify the play style it will not be able to put the opponent off.

Before we can narrow down how our bot will work we will have to determine which race the bot will play. It could be able to play all three races, but this will be time consuming and not interesting in relation to Machine Intelligence. We have therefore settled on one race, which is Terran.

2.3 Terran Tactics

This section will feature some concrete techniques a Terran player can use to win a game. These tactics are important to analyze because a bot will need to utilize these tactics in order to win.

2.3.1 Timing Attack

A timing attack is an attack which either hits when a specific upgrade is done and/or when the opponent is vulnerable in his specific build order. When a bot does a timing attack it can be deadly because a bot can be very precise. Making the bot attack when a specific upgrade is completed is trivial, but for it to analyze when the opponent is weak may be a more difficult task. This would require the bot to research which build order the opponent is using and change its build order or strategy to fit a timing attack. If the bot could succeed in doing this, it could take opponents off guard.

2.3.2 Pushing

The term *pushing* refers to an attack that occurs when the Terran player has gained a large army after being defensive for a long period of time. It is called pushing because the idea is to push the opponent all the way back to their base where you can deal the finishing blow. This Terran tactic is relatively easy to execute because it is generally easy to defend your base as Terran. When you are ready to do the push your army will be difficult to beat.

2.3.3 Harassment

Harassing is a tactic that every race can use, but here we will give some ways for the Terran race to do it. Doing drops is one way to harass. Dropping refers to transporting units into the opponents base. The most common type of drops for Terran are done with vultures and marines. Vultures are fast and can take out the workers in the base. They can also place mines so enemy units can not easily come to defend the workers. When dropping with marines the purpose is generally just to destroy important enemy buildings rather than kill workers (note this is not always the case). Vultures can also be used without dropping them. Because of their superior speed vultures can easily slip past enemy units and into the enemy's main base. Doing drop harassment requires good multitasking ability. If you keep your units out of sight for one second they might be destroyed, and if you forget to macro you can get far behind economically. This is a field that the bot can really prove its abilities because multitasking is something that is hard for a human to do but easy for a computer.

2.3.4 Mining the Map

This tactic is used to gain map control by placing mines all over the map. A mine is set by a vulture if an enemy unit is near it. The mine will unburrow and run after the enemy unit dealing massive damage if hit. To see a burrowed mine you will need detection. By mining the map you can slow the opponent down because he needs to kill the mines before he can attack. This can give the Terran player security and a way to establish a superior army and economy. Mining the map is also a tactic that requires a lot of multitasking and is quite closely related to the harassment tactic because they both act in a aggressive way.

2.4 Bot Analysis

This section will discuss the way we will apply Machine Intelligence to control our bot. It is interesting to look at how a bot compares to a human both physically and mentally.

Outplaying a human player

A bot can easily physically outplay a human player in Starcraft Broodwar. Doing micromanagement is really demanding for a human because it requires a lot of actions and fast reaction. A bot may be able to control each attacking unit individually and make intelligent decisions on attacking according to health, cooldown and other factors. This is nearly impossible for a human to do in a large scale. This is why a bot that is focusing on micromanagement could be interesting to make. Another point is that while a player is doing micromanagement a lot of other actions are needed elsewhere. If these tasks are not handled this could lose one the game.

Outsmarting a Bot

Outsmarting refers to either bluffing, tricking or just making good decisions. This will be hard for a bot to do, because there are so many different strategies. Being able to reacting to everything and be able to counter these strategies will be difficult. Something unexpected can break the bot down completely.

2.4.1 Choice of Main Focus

The area we think the bot will really be able to out play a human is in micro. It can do this by controlling a lot of units very precisely. This is interesting since it will be able to do something that is impossible for a human player, i.e. the bot will be able to control one unit at a time. It will also be easier to spot the learning part of the bot because the performance can be measured, e.g. units killed.

2.5 Unit Analysis

We will now take a look at three Terran units which could be interesting to micro. In the analysis of the units we will focus on the range, movement speed, damage, armor, availability, micro benefits, and fire cooldown.

2.5.1 Marines

Hit points	40
Range	4
Weapon Cooldown	7.5/15
Price	50m
Building needed	Barracks

Marines are interesting because they can shoot air and ground units. The upgrade *stimpacks* increases the marines movement speed and doubles the fire rate. This makes the unit very dynamic and gives it high micro benefits. Marines are easy to get and are normally the first unit a Terran uses in their army. Marines only have a range of 4 and are the slowest unit that we are going to take a look at. Their hit points are only 40. Since marines can be gotten so early, they are a good candidate for the project. On the other hand, to get the full benefit from the marines they need upgrades and support from medics to heal them.

2.5.2 Vultures

Hit points	80
Range	5
Weapon Cooldown	30
Price	75m
Building needed	Factory

Vultures are a little car with a high speed. Because of a longer range the vulture can out range the marine. Like the marines, the vultures also have upgrades. The most interesting upgrade is the spider mines that allows the vultures to place three mines. When another unit goes near the mine it will detonate dealing a large amount of splash damage. The vultures are made from the Factory which means you will be able to get marines before you get vultures. The good thing about vultures is the price is only 75 minerals. Having all this in mind, it is clear that vultures have much to offer in terms of micro. That being said the vultures fire cooldown is 30 which is twice the marine's cooldown (see reference [2]).

2.5.3 Wraiths

Hit points	120
Range	5
Weapon Cooldown	22/30
Price	150m 100g
Building needed	Starport

Wraiths are air units that can attack ground and other air units. This means that they can move over edges and cliffs which allows it to move over large distances within a short time period. They are interesting because they have an upgrade that allows them to turn invisible while they have energy. It has the same fire cooldown as a vulture when attacking ground units and 22 fps when attacking air units. The down side of the wraiths is the price (150 minerals and 100 gas) and we need a Starport to built it. This means there is a long time before you can get actually buy wraith. It is clear there is a lot of micro benefits from choosing this unit.

Chapter 3

Design

In this chapter we will distinguish between the word agent and bot, where an agent is a program thats able to learn and chance itself, and the bot it the more static.

In this chapter we are going to look at the design of the bot. First we will take a look at the different managers we use for controlling units, producing buildings, scouting bases and choosing strategies. Then we explain Potential fields, and how we use them to micro the bot. In order to correct the bot and make sure that it will be able to learn, we will explain reinforcement learning. For the macro part we are going to use Bayesian networks to detect the opponent's build order, predict the opponent's start location, and find the threat level of the opponent.

The purpose of the design is to make an agent that is able to out micro its opponent in combat. To limit the scope we will only look at vultures. This implies the following:

- Construct the necessary buildings to produce vultures.
- Construct a squad of vultures.
- Scout and find the enemy's base.
- Move the squad to the enemy's base and attack.

3.1 Bot Managers

The managers for the bot are used to control different parts of the bot. The part which is connected to the micromanaging of units will use reinforcement learning and Potential fields. The part which manages scouting information will use Bayesian networks. The purpose for these managers are to separate parts of the game as much as possible. They should only use the other managers when a task needs to be handled by another manager. The following subsections will explain the different managers.

Tactics Manager

This manager is responsible for managing our offensive units in and out of small scale combat. It manages the units by controlling squads of different units.

Production Manager

This manager is responsible for producing units, constructing buildings, upgrading technology and researching technology. The manager will be able to know what to build by following build orders.

Scouting Manager

The Scouting Manager keeps track of the opponent. It saves every enemy it has seen and is responsible for scouting the enemy's spawn position. It also keeps track of which buildings the enemy has produced.

Strategy Manager

The Strategy Manager is responsible for making high-level decisions based on the information we have. These decisions include setting the Tactics Manager to attack the enemy or setting the Production Manager to change build order.

Worker Manager

The Worker Manager sets the workers to mine minerals, mine gas, scout, construct buildings and perform other tasks.

3.1.1 Managers in Game Scenario

The game starts and the Production Manager begins following the build order which produces SCVs. Every time an SCV is created the Worker Manager sets the worker to a free mineral patch. The Scouting Manager sends out an SCV to the opponent's base. The scout finds the base and the Strategy Manager still predicts that the threat level is low. The Production Manager keeps following the build order, and now we have marines assigned to a Tactics Manager. The Scout Manager sends out another SCV, but this one is killed before reaching the base of the opponent. The Strategy Manager sets the threat level to high and makes the Production Manager stop producing workers. The production manager will then start producing only offensive units. The opponent does the attack, and the Tactics Manager micromanages the units which defeats the enemies.

3.2 Potential Fields

Potential fields can be used to move a computer AI through a dynamically updated environment. If one would program an AI which should move from one point to another, one would most likely use a normal shortest path algorithm. The problem is that if there are a lot of dynamic influences that effects the route, this calculation becomes very complex. It could be enemies that should be avoided or other dynamic influences.

This is why Potential fields are good for such a problem. They work by generating either attractive or repulsive fields of vectors $v = (m \times d)$. Where m is the magnitude and d is the direction. If we create an attractive Potential field the point will be surrounded by vectors pointing toward this point, as seen in figure F3-1. The bot will be attracted towards this point. The magnitude determines how attractive the field will be. The direction shows in which way the bot will move. This is called an Attractive behaviour

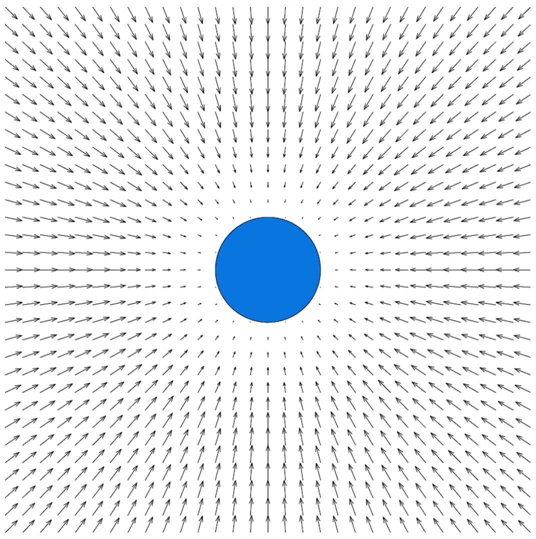


Figure F3-1: Attractive behavior[3]

Likewise if we assume there is only a single obstacle in the area (a unit we do not want to attack) it would generate a repulsive field around it, see figure F3-2. This is called a repulsive behaviour because it causes our own units to try and avoid it.

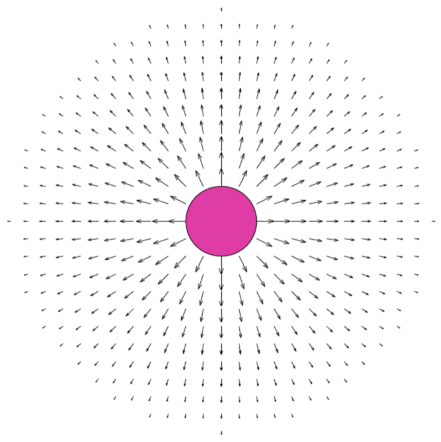


Figure F3-2: Repulsive behavior (see reference [3])

These two kinds of behaviours can be combined to make a map that can tell our unit how to move around enemy units and reach a specific target as seen in figure F3-3.

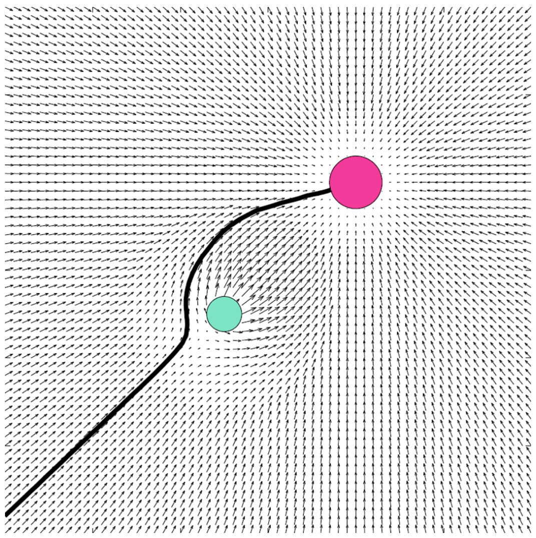


Figure F3-3: Combined behavior (see reference [3])

3.2.1 Designing our Potential Field Functions

The Potential fields in our bot will reduce the vectors into constant values that represent their importance or magnitude. We arrange the vector to fit linear values that indicate how attractive or repulsive a field is. This number is calculated by functions, which is either attractive or repulsive and are related to the obstacles in the game world.

The reason we can use numbers instead of vectors is because we only care about the Potential on the tiles [1] immediately surrounding a unit, so the direction is given by just taking the highest number. As seen in figure F3-4.

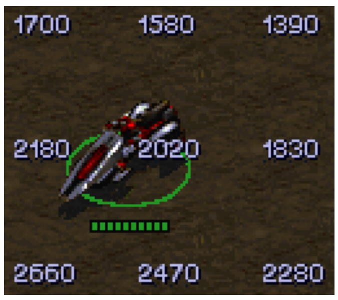


Figure F3-4: Vector direction given by numbers

[1]A tile represents an $8px \times 8px$ square. This is the smallest area units can stand on or move to.

Potential Field Basic Structure

Our functions will be described with the following math (see reference [4]:)

Variables:
f = force
s = size of the Potential field
c = constant
d = distance

$$Attractive = \begin{cases} f * c & \text{if } d > s \\ 0 & \text{else} \end{cases}$$

$$Repulsive = \begin{cases} -f * c & \text{if } d > s \\ 0 & \text{else} \end{cases}$$

3.3 Functions for the Potential Fields

This section contains the functions which will be used to calculate the potential of a field. Here is introduced some variables that are going to be used in the functions.

Variables:
c = A constant used to adjust the size of the different potential fields.
f_c = A force used to adjust the potential field.
wr = Boolean denoting whether or not the weapons are ready to fire.
sr = Units maximum shooting range.
da = Distance from unit tile to nearest ally unit.
dua = Distance from current tile[2] to nearest ally unit.
ds = Distance from center of army to unit tile.
dsv = Distance from center of army to current tile.
de = Distance from unit tile to enemy.
due = Distance from current tile to enemy.
dc = Distance from unit tile to nearest cliff or edge.
duc = Distance from current tile to nearest cliff or edge.

Where f_c and c are specific to each part of potential field and named after each e.g. f_S for force for the squad and $c_{squadSize}$ for the size of the squad. A negative force will result in a repulsive potential field, and a positive force will yield an attractive potential field.

The calculation $(2ds - dsv)$, found in section 3.3.1 and other calculations with the same structure, are used to invert the distances. This calculation is needed because a lot of the potential fields depend on distance. But normally when calculating the distance from one unit to another, the distance closer to the other unit would be smaller, and the distance further away would be larger. Because the highest number in a potential field is the most attractive one, we sometimes need to make the closest distance the most attractive one which is what $(2ds - dsv)$ does. An example can be seen in figure F3-5.

[2]Current tile is the tile we are currently calculating the potential field for.

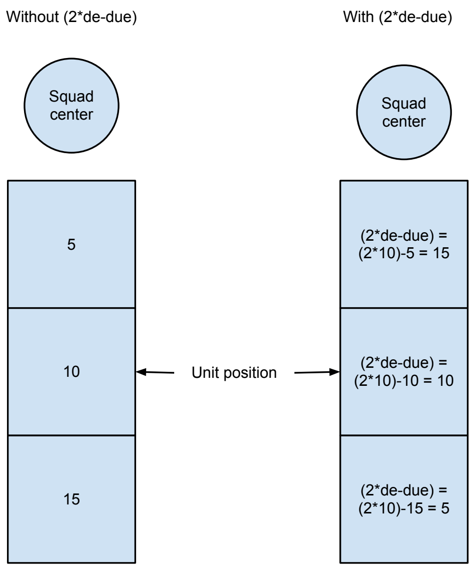

Without (2*de-due)
With (2*de-due)
Squad center
Squad center
5
10
15
Unit position
(2*de-due) = (2*10)-5 = 15
(2*de-due) = (2*10)-10 = 10
(2*de-due) = (2*10)-15 = 5

Figure F3-5: Correcting the distance

3.3.1 Squad Center (Attractive)

A vulture is more likely to survive if it sticks together with its squad. It will both give a better damage output and also spread out damage taken.

$$SquadCenter = \begin{cases} 0 & \text{if } ds < c_{squadSize} \\ f_S \times (2ds - dsv) & \text{if } ds >= c_{squadSize} \end{cases}$$

First case The units are within the desired squad size. In that case we do not need to attract the units more towards the squad center.

Second case The unit is outside the desired squad size. In that case we need to attract it towards the squad center.

3.3.2 Maximum Distance Positioning (Attractive)

The vultures have to utilize their range, so they will be attracted to the position which gives them the maximum distance to the enemies.

$$MaximumDistancePositioning = \begin{cases} f_{MDP} \times (2de - due) & \text{if } de < sr \\ 0 & \text{if } de > sr \end{cases}$$

First case We want to get closer to the enemies because we are not within shooting range.

Second case The unit is within range and doesn't need to get closer. One could argue that we need a third case to handle if we are getting to close, but this is handled by *Weapon cooldown*.

3.3.3 Ally Units (Repulsive)

We do not want all of the vultures to clump completely together during a match. This can lead them vulnerable to splash damage attacks. If we keep the vulture spread out a reasonable distance the splash damage will not have any effect.

$$AllyUnits = \begin{cases} 0 & \text{if } da > c_{allyDistance} \\ -f_{AU} * (2da - dua) & \text{if } da < c_{allyDistance} \end{cases}$$

First case The unit is sufficiently far away from the enemies to not sustain splash damage, so we do not need to get any further away.

Secound case Active when the units are too close, and we need them to move away from each other.

3.3.4 Weapon Cooldown (Repulsive)

A vulture cannot do any damage right after it shoots. This is because there is a certain amount of time in between each shot. Right after a vulture shoots we want to move it out of the battle until it can fire again. This way each vulture will be less vulnerable to taking damage when it cannot give any damage.

$$WeaponCoolDown = \begin{cases} 0 & \text{if } wr \\ -f_{WCD} * (2de - due) & \text{else} \end{cases}$$

First case The weapons are not on cooldown, so nothing happens.

Second case The weapons are on cooldown, and we need to flee.

3.3.5 Edges and Cliffs (Repulsive)

We do not want our vultures to get stuck against any walls so we will add a potential field for any edge where a unit can get stuck.

$$EdgesAndCliffs = \begin{cases} -f_{EAC} \times (2 * dc - due) & \text{if } dc < c_{edgeDistance} \\ 0 & \text{if } dc > c_{edgeDistance} \end{cases}$$

First case We are too close to the edges and need to move away.

Second case Nothing happens.

3.4 Agent Learning

The purpose of agent learning is to be able to adjust each force from the potential field, to help improve its performance, and to learn how to fight efficiently.

This section addresses various learning techniques and methods that could be used in the AI agent. There will be a brief explanation of the the two options we considered. First, we give a brief overview of neural network. This is followed by the examination of its usability in our project. Then the we talk about different reinforcement learning techniques.

Neural Networks

Neural networks cover different learning methods that an agent can execute to approximate values or target functions. They are ideal for interpreting complex real world data and are widely known for being one of the most effective learning methods. These methods can be used to teach behaviour patterns in a human-like way. This is because neural networks are biologically based in how the brain obtains, stores, and uses new information.

Neural Networks model a very complex web of interconnected nodes that take large sets of numbers and reduces all the input into one single number for output. This network is constructed of several individual neurons. Each neuron takes 3 numbers as input and delivers one as an output. Using this principle, several input numbers are forwarded through all the connections (both expansions and simplifications) until one final choice or signal is produced.

We have chosen not to use a neural networks. The main reason is that neural networks are very slow. The complication with neural networks is the number of calculations that are made for making every decision grows exponentially in time complexity. In a real-time-strategy game where we can have around 50.000 states, the computation time will take too long and the input will be way to high since every minute 50.000 states have to be computed. A less analytic generalization method would be a better approach because it approximates the correct decision and does not have to compute the entire state-space in a game.

Reinforcement Learning

Reinforcement learning (RL) is a method used to build models or functions that learn from experiences and examples. The basic idea is that for every action in an environment, there is a reward or some feedback that reinforces all actions that have a bigger reward. We consider a reward a numerical value that grades the result of any action. On a larger scale, the task of reinforcement learning in this report is the process to discover the optimal path; the series of actions that accomplish the best possible total reward at the end of the process. This reward depends entirely on the agent's policy, better defined as the strategy it follows for accomplishing something.

There are different RL techniques that depend on the amount of information we have available for learning. They can be classified into passive and active RL methods. *In passive RL the agents policy is fixed and the task is to learn the utilities of each state.* (see reference [5, p764]) This implies that the environment is fully observable and the agent knows the future impact of its actions. The learning part of the algorithm is only in charge of finding the best strategy for the already defined probabilities. *The active reinforcement learning does not have a fixed policy to begin with, and the agent must decide what actions to take* (see reference [5, p771]). So the agent basically explores considering that it can't look ahead for more than a move or predict the effects of its actions in the future.

Markov Decision Process

With the intent of further expanding on the RL capabilities, we need a way to define the task of the agent. A general formulation of the problem starts based on Markov Decision Processes: *In a Markov Decision Process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform. At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions δ and r are part of the environment and are not necessarily known to the agent. In an MDP, the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.* (see reference [6, p370])

With this we almost have enough information to build the problem structure of the agent. We need to consider a function that describes the *total cumulative reward* of a set of actions. It could be any function: discounted cumulative reward ($\sum_{i=0}^{\infty} \gamma^i r_{t+i}$), average reward($\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$), finite horizon reward ($\sum_{i=0}^{h} r_{t+i}$). This reward function varies depending on what the agent needs to learn. The most common example is the discounted cumulative reward which is just the sum of all the rewards with a discount factor (γ) that progressively reduces the importance of past experiences.

Once we have the reward function, we can define the learning task of the agent. In passive RL it is to find a *utility function* or how good a certain policy is. In active RL is to find a policy that maximizes the value of the reward function, in other words, finding the *optimal policy.*(see reference [6])

Direct Utility Estimation & Bellman Rules

In passive RL, the method of Direct Utility Estimation (DUE) follows the idea that the utility of a state is the expected total reward from that state into the future. At the end of a trial, the RL algorithm for DUE will trace back through all the observed rewards and calculate the estimated utility for every state. It is basically a process of inductive learning that observes a set of data that is completely known. The formula for calculating the utility values follows the Bellman equations for a fixed policy (see reference [5]) The problem with this technique is that it ignores the relationship between the states (they are not independent from each other), and it only learns at the end of a trial, therefore missing several opportunities for learning and converging very slowly (see reference [5]).

Adaptive Dynamic Programming

Searching for a way of considering the constraints between states, we come across Adaptive Dynamic Programming (ADP). This is an agent that learns the transition model for an environment while solving the MDP. We call transition model the function that evaluates form the current state and action to the new state. In a fully observable environment, this means that you use a transition model and the observed rewards into the Bellman equations to calculate the utilities of a state. In simpler terms, it means that while learning each step (state-action pair) you keep track of the outcome and then save it into a table of probabilities. At the end you have a comprehensive probability table for all the transitions, in this way you know a reliable model for knowing what is going to happen with every state-action pair (see reference [5]).

But we have to take into consideration all the possible states in a game. We call all the possible states that can be achieved by any action throughout the entire game the state space. The problem is it is impossible to calculate ADP tables for large state spaces, specially if you have to run several trials to exhaust every reasonable possibility for every transition. Therefore not usable for our agent.

Temporal Difference Learning

The combination of the previous two methods, *using the observed transitions to adjust the values of the observed states so that they agree with the constraint equations*(see reference [5, p767]), produces the following rule (to be applied every time a transition occurs form state s to s'):

$$U^\pi(s) = U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)) \tag{3.1}$$

This update rule (temporal difference equation) uses the difference in utilities between succesive states. It shifts or updates the estimates towards the ideal equation. There are several things to notice here. The first is that since it updates with the next state (s'), it might seem like it adapts too much to every trial, but in reality this update rule is applied several times, therefore producing an average and isolating rare cases. (see reference [5])

The second is the appearance of α, also known as the learning rate parameter; how much it learns from an specific trial. Normally the value of α would be something like: $\frac{1}{1+numberOfVisits(s,a)}$ (see reference [6, p382]). Which then decreases the magnitude of the update proportionally to how many times you visited a state-action pair.

Q-learning

All the previous methods have been explained considering there is a fixed policy that determines the behaviour of the agent. Now the task shifts to active reinforcement learning because we need our agent to decide which action to take in each state-action pair. We need an active temporal difference learner (that learns the utility function U). In Q-learning this utility function is:s (see reference [5])

$$U(s) = max_a(Q(s,a)) \tag{3.2}$$

Where $Q(s,a)$ represents the value of making and action a in a state s. And if we use the temporal difference approach for Q-learning we have the following updating rule: (see reference [5])

$$Q(a,s) \leftarrow Q(a,s) + \alpha[R(s) + \gamma max_{a'}(Q(s',a')) - Q(a,s)] \tag{3.3}$$

Q-learning is a temporal difference learner that *does not need a model for either learning or action selection* (see reference [5, p775]). It simply executes the updating rule every time and action a is executed in a state s that leads to an state s'. The only restriction we have left is that the algorithm for exploration for this Q-learning temporal difference agent is the same as in the ADP agent (see reference [5, p776]) . That means that it keeps track of every movement and saves the statistics in a table. It easily learns and saves the *optimal policy* for small state spaces, but it is impossible to keep track for larger ones.

Generalization of Q-Learning

The choice of using reinforcement learning was based on the necessity to train our AI to improve while it plays. Since our environment covers a lot of different factors and variables, we decided to use a form of active reinforcement learning that simplifies the complexity and size of all the different states.

Generalization in reinforcement learning takes into consideration huge state spaces by representing them as function approximations. This function reduces the complexity of mapping all the states considerably and allows the learning agent to generalize from the visited states to the non-visited ones.

There is the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well. As in all inductive learning there is a trade-off between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means the convergence is likely to be delayed. (see reference [5, p778])

When a function approximation reaches the optimal values or closes around them can say it converged. The option of choosing linear function would ensure that the convergence is not excessively delayed. For example, if there is a function approximation in the following form:

$$\hat{Q}(x, y) = f_0 + f_2 x + f_3 y \tag{3.4}$$

We can use the updating rule or $\hat{Q}$-learning equation that evolves from the Q-learning temporal difference formula, now taking into consideration the values of the function approximation: (see reference [5])

$$f_i \leftarrow f_i + \alpha[R(s) + \gamma(max\hat{Q}_f(a', s')) - \hat{Q}_f(a, s)]\frac{\partial \hat{Q}_f(a, s)}{\partial f_i} \tag{3.5}$$

Where f_i is each one of the coefficients in the Q-approximation.

α - Is the learning rate. As mentioned before it means how much you modify the value of the coefficient f_i to fit the current example or situation, it learns from each visit to every state. Its a number, $0 < \alpha < 1$, that in a normal temporal difference equation (see reference 3.4) would (optimally) decrease according to how many times a state is visited (see reference [5]). Since every one of our states is visited infinitely many times, the value α can be a fixed number that we modify manually. The higher the value, the more you learn from every specific case.

γ - Is the discount factor, this determines the importance of the future rewards. Its a number, $0 < \gamma < 1$, the closest γ gets to 1 the more it takes into account future rewards. A γ value close to zero would maximize the immediate rewards.

$R(S)$ - represents the reward function for the current state.

$\hat{Q}_f(a, s)$ - Is the value of the $\hat{Q}_f$ function for the next position (one step ahead). The next position depends on the current state s and the action a performed by the agent from that state.

$max(\hat{Q}_f(a', s'))$ - Is the highest possible $\hat{Q}_f$ value calculated from the next position (two steps ahead). The highest possible option (for every a') once the current state s has performed an action a and is in a new state $s' = \delta(a, s)$.

$\frac{\partial \hat{Q}_f(a,s)}{\partial f_i}$ - Is the partial derivative of the $\hat{Q}_f$ function with respect to the current f_i, in other words the variables or factors that afect only that coefficient f_i. And in the case of our linear $\hat{Q}_f$ function, always a constant representing some distances in a certain state.

Convergence in Aproximation of Q-Learning

In a normal Q-learning process the rules are the following: *First, we must assume the system is a deterministic markov decision process. Second, we must assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, $r(s, a) < c$. Third, we assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often.* (see reference [6, p377])

The only difference between this process and the approximation to Q-learning is the generalization of unvisited states. But this generalization is also guaranteed: *These update rules can be shown to converge to the closest possible approximation to the true function when the function approximator is linear in the parameters.* (see reference [5, p779])

3.5 Application of Generalization of Q-Learning

The Starcraft Broodwar environment has several factors that could be considered important for defining a moment in time; units, frames, enemies, distances, map elements etc. Since formulating a model considering all the factors made the function too complex, we focused on the factors that could model the environment as closely as possible without increasing the size and computability time of each calculation. We have to also consider how long is the learning time.

3.5.1 Q-Learning Functions

We combine the generalization of Q-learning with the potential fields to obtain a reasonable model of the Starcraft Broodwar environment. We do this by transforming all the potential fields (per unit) into a simplified version of the Q function used by the agent. The first thing we needed to specify was a representation of all the data relevant for a specific time or frame, our hypothesis space.

State - Hypothesis Space

We define a state in our environment as a combination of the most important factors that interact with the agent and the game. It consists of all the distances used in the potential fields plus the numbers required to calculate a comprehensive reward function.

$$State = \begin{cases} da \\ dua \\ ds \\ dsv \\ de \\ due \\ dc \\ duc \\ wr \\ sr \\ numberOfUnits \\ healthLost \\ damageDealt \\ numberOfKills \\ time \end{cases}$$

The variables numberOfUnits, healthLost, damageDealt, numberOfKills and time are variables accessible through the entire game, therefore used as part of our reward function. The description the rest of the factors or distances is the same as mentioned before in the potential field's documentation (see reference 3.3).

Function Approximation

After defining a state in the game we created a linear function approximation to ensure convergence of each value. The function takes all the forces that determine the magnitude of the potential field vectors as coefficients or weights in the $\hat{Q}$ function.

$$\hat{Q}_f = f_{MDP}(2de - due) + f_{AU}(2da - dua) + f_{EAC}(2dc - duc) + f_S(2ds - dsv) + f_{CD}(2de - due) \tag{3.6}$$

This function is not a thorough model of the true utility function, but it covers all the factors that affect the movement of a unit. Since every unit is controlled independently with this movement model/function, it covers the purpose of using the computer's capabilities of controlling each unit's movement separately and optimally (micro).

Notice that there are forces that are dependant on the exact same variables, like Cooldown and Maximum Distance Positioning. This coefficients vary in magnitude because they are updated and calculated under different circumstances and throughout different moments in the game. This is furthered explained in the implementation documentation (see section 4.4.1).

Updating Rules

Then we apply the updating rules (equation 3.6) mentioned for the $\hat{Q}$ learning. In the context of our $\hat{Q}_f$ function, it would represent the forces: f_{MDP}, f_{AU}, f_{EAC}, f_S, f_{CD}. The exact same forces that represent each one of the potential fields. So we are left with the following updating rules for each of the coefficients/forces:

Maximum Distance Positioning

$$f_{MDP} \leftarrow f_{MDP} + \alpha[R(s) + \gamma(max(\hat{Q}_f(a', s'))) - \hat{Q}_f(a, s)](2de - due) \tag{3.7}$$

Ally Units

$$f_{AU} \leftarrow f_{AU} + \alpha[R(s) + \gamma(max(\hat{Q}_f(a', s'))) - \hat{Q}_f(a, s)](2da - dua) \tag{3.8}$$

Edges and Cliffs

$$f_{EAC} \leftarrow f_{EAC} + \alpha[R(s) + \gamma(max(\hat{Q}_f(a', s'))) - \hat{Q}_f(a, s)](2dc - duc) \tag{3.9}$$

Squad

$$f_S \leftarrow f_S + \alpha[R(s) + \gamma(max(\hat{Q}_f(a', s'))) - \hat{Q}_f(a, s)](2ds - dsv) \tag{3.10}$$

Cooldown

$$f_{CD} \leftarrow f_{CD} + \alpha[R(s) + \gamma(max(\hat{Q}_f(a', s'))) - \hat{Q}_f(a, s)](2de - due) \tag{3.11}$$

These updating rules should eventually converge to values that are very close to the optimal Q function, since we considered all the restrictions for convergence. The final result (after converging) should be the perfect magnitude for the potential fields to guide every unit's movement.

Reward Function

We created a reward function that takes into consideration all the factors to grade the performance of the agent. The reward function gives positive points for keeping the highest number of units alive, negative points for loosing health, positive points for both killing or damaging the enemies, and a negative reward for every time frame in the game that goes by. This way we ensure that the agent wants to attack the enemy while protecting its units; but not prioritizing protecting the units. We control that the agent chooses attacking over hiding or running away by making the reward proportional to how short the match is.

$$R(s) = C_1 numberOfUnits - C_2 healthLost + C_3 damageDealt + C_4 numberOfKills - C_5 time \tag{3.12}$$

We ensure that the reward complies with the convergence restrictions for Q-learning (see reference 3.4). The reward function is bound, $R(s) <= C$. The upper bound of the constant C defined by $C = C_1 startingNumberOfUnits + C_3 maximumDamageDealt + C_4 maximumNumberOfKills$. The lower bound defined by $C = -C_2 maximumHealthLost - C_5 frameCount$.

3.6 Bayesian Networks and Decision Trees

In this section we are going to define Bayesian networks and decision trees. Then we are going to compare the two decision models and choose the best model that our bot can use for analysing information.

3.6.1 Bayesian Networks

Bayesian networks are simple graphical models where each probability for the node is calculated. Therefore the Bayesian networks are used for calculating new probabilities whenever new information is gathered. Bayesian networks have a set of nodes (F3-6- C1,C2,C3) that are connected with directed edges. Each node in the Bayesian network must have a finite set of mutually exclusive states. By this we mean that we can only be in one state in each node. To make sure that we can calculate a result, the Bayesian network needs to be an acyclic directed graph. A node needs a conditional probability table for each of its parents. This means that the amount of calculations in a Bayesian networks depends on the number of nodes and the edges that connect them (see reference [7, p. 33]).

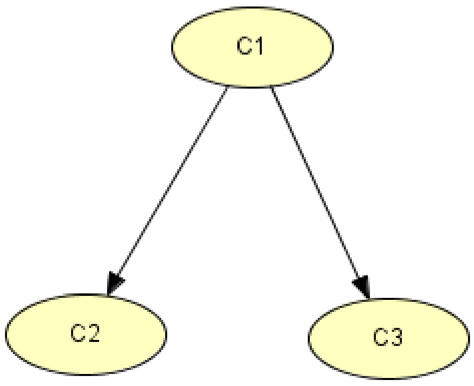


Figure F3-6: A simple Bayesian network

3.6.2 Decision Trees

Decision trees are used to represent decision problems. The figure F3-7 gives an example of a decision tree that helps investment decisions. A decision tree consists of three types of nodes: decision nodes (F3-7 square boxes), chance nodes (F3-7 circles), and utility nodes (F3-7 triangles). The link from a decision node to a chance node is called an action, and a link from a chance node to a decision node is called a state. The idea of the decision tree is to find the path that will give us the highest utility (reward). To make a decision tree over a decision problem, every possible path of decisions have to be shown in the tree. This will create a tree that grows exponentially with the number of decision and chance nodes. Small decision problems will require big trees if not reduced.

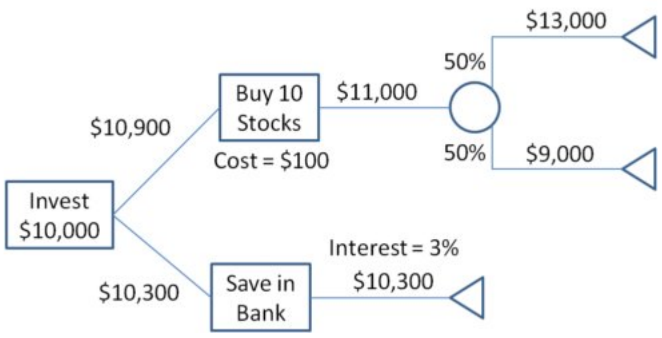


Figure F3-7: A simple decision tree (see reference [8])

3.6.3 Choice of Model

We choose to use a Bayesian network solution for our problems because decision trees grow exponentially with the number of nodes, and Bayesian networks are easier to handle and manipulate.

3.7 Designing Bayesian Networks for Prediction

This section will describe the Bayesian networks we made to predict the enemy's spawning position, what build order they are doing and what the current threatlevel is. The next subsection describes the network created for predicting the enemy spawn position.

3.7.1 Prediction of Enemy Spawn Position

The purpose of this Bayesian network is to find our opponent's spawning position. Some factors taken into account are our spawn position, enemy scouts, and time into account to help us predict this. We wanted to make this network usable on a variety of maps. Below is a visual representation of how each node is connected in the network.

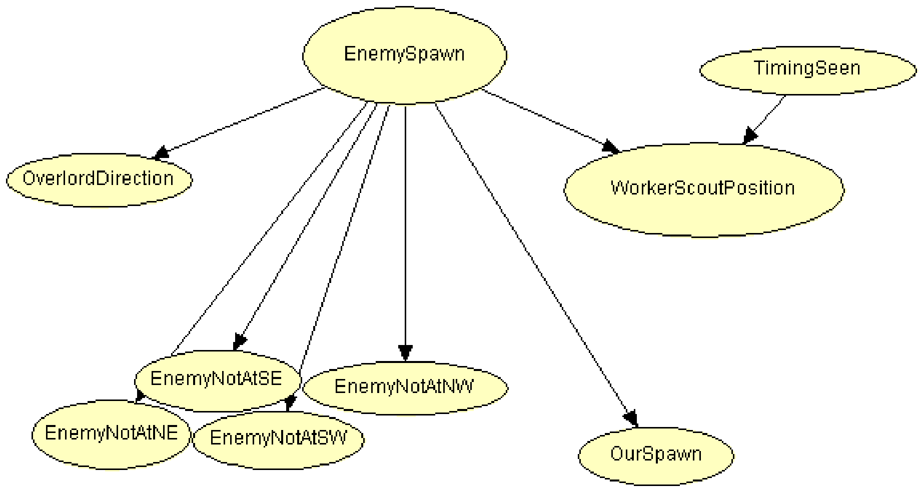


Figure F3-8: A graph of the nodes in the Spawn Prediction Bayesian network

EnemySpawn

The correct state in this node is the value we are trying to discover. Ultimately, all of the evidence we collect, will be used to find this state. Most of the other nodes have direct links from this node. Once we know the state of this, we are satisfied and do not need to use the network to infer more information because we do not care about predicting any of the other states.

OurSpawn

This node is a child of EnemySpawn. We can use the values in this node to determine where the enemy does not spawn. We simply say that the enemy cannot spawn where we spawn at. At the beginning of a match we know where we spawn and can instantly put evidence on the OurSpawn node. This instantly reduces the change for our opponent to be at any other spawning location to 33%.

EnemyNotAt(NE,SE,SW,NW)

These nodes are used to keep track of positions we know our opponent did not spawn. Since a node can only be in one state at a time, we made four nodes that can influence the probabilities in enemy spawn. Ways we would generally get these values could be our own scouts arriving at a position and not finding our opponent.

WorkerScoutPosition

When we observe an enemy worker at a certain position on the map, we can influence our belief on our opponents spawn position. This is because the opponent's scout will take a certain amount of time to get to a certain position on the map. TimingSeen is a parent of this node. We use the information from this node to help our prediction. Just seeing an enemy at a certain position is not enough. We need the time we see the enemy scout to help us form our beliefs. Whenever we obtain evidence on WorkerScoutPosition we will also always gather information on TimingSeen.

TimingSeen

This node is used to help us use the information from WorkerScoutPosition. It has the values: Almost None, Early, Middle, and Late. These values are the times we may see the opponent's scout. Since the values for timing depends on both map size and map layout, we purposely made the values broad.

OverlordDirection

Since overlords are so slow, a player will be able to use the direction the overlord is coming from to predict the spawn location. By the time an overlord would have visited two bases we probably would have already known where the enemy's base is. This node only helps when fighting against Zerg players. EnemySpawn is a parent of OverlordDirection. Once we know the direction the overlord is coming from we gain a lot of information on the enemy's spawn position.

3.7.2 Build Order Prediction

This subsection describes the Bayesian networks that was created for predicting that build order the enemy is doing. The networks have some similarities between them. There are three different networks, one each for every race that the bot can encounter: Terran versus Terran, Terran versus Protoss and Terran versus Zerg.

They all have a node called BuildChosen, which is the node containing the beliefs for what the enemy's build order is. The other nodes are buildings (Except an upgrade node in the Terran versus Protoss network) all with the states Seen and NotSeen, though Seen is only used when presenting evidence, because the bot will never be able to prove that the enemy building does not exist. These states determines if we have scouted the building. The order in which the buildings come in is modelled with arrows going from the nodes representing the buildings needed to the node representing the building which needs that building. As buildings are scouted evidence are put on the state Seen on the nodes and the probabilities for the most probable build order is increased.

Terran versus Terran

This network will try to predict the following buildorders: Proxy Rush, 2 Factory Vulture Pressure, 1 Factory Expand, 1 Starport Wraith and Stim Rush. The numbers in the node names are how many of the given unit the enemy have, e.g. Barracks2 means the second barracks the bot scouts.

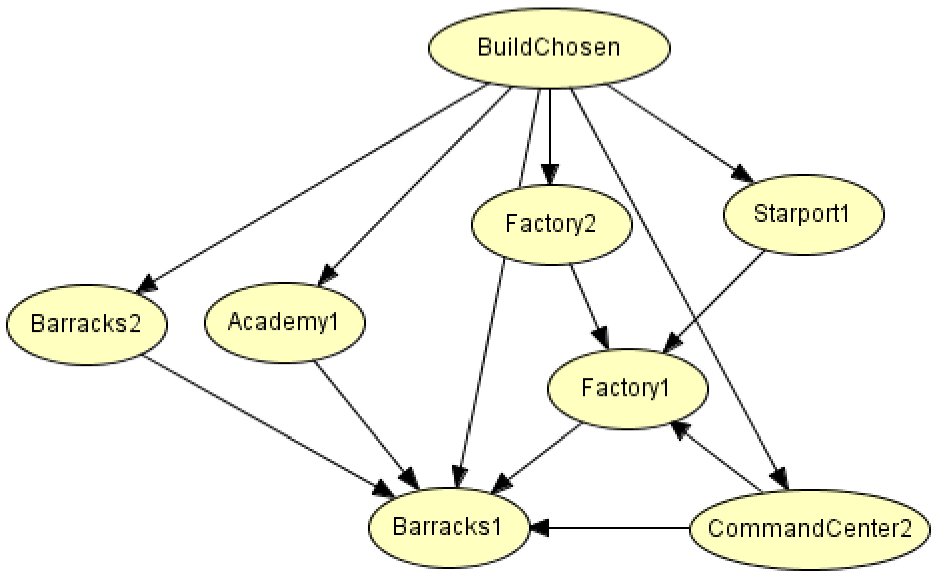


Figure F3-9: Terran versus Terran prediction network

Terran versus Protoss

This network is bigger than the other two prediction networks. Instead of having only having a node for BuildChosen it has a node for the opening used. The reason for this is that Protoss is very versatile race and the different openings affect when the build order hits and what kind of build the player is more likely to use. After the opening is determined all the nodes affecting the openings are not updated any more. The build orders the network can predict is : 2 Base Carrier, 2 Base Carrier and 2 Base Arbiter. The openings the bot can predict is: One Gate Tech, Two Gate Range, Nexus First and Two Gate Zealot Rush.

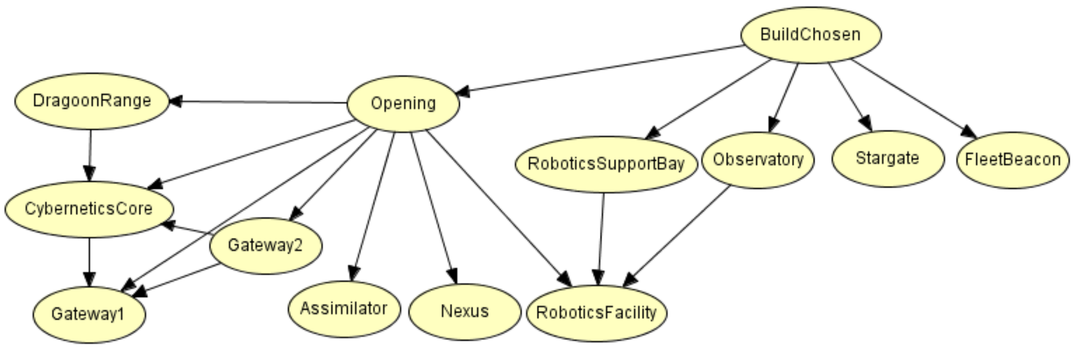


Figure F3-10: Terran versus Protoss prediction network

Terran versus Zerg

This network will try to predict the following buildorders: 2 Hatch Muta, 3 Hatch Muta, 3 Hatch Lurker and Early Pool Rush, where Early Pool Rush is a 4pool or 5pool. The nodes have names as in the Terran network, where a node name ends on how many of that type is scouted. E.g Hatchery3 means that the bot have scouted 3 hatcheries.

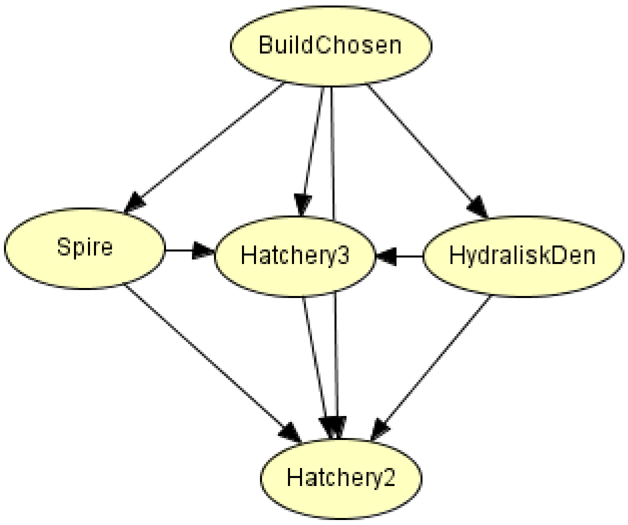


Figure F3-11: Terran versus Zerg prediction network

3.7.3 ThreatLevel Prediction

This prediction is closely related to the prediction of the build order the enemy is doing. A build order have a certain time where it is effective or where it hits. So determine what the current threatlevel is two nodes have to be added to each of the build order prediction networks, ThreatLevel and Time. ThreatLevel have the states Low, Medium and High, and to make the network simple it only has three states for time: 0-5 minutes, 6-8 minutes and 9-11 minutes. By presenting evidence as before and on the time node the current threatlevel can be read.

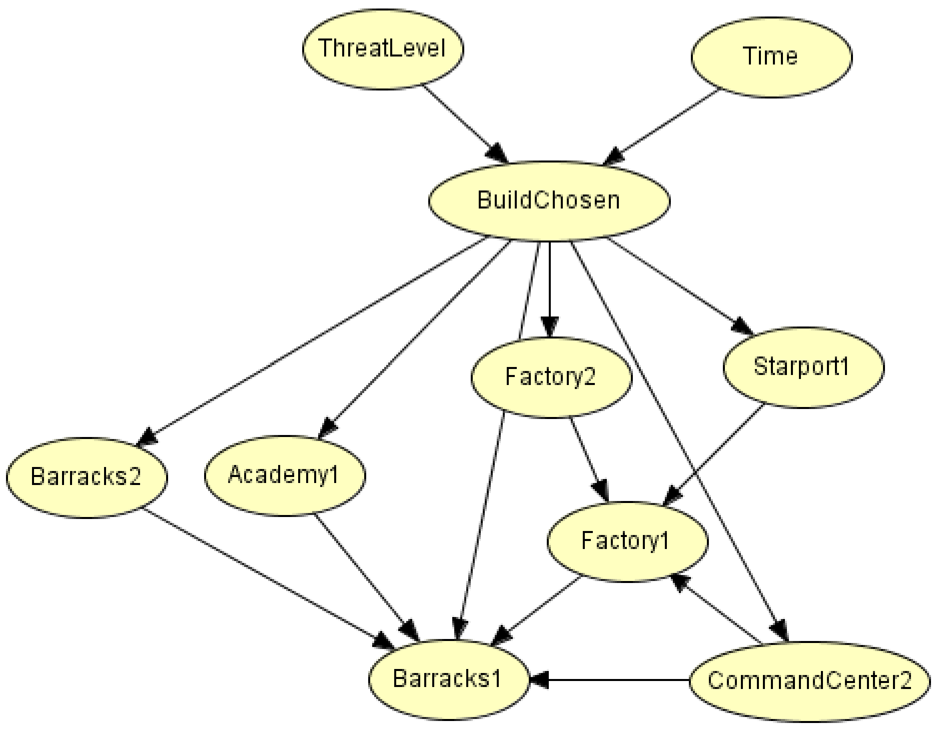


Figure F3-12: ThreatLevel for the Terran versus Terran matchup

3.8 Design Summary

In this chapter we compared the different tools we use in constructing our bot. We separate different parts of the games into different managers so we can easily manage our bot. We then decided to use Potential fields for movement. This is because it would be impossible to come up with a generic algorithm that takes all of the variables in a game into account. To compute the best forces for the different Potential fields we use a generalization of Q-learning. Generalization of Q-learning works well in our case because of an extremely high number of states in a Starcraft Broodwar game. For analyzing information we decided to use Bayesian networks. We choose to come up with networks for enemy spawning positions, enemy build orders, and enemy threat. In the next chapter we will discuss how we implement these concepts.

Chapter 4

Implementation

This chapter will explain implementation of the parts of the bot. This is done by showing code snippets and examples. First we will look at the core of Bayesian Networks then we look at the overall structure of the managers. Next we look at implementation of the different Potential fields. Finally we will look at how we implemented Q-Learning.

To communicate with Starcraft Broodwar we use an API called BWAPI. A link can be found in the bibliography (see reference [9]).

4.1 Class BayesianNetwork

For handling Bayesian networks we use the Hugin API (see reference [10]) which can handle loading networks, inserting evidence, and do other manipulations of .net files. The way to insert evidence into the nodes can be a hard task because many steps are needed. To make it easier for classes to use the networks we wrote a class called BayesianNetwork.

When a class needs a Bayesian network it uses this class. The constructor takes a file name, which then loads the given Bayesian network. The class also contains methods for printing nodes, retracting evidence, and inserting of evidence, and getting the probability of a state. The method for inserting evidence will be explained beneath.

Inserting evidence

```
void BayesianNetwork::EnterEvidence(std::string nodeName,std::string stateName){
domain->uncompile();
NodeList nodes = domain->getNodes();
for (NodeList::const_iterator it = nodes.begin(); it != nodes.end(); ++it)
{
 Node* node = *it;
 if(nodeName == node->getName())
 {
❶  DiscreteChanceNode* evidenceNode = dynamic_cast<DiscreteChanceNode*>(node);
  size_t index = evidenceNode->getStateIndex(stateName);
❷  evidenceNode->selectState(index);
  break;
 }
}
domain->compile();
domain->propagate(H_EQUILIBRIUM_SUM, H_MODE_NORMAL);
}
```

Listing 4.1: EnterEvidence method

The method starts by uncompiling which is needed to manipulate nodes. After this it loops through all the nodes until it finds the specified node. At ❶ the node is converted to a DiscreteChanceNode because evidence cannot be presented to a normal node. The reason for this is that there are many types which inherit from the Node class which does not all use evidence. The index of the wanted state is retrieved and the state is selected at ❷ which is the same at presenting evidence at the state. After the evidence is presented, the domain is then compiled and propagated. The propagate function calculates the new probabilities for the states in the network.

4.2 Managers

A manager is responsible for controlling different sections of the bot. Sometimes they still have to pass tasks to each other. Each manager has a method called *Update* which is called on each frame and is used for running their code.

4.2.1 Production Manager

The production manager is responsible for producing units, constructing buildings, and researching technology. Every time a new production facility or research facility is produced it will be saved here, so we are can use them later. The tasks that the production manager is responsible for are passed from a class called BuildOrder-Handler.

BuildOrderHandler

This class contains the build orders the bot can use and passes the items as tasks for the production manager. A BuildOrder contains a list of BuildOrderItems which every class that inherits from this class can use. The items the build order can store are:

- BuildingItem - Contains a building to be constructed
- ProductionFocusItem - Contains a new Production Focus
- ResearchItem - Contains a research technology that needs to be bought
- UpgradeItem - Contains an upgrade that needs to be bought
- ScoutItem - Sends an SCV to scout
- UnitProductionItem - Contains a unit to be produced

In the constructor each these items conditions will be placed so the item first will get carried out when the conditions are fulfilled. The conditions we have are:

- SupplyCondition - Is fulfilled when the supply is reached
- ThreatLevelCondition - Is fulfilled when the threatlevel is reached
- UnitProductionCondition - Is fulfilled when the number of units or buildings reaches the number specified

The following code snippet shows the main loop of the BuildOrderHandler which goes through the items in the build order and checks if the conditions are all fulfilled.

```
std::list<BuildOrderItem*> items = _currentBuildOrder->
    GetBuildOrderItems();
❶ for(std::list<BuildOrderItem*>::iterator item = items.begin()
    ;item!=items.end();++item)
{
 bool allConditionsFulfilled = true;
 std::list<Condition*> conditions = (*item)->GetConditions();
 for each(Condition* condition in conditions)
 {
  if(!condition->IsFulfilled())
  {
   allConditionsFulfilled = false;
   break;
  }
 }
 //If all the conditions of the item was fulfilled we save it
     to the right list of tasks
 if(allConditionsFulfilled == true)
 {
  SaveAsTask(*item);
  _currentBuildOrder->items.remove(*item);
 }
}
```

Listing 4.2: BuildOrderHandler main loop

This loop grabs the BuildOrderItems in the current buildorder and in the loop at ❶ we check if all the conditions of the item were fulfilled. Each condition object inherits from the class Condition where they inherit a method called IsFulfilled. Each condition that overrides the method has to specify when the given condition will be fulfilled. If all the conditions were fulfilled, we convert the item to a Task and save it so the ProductionManager can retrieve the Task. The difference between a BuildOrderItem and a Task is that the item is stripped of all information except the information that is important in producing the task.

Retrieving and Executing Tasks

The ProductionManager contains methods for retrieving tasks from the BuildOrderHandler. There is a method for each type of task it can produce. When it has retrieved all the tasks it tries to see if it can execute it right away. If we can afford to do the task it will be passed on to the appropriate method for execution. These methods are called TryProduceUnit, TryConstructBuilding, TryUpgradeTech, and TryResearchTech. There is a method for both upgrades and researching because the BWAPI makes a distinction between these two types. The methods all follow the same basic pattern.

- Check if we can afford the task
- Find a building that can execute the task
- If both succeed the task will be executed

The only method that does not follow this pattern is the method TryConstructBuilding. The reason for this is that the ProductionManager is not able to construct a building without an SCV. Because of this we pass this task on to the WorkerManager that will try to construct the building.

4.2.2 WorkerManager

This manager controls the SCVs the bot owns and is responsible for constructing buildings. When an SCV is created it is saved to a list of SCVs and is given a state. An SCV can be in one the following states: Constructing, BeingBuild, MiningGas, MiningMinerals, Defending, Evading, or Nothing. When the SCV is completed from the Command Center it will get the state Nothing. The Update method will then determine what the SCV should do. Usually the SCV will be set to mine minerals, so it gets the state MiningMinerals.

Efficient Mining

To ensure that the bot will get the most minerals in relation to how many SCVs it has, we have implemented a way to mine more efficiently by choosing the right mineral or refinery to mine from. The following code snippet is from the method SendToMineral, which sends a single SCV to a mineral patch.

```
int fewestScvs = 100;
int distanceToMineral = 10000;
BWAPI::Unit* bestMineral;

❶ for(std::map<BWAPI::Unit*,int>::iterator m = _workersOnMineral.begin();m != _workersOnMineral.end();m++)
{
 if((*m).second < fewestScvs)
 {
  fewestScvs = (*m).second;
  bestMineral = (*m).first;
  distanceToMineral=scv->getDistance(bestMineral);
 }
 else if((*m).second ==fewestScvs && scv->getDistance((*m).first)<distanceToMineral)
 {
  fewestScvs = (*m).second;
  bestMineral = (*m).first;
  distanceToMineral=scv->getDistance(bestMineral);
 }
}
❷ _workersOnMineral[bestMineral]++;
_scvResourceGoals[scv]=bestMineral;
scv->rightClick(bestMineral);
```

Listing 4.3: SendToMineral method

Each mineral patch and refinery are mapped to an int which represents how many SCVs are using this patch or refinery. To find the best mineral patch every mineral patch we are mining from is iterated through ❶ and checked if there are less SCVs than the preceding mineral patch. The distance to this mineral patch is also saved because there might be a mineral patch which has as few SCVs on the patch but is closer. When the best mineral patch is found it is noted❷ that an additional SCV is working on that mineral patch- After this we save the mineral to the SCV and sends the SCV to mine the mineral patch by right clicking it.

Constructing Buildings

Below is the code that the Production Manager calls when it has a ConstructionTask that needs to be executed. The method is called ConstructBuilding and has the parameters of type UnitType and BuildingPlacement, where BuildingPlacement is an enum that can have the values MainBase or MainChokepoint.

```
❶ BWAPI::TilePosition position = finder.FindBuildLocation(buildingType,placement);

❷ BWAPI::Unit* scv = GetAvailableScvNearPosition(BWAPI::Position(position.x(),position.y()));


❸ SwitchState(scv,WorkerManager::Constructing);

❹ buildingToConstruct building;
building[position] = buildingType;
_workersOnConstruction[scv] = building;
scv->build(position,buildingType);
```

Listing 4.4: ConstructBuilding method

To find a location to place a building we use an instance of the class Building-PlacementFinder called finder (see ❶). This class can find a suitable location for a building to be constructed in relation to the enumeration specified. After this we find an SCV for the task ❷ by using the method GetAvailableScvNearPosition which gets the nearest SCV that is not carrying minerals, mining gas, or constructing buildings. With the position and an SCV the building can now be constructed. The state of the SCV is changed to Constructing ❸, and we save the building information, so we are able to try again if it fails in constructing the building. ❹ is a type containing a map between a position and a building type. After this the SCV tries to construct the building.

4.2.3 Scouting Manager

The scouting manager is used to find the opponents base and obtain information on what the opponent is doing. It uses a Bayesian network to find the most probable location for the enemy base. At the beginning of the match evidence is instantly inserted on the "OurSpawn" node of the Bayesian network. Whenever an opponent's worker is found, evidence for the worker location and current game time are placed as evidence in the Bayesian network and our scout is sent to the new most probable worker location. Once the scout finally finds the opponent's base, it gathers information on which buildings the opponent has in order to help predict the build order.

4.2.4 Strategy Manager

This manager was supposed to be responsible for making high level decisions for the bot, but we have not finished the manager completely. We have made the necessary analysis needed to make these prediction. By using the class BuildOrderPredictor we are able to retrieve the most probable build order and threat level.

Build Order Predictor Class

This class uses a Bayesian network to analyze what build order the enemy is currently doing. This is related to the current threat level which the strategy manager retrieves.

The class is instantiated by loading the proper prediction network in relation to the match up and saving the match up as a variable. The class have a public method for updating the prediction network which then calls the proper method for updating the correct network related to the current match up. The nodes in the Bayesian network have similar names, so that the conversion from the unit type to the node name can be done more easily. When an enemy unit is scouted the strategy manager passes this to the BuildOrderPredictor which then tries to put evidence on the correct node. Such a conversion can be seen below.

```cpp
if((building == BWAPI::UnitTypes::Terran_Academy ||
  building == BWAPI::UnitTypes::Terran_Starport ||
  building == BWAPI::UnitTypes::Terran_Barracks ||
  building == BWAPI::UnitTypes::Terran_Factory)&&
  enemyBuildingsOwned[building]==1)
{
 //Converts the building type to how the nodes are written
 std::string nodeName = building.getName();
 nodeName.erase(0,7);
 std::remove(nodeName.begin(), nodeName.end(), ' ');
 char buffer [2];
 std::string nodeNumber = itoa(1,buffer,10);
 predictionNetwork.EnterEvidence((nodeName+nodeNumber),"Seen")
    ;
 BWAPI::Broodwar->printf("Updated the prediction network");
 predictionNetwork.PrintMostProbableState("BuildChosen");
}
```

Listing 4.5: Conversion from unit type to node name

The code removes Terran string and the white spaces and saves it to a string. The number of buildings of the type is converted to a string too. Then the evidence is entered and prints the most probable build order.

4.2.5 Tactics Manager

The TacticsManager is responsible for managing the offensive units the bot owns. When units of the same type are near each other they are saved to a Squad object. The TacticsManager executes all the tactics of the Squad objects in it's Update method. This tactic that the squad executes are from the Reinforcement Learning and Potential fields which will be explained in the next sections.

4.3 Implementing Potential Fields

In this section we are going to take a look at the implementation of the Potential field. First we will look at the main loop and the general structure. Then we move on to each part of the Potential field itself.

4.3.1 General Structure

The Potential field is calculated for individual units, these units belong to a squad. When we calculate the Potential field we also pass a reference to the squad called *mySquad* because it is needed to calculate *Squad center* and *Ally Potential*.

```
❶ BaseTactic::InitializeParameters(mySquad);
  Position centerPosition = _unit->getPosition();
❷ std::list<BWAPI::Position> listOfPositions = MathHelper::
      GetSurroundingPositions(centerPosition,48);
  BWAPI::Position bestQPosistion = BWAPI::Position(1,1);
❸ double centerQ = BaseTactic::CalculateQPotentialField(_unit->
      getPosition(),false);
  double higestQfound;
  bool firstCalculation = true;
❹ for each(BWAPI::Position position in listOfPositions)
  {
   double currentQ = BaseTactic::CalculateQPotentialField(
       position,false);
❺  if(firstCalculation)
   {
    firstCalculation = false;
    higestQfound = currentQ;
    bestQPosistion = position;
   }
   if(currentQ > higestQfound)
   {
    higestQfound = currentQ;
    bestQPosistion = position;
   }
   //used to print the Potential around the unit
   Broodwar->drawTextMap(position.x(),position.y(),"%d",(int)
       currentQ);
  }
❻ if(centerQ == higestQfound)
  {
   bestQPosistion = centerPosition;
  }
```

Listing 4.6: Main loop

In ❶ we initialize all the parameters because they are the same for all the tiles:

- int da - Is the distance to closed ally unit.
- int ds - The distance from center of army to unit.
- int sv - The units maximum shooting range. -1 if there is no weapon for this type.
- int sva - The units maximum shooting range for air. -1 if there is no weapon for this type.
- int de - Distance to nearest known enemy.
- bool wr - A boolean denoting whether or not the weapons are ready to fire.
- BWAPI::Position squadPos - The center of the squad.

They are all used in the calculation of the Potential field later in this section.

Next we need to find the tiles we want to calculate the Potential for. This is done using the *MathHelper::GetSurroundingPositions* function in ❷. This function takes two parameters, the tile the unit is currently on and the distance to the center of the adjacent tiles. From this we calculate all tiles around the unit and and them to a list.

Now in ❸ we calculate the Potential for the center tile. This is used in ❻ to compare it to the best Potential found. If they are equal we choose the center tile. This is a rare case and mostly only occurs when all the Potential fields are zero.

In that case we do not want to move. If we did not choose to make *bestQPosition* the *centerPosition* and returned any other tile, the unit would move to this tile.

In ❹ we have the main loop. This loop iterates trough all the tiles around our unit and selects the highest one. There is one special case in ❺ which is the first calculation. In this calculation we always select it as the current best option. This is done so we have something to compare with rather than just set it to an arbitrary tile and number.

In the calculation of the Potential field we use three different constants listed below. These constants are initialized to a parameter we found by trial and error.

- _variables.SQUADDISTANCE_CONSTANT - Is set to 150 px.
- _variables.ALLYDISTANCE_CONSTANT - Is set to 50 px.
- _variables.EDGESDISTANCE_CONSTANT - Is set to 250 px.

An important thing to note before we look at the individual function is a small change from the design. In the design we use negative forces in repulsive fields, but due to the reinforcement learning this has changed. The reinforcement learning will change the forces, and it will be able to make them both positive and negative. In other words the reinforcement learning will be able to change whether or not a Potential field is attractive or repulsive.

4.3.2 Squad Center

```
❶ _parameters.dsv = pos.getApproxDistance(_parameters.squadPos);
  int useSquad = 0;
❷ if(_parameters.ds > _variables.SQUADDISTANCE_CONSTANT)
   useSquad = 1;

❸ squad += ((double)_variables.FORCESQUAD * (2*(double)
      _parameters.ds-(double)_parameters.dsv))*useSquad;
❹ squadQ += (2*(double)_parameters.ds-(double)_parameters.dsv)*
      useSquad;
```

Listing 4.7: Squad center

In ❶ we calculate *dsv* by using the build in function getApproxDistance which returns the distance between two tiles as an integer. *pos* is the current tile we want to calculate the Potential for and *squadPos* is the tile in the center of the squad.

Then in ❷ we check weather or not to use the *Squad center* which depends on the distance from the unit, know as *ds*, to the center of the squad. This is used to handle the two cases found in the design of the Potential field.

Afterwards we calculate the Potential in ❸ where *_variables.FORCESQUAD* is f_S. *_variables.FORCESQUAD* is learned by the reinforcement learning in section 4.4. Unlike in the design we multiply useSquad on the end. This is to make it zero if we do not want to use Potential field which would make the entire calculation become zero. This is way to control that the use of the Potential field is not only used in *Squad center* but also in all the other Potential fields.

squadQ in ❹ and likewise will be explained in the implementation of reinforcement learning in section 4.4. The *Squad center* implemented and running can be seen in figure F4-1.

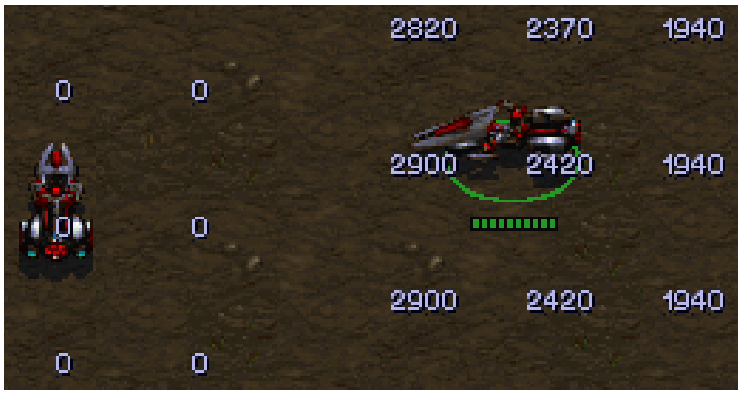


Figure F4-1: Squad center

4.3.3 Maximum Distance Positioning

```
❶ int distanceToEnemyFromUnit = _parameters.de;
❷ _parameters.due = MathHelper::GetDistanceToNearestEnemy(pos);
❸ int correctedDistance = (2*_parameters.de - _parameters.due);
  int useMaxDist = 1;
  if(_parameters.sv > distanceToEnemyFromUnit)
   useMaxDist = 0;

❹ maxdist += ((double)_variables.FORCEMAXDIST * (double)
     correctedDistance)*(double)useMaxDist;
  maxdistQ += (double)(correctedDistance)*(double)useMaxDist;
```

Listing 4.8: Maximum distance

In *Maximum distance positioning* we start by calculating *de* and *due*in ❶ ❷. These are used to calculate $(2 * de - due)$ in ❸, known in the code as *correctedDistance*.

correctedDistance is being calculated even if we do not use *Maximum distance positioning* because it is also used in *Weapon cooldown*.

In ❶ and ❷ we make use of *MathHelper::GetDistanceToNearestEnemy* which is a function that takes a tile as input and then iterates trough all visible enemies. Then it calculates the distance to the nearest enemy and returns it.

The last thing to do is calculate the Potential which is done in ❹. The *Maximum distance positioning* implemented and running can be seen in figure F4-2.

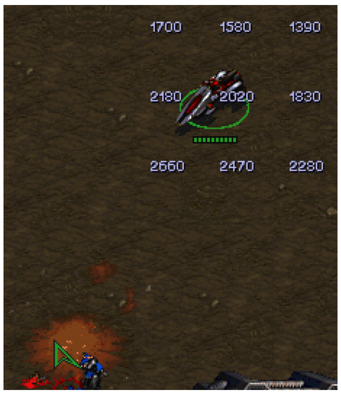


Figure F4-2: Maximum distance positioning

4.3.4 Weapon Cooldown

```
int toCool = 1;
if(_parameters.wr)
 toCool = 0;

cool += _variables.FORCECOOLDOWN*correctedDistance*toCool;
coolQ += correctedDistance*toCool;
```

Listing 4.9: Weapon cooldown

Weapon cooldown looks a lot like *Maximum distance positioning* and does almost the same thing. The only difference is when they are applied. An example of *Weapon cooldown* can be found in figure F4-3.

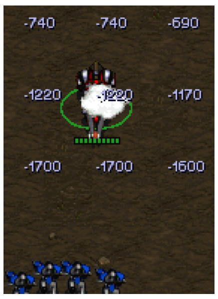


Figure F4-3: Repulsion from Weapons Cooldown

4.3.5 Ally Units

```
❶ _parameters.dua = MathHelper::GetDistanceToNearestAlly(pos,
    _unit->getID());
int useAlly = 0;
if(_parameters.da < _variables.ALLYDISTANCE_CONSTANT)
 useAlly = 1;

ally += (double)_variables.FORCEALLY*(double)(2*_parameters.da
    - _parameters.dua)*useAlly;
allyQ += (double)(2*_parameters.da - _parameters.dua)*useAlly;
```

Listing 4.10: Ally units

In *Ally units* at ❶ we use a function called *MathHelper::GetDistanceToNearestAlly* which takes as input a tile position and the id of the current unit. We need the unit id because the function iterates trough all our units, and we want to avoid calculating the distance to the current unit. If we did not use the unit id we would always get the distance from the current unit to itself because the distance would always be less than the distance to other units. *Ally units* can be seen in figure F4-4.

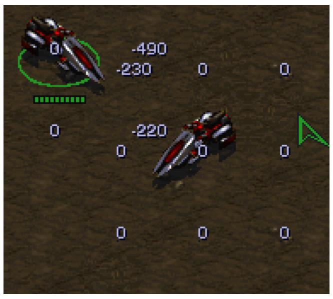


Figure F4-4: Repulsion of ally units

4.3.6 Edges and Cliffs (Repulsive)

```
❶ _parameters.duc= (int)MathHelper::GetDistanceBetweenPositions(
       BWTA::getNearestUnwalkablePosition(pos),pos);
❷ _parameters.dc = (int)MathHelper::GetDistanceBetweenPositions(
       BWTA::getNearestUnwalkablePosition(pos),_unit->getPosition()
       );

int useEdge = 1;
if(_parameters.duc > _variables.EDGESDISTANCE_CONSTANT)
 useEdge = 0;

edge += (_variables.FORCEEDGE)*(2*_parameters.dc-_parameters.
       duc)*useEdge;
edgeQ += (2*_parameters.dc-_parameters.duc)*useEdge;
```

Listing 4.11: Edges and cliffs

In ❶ we use the function *MathHelper::GetDistanceBetweenPositions* to get the distance between the nearest unwalkable position and the current tile. To get the nearest unwalkable position we use the built in function *BWTA::getNearestUnwalkablePosition* which takes as input a position and then find the nearest unwalkable position.

Normally *dc* from ❷ would be calculated in the initialization, but because *BWTA::getNearestUnwalkablePosition* should depend on the current tile we have to recalculate it for each tile. The reason is that for each tile the nearest unwalkable position can differ, and we need to use the same unwalkable position in the calculation of the center tile. If we didn't do this, there could be cases where we took the wrong tiles into consideration. An example could be a corner, where some tile would depend on one side, and other tiles on the other side. The *Edges and cliffs* can be seen in figure F4-5

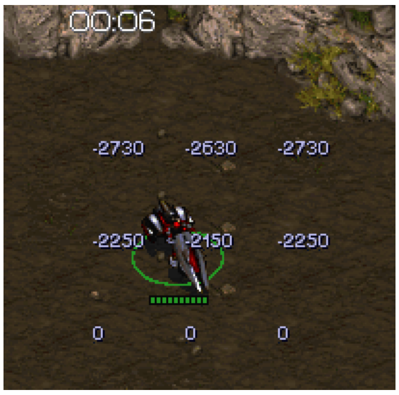


Figure F4-5: Repulsion from cliffs

4.4 $\hat{Q}$-Learning

In this section we will address the implementation of the Reinforcement Learning Approximation. First we will explain the $ReinforcementLearning$ class and the methods that calculate the value of the $\hat{Q}$ function and updating rules. Then we go through all the necessary calculations made throughout the game to create the data needed in order to use this class and save the relevant information for testing.

The $ReinforcementLearning$ class contains all the variables and processes necessary to calculate all the formulas of the $\hat{Q}learning$. It is used throughout the game to update each coefficient of the $\hat{Q}_f$ function, calculate the reward for each state-action pair and saving all the necessary data to continue the calculations in the next iteration of the game (also for saving a data log with all the numbers for further analysis and testing). Further detail of the class implementation can be found in the appendix section 8.7.

4.4.1 $\hat{Q}_f$ Decision

Throughout the game, the squad management uses the $\hat{Q}_f$ method *GetBestPositionBasedOnQPotential* in order to select the next action. It starts with saving the value off all the derivatives ($\frac{\partial \hat{Q}_f(a,s)}{\partial f_i}$). This part is explained in detail in the implementation for every one of the Potential fields.

```
_parameters.dua = MathHelper::GetDistanceToNearestAlly(pos, _unit->getID());
int useAlly = 0;
if(_parameters.da < _variables.ALLYDISTANCE_CONSTANT)
 useAlly = 1;

ally += (double)_variables.FORCEALLY*(double)(2*_parameters.da - _parameters.dua)*useAlly;
❶ allyQ += (double)(2*_parameters.da - _parameters.dua)*useAlly;
```

Listing 4.12: Ally units

Every time a Potential field is calculated we save the values that correspond to the derivative. In the code shown before for the Ally Units Potential the derivative corresponds to the value of *allyQ* in ❶. We do the same thing to calculate the rest of the derivatives (*edgeQ*, *squadQ*, *coolQ*, *maxdistQ*). Finally just saving them for the future update calculations.

```
  StarcraftAI::reinforcementLearning.WriteLiveValue(allyQ);
  StarcraftAI::reinforcementLearning.WriteLiveValue(squadQ);
  StarcraftAI::reinforcementLearning.WriteLiveValue(maxdistQ);
  StarcraftAI::reinforcementLearning.WriteLiveValue(coolQ);
  StarcraftAI::reinforcementLearning.WriteLiveValue(edgeQ);
```

Listing 4.13: Saving the values of each derivative

After saving all the derivatives values for the updating rules, continues with the calculations for the $\hat{Q}_f(a,s)$ as shown before in the Potential fields section 4.3. Finally it calculates the value of $max\hat{Q}_f(a',s')$). The code used for this calculation follows the same pattern explained for the $\hat{Q}_f(a,s)$ function, the only difference being that the center point the previously calculated $\hat{Q}_f(a,s)$.

4.4.2 Reward Function

CalculateReward Method: This method applies the formula explained in section 3.5.1 for calculating a reward. This method is called through several frames in the game to calculate the reward value needed for the update function methods.

```
double ReinforcementLearning::CalculateReward(std::set<BWAPI::Unit*> squad)
{
❶ //__________________________________________________________
 double reward = 0.0;
 double maxEnemieHealth = startingEnemies * startingEnemyMaxHealth;
 double maxUnitHealth = startingUnits * startingUnitMaxHealth;
 double enemyCurrentHealth = 0.0;
 double currentUnitHealth = 0.0;
 double numberOfEnemies = 0.0;
 double squadSize = 0.0;
❷ //__________________________________________________________
 std::set<BWAPI::Player*> enemies = BWAPI::Broodwar->enemies();
 std::set<BWAPI::Unit*> enemieUnits;

 for(std::set<BWAPI::Player*>::const_iterator i = enemies.begin(); i != enemies.end(); i++)
 {
  std::set<BWAPI::Unit*> tempUnits = (*i)->getUnits();
  for(std::set<BWAPI::Unit*>::iterator j = tempUnits.begin(); j != tempUnits.end(); j++)
  {
   enemieUnits.insert((*j));
   numberOfEnemies++;
  }
 }
❸ //__________________________________________________________
 for(std::set<BWAPI::Unit*>::iterator j = squad.begin(); j != squad.end(); j++)
 {
  if((*j)->exists()) squadSize++;
  currentUnitHealth += (double)(*j)->getHitPoints();
 }
❹ //__________________________________________________________
 for(std::set<BWAPI::Unit*>::iterator j = enemieUnits.begin(); j != enemieUnits.end(); j++)
 {
  enemyCurrentHealth += (double)(*j)->getHitPoints();
 }
//__________________________________________________________
❺ reward = c1 * (startingUnits - squadSize) + c2 * (maxUnitHealth-currentUnitHealth) + c3 * (maxEnemieHealth-enemyCurrentHealth) + c4*(startingEnemies-numberOfEnemies)+ c5*(BWAPI::Broodwar->getFrameCount());

❻ return reward/1000;
}
```

Listing 4.14: CalculateReward Method

This method begins with the initialization of all the values needed for the reward formula (contained in section ❶). The values use some of the already indicated constants in the class fields (view the Field Summary here 8.7).

In section ❷, we obtain the array of all the enemy players (*enemies*) and create an array of saving all the enemy units (*enemyUnits*). Then we iterate through the enemy players and save each of their units in the *enemyUnits* array also counting the *numberOfEnemies*.

In section ❸, we iterate through the *squad* (received as a parameter) to count the $squadSize$ and accumulate the total $currentUnitHealth$. Similarly, in section ❹, we iterate through the $enemieUnits$ to accumulate the total $currentEnemyHealth$.

Then we use all the previous calculated values to evaluate the reward function in ❹. Note that we already indicated in the Field Summary which constant value we used for each coefficient. We choose those specific values because they seem to reflect a reward/punishment value correspondent to what we want our agent to learn. Finally, also note that we use this reward value scaled, or reduced (as observed in ❺), to impact the rate of change of the updating formulas and reduce it slightly.

4.4.3 Updating Rules

CalculateTheta Method: This method simply applies the formula explained before in $\hat{Q}$ learning (see reference 3.5.1). It corresponds to the updating rule for Q-learning function approximation considering temporal difference.

```
double ReinforcementLearning::CalculateTheta(double theta, double reward,double currQ, double nextQ, double derivative)
{
❶ double newtheta = theta + alpha * (reward + gamma * nextQ - currQ) * derivative;
 return newtheta;
}
```

Listing 4.15: CalculateTheta Method

In section 3.5.1 there are several different formulas for updating each coefficient in the $\hat{Q}_f$ function, but that distinction between updating rules is external to the $ReinforcementLearning$ class. Therefore, the formula directly translated in ❶ is the general updating rule. The specific $\hat{Q}$ value, current coefficient and the rest of the parameters are calculated separately because we need them in every frame throughout the game.

Calling the Updating Rules

We choose to calculate new weights or coefficients for the $\hat{Q}_f$ function every n frames. Trying to update for every single one created a lag in the performance of the game. The optimization we created for updating while still preserving the game speed is the following:

```
❶ if(BWAPI::Broodwar->getFrameCount() % 25 == 0){
❷   \\_________________________________________
    double* liveBufferPointer = StarcraftAI::
        reinforcementLearning.GetLiveBuffer();
    int liveCount = StarcraftAI::reinforcementLearning.
        GetLiveCount();
    StarcraftAI::reinforcementLearning.ClearLiveBuffer();
    \\_________________________________________
    double edge, cool, mde, squad, ally, currQ, nextQ, reward;

❸   for(int i=0; i<liveCount; i++){
     switch(i%8)
      {
      case 0:
       ally = liveBufferPointer[i];
       break;
      case 1:
       squad = liveBufferPointer[i];
       break;
      case 2:
       mde = liveBufferPointer[i];;
       break;
      case 3:
       cool = liveBufferPointer[i];
       break;
      case 4:
       edge = liveBufferPointer[i];
       break;
      case 5:
       currQ = liveBufferPointer[i];
       break;
      case 6:
       nextQ = liveBufferPointer[i];
       break;
      case 7:
       reward = liveBufferPointer[i];
❹      \\_____________________________________
       _thetas.edge = ReinforcementLearning::CalculateTheta(
           _thetas.edge,reward,currQ,nextQ,edge);
       _thetas.cool = ReinforcementLearning::CalculateTheta(
           _thetas.cool,reward,currQ,nextQ,cool);
       _thetas.mde = ReinforcementLearning::CalculateTheta(
           _thetas.mde,reward,currQ,nextQ,mde);
       _thetas.squad = ReinforcementLearning::CalculateTheta(
           _thetas.squad,reward,currQ,nextQ,squad);
       _thetas.ally = ReinforcementLearning::CalculateTheta(
           _thetas.ally,reward,currQ,nextQ,ally);
       \\_____________________________________
       break;
      default:
       break;
     }
    }
  }
```

Listing 4.16: Calculations for the Updating Function Values

In the first place, we make these calculations every 25 frames as shown in ❶, trying to simulate updating once a second. Then we use some of the methods in the $ReinforcementLearning$ class (indicated in section ❷) to call all the values that were previously saved in the data buffer. This values includes all the above explained derivatives and $\hat{Q}_f$ values. We separate the values, as shown in ❸ and then save them in variables for parameter passing to the updating function. Finally in section ❹ we use the $ReinforcementLearning :: CalculateTheta$ method that corresponds to each of the weights and coefficients of the new $\hat{Q}_f$.

Chapter 5

Tests

In this chapter we will look at a few different tests. First we will test the built in AI against itself, so we have a base to compare our bot against. Then we will make a test using the potential field without any learning and a test with the same number for all the forces. Afterwords we will use reinforcement learning to try to improve our results. We will try and change the alpha and gamma to change the way our bot learns to see the difference. Afterwords we will test the Bayesian networks to see how well they perform.

5.1 Test Without Reinforcement Learning and Potential Fields

In this test the two forces are just attacking each other without any use of potential fields or reinforcement learning. This test is without any micromanagement control. This means the vultures are losing in both maps. They should lose so we can prove that if a unit is being controlled correctly victory is possible. Below is a list of the results from the test in the two maps.

After this first test we know how weak the player with the vultures is compared to the opponent

5.2 Test Using the Potential Fields Without Reinforcement Learning

This is the first test where our bot uses only potential fields to fight the built-in bot. There is no reinforcement learning used in this test. In other words this means that the reinforcement learning weights are all set to a constant value.

Our bot got this impressive result by not overextending itself and thereby only be in sight of a few Zerglings. This forced some Zerglings out of the group and made them an easy target. This test went better than expected because it took out a few Zerglings at a time and won the match with 5 vultures with full health.

Test results from first map

Players	Produced units	Killed units	Lost units
Player with vultures	5	9	5
Player with Zerglings	30	5	5

Test results from second map

Players	Produced units	Killed units	Lost units
Player with vultures	5	5	5
Player with marines	20	5	5

Test results from second map

Players	Produced units	Killed units	Lost units
Player with vultures	5	30	0
Player with Zerglings	30	0	30

Test results from second map

Players	Produced units	Killed units	Lost units
Player with vultures	5	6	5
Player with marines	20	5	6

The result from this test wasn't as good as one would expect. The result of the vultures moving back and forth should be way better than static movement. But the fact is that the vultures only killed one more marine. The reason why this is the case is that the vultures couldn't use the same attack pattern as effectively as with the Zerglings because the marines range attack sometimes dealt damage to the vultures. So the vultures get to close before attacking. We see from this test that potential fields in itself are not enough. If we would like to win or have less of a loss than the opponent. In the next test we will test if there is an improvement by using reinforcement learning on the bot. Since the vultures did well against Zerglings the tests will be changed to only be vultures against 12 marines. If they should learn with more marines the vultures will learn that it's okay to run away and only attack when the cooldown is 0, and this is not the right way to defeat the opponent.

5.3 Comparing α and γ Values

This section is about the tests with the learning rate of the agent. The more thorough explanation of the generalization can be read in section 3.5. By changing the γ and α values the agent will learn differently, so we have made a test where we run between 15000 to several hundred thousand iterations. All the test have been run with 5 vultures against 12 marines. The graphs from the test can be seen in full size in the appendices chapter 8. We will make tests over how much damage the vultures have dealt, how many units we have lost, and how many units we have killed. As mentioned before these test will be performed with different learning values. It is worth mentioning that 5 vultures against 12 marines is a very difficult battle for the vultures, and the marines are favoured to win.

Learning Rate Test 1.1

In this test we are using the following values in the table T5-1 for the reinforcement learning. Iterations occur every time the agent has played a game on our test map.

Alpha value (α)	Gamma value (γ)	Iterations
0.9	0.2	40182

Table T5-1: Alpha and Gamma values for the learning algorithm

The figures shown in the test can be watched full sized in the appendix chapter 8.

In figure F5-2 the blue graph is how many marines were killed. When it peaks to the 12 mark our agent has won a battle.

In figure F5-1 one can see every time the y-axis peaks (blue graph) the agent has killed all of the opponents like we had hoped for. The red graph is damage taken. If it's on 400 all the 5 vultures have died, and the agent has had a lost.

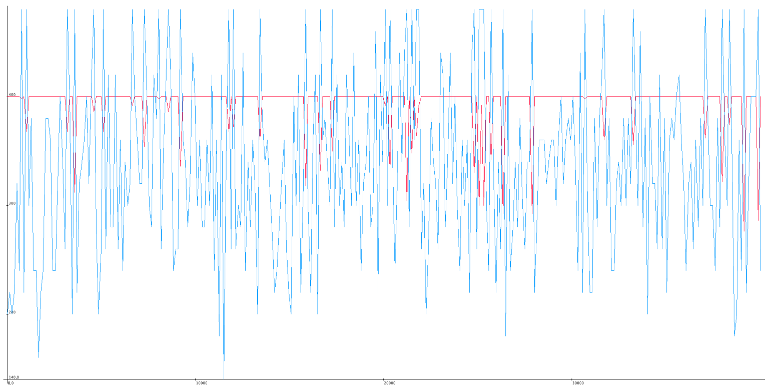


Figure F5-1: Alpha 0.9 Gamma 0.2 Damage - Blue: Damage given - Red: Damage taken (Can be seen fullsized here F8-2)

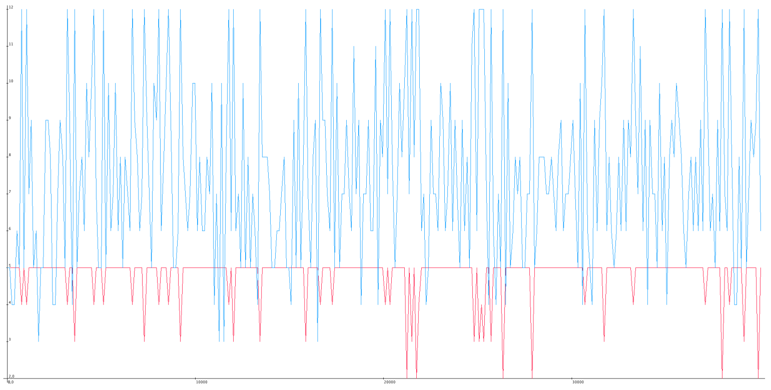


Figure F5-2: Alpha 0.9 Gamma 0.2 Lost and killed - Blue: Enemies killed - Red: Vultures left (Can be seen fullsized here F8-3)

Average results from test 1.1

Damage taken	Damage given	Units lost	Enemies killed
392,71	337,26	4,79	7,51

Table T5-2: Average numbers of A9G2

Comparing the Potential field data with winning streaks could tell us how the agent reacts when it wins. Damage taken and damage given are self explanatory, but the rest are all values of the Potential fields. Ally is how far away each of our own vultures can be from each other. A negative value means repulsive and a positive value means attractive. The squad value is similar to the ally value but the squad means the center point in the group of units. The maximum distance is the distance to an enemy. The value is only for the Potential fields and not real distance to the enemies. Cooldown is when the vultures have fired. They use the cooldown value if they should be attracted to the enemy or not. Finally the edges are how attractive or repulsive the vultures are to the edges. Just like before, negative values mean repulsive and positive values mean attractive.

Average results from test 1.2

Damage taken	Damage given	Units lost	Enemies killed
392,35	334,26	4,79	7,44

Table T5-4: Average numbers of A6G4

Learning Rate Test 1.2

In this test we are using the following values in the table T5-4. Here we try with different values in the learning algorithm to see which ones would fit the best and which values make the agent learn or converge the fastest.

Alpha value (α)	Gamma value (γ)	Iterations
0.6	0.4	17992

Table T5-3: Alpha and Gamma values for the learning algorithm

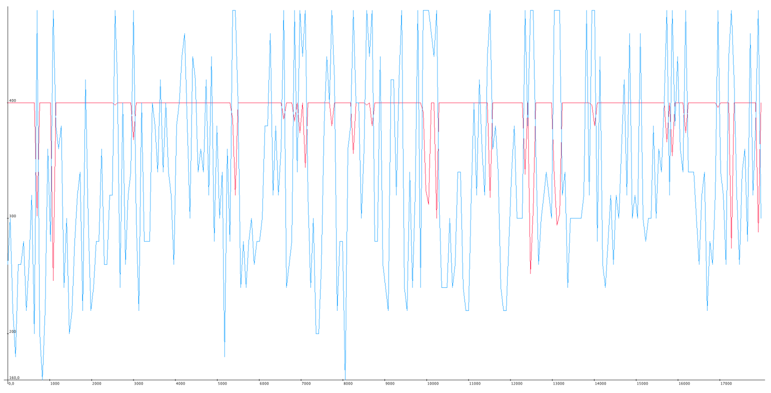


Figure F5-3: Alpha 0.6 gamma 0.4 Damage - Blue: Damage given - Red: Damage taken(Can be seen fullsized here F8-4)

In figure F5-3 just as before but with different α and γ values the agents graph over it's performance.

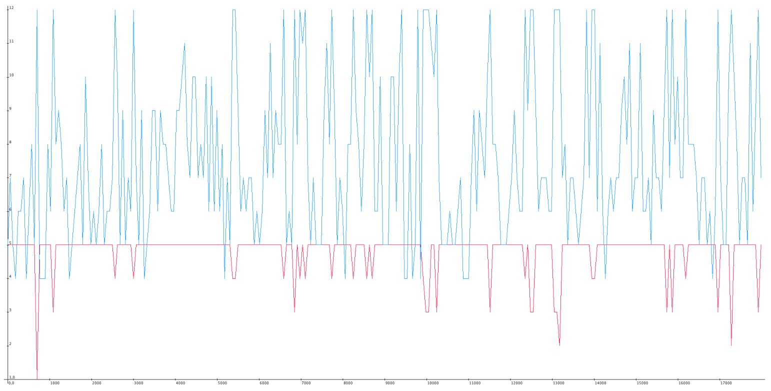


Figure F5-4: Alpha 0.6 Gamma 0.4 Lost and killed - Blue: Vultures left - Red: Marines killed(Can be seen fullsized here F8-5)

Comparing the tests T5-2 and T5-4 one can see that the agent needs several more iterations than it has right now. Since it's not converging from just 40000 iterations. The next tests will be with around 100000 iterations. After that amount of time we can be more certain that it's near converging. The average values are

Average results from test 1.3

Damage taken	Damage given	Units lost	Enemies killed
395,51	338,29	4,86	7,52

almost the same which tells us that either it needs way more iterations or the small change in the values does not give a huge difference.

Learning Rate Test 1.3

In this test we are using the following values in the table T5-5.

Alpha value (α)	Gamma value (γ)	Iterations
0.4	0.6	135936

Table T5-5: Alpha and Gamma values for the learning algorithm

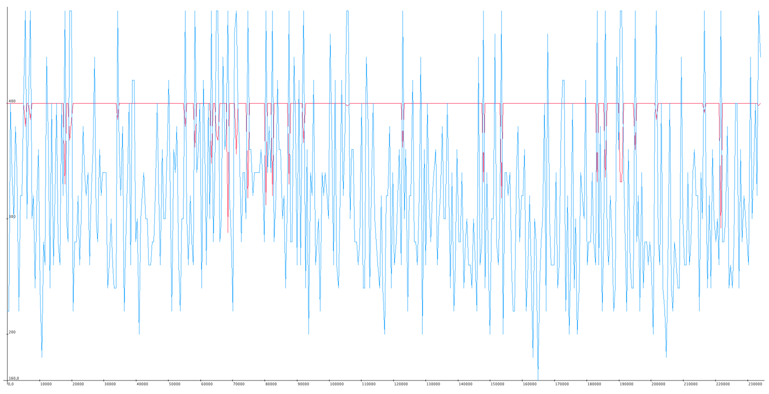


Figure F5-5: Alpha 0.4 gamma 0.6 Damage - Blue: Damage given - Red: Damage taken(Can be seen fullsized here F8-6)

Figure F5-5 is just as before but with different α and γ values the agents graph over it's performance.

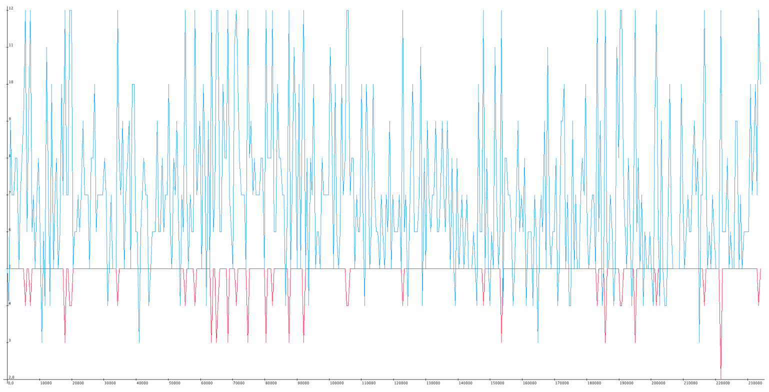


Figure F5-6: Alpha 0.4 Gamma 0.6 Lost and killed - Blue: Vultures left - Red: Marines killed(Can be seen fullsized here F8-7)

The table in the appendix T8-1 shows some numbers when the agent have had a winning streak. All the values get saved after every iteration, and the last 5 values are the potential fields *[Ally, Squad, Maximum distance, Cooldown, Edge]* and the first 2 values shows *[Damage Dealt, and Damage Taken]*. One can see that the damage given are very high and the ones with 480 in damage given is when the vultures have won over the 12 marines. The values are taken from iterations 214385 to 214392. Iterations are saved every time there has been a won, draw or loss. By looking at the numbers in ally and squad we can see that the units like to stick to each other, in other words they like to team up. The reinforcement learning have changed the values and it found out that sticking together is better than dealing with a bunch of marines on it's own. The maximum distance, which means the attractive level towards the marines, are very high in a negative value which makes them very attracted towards the marines and in other words very aggressive. The cooldown is a high positive value which means that when the vultures have fired their weapons they use the potential field of cooldown, which means they get repulsed by the enemies when they are in a cooldown.

The table in the appendix T8-2 also shows a winning streak but this time is in the later stages this is a portion from iterations 260284 to 260291 to compare the numbers with the early stages in winning streak tests T8-1, than the later stages here in test T8-2. One can see that the numbers differs in many ways, ally is higher than it was in the previous test, but then the squad values doesn't differ that much from the early stages T8-1. The maximum distance is attracted towards the enemy which means they have an aggressive behaviour, and by looking at the values of the edges, the vultures are attracted to the cliffs and edges. The cooldown value is a positive number as it's supposed to be, because the vultures should attack and then withdraw until the cooldown has settled.

Learning Rate Test 1.4

In this test we are using the following values from the table T5-6.

Alpha value (α)	Gamma value (γ)	Iterations
0.2	0.9	30852

Table T5-6: Alpha and Gamma values for the learning algorithm

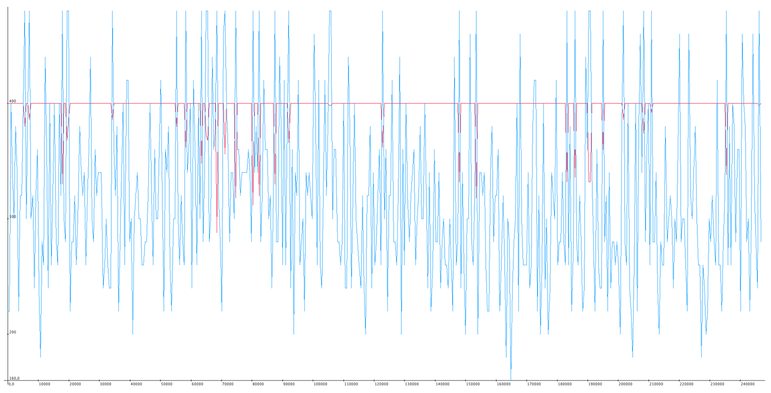


Figure F5-7: Alpha 2 gamma 9 Damage - Blue: Damage given - Red: Damage taken(Can be seen fullsized here F8-8)

In figure F5-7 just as before but with different α and γ values the agents graph over it's performance.

Comparing tests 1.4 and 1.3 where the 1.3 test has run 135936 iterations one can clearly see that the average damage dealt is higher than from the test 1.4 where

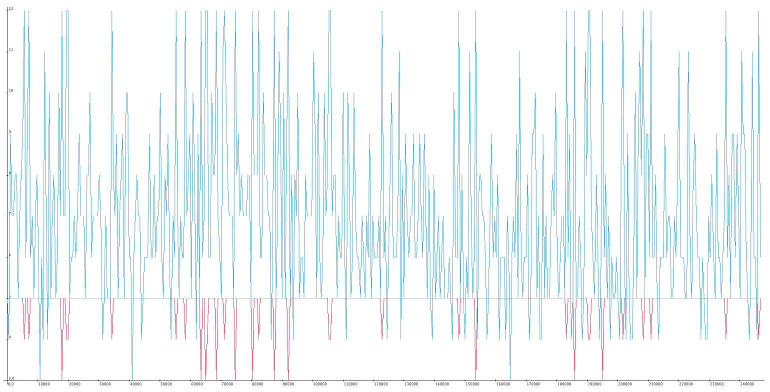


Figure F5-8: Alpha 2 Gamma 9 Lost and killed - Blue: Vultures left - Red: Marines killed(Can be seen fullsized here F8-9)

Average results from test 1.4

Damage taken	Damage given	Units lost	Enemies killed
398,78	282,28	4,96	6,31

Table T5-7: Average numbers from A2G9

the iterations are 30852 times. To make the agent converge is has to run even more iterations. If we cut the test 1.3 down to 30852 iterations and compare the numbers again it has an average of *Damage taken: 396,05 - Damage given: 324,81* which is closely to the tests with few iterations.

The table T8-3 shows some numbers when the agent have had a winning streak. One can see that the damage given are very high and the ones with 480 in damage given is when the vultures have won over the 12 marines. The values have been saved from the agent in the iterations 208280 to 208288. By looking at the numbers in ally and squad we can see that the units like to stick to each other, in other words they like to team up. The reinforcement learning have changed the values and it have found out that sticking together is better than dealing with a bunch of marines on its own. The maximum distance, which means the attractive level towards the marines, are very high in a negative value which makes them very attracted towards the marines and in other words very aggressive.

The table T8-4 also shows a winning streak but this time is in the early stages this is only a cut out from 40081 to 40086 iterations to compare the numbers with the later stages in winning streak tests T8-3, than the early stages here in test T8-4. One can see that the numbers differs in many ways, ally is not that high as it was in the previous test, but then the squad values are high but still not as high as from the other test T8-1. The maximum distance is repulsed from the enemy, and by looking at the values of the edges the vultures are attracted to the cliffs and edges and repulsed by the marines. The cooldown is not that interesting to talk about it has the same drop in value as the other values.

5.4 Convergence Analysis

As mentioned before, we considered a lower α value and a higher γ value would deliver better results in the training for the Reinforcement learning agent. Therefore we prioritized the training of two particular combinations of them: $\alpha = 0.4$ - $\gamma = 0.6$ and $\alpha = 0.2$ - $\gamma = 0.9$. We ran a couple hundred thousand game iterations in each one of these training values. The results obtained from this training show the different trends the weights follow throughout the entire learning process. We considered it important to analyze this behaviour to show the training results for our agent.

Convergence for $\alpha = 0.2$ and $\gamma = 0.9$

By convergence we mean that the agent has found the perfect values that work every time. In other words, the agent learns to get the highest reward. In figure F8-11 one can clearly see that the agent tries with different values and after the almost 250.000 iterations it's beginning to narrow them down to a more stable line instead of trying higher values. The blue line which says *edge* has a drop way below zero, and that might be a buffer overflow, but if we only look at the *ally* and *squad* we can see that the agent learned that sticking together in a group is better than attacking marines individually. The *cooldown* is beginning to have a more stable line which means that the agent learned that after firing, its good to flee to avoid getting injured. The *maximum distance* is not as high as we expected, since the higher it is the more aggressive the vultures are. Even more iterations could perfect these numbers to a convergence and the vultures would then win every game.

Convergence for $\alpha = 0.4$ and $\gamma = 0.6$

As shown in figure F8-10, the weights followed a consistent learning pattern throughout most of the iterations. The first thing the agent learns is that maintaining the squad tightly together (high values of Ally and Squad weights), running from the enemy (low MaxDist weight) and running towards the edges (high Edge weight) delivers the best result (higher number of kills and less damage). But, since we made our reward function punish the agent for taking a long time to destroy the enemy, the weights always adapt trying to make the units more aggressive (it can be observed in the changing values of the MaxDist weight).

It goes back and forth between running and attacking up front until it finally stabilizes in very high values of Ally and Squad (staying together), almost identical values of Cooldown and MaxDist (attack with the same strength that you run away when the weapons are down) and not having a very strong opinion about the Edge (this weight is the closest to 0 and varies around it). We definitely need to keep running training sessions for this values to converge fully but we believe the agent is reaching a consistent point. It manages to win or kill a high number of marines periodically.

5.5 Spawn Prediction Test

In these test we wanted to make sure our spawn predicting Bayesian network had the Potential to accurately predict the enemy's spawn.

We tested our bot first against itself. This was useful because we could check two sets of data for each game ran. By default without encountering an opponent's scout our SCV would go in order to bases NE, SE, SW, NW. Our hopes were that our scout would change direction upon encountering the opponent's scout. In our test we observed that our bot's belief on enemy base position would change when encounter the opponent scout. However, the new probabilities did not effect the bot's belief enough to change the current path. This is due to the fact that we manually entered the probability tables for the Bayesian network without knowing too well what the actual probabilities were. To get the bot to better predict the enemy spawn we would have to change the probabilities probably through machine learning.

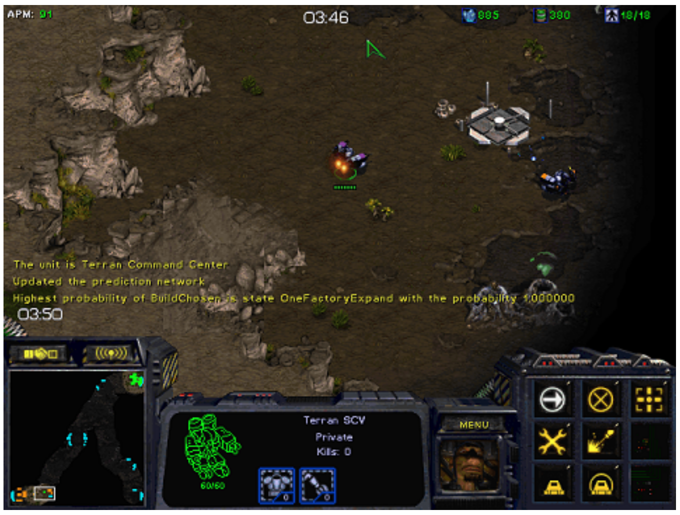
We also did some test against our bot when a human played as the other player. This way we could manipulate the time and position that our bot would encounter the opponent scout. This test had similar outcomes as when our bot played against itself. This further supports our belief that this network has Potential even if it does not current function perfectly.

5.6 Build Order Prediction Test

These test are meant to show the ability of our Build Order Bayesian network to successfully predict an opponent's build order. During these tests we had a human player play against our bot. The human player would play using a specific build order that the bot should be able to predict. Once the bot scouted, it would use the information obtained from the buildings seen and print out the build order. We keep a list of all of the enemy buildings we have seen. Once we see a new building we add to our list and update our Bayesian network.

The bot did successfully predict the build orders. When the bot's SCV first scouts and only sees a few of the buildings to support a particular build order, it will increase the probability for some of the builds. For example, when the scout was inside the enemy base, the probability that the build was a one factory expand build and a two factory pressure build both increased to about 50% with the one factory expand being slightly favoured. Once the scout was able to see the second command center the probability of a one factory expand increased to 100% as shown in F5-9.

The build order predictor works fairly well, but still has some needed improvements. First off there could be a build order we have not included in the Bayesian network. If the bot encountered an unknown build it would treat it as one of the builds in the network and may not act in a responsible way. Another problem is if we do not collect any information on the enemy at all. The bot's scout could get killed before getting any information on the enemy base. This would give us the belief that there is an equal chance for all of the different builds so the bot may not act correctly.

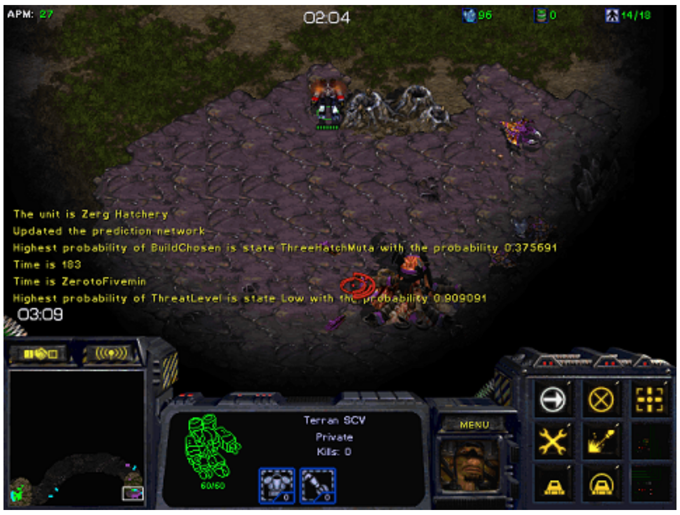


```
The unit is Terran Command Center
Updated the prediction network
Highest probability of BuildChosen is state OneFactoryExpand with the probability 1.000000
```

Figure F5-9: Predicting one fact expand build upon seeing second command center

5.6.1 Prediction of Threatlevel test

In this test we wanted to see if the prediction network could analyze what he current threatlevel is based on what it sees and what the time is. The bot scouts the opponent and sees 2 hatcheries, which it predicts is going to be a 3 Hatch Muta build. Because that build is effective later in the game the network predicts the current threatlevel to low. In the picture below (See F5-9) this situation is shown. This test just showed that the network worked exactly as assumed.



```
The unit is Zerg Hatchery
Updated the prediction network
Highest probability of BuildChosen is state ThreeHatchMuta with the probability 0.375691
Time is 183
Time is ZerotoFivemin
Highest probability of ThreatLevel is state Low with the probability 0.909091
```

Figure F5-10: Predicting the current threatlevel

Chapter 6

Conclusion

In this chapter of the report we summarize all the work done, and in what measure our goals were fulfilled.

For this project we wanted to create an intelligent bot that can play Starcraft Broodwar. With this idea in mind, we constructed two different intelligent characteristics. The first one is the making of decisions during a real-time combat. It takes this decisions using reinforcement learning Q approximation and potential fields. The second one is the prediction of build orders and analysis of threat level. We accomplish this task by using Bayesian networks.

Addressing the Reinforcement learning tests. In the results for bot with different alpha and gamma values, we can conclude that a lower alpha value, and a higher gamma value is the best for learning fastest in our environment. We have noticed that it takes several hundred thousand of iterations before it converges. It has learned how to play against the marines and has several wins after playing against the marines for playing about 250.000 iterations.

The Bayesian networks that are implemented are very simple, which means that the bot will only be able to recognize a small set of build orders. Even if these networks were made more complex, there will always be build orders which will not get recognized. So enabling the bot to recognize the most known build orders seems sufficient. The Reinforcement learning and Potential fields merge preformed as expected. The bot can control the units more efficiently than a human player, and it learns continuously to better achieve the goal of killing all the enemy units.

Another of our goals was to effectively apply Machine Intelligence theories in the modelling of our bot, which goes hand in hand with the third goal we defined in the problem statement, create a bot that improves by playing Starcraft Broodwar. This goal was achieved with the Reinforcement learning merge with Potential fields. We started defining the potential fields' vectors for modelling the movement of our units and then using this completed product to create the Q-function of our Reinforcement Learning. By doing this merge we managed to create a simple, linear, and comprehensive function that models the combat environment in Starcraft Broodwar. With this function the Reinforcement learning algorithm had all the elements needed to learn and improve. After several training sessions our bot manages to increase the number of kills and almost converge to the true values for the Q-function and Potential.

The Bayesian networks predictions are not improved by playing. All the probabilities for the networks are obtained from replays with professional Starcraft players. The numbers might be unrealistic, but as we showed in the test of these networks, it was able to predict the right build order and threat level. But it would definitely be improved using data mining from several match replays.

We also described that we wanted to make a full bot, completely able to play a normal game against another player. And in theory, the bot is able to do this. But for it to preform well, better techniques and more precise information is needed in the Strategy Manager and the placing of buildings. The Strategy Manager can not control the other managers as we intended completely. The Reinforcement learning and potential fields only work well for the training maps right now and not for the normal maps, because it has not been trained to move in environments where there are buildings and other units that can block the path. Nevertheless, if the managers controlled the game enough to only activate the reinforcement learning and potential fields when needed, the bot could be used in real games.

We were successfully able to use Machine Intelligence to create a bot in Starcraft Broodwar, that fulfilled the goals we created for it.

Chapter 7

Future work

7.1 Future Work on Potential Fields

If we had more time to work on the Potential fields they could use a lot of adjustments. They work well for our test case against the marines, but when used in most normal maps during a real game they need to take a lot more things into account. An example would be buildings, right now they are not mapped in the Potential field, so the units will not take them into account only choose not to move directly on top of them due the the tile being invalid. The same goes for mineral patches and gas geysers.

We could look into adding more Potential field for all kind of scenarios, for low health etc. This should make the bot even more cleaver but it would also mean more variables to learn and we would need to relearn all the other variables. The low health vultures should be repaired by a SVN, which mean that a SVN should stop mining

Another thing to look at would be using a different set of forces for different scenarios, like defending the base and attacking other bases. And learning different numbers for different kind of units. There should not be anything in the way of using the Potential field on other kind of units.

We could also use more time adjusting the constants we filled in our selves[1], to try and see what works best.

As we later found out the vultures have a movement speed upgrade, this could be interesting to test the effect the upgrade would have on the variables in reinforcement learn.

7.2 Future Work on Reinforcement Learning

If we had more time to work on this project there are quite a few thing we would like to try out. First of all we would like to consider different rewards function, to see how that would impact the learning. Right now we only consider the same five variables in the reward function, mainly the total health of the squad and squad sizes. But we would considerer the percentage of health of each unit, to make it try to distribute the damage more. Or just something as simple as trying with our current reward function but with different numbers.

In the Potential fields we only consider distances, and all calculations are done for one unit at a time. We would look at during the calculations for the entire group at ones. To make them do what is collective best for the group. And instead of just considering distances we would consider the health of a unit, so that units on low

[1]Size of the squad, minimum distance to ally and maximum distance to the edges

health will be more defence then units with full health.

Another thing to look at would be the enemies race or even the type of units we fight. Right now we use the same forces no matter who or what we fight. But it would most likely preform better if we learned different set of numbers of each different kind of unit.

The last thing we would like to do is keep the bot running until all the forces has converged.

7.3 Future Work on Prediction Networks

The project we use Bayesian networks for prediction of the enemy base position, the enemy's buildorder and the current threatlevel. It is hard to make the prediction of the enemy base position, because not a lot of data can be analyzed in the first minutes in the game. The SCV have to go from base to base until it finds the opponent's base and there is no way to improve this, because the bot have fog of war. The Bayesian networks is made for predicting the buildorder of the enemy, yet it is very simple and can only predict a few buildorders, so if the enemy is using a build that is not in our network it can not predict it. Both these networks could be better at predicting by using data mining. If the bot could train the networks by watching replays or by using the data from a replay. If we had more time we would make the buildorder prediction more precise using a node for early game, mid game and late game. That way it will be a better identification of the buildorder and of the current threatlevel.
We could have liked to use Bayesian network for is predicting when the opponent is going to attack. This would be based on what infomation the bot gets from scouting, where the must importent node in the Bayesian network would be the build order and the units the opponent have.

Chapter 8

Appendices

8.1 Picture of a Normal Game

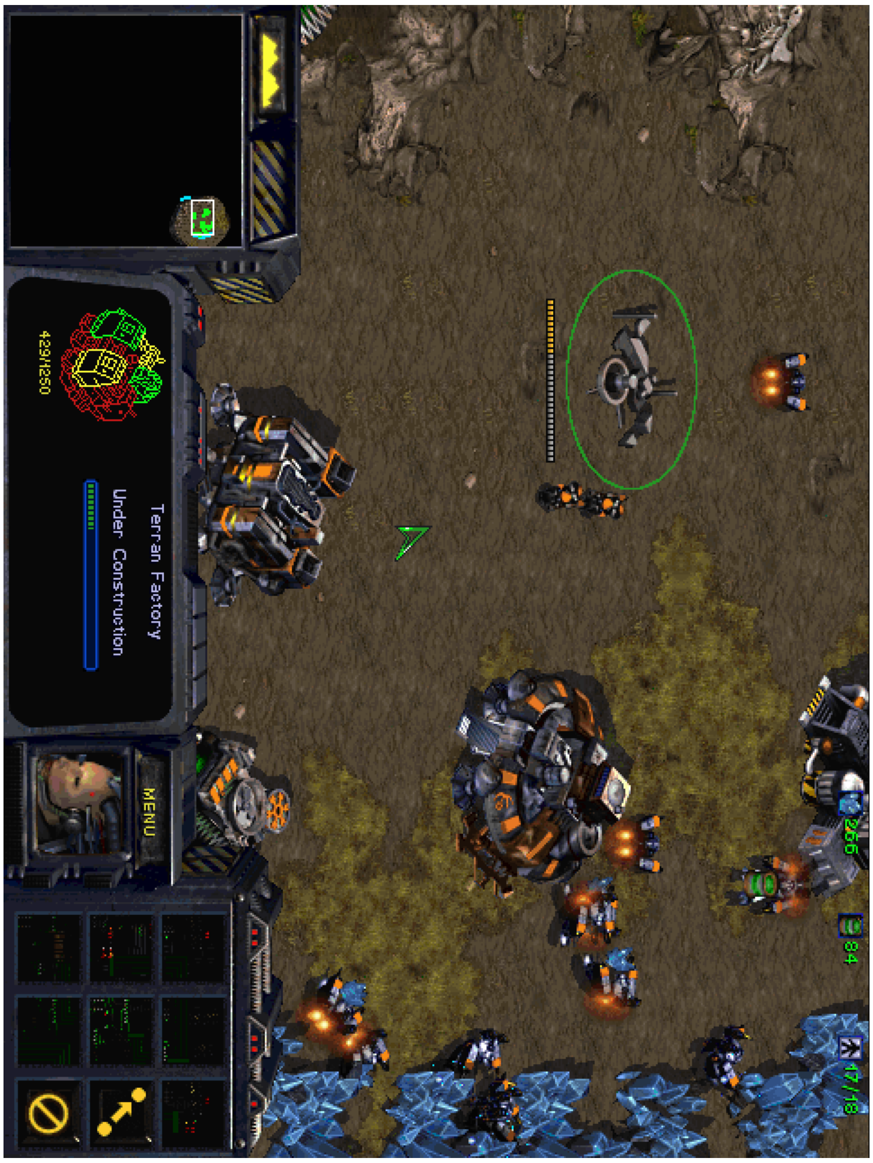


Figure F8-1: A normal Game

8.2 A9G2

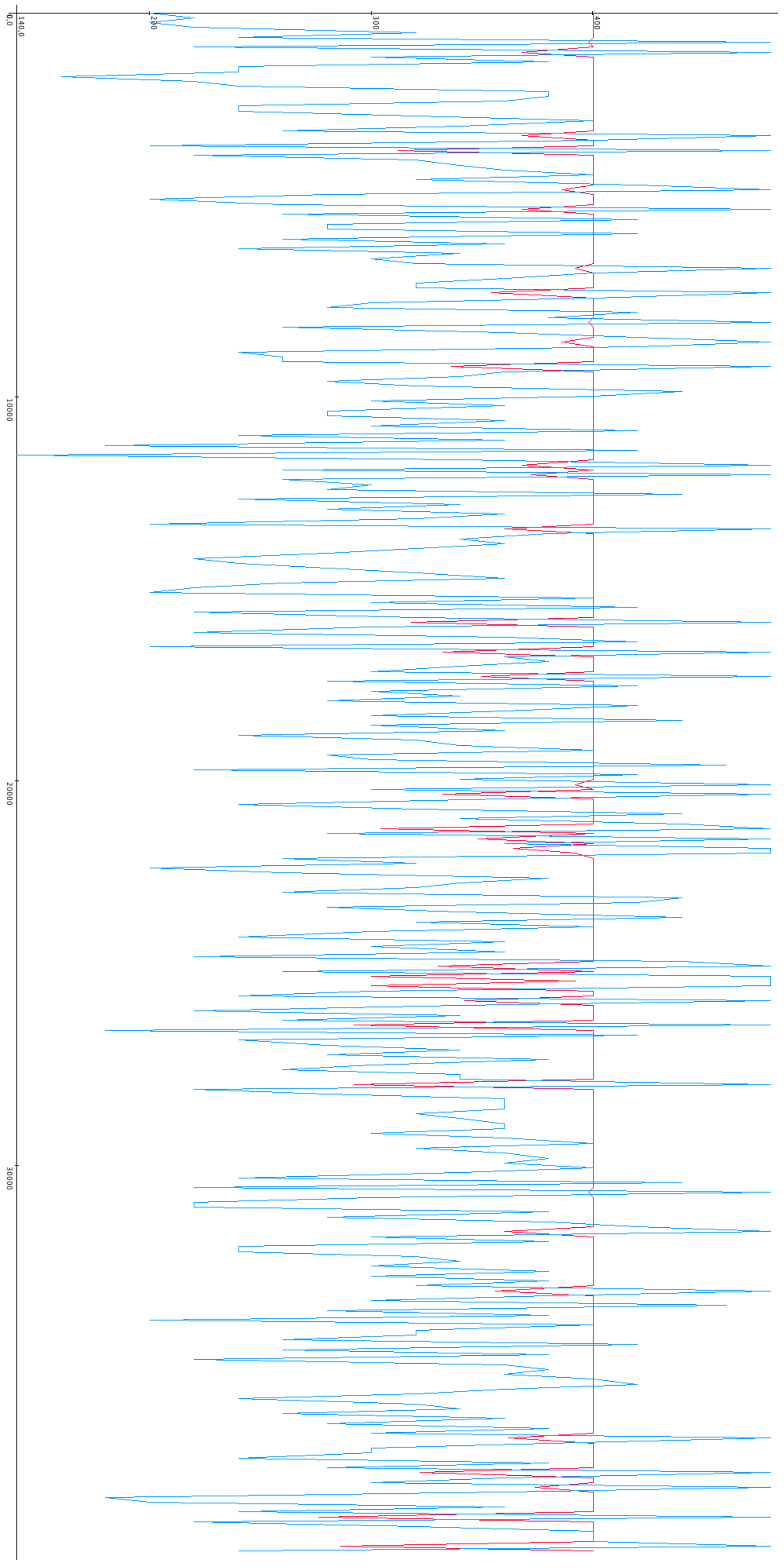


Figure F8-2: Alpha 9 Gamma 2 damage - Blue: Damage given - Red: Damage taken

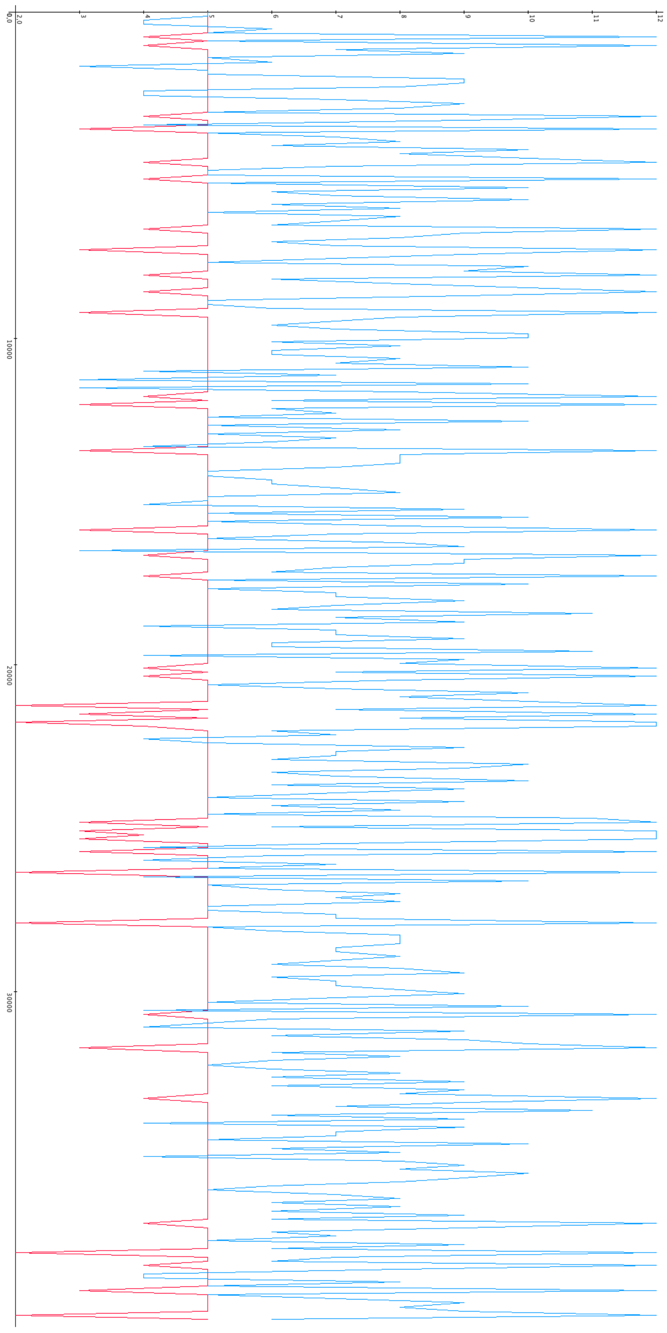


Figure F8-3: Alpha 9 Gamma 2 units lost and killed - Blue: Enemies killed - Red: Units lost

8.3 A6G4

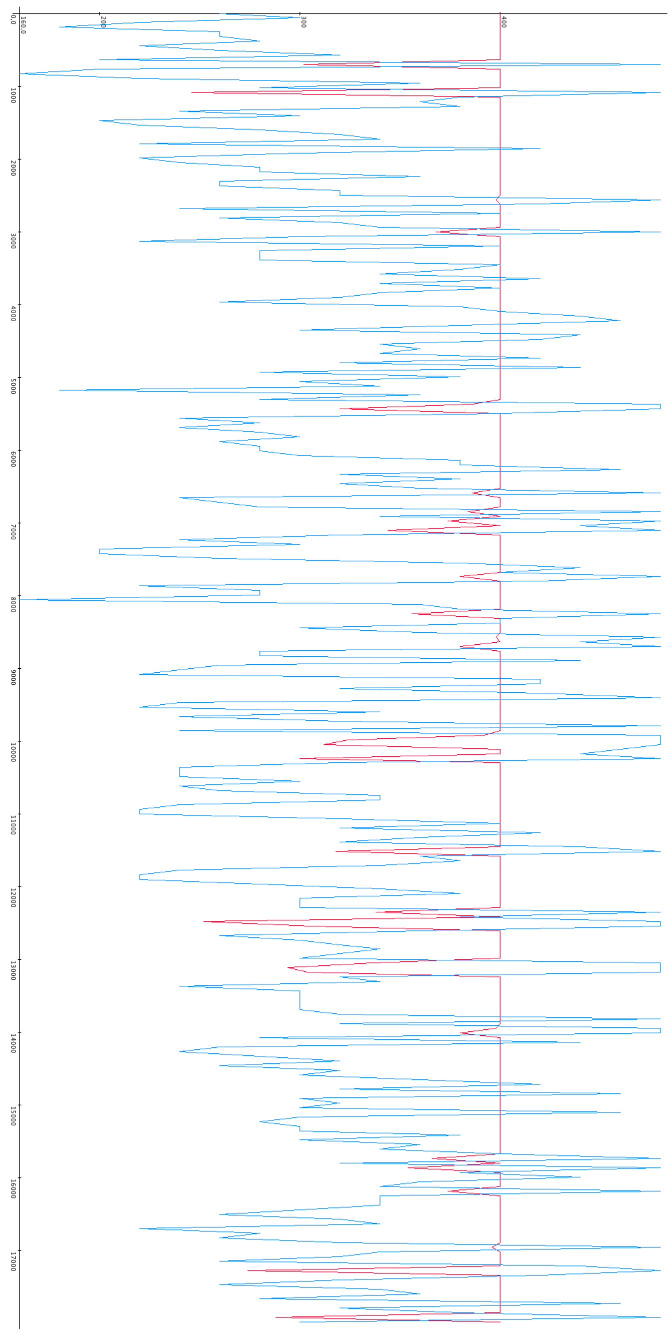


Figure F8-4: Alpha 6 Gamma 4 damage - Blue: Damage given - Red: Damage taken

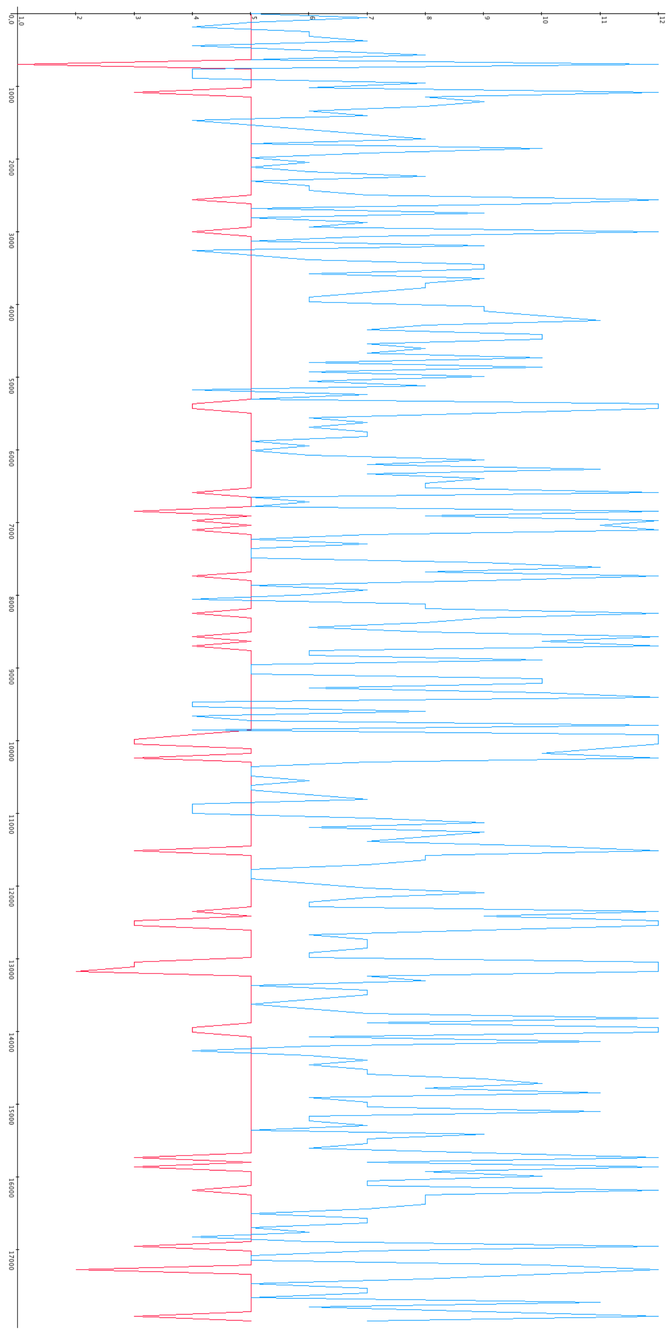


Figure F8-5: Alpha 6 Gamma 4 units lost and killed - Blue: Enemies killed - Red: Units lost

8.4 A4G6

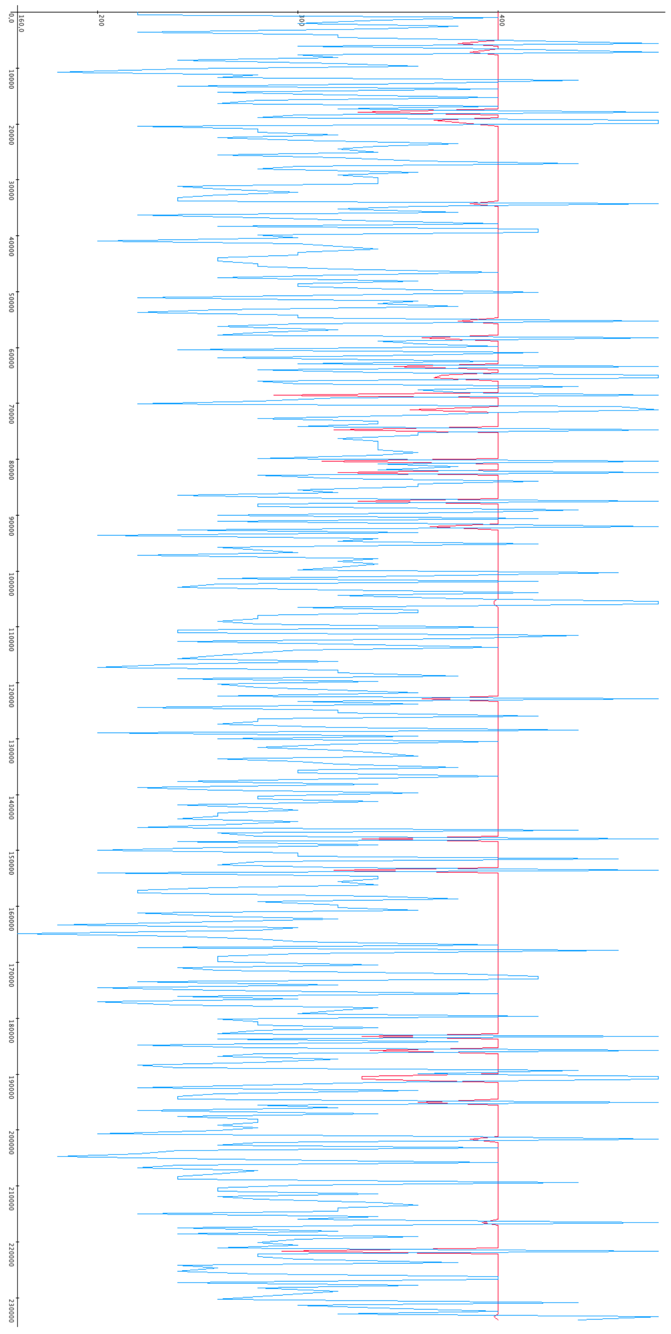


Figure F8-6: Alpha 4 Gamma 6 damage - Blue: Damage given - Red: Damage taken

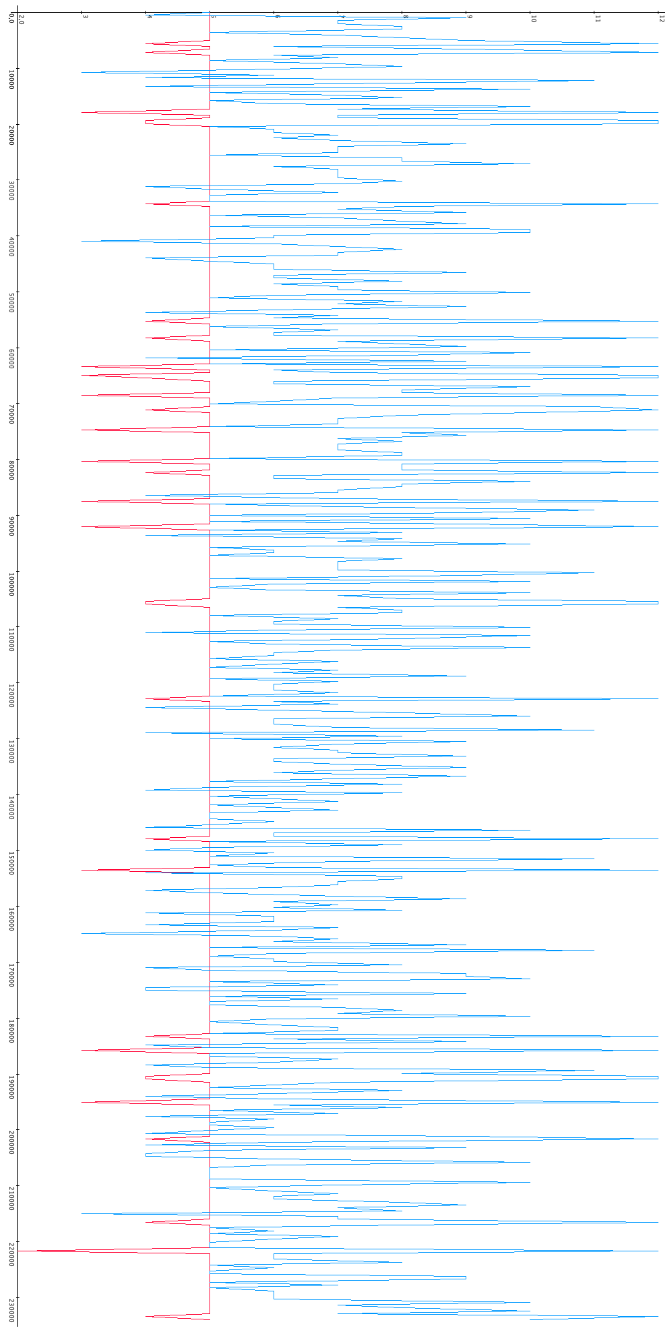


Figure F8-7: Alpha 4 Gamma 6 units lost and killed - Blue: Enemies killed - Red: Units lost

8.5 A2G9

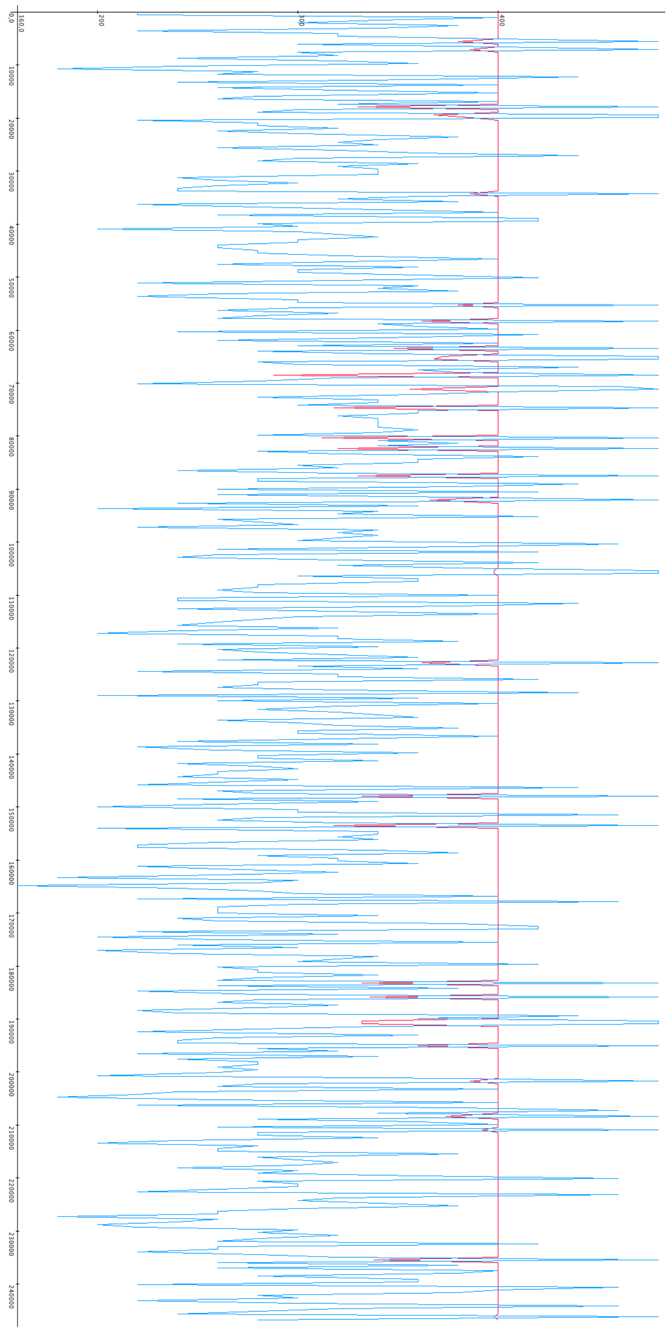


Figure F8-8: Alpha 2 Gamma 9 damage - Blue: Damage given - Red: Damage taken

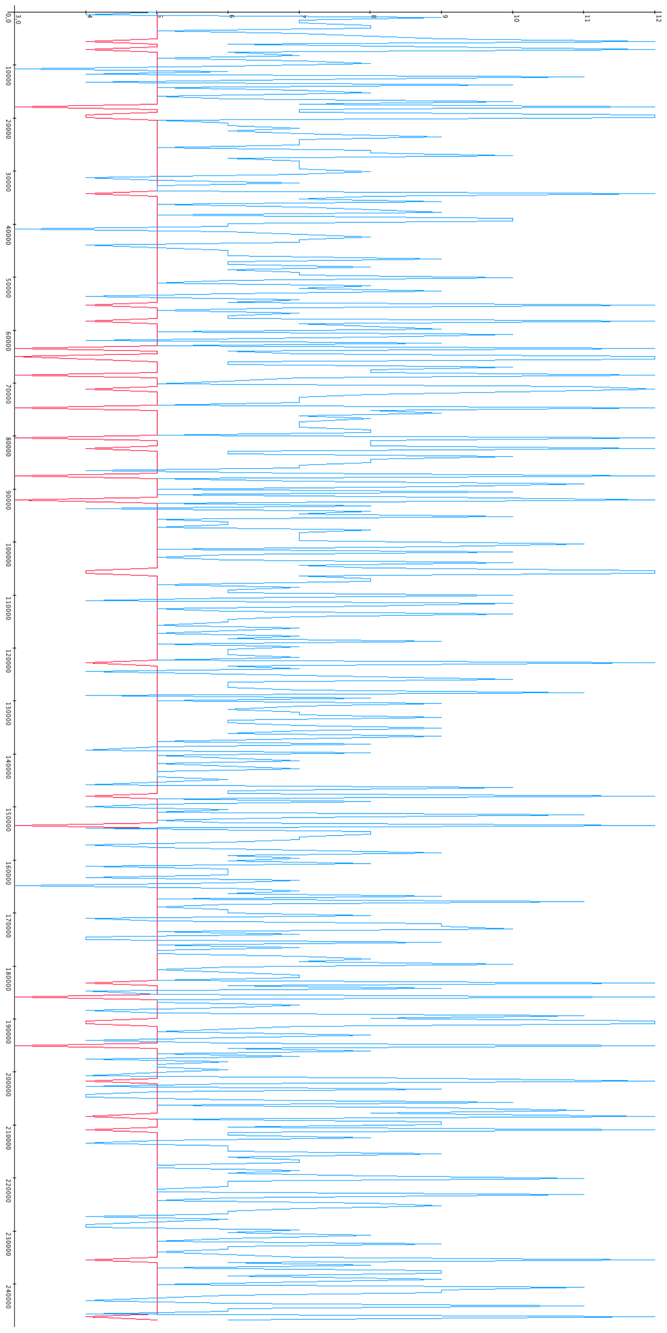


Figure F8-9: Alpha 2 Gamma 9 units lost and killed - Blue: Enemies killed - Red: Units lost

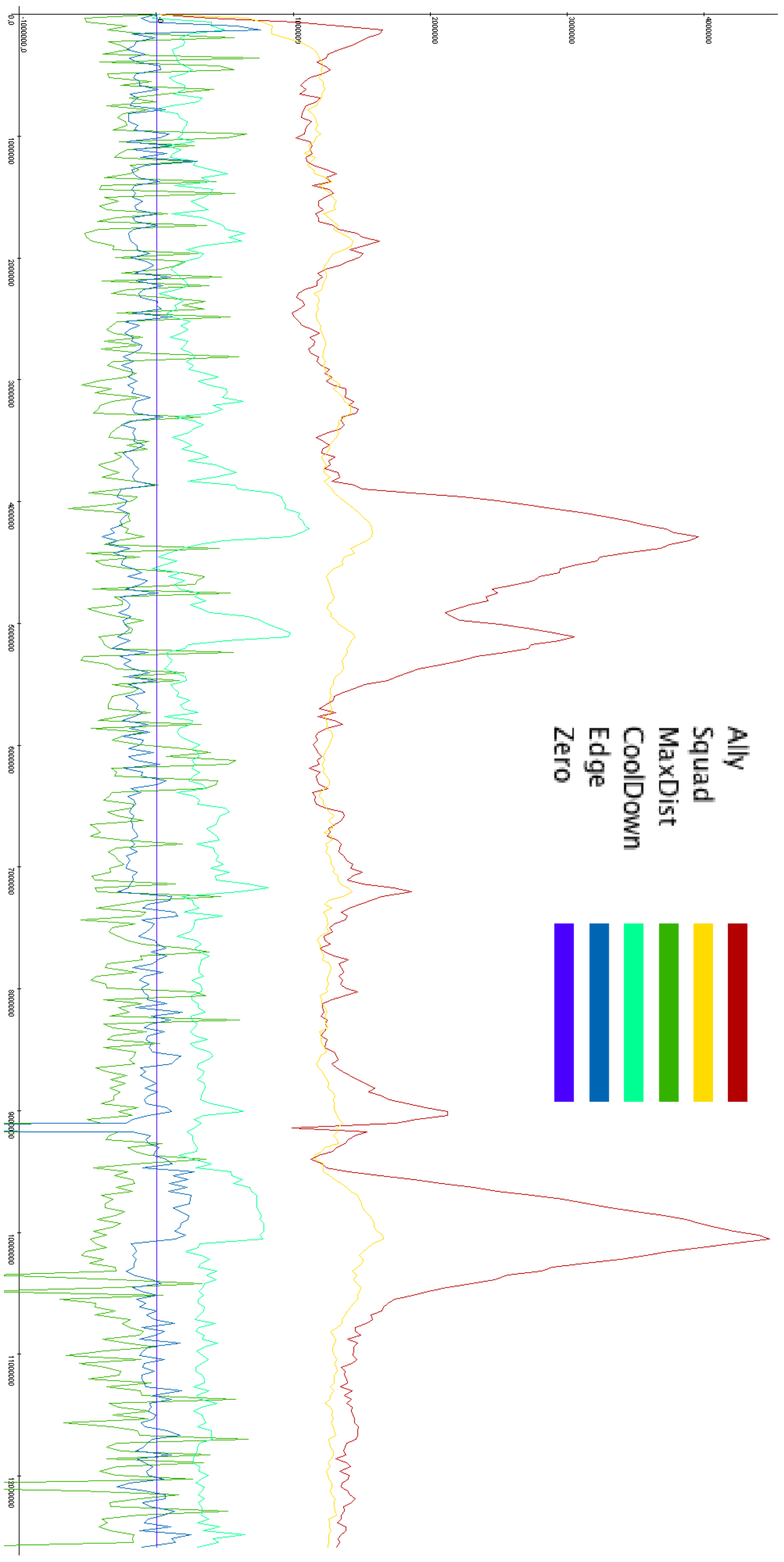


Figure F8-10: Alpha 4 Gamma 6

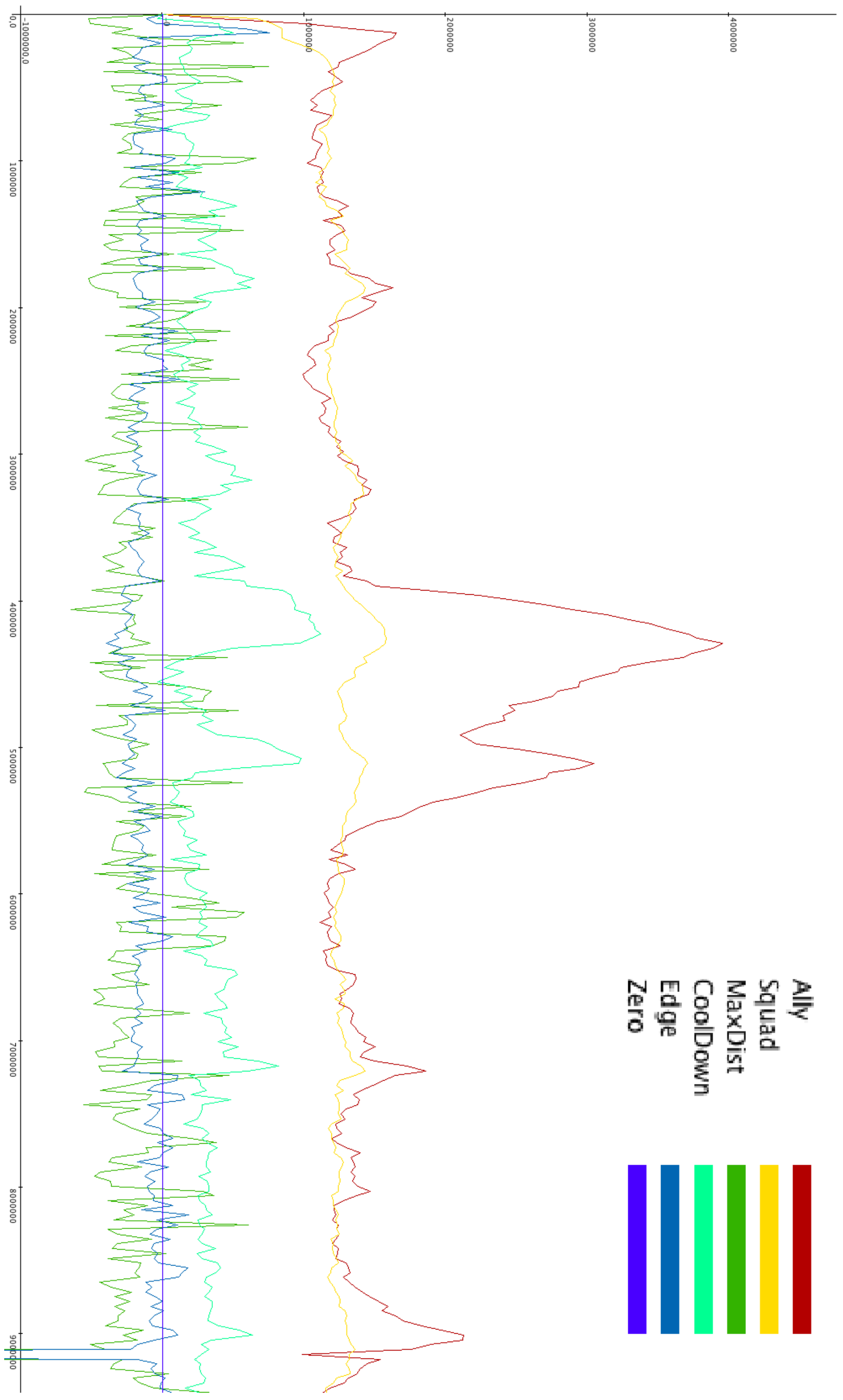


Figure F8-11: Alpha 2 Gamma 9

8.6 Winning Streak

A4G6 winning streak 1

D. taken	D. given	Ally	Squad	Max. dist.	Cooldown	Edge
380	480	2.39033e+006	1.40153e+006	-282631	698463	277493
392	480	2.39e+006	1.40181e+006	-278968	696251	274540
400	300	2.39058e+006	1.4019e+006	-273958	698558	279631
400	380	2.39196e+006	1.40196e+006	-268354	705666	293743
368	480	2.39249e+006	1.40228e+006	-264404	708283	296838
400	360	2.39315e+006	1.40233e+006	-260781	711552	301867
400	340	2.39447e+006	1.40238e+006	-256977	719614	312891
324	480	2.39407e+006	1.40262e+006	-256418	716551	307314

Table T8-1: First winning streak of A4G6

A4G6 winning streak 2

D. taken	D. given	Ally	Squad	Max. dist.	Cooldown	Edge
368	480	4.09754e+006	1.551e+006	-423034	341397	-173826
400	320	4.09229e+006	1.55017e+006	-487204	339150	-196334
392	480	4.09461e+006	1.55022e+006	-405424	342703	-194759
400	320	4.09038e+006	1.5499e+006	-469782	340807	-193842
398	480	4.09328e+006	1.5499e+006	-400091	344188	-178773
386	480	4.09453e+006	1.55029e+006	-384631	348976	-172204
400	340	4.0933e+006	1.55037e+006	-416490	348259	-183012

Table T8-2: Second winning streak of A4G6

A2G9 winning streak 1

D. taken	D. given	Ally	Squad	Max. dist.	Cooldown	Edge
398	480	1.40801e+006	1.574e+006	-761798	281321	298668
362	480	1.40872e+006	1.57429e+006	-756848	287031	305837
400	480	1.4091e+006	1.57471e+006	-752924	288704	310040
400	420	1.40933e+006	1.57481e+006	-749240	288771	312032
400	360	1.40947e+006	1.57496e+006	-745253	289079	313754
400	400	1.40982e+006	1.57511e+006	-742978	290117	314484
400	440	1.40994e+006	1.57522e+006	-739405	290281	316040
368	480	1.41033e+006	1.57547e+006	-733273	293577	323478
282	480	1.4105e+006	1.57584e+006	-802858	293081	311030

Table T8-3: First winning streak of A2G9

A2G9 winning streak 2

D. taken	D. given	Ally	Squad	Max. dist.	Cooldown	Edge
374	480	894888	4.65666e+006	-1.94865e+006	12333.4	-96592.9
238	480	895250	4.65597e+006	-1.93662e+006	14373.5	-92902.4
400	300	893479	4.65594e+006	-1.99003e+006	12629.4	-114838
398	480	892553	4.65602e+006	-2.00792e+006	12309.4	-119357
400	280	892857	4.6556e+006	-1.99505e+006	12454.5	-116666
400	480	892257	4.6547e+006	-2.0309e+006	11716.2	-124105

Table T8-4: Second winning streak of A2G9

8.7 Reinforcement Learning Field Summary

double const	*alpha* The α value of the $\hat{Q}_f$ function updating rules.
double const	*gamma* The γ value of the $\hat{Q}_f$ function updating rules.
double const	$startingEnemies$ Number of enemy units (manual input).
double const	$startingEnemyMaxHealth$ Maximum Enemy Health (manual input).
double const	$startingUnits$ Number of units (manual input).
double const	$startingUnitMaxHealth$ Maximum Health (manual input).
double const	$c1$ the C_1 value of the reward function, set to -180, coefficient for the number of units.
double const	$c2$ the C_2 value of the reward function, set to -1, coefficient for the amount of health lost.
double const	$c3$ the C_3 value of the reward function, set to 2, coefficient for the damage dealt.
double const	$c4$ the C_4 value of the reward function, set to 40, coefficient for the number of kills.
double const	$c5$ the C_5 value of the reward function, set to -0.025, coefficient for the frame count inside a game (time).
double[ ]	$liveBuffer$ Array used for saving the different $\hat{Q}_f$ updating values, its purpose is to optimize the test d data saving (to files) to only a few occurrences throughout the game.
int	$liveCount$ Counter for controlling the $liveBuffer$.
struct Weights	$\_weights${double FORCEALLY, double FORCESQUAD, double FORCEMAXDIST, double FORCECOOLDOWN, double FORCEEDGE} Struct used to save all the values of the f_i's in the $\hat{Q}_f$ function throughout all the calculations in the game.

Method Summary

static double	**CalculateTheta(double theta, double reward, double currQ, double nextQ, double derivative)** Returns the value of the updating rule for the current coefficient f_i of the $\hat{Q}_f$ function.
static double	**CalculateReward(std::set<BWAPI::Unit*>squad)** Returns the value of the reward function $R(s) = C_1 numberOfUnits + C_2 healthLost + C_3 damageDealt + C_4 numberOfKills + C_5 time$.
static void	**LoadWeightsFromFile()** Loads the weights (f) of the $\hat{Q}_f$ function into the *_weights* field.
static void	**SaveCurrentWeightsToFile()** Saves the last weights (f) of the $\hat{Q}_f$ function into the weight's file.
static void	**WriteLiveValue(double value)** Writes a value into the array $liveBuffer$ for future use in the calculations.
static double*	**GetLiveBuffer()** Returns the $liveBuffer$ array.
static int	**GetLiveCount()** Returns the $liveCount$ value that indicates how many numbers have been saved in the $liveBuffer$.
static void	**ClearLiveBuffer()** Clears the current values in the $liveBuffer$ and initializes $liveCount$ to 0.
static void	**WriteToDataFiles()** Saves game data into files for future analysis. It saves the game count, remaining health, remaining enemy health, remaining squad size, remaining number of enemies. It is only to be used at the end of each game.
static double	**GetForceAlly()** Returns the value of *_weights.FORCEALLY*.
static double	**GetForceSquad()** Returns the value of *_weights.FORCESQUAD*.
static double	**GetForceMaxDist()** Returns the value of *_weights.FORCEMAXDIST*.
static double	**GetForceCooldown()** Returns the value of *_weights.FORCECOOLDOWN*.
static double	**GetForceEdge()** Returns the value of *_weights.FORCEEDGE*.
static void	**SetForceAlly(double ally)** Sets the value of *_weights.FORCEALLY* to *ally*.
static void	**SetForceSquad(double squad)** Sets the value of *_weights.FORCESQUAD* to *squad*.
static void	**SetForceMaxDist(double mde)** Sets the value of *_weights.FORCEMAXDIST* to *mde*.
static void	**SetForceCooldown(double cool)** Sets the value of *_weights.FORCECOOLDOWN* to *cool*.
static void	**SetForceEdge(double edge)** Sets the value of *_weights.FORCEEDGE* to *edge*.

Bibliography

[1] Team Liquid. Build order. Wiki article, September 2011. 4

[2] Vulture. Wiki article. 8

[3] Michael A. Goodrich. Potential fields tutorial. 11, 12

[4] Lynne Parker. Potential fields. 13

[5] Stuart Russell and Peter Norvig. Artificial intelligence a modern approach 2nd edition. 16, 17, 18, 19

[6] Tom M. Mitchel. Machine learning, March 1997. 16, 17, 18, 19

[7] Finn V. Jenen and Thomas D. Nielsen. Bayesian networks and decision graphs. Book, 2007. 22

[8] The decision making tree - a simple way to visualize a decision. 23

[9] Broodwar application programming interface. 29

[10] An api for creating and altering bayesian network. 29