Battlecode
— Final Report —

Mohammed Al-Hakim, Peter Efstathiou, Karlson Lee, Timothy Lim, Robin Marle, William McGehee
{ma1015, ple15, kl909, tl5515, rm1715, wm2315}@doc.ic.ac.uk

Supervisor: Dr. Stephen Muggleton
Course: CO530/533, Imperial College London

20th November, 2016

Abstract

Battlecode is a real-time strategy game played by two autonomous artificially intelligent robots hosted by Massachusetts Institute of Technology. Two teams of virtual robots manage resources and attack each other with different weapons. A third faction called "Zombies" attack both teams. The goal of the game is to either destroy or outlast the opponent team. A working robot was completed and submitted to Battlecode 2016. The competition was divided into a "seeding" tournament in which the robot survived to the 4th round of play, corresponding to an approximate rank of 32th-48th place out of a total 150 teams. In the final tournament, a rank of 64th-96th place was achieved. Upon completion of the tournament, an automated testing suite was developed to allow rapid and precise prototyping of new iterations of the robot. In order to present these statistics in a more understandable way, the testing suite generated replay diagnostics in the form of bar charts and line graphs. Once a hand coded solution to the Battlecode tournament was achieved, the team attempted to produce an alternative solution based on machine learning techniques. Specifically, the team implemented a Q-learning algorithm with the goal of navigating a simplified Battlecode scenario.

Contents

1 Introduction

Battlecode is an MIT programming competition held every year that combines battle strategy, software engineering and artificial intelligence. It is a Real Time Strategy (RTS) game, pitching two teams of virtual autonomous robots against each other.

The goal of the game is to defeat the opponent by destroying all of their Archons (a mobile headquarters). This simple objective can be achieved in a wide variety of ways. A bot will not only need to form effective plans, it must also anticipate, react, and disrupt the opponent's plan. The game features imperfect information of the state of the game, the state of the opponent, and also the state of the robot's teammates. Finally, there are multiple resource constraints to manage, including financial and computational resources.[4]

The scenario presented in this year's competition is "Zombie Apocalypse". The two teams must now struggle for resources and attempt to survive amidst swarms of zombie robots. Zombie robots will periodically spawn on the map and attack the players. Each player will begin with several Archons, and the first player who loses all Archons first loses the game. Players must scavenge the map for resources and units as they evade or fight the zombies and opposing player faction.[4]

In addition to submitting a hard coded bot to the Battlecode tournament, the team also conducted a literature review of alternative approaches of engineering artificial intelligence. This review pays special attention to machine learning techniques and implementations of these techniques in the context of real time strategy games. As machine learning has received much publicity since Google DeepMind presented solutions to several Atari games and the board game go, it is useful to consider how applicable these techniques are to real time strategy games.

In addition to a literature review, the team attempted to implement a machine learning solution to a simplified Battlecode scenario. As mentioned previously, Battlecode, like most other real time strategy games, is a complex game. Not only must units navigate a wide variety of game boards and combat situations, they must also operate under economic and time constraints. A machine learning solution that successfully navigates this complex web is well beyond the scope of this project. Instead, the Battlecode engine was used to create a simple scenario in which one unit of each faction navigates a small and obstacle free game board. The goal of our implementation was to use machine learning techniques to train the target unit to avoid the enemy unit. Such a scenario has the advantages of limiting the scope of our implementation and providing testing ground from which a more complex implementation could be built.

This report describes the creation of a virtual robot that competes in the MIT Battlecode competition. This project is divided into three components: the Battlecode competition, the automated testing suite and incorporating reinforcement learning as an extension of the project. This report takes the reader through the following sections, each of which branch off into the three project components. The first section is the specification which provides the original outline of the project, and the specific requirements set for each objective. The specification section is followed by the design section where we have analysed and discussed all the design choices we have made and alternative options we have considered. The report then outlines the development methodology used throughout the project, including unit testing and system testing. In an almost independent section, a brief literature review of applicable machine learning techniques and RTS implementations of those techniques is presented. Finally, the performance of our product is shown in the Final Product section. Within the Final Product section, challenges encountered during implementation are highlighted as well as the solutions that were employed to resolve them.

2 Specification

The main aim of the project is to produce a bot that is able to play and compete in the Battlecode tournament. As the final goal is well defined, capturing the minimum, medium and super requirements was essential to the success of the project.

2.1 Requirements Overview

1. Minimum Requirements
    - Working Bot
    - Defensive Strategy (Turtle)
    - Offensive Strategy (Napoleon)
2. Medium Requirements
    - Automated Test Suite
    - Beat MIT Reference Bot
3. Super Requirements
    - Compute success indicators
    - Literature review of relevant machine learning techniques
    - Implement and evaluate machine learning techniques such as Neural Nets and Reinforcement Learning
    - Implement and evaluate self Program Induction Technique

Throughout the development of the project and the Battlecode tournament, the team were required to continuously implement, test and discard various strategies, depending on their efficiency. For example, one strategy which was not implemented was a comprehensive "scouting" strategy. Such a strategy would use a Scout robot to ascertain the locations of various enemy robots and communicate those locations to friendly units. A more detailed discussion of the strategies pursued and discarded will be presented in the Design and Final Product sections of the report.

2.2 Requirements Outline

2.2.1 Minimum Requirement: Working Bot - Achieved

The team aimed to submit at least one working AI to the MIT Battlecode competition. A fully functional bot consists of basic functionality such as the ability survive and either destroy or outlive in an environment with a trivial enemy. Additionally the team aimed to develop two main strategies, one defensive and one offensive.

1. Basic Functionality
    - Produce units
    - Engage enemy units
    - Implement a defense strategy
    - Implement an offensive strategy
2. Defensive Strategy (Turtle)
    - Build turrets around the Archon (home base)
    - Turrets distribute themselves autonomously to maximise coverage
    - Build soldiers

- Soldiers distribute themselves in the proximity of the turrets to protect them
- Repair nearby units

3. Offensive Strategy (Napoleon)
    - Engage a defensive strategy until the enemy has been scouted and enough units are available
    - Pack up turrets and move in formation to the enemy location
    - Eliminate the enemy

This requirement has been met as two working bots were entered into the Battlcode competition and competed successfully in the tournament. There was no need to change these requirements during the project.

2.2.2 Medium Requirement: Automated Test Suite - Achieved

An automated test suite is critical in order to quickly iteratin through different versions of the bot. MIT provides a basic test framework called The Battlecode Headless Client; the goal is to extend this into a more sophisticated testing environment in which useful statistics and insights can be generated. This testing suite will run a large number of matches across different maps and parse replay file information into a more useful format.

- Automate games between two players across a number of maps (at least 20)
- Output results in XML format
- Randomized starting locations for players

This requirement has been met as the team created an automated testing suite using three stages discussed further in the design and final product sections. There was no need to change these requirements during the project.

2.2.3 Medium Requirement: Beat MIT Reference Bot - Achieved

As part of the competition, MIT releases a Reference Bot that students are encouraged to beat. A medium requirement of this project will be to beat the Reference Bot in at least 50% of the available maps.

This requirement has been met as the 'Victorious Secret' robot was able to beat the MIT reference bot 87% of the time.

2.2.4 Super Requirement: Literature Review - Achieved

The team will read and review literature of relevant machine learning techniques and alternative real time strategy game solutions. Each member will hold a workshop on their paper, to ensure that the knowledge is effectively shared. Upon completion of this review, a technique will be chosen upon which to base a machine learning solution to a simplified Battlecode scenario.

This requirement has been met as the team met regularly to discuss and present machine learning research. Based on this research, the team decided to explore the use of Q-learning.

2.2.5 Super Requirement: Implement Machine Learning For Combat Micro - Achieved

Combat micro is a subtle and complex art, where different conditions require different combat strategies. The team will explore the use of machine learning techniques as a method of navigating combat situations. A simplified Battlecode scenario will be used as testing ground for such techniques.

This requirement has been met as a Q-learning solution to a simple Battlecode scenario was produced. However, the results of this implementation are mixed and discussed more fully in the relevant section.

2.2.6 Super Requirement: Implement Program Induction - Uncompleted

Program induction is a modern technique in self-writing programs. Flags enabling or disabling behaviours will be set and a learning algorithm will find optimal combinations of behaviours. [16] [15]

This requirement has not been met as the team did not have enough time to complete this super requirement.

2.2.7 Super Requirement: Success Indicators - Semi-Achieved

In order to demonstrate progress, we will compute metrics that describe the desired behaviours. The indicators can be divided by the behaviour which they measure, facilitating unit testing and allowing objective measure of marginal code improvements.

1. Macro indicators (e.g Win percentage)
2. Combat indicators (e.g Combat event win percentage)
3. Movement indicators (e.g Time taken to reach destination)
4. Scouting indicators (e.g Percentage map explored)
5. Code Quality indicators (e.g Number of exceptions thrown)

This requirement was partially met. Macro indicators, such as win percentage, and Combat indicators, such as number of units over time, were produced using the headless client and testing suite. However, Movement, Scouting and Code Quality indicators were left unimplemented. The Battlecode data provided per round did not lend itself to providing these types of indicators.

2.2.8 Uncompleted Sub-Requirements

Implement Viper Strategy
Initially Viper robots were considered by the team as an optional strategy to use within both the defensive and offensive robots during sprint 4. However, the deadline of implementing a viper strategy was not met because, as the tournament progressed, it was identified that Viper robots are not crucial to the success of a robots overall game plan and time could be more effectively spent elsewhere.

Identify Zombie Dens using Scout
The team has implemented a simple version of scout strategy that identifies Zombie Dens and broadcasts the location to all of the teams attacking units. However, it became apparent that this strategy was not as effective as expected and was thus discarded. This has been discussed in the Further Development section.

Implement Kiting in Attack Micro
The team had started on the kiting strategy during sprint 4. However, the team was unable to complete the task as the original allocated time was not too short. This was an oversight as the team originally considered the task as trivial. The team decided to discard this task in order to start work on the super requirements ensuring adherence to the scheduled sprints.

Implement program Induction Technique
This super requirement was seen to be too ambitious in the later stage of development. Within Sprint 6 and 7, the team realised there would not be enough time to complete both implementations on program induction and machine learning techniques. In hindsight, this was the right decision, as the team was able to complete the most important super requirement with the limited time available. Further development of the project would expand on the current level of machine learning used and how program induction can be introduced.

Implement meta-level program for offline learning
As Neuronph can be connected directly to the Battlecode API, there was no need to implement a meta-level program that connects the machine learning and java components. However, as TensorFlow is a python based package, the testing suite code was expanded in order to act as a bridge between the Java and Python implementations.

3 Design

The main design objective of the project is to efficiently interface artificial intelligence with the Battlecode API client. The API client is defined as the 'RobotController' class which provides functions such as the movement and fighting of robots within the simulation.

For a simple robot it is possible to encapsulate all the necessary logic in a single file. In fact, this was the case for the first submission during the sprint tournament. However, as the tournament progressed, it was decided that the code should be organized in the manner shown in the UML diagram on the next page.[12]

The Battlecode API was used to build complex functions for the behaviour of the units. The foundational logic is encapsulated in the 'Behaviour' package. Foundational logic refers to code that is one level of abstraction more complex than a simple Battlecode API command. For example, the API provides methods to check if the core is ready (i.e., whether the robot can execute an action), check if a target is within a certain radius and finally, to shoot at a target. The 'Behaviour' package has a 'Fight' class within which an 'Attack' method would call an attack function that would invoke the previous three functions as given by the API.

The 'Strategy' package contains so called 'meta-behaviours' that add an additional layer of complexity atop the Behaviours defined previously. Strategies execute behaviours with the aim of achieving a goal. For example, the 'flee' strategy aims to move a unit away from opponents and to stay alive for as long as possible, and as such it calls the 'navigation' behaviour dependent on environmental inputs.

Finally, 'units' are agents supplied by the Battlecode framework that act autonomously within each game instance. Each unit runs in its own thread and has its own shared memory. In addition, the units encapsulate different strategies depending on their class.

3.1 Navigation

Navigation is the central component of both robot strategies (offensive and defensive). Navigation will be used by the Archon to navigate to other Archons and provide the movement behaviour of all units throughout the duration of the game. The main navigation techniques we have looked at are depth-first search, breadth-first search, bug navigation and A* search. [18]

A* was considered at the beginning as it would be very applicable and powerful pathing algorithm for the robot. From several Mathmatica simulations, it was shown to have required behaviour compared to the other choices. It is very effective and thus is chosen as the preferred pathing algorithm for the robots.

DFS/BFS is the most used algorithm for previous Battlecode tournaments due to its simplicity. [18] However, it has the highest bytecode usage and thus was not considered.

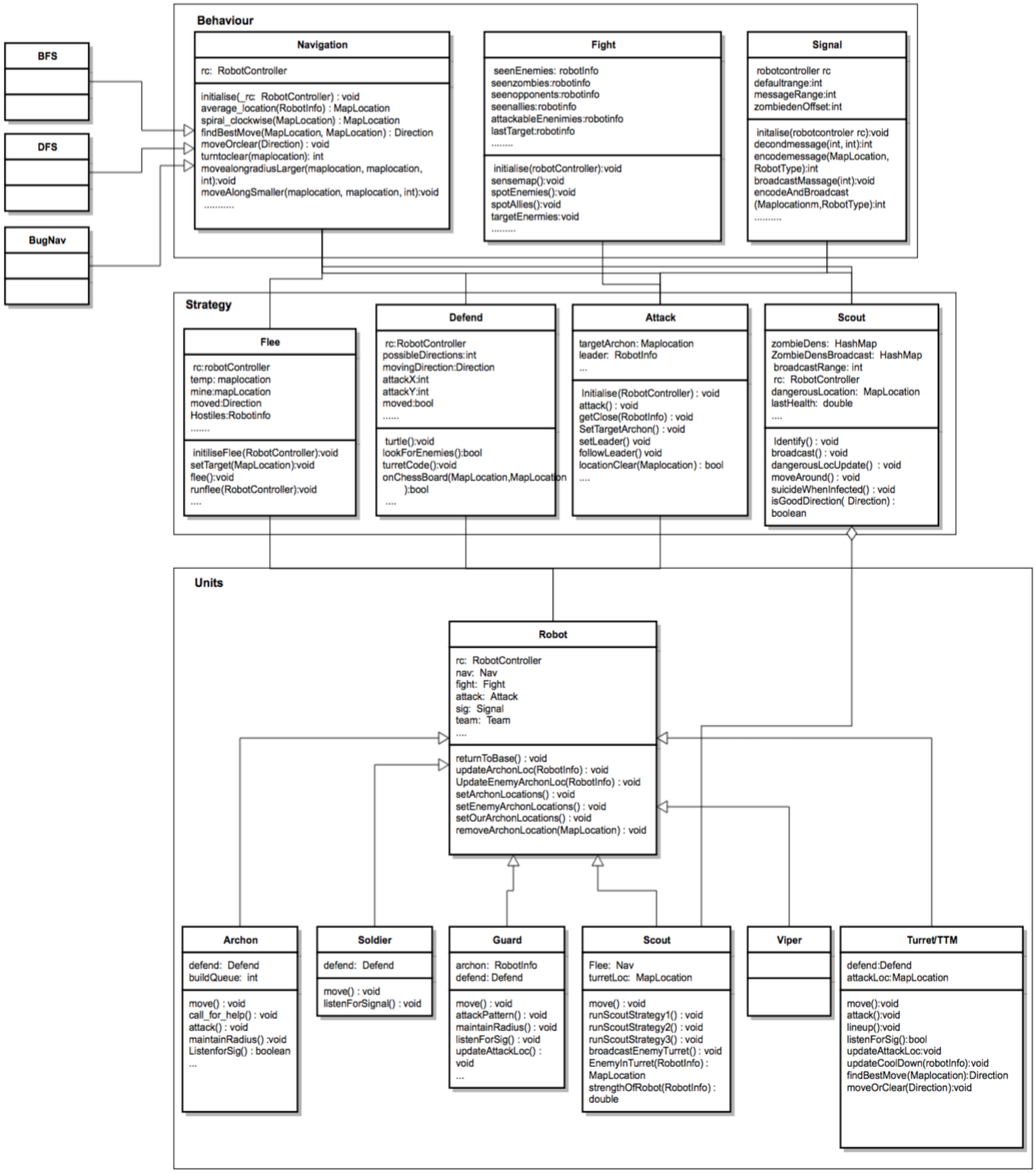
3.2 Scouting

Scouting is an essential component of any real time strategy game. All units operate under incomplete information with only a limited sensor range within which they can spot enemies and map information. In Battlecode, a dedicated scout unit exits which has above average movement speed, sensor range and messaging capabilities. Any scouting implementation is an exercise in leveraging the advantages of this unit. Therefore, unlike other basic behaviours, scouting is not best implemented as a behaviour class but instead as an extension of the scout unit.

Several scouting algorithms were explored but all were variations on several discrete steps. First, the scout unit would pick a direction of exploration. Second the scout would sense map and enemy features at each step of this exploration. If valuable information is found, the scout will return to friendly territory, communicate that information, and return to step one to repeat the loop. If no valuable information was found during exploration, the scout would return to step one to repeat the loop.

A special case in which scouting does not conform to this algorithm is that of turret enhancement. The turret unit is slightly unusual in that its sight range is smaller than its attack range. Thus, the turrets efficiency can be enhanced by pairing it with a scout. The scouts large sight radius can be

Figure 1: Simplified UML diagram to illustrate organization of code



used to identify attack-able enemies outside of the turrets sensor range. Once these enemies have been identified, the scout can communicate their location to the turret, thus effectively increasing the turrets line of sight. This scout/turret pairing was used to great success in the robots turtle strategy.

3.3 Signalling

The Battlecode game engine enforces a strict separation between robots, preventing any sharing of resources or information except through a very limited signalling interface. This interface has two

forms, a complex one that transmits two signed integers and a simple one that only allows four binary signals. All units can receive all types of signals and can transmit the simple signals. The ability to transmit the more complex signals were restricted to only Archons and Scouts. Thus, the ability to exchange information to facilitate coordination is severely curtailed.

Units that can only use the restricted signalling system can communicate using a base-four binary system to transmit information efficiently between units. As receiving units were only able to detect the presence of a signal, rather than it's absence, and units could send a maximum of four beeps per turn, the most efficient representation is a quaternary system.

Archons and Scouts are allowed to use a more descriptive messaging system, able to transmit two integers up to four times per turn. As mentioned in the scouting section above, scouts will need to be able to communicate what they have discovered and where. As messages will almost exclusively be in the form of map coordinates, and there are only a small number of feature-types to discover, and the range of the integer is substantially greater than the range of possible coordinates, feature description information will be encoded by adding an offset to the coordinates.

A small space will also reserved to allow the Archon to transmit high level change in strategy commands. Thus, an Archon could signal to units that it was time to flee and later that it was time to stop fleeing.

3.4 Fleeing

Fleeing will use the navigation algorithm as described above to implement a pathing dedicated to fleeing from hostile units. It will be part of the strategy package and will be used by all units.

The objective of fleeing is for the robots to survive as long as possible. Fleeing will require digging through walls when surrounded and the ability to pick the best direction to move using the navigation algorithm. The team decided that if the robot health is below 40%, fleeing will be triggered to ensure unit survival.

3.5 Offensive Strategy

The offensive strategy - the behaviour of all units together - was centered around building lines of units that then slowly converge on the last known enemy location. There are two opponents in Battlecode, the Zombies and the enemy player. An offensive strategy has to decide who attack as well as how to attack them. We decided to focus on attacking the opponent, rather than the Zombies, as we suspected this would be the most efficient way of winning each battle. It involves a substantial amount of risk taking, as it is substantially more difficult to coordinate a successful attack than a defense.

The goal of this strategy is to proactively destroy the opposing team, before they can destroy you.

3.6 Defensive Strategy

The 'Defend' class implements the Defensive strategy with objective of gathering units to circle a designated Archon. For the purposes of this report, this strategy will be referred to as 'turtling'[2].

Turtling is a game play strategy that relies on heavy defense, with no offense. It minimizes risk while baiting the opponent to take risks in trying to overcome the defenses. Consequently, while turtling strategies are usually simple enough to implement and are effective as such, they are easily defeated by experienced opponents that implements counter turtling strategies.

The Defensive strategy defines a perimeter around an Archon to be protected which evolves when the area is populated with units depending on the space available.

The goal of this strategy is to survive longer than the opponent team as Zombie units become stronger and stronger throughout the rounds.

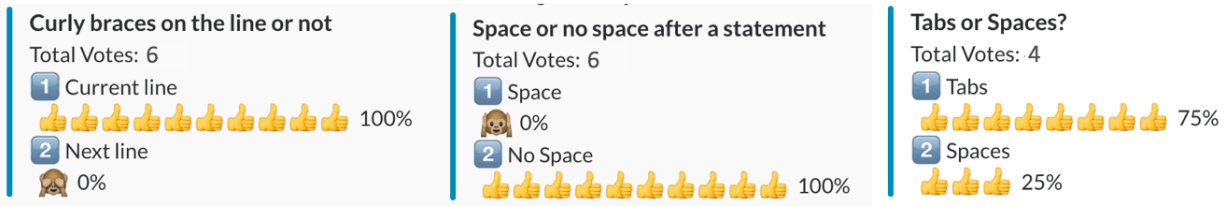
3.7 Coding Conventions

Since the main programmatic constraint in the competition was a limit on the number of bytecode that were allowed to be computed each round, it was decided that the majority of the functions would be static. This is okay as each bot in Battlecode is ran through multiple independent JVMs and thus

there is no need to defined extra overhead through defining objects. This served two purposes: firstly, there was a performance increase (and hence a subsequent decrease in the number of bytecode used) and secondly, it simplified the calling of methods between classes. The main disadvantage of using static methods was the loss of flexibility in the design of the code base as the project progressed. This is due to fact that static methods cannot be overwritten nor be declared in an interface.

The team decided that it was imperative to establish a coherent coding conventions that all team members must follow. An example of these decisions was conducted using slack polls shown below.

Figure 2: Slack polls to conduct design decisions as a group



3.8 Other Implementations

As an addition to the BugNav and fleeing code, in later stages of development there was a requirement for the robots to pick up parts and activate neutral unit(e.g Archons). This was seen as a vital implementation as units and parts contribute to the success of strategy as the winners are decided by which team has the most surviving archons and parts provided resources to build more turrets and soldiers for a stronger base.

3.9 Machine Learning

In order to implement a machine learning based solution to a simplified Battlecode scenario the team conducted a literature review of a variety of techniques. After reviewing each of these techniques, the team decided to pursue an implementation based on reinforcement learning, specifically Q-learning. [19]

In general, reinforcement learning is concerned with how an agent is able to learn an optimal policy of behaviours from interacting with its environment. An agent is able to observe the state of the environment and the actions it can perform in that state [20]. Each action is assumed to alter the state of the environment in some way. It is further assumed that the agent has a specific set of goals it wishes to accomplish and that these goals are represented by a reward function. [19]

At any given state, the value of an action can be divided into two components: an immediate reward, as calculated by the reward function, and the cumulative future reward of new state that results from that action. An optimal policy is one that chooses those actions with the highest value at each state.

While the immediate reward is likely straightforward to calculate, the cumulative future reward of the resulting state is very difficult to interpret. Q-learning presents an algorithm that attempts to approximate this cumulative future reward function using a Q-function. A Q-function is a function that takes as inputs a state and an action and returns as output the cumulative future reward of that state action pair. Specifying this function is possible due to a recursive definition of cumulative future reward. For each State/Action pair the cumulative future reward is the immediate reward received for performing the action from that state plus the value of following an optimal policy thereafter. Using this definition it is therefore possible to implement this Q-function as a large look up table of state action pairs. Indeed, this implementation has been used for small environments. However, once the state space of the environment grows beyond a certain threshold, expressing this Q-function in terms of look up tables becomes impractical. Instead neural networks are often used as Q-function approximates instead. [19]

The team chose to pursue Q-learning for several reasons. The main reason was the algorithm's generality. In order to implement a solution, one simply needs an agent, an environment, a reward

function and a correct representation of the Q-function. The second reason the team chose to pursue a Q-learning approach was how easily real time strategy elements could be mapped to the Q-learning components of game state, robot actions and reward.

Thus, in designing a Q-learning based solution it became apparent that the team would need to specify the aforementioned components, namely a reward function, a neural network with which to represent a Q-function and a method of codifying the states of the environment as well as robot actions. The exact implementations of these components are discussed in the Final Product section.

3.10 Testing Suite Design

The default method of checking the effectiveness of a strategy is to watch each match unfold in real time, making observations of the robot's behaviour over the course of the match. This method is both inefficient and imprecise. Each match takes between one and five minutes to run, making it infeasible to interpret a large number of matches. Furthermore, any interpretation of the strategy's efficiency is subjective at best. One may be tempted to rely entirely on the robot's win percentage. However, this metric is only informative when the strength of the opponent is taken into consideration (e.g. A win percentage of 40% might be excellent if the opponent is a top seeded robot). To provide an accurate and objective measure of how well a robot plays the game, a more advanced set of testing tools are needed.

The testing suite will perform its analysis in three stages. In the first stage, a large number of individual "headless" matches (i.e. matches run without the viewing the heads-up-display) are run. Each match should produce a "replay" file that specifies the state of play at each round in that match. The second stage would take these replay files and extract the data into a directory for further analysis. In the third stage of testing there will be a script that extracts relevant information and generates a series of descriptive statistics and graphs that can be used to determine the efficiency of the strategy over the course of the match. These provide the success indicators such as macro, combat, movement and scouting indicators.

4 Methodology

4.1 Project Management Overview

Throughout the project, the team successfully followed the agile development methodology[14]. Each week, a sprint backlog was compiled, specifying tasks to be completed in the current sprint. The team attended scrums every other day, bi-weekly review meeting on Tuesdays, and 15 minutes mid-week progress checks on Fridays. Images of the scrum meetings discussion points can be seen in the Appendix 4.1. Furthermore, certain design decisions were conducted through polls. An example can be seen in Appendix 4.1. Tasks were always delegated to pairs of team members as such "paired programming" was found to produce higher quality code with fewer errors.

The regular competition deadlines naturally encouraged a scrum methodology[14], as each inter-tournament period could be easily mapped to individual sprints. Furthermore, the performance of the robot in each tournament provided valuable feedback that informed the objectives of the sprint to follow. Observing the strategies of other competitors also provided inspiration for our regular design and analysis sessions.

The master/production branch of the team's Git repository is updated only when key functionality has been fully tested and agreed on by all members. This "master version" is a stable version of our robot and demonstrates to key stakeholders the incremental improvements in the robot's functionality.

4.2 Sprints

The 14 weeks between the submission of this report and the final project deadline will be divided into seven two-week-long sprints. Each sprint will begin with a planning session. During these planning sessions, the team and project leaders will determine the set of features to implement during that sprint. A sprint backlog will then be prepared specifying the tasks that need to be completed in order to accomplish the selected features. Tasks will be assigned to team members either by group leader assignment or self-selection. [14]

4.3 Scrums

During the sprint period, short meetings (scrums) will be held every other day. Each member of the team will discuss what he has accomplished since the last scrum, what he intends to do over the next two days, and any obstacles he has/expects to experience in meeting the sprint objectives. Any obstacles, delays or other issues will be addressed by the team leaders. [6] [11]

4.4 Sprint Reviews

At the end of each sprint period, a review session will be held. Work that was completed during the sprint period will be reviewed and discussed. Any issues will be highlighted. Tasks that were not completed will be identified. The estimated completion time and feature backlog will be adjusted accordingly. Progress and areas of difficulty will be presented to Dr Stephen Muggleton after each review session.

4.5 Documentation

Documentation will be maintained as in-line comments conforming to the Javadocs standard. Javadocs allows automated creation of HTML documentation pages, and are supported by many IDE's such as the one we are using, IntelliJ. Documentation will also be supported by a Github-based wiki.

4.6 Team Working Tools

4.6.1 Version Control

The team consists of six team members. Therefore version control is essential to ensure the tracking of changes, reverting changes, merging individual development branches and resolving conflicts. There are many version control systems available. However, this project requires a free, open-source, distributed and disconnected environment. It is also important that the version control system (VCS) contains adequate documentation, support and is popular amongst developers. [11] [6]

Git, is an open-source environment that meets all these requirements. It also used by many popular projects such as Googles Android, Eclipse and the Linux Kernel. All the team members have working knowledge of Git which drastically reduces the initial learning curve and enables focus to be on the requirements at hand.

Github is an online hosting service for Git. It offers a simple UI for browsing history, bug-tracking, wiki. It is distributed, allowing team members to work separately at different times and locations. It is also the most popular Git repo, and thus has a large body of resources available online and excellent support.

4.6.2 Collaboration Tools

Communication will be done using Slack, a project management platform for team communication. It provides full visibility to all aspects of the project. Relevant materials are shared, discussed, segmented using Slack channels. This is also convenient for organising sprints.

4.6.3 Workflow Control

To encourage agile development technique, Trello is used as the task scheduling tool. This tool gives access to a visual board that displays the ongoing tasks. These tasks are represented as cards with labels and priorities.

4.7 Feasibility and Risks

Although the minimum requirements are relatively feasible, risks remain as most team members are unfamiliar with Java and the proposed project management tools. However, the MIT competition has been running for the last 10 years and therefore there is a great deal of relevant material online.

The medium requirements are more challenging. However, they are still a manageable risk as they can be accomplished with adequate time, planning and preparation.

There are several risks involved in the super requirements of the project. Most members have limited experience with state of the art artificial intelligence and machine learning techniques. This knowledge has to be acquired concurrent to the completion of the more basic requirements of the project. Secondly, there has not been an implementation of neural networks and program induction techniques in Battlecode before. Although this will demonstrate genuine innovation from the team, it will involve a process of trial and error.

We will continuously adjust the schedule to these risks by working intensively with our supervisor, team sprints, efficient project management and code development integrity as specified in this report.

4.8 Project Boundaries

4.8.1 Software

The software will be written in Java, using the IntelliJ IDEA IDE. The Battlecode Client suite will be used predominantly for testing AI features, map-making, etc.

4.8.2 Testing Methodology

A Grey-Box testing approach was used, with each development group building tests for the code they wrote, but with additional key areas also black-box tested by the consumers of that code.[1]

The robot was built on the MIT-supplied Battlecode API, which the team decided was not within the testing scope. However, to facilitate the testing of code that depended upon this, the team developed extensive testing stubs to simulate responses from the API. This allowed us to craft tests that had well-defined behaviours.

JUnit was selected as the unit testing framework for our code, as it is the most popular Java uniting framework and it enabled straightforward testing of our Java code.[7] [10]

IntelliJ IDE was used as the code coverage tool as it is the IDE that the group used for this project and it has a integration with JUnit, allowing tests to be run and coverage to be computed at the same time. Statement, branch and function coverage is also easily seen.

All classes, methods and statements were extensively documented using Java Docs, which ensures transparency between the teams and stakeholders.

4.8.3 Unit Testing

Unit Testing is conducted by testing each method in a class under different conditions. These test-cases are created by setting the state of the Battlecode client so as to generate an expected result, and comparing the actual results with the expected results. This highlighted a number of minor programming errors, such as a "spiralClockwise" function that made robots spiral anti-clockwise, to more substantial errors, such as failing to defensively guard against nullPointerExceptions. Unit-tests are completed alongside the development as much as possible to ensure components are tested as soon as they are built.

The team decided that partition testing was vital to the testing framework of this project, as the code depends very much upon the RobotController class from the Battlecode API which is specified by more than 30 methods. However, in many cases each function only depends on a small subset of these parameters. Comprehensive tests over the full spectrum of possible methods would have been infeasible and unnecessary so the team decided to test only on the combination of methods that mattered. [1][7]

Each project group was designated an area to develop and to provide unit tests. The program is composed of several layers, where high level functionality is provided by combining sequences of lower level behaviours. For example, the Attack class builds various high-level attack profiles by combining low-level targeting and situation-evaluation functions provided by the Fight class. It was typically much easier to effectively test low-level behaviours as they were typically simple and atomic. Higher-level functionality often contained many branches and complicated dependencies. The partition tests on the low-level functions provided examples of relevant partitions on the high-level functions. Due to the high branching factor it was typically only possible to test the most relevant execution paths.

Individual sections that were considered particularly important were also black-box tested. For example, the robot's path finding implementation is a critical function. As such, a separate group implemented a set of black box unit tests, thus providing further validation of that unit's correctness. These black-box tests also provided verification that each team was correctly implementing the classes, and that the documentation was clear, relevant, and useful.

4.8.4 System Testing

In order to effectively test our project on a systems level, the team has developed a testing suite that automates the process of running and interpreting the outcome of Battlecode matches. [1] [10]

4.8.5 Testing Summary

The main difficulty of unit-testing is interacting with the Battlecode engine. Because the robot player's code is built on top of a relatively complex platform, much of the robot's behaviour is predicated on the state of that platform (i.e., what signals have been sent, what robots are in what locations, the robot's current health, etc.). This complication was handled by building testing stubs that specified these states with pre-defined constants.

The Robot Controller controls and runs the Battlecode engine implementing the methods available to players for use. The rewritten Robot Controller stub ensures that the code coverage targets can

be achieved. The team has actually found a bug within the Battlecode implementation through this process showing the thorough analysis conducted and the stress-testing the team has implemented.

Figure 3: Table with code coverage for each section

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	88% (22/ 25)	75,3% (189/ 251)	72,5% (1036/ 1428)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
victorious_secret.Behaviour	100% (8/ 8)	90,5% (67/ 74)	92,1% (397/ 431)
victorious_secret.Strategy	80% (4/ 5)	70,5% (31/ 44)	54,6% (273/ 500)
victorious_secret.Testing_stubs	100% (2/ 2)	62,6% (57/ 91)	73,4% (94/ 128)
victorious_secret.Units	80% (8/ 10)	80,6% (34/ 42)	73,7% (272/ 369)

generated on 2016-03-11 11:53

4.8.6 Code Coverage Discussion

Code coverage was reasonably high. However, particular packages were more difficult to test. For example, *victorious_secret.Units* and *victorious_secret.Behaviour* has the highest coverages rate. On the other hand *victorious_secret.Strategy* has a significantly lower coverage. This is due to the testing stubs used for testing was much easier to implement for Units and Behavior. Units and Behavior code used less Battlecode API, whereas Strategy was more integrated and testing stub has to be written for each Battlecode function that we use.

5 Group Work

Timothy Lim and Peter Efstathiou were selected as co-leaders of this project, responsible for code integration and project coordination. These tasks were later handed to Karlson Lee and they focused on leading each team for the defensive and offence robots respectively.

- Karlson Lee was responsible for organising meetings, taking minutes and scheduling. Our project was split into three segments - minimum, medium and super requirements, with team members working on separate streams depending on the stage of development.
- Mohammed Al-Hakim and Karlson Lee were in charge of implementing navigation strategies such as BugNav and BFS/DFS. They successfully implemented the navigation strategy for all robots.
- Robin Marle and Timothy Lim worked on the defensive bot using the turtle strategy that competed in the tournament as 'Victorious secret'. They were supported by Karlson Lee and Mohammed Al-Hakim.
- Peter Efstathiou and Will McGehee worked on the offensive bot using the turrets attacking formation strategy that competed in the tournament as 'There is no 'i' in A Team'.
- Will McGehee was in charge of the scouting strategy which each team implemented, when required.
- William McGehee and Mohammed Al-Hakim took charge of the automated testing suite and ensured that this requirement was met. They were supported by Karlson Lee.
- Robin Marle was in charge of Beating the MIT reference bot. He was supported by Peter Efstathiou and Timothy Lim.
- Peter Efstathiou and Will McGehee worked on the implementation of neural networks for implementing the machine learning super requirement.

  An outline of the task allocations are provided below as a table.

Figure 4: Task Allocation Final Revision

	Minimum			Medium		Super	
	Working Bot	Offensive (Napoleon)	Defensive (Turtling)	Beat MIT teaching assistant bot	Automated Testing Suite	Implemented Machine Learning	Implement Program Induction
Timothy Lim	✓		✓	✓			✓
Peter Lewis Efstathiou	✓	✓		✓		✓	
William McGehee	✓	✓			✓	✓	
Mohammed Al-Hakim	✓		✓		✓		✓
Karlson Lee	✓		✓		✓		✓
Robin Marle	✓		✓	✓			✓

6 Machine Learning Literature Review

The use of learning techniques in developing video game artificial intelligence is an area of active and abundant research. The often complicated but well-defined mechanics of video games present a useful environment in which to explore and evaluate different methodologies. Furthermore, complex games such as real time strategy games can be seen as simplifications of more complicated problems encountered in the real world, such as resource management, planning under uncertainty, adversary modeling and many others.[13]

However, real time strategy games such as Battlecode present a unique set of problems to AI research. Unlike board games such as chess, backgammon, and go, real time strategy games are orders of magnitude more complex in both the states a player will encounter and the actions a player can take. One method of measuring this complexity is to calculate the depth and branching factor of a game. For the purposes of this method, the depth of a game is the average number of rounds of play, while the branching factor is the average number of actions available in each round. It is estimated that chess has a depth of approximately 80 turns and a branching factor of 35 actions. A more complex game like go has a depth between 150 to 200 turns and a branching factor between 30 and 300 actions. Dwarfing both of these, Ontanon et al estimate that real time strategy games have depths of approximately 36,000 rounds and branching factors between 10^{50} to 10^{200}[13].

For this reason, artificial intelligences capable of playing real time strategy games well are notoriously difficult to create. For this reason, it is useful to briefly compare and contrast the different techniques that could potentially offer solutions to this problem as well as the efficacy of existing implementations.

6.1 Finite State Machines and Hard Coded Solutions

Finite state machines (FSM) are a method of computation in which a program can be in one of a certain number of states at any point in time. The state of the program determines the behaviour of the program and there are well defined rules governing the transitions between states. [3]

FSMs are the baseline of computer game artificial intelligence. They are straight forward to write, deterministic and follow a clear, easily understandable structure. Furthermore, the concept of state-specific behaviour often maps well onto the mechanics of video games.[3] The team's solution submitted to the Battlecode tournament is an example of such a FSM implementation.

FSMs do not fall under the category of machine learning. However, as they represent the default implementation of most video game AIs, they are important starting point when seeking to understand more complex Machine learning solutions.

6.2 Bayesian Networks

Bayesian Networks (BN) are directed acyclic graphs that represent the probabilistic relationships between different variables. These variables can be observations, unobserved latent variables or hypotheses. Variables are connected in the graph when the state of one variable is conditionally dependant on the state of another.

The probabilities of variables in the networks are computed using Bayes' Rule, in which the prior probability of a variable is weighed by evidence provided by connected variables.

BNs are designed to model uncertainty about a topic and are able to adjust their predictions once more evidence is obtained. In the context of games in which agents must act under incomplete information, this is a potentially significant advantage.

Examples of BNs being used in real time strategy game AI are presented by Dereszynski et al[5], Synnaeve et al[17] and Walther[21]. Dereszynski et al uses a Hidden Markov Model to predict

the most likely building order sequences while Synnaeve et al uses a Baysean network to predict the likelyhood of certain opening strategies. Both of these were implemented using replay data from the real time strategy game Star Craft. Meanwhile, Walther attempts to use BNs to create a resource management AI for a custom real time strategy game.

6.3 Evolutionary Learning Techniques

Evolutionary algorithms are a class of learning algorithm which seek to mimic the evolutionary pressures of natural selection. These techniques begin with a large "population" of potential solutions and a fitness function. In each iteration, the potential solutions are graded according to the fitness function. The most successful algorithms are selected and "chromosomes" (component characteristics) of each solution are mixed and matched to create a new generation of algorithms. Eventually, after many iterations of this process, one hopes that a particularly robust algorithm is produced.

A use of this technique in the context of real time strategy games is presented by Young et al[24]. They used an evolutionary algorithm to create an AI which decides which goals should be prioritised by its goal management system. This algorithm was able to improve on statically set priorities but was expensive both in terms of time and computational power required.

6.4 Supervised Learning

Supervised learning is a broad category of techniques in which labeled training data is used to infer a general function. Ideally this general function can then be used to predict the labels of future data points. A wide variety of supervised learning techniques exist and will not be cataloged in this section.

One obvious disadvantage of supervised learning approaches is that of obtaining and classifying data with the correct labels. Furthermore, a general real time strategy AI will need to operate strategically on a variety of levels from micro combat to macro resource management. Each problem is likely to require a specialized data set with its own distinct set of labels.

Weber et al provide a discussion of how data mining and supervised learning might be used by real time strategy game AI to make predictions of an opponents actions and strategy.[22]

6.5 Unsupervised Learning and Q-Learning

Unlike supervised learning, unsupervised learning is the process of inferring a function from a set of unlabeled data. This presupposes that there is an underlying structure to the unlabeled data set that can be used to produce a function of interest. As with supervised learning, unsupervised learning is a broad category with a multitude of algorithms and techniques falling under its purview.

The team researched a number of different unsupervised learning techniques but the one that seemed most suitable to Battlecode was a type of reinforcement learning called Q-Learning. After reviewing the characteristics of this technique the group decided to attempt an implementation of this technique.

It should be noted in section that several others have used Q-learning to implement real time strategy AI. Jaidee et al use a Q-learning based program to play Wargus, a Warcraft 2 variant.[19] A variation of Q-learning was also used by Google DeepMind to train an AI to play a variety of Atari games.

7 Final Product

7.1 Implementation Challenges

7.1.1 Navigation

Challenge: Choosing the right algorithm for pathing
Originally A* were implemented. However, one of the major limitation this year is that map will not be provided to our robots prior to the beginning of the match and only local information is provided for each robot with limited visibility, thus it was difficult to find a good heuristic. The other problem is that the use of the stack/queue for implementation of this algorithm took up a lot of bytecode/computational overhead and thus makes the robot much slower than the others.

BugNav was chosen as the better algorithm as means of navigation as the advantages outweighed the disadvantages for this year's Battlecode specification. The main advantage was the implementation of a reliable way of navigating around obstacles and enemies without engaging them directly. It was also important that it did not require the implementation of 'costly' data structures such as a stack/queue. It was very effective in most of the maps compared to A*/DFS/BFS and thus was used as the backbone for the other classes.

In hindsight, we have also realised that a pathing algorithm might not have been necessary for all robot units, as this year, competition robots can dig through walls and we could have a much simpler pathing algorithm to go from point A to B and dig if there is a obstacles. This was a strategy used by the top team in the tournament which came as a surprise.

Algorithm 2 and 3 briefly outline the pseudo-code of the implementation of the pathing algorithms. The outline is the simplified version where in our implementation, the team also included a list of prohibited directions to avoid repeatedly BugNav-ing the same position.

Algorithm 1 Bug Nav algorithm pseudocode

```
1:  procedure BugNav(Robotcontroller robot, MapLocation target)
2:      currentLocation = robot.GetCurrentLocation()
3:      if self.state in BUGGING then
4:          if robot.canMoveTo(Target)AND thedistanceisshortertotargetthanlastlocation then
5:              self.state = FLOCKING
6:      if self.state in FLOCKING then
7:          rc.MoveTo(Target)
8:      else
9:          Directiondir = robot.getDirection(target)
10:         Hug(dir, FALSE)
```

Algorithm 2 Hugging sub-function simplified pseudocode

```
1:  procedure Hug(Direction desiredDir, Boolean recursed)
2:      currentLocation = robot.GetCurrentLocation()
3:      if robot.canMove(desiredDir) then
4:          return desiredDir
5:      if HugLeft then
6:           rotate right until robot.canMove(RotatedDir)
7:      else
8:           rotate left until robot.canMove(RotatedDir)
9:      if !robot.canMove(RotatedDir)  OR  robot.IsOffMap(RotatedDir) then
10:         HugLeft = !HugLeft
11:         Return  Hug(desiredDir, TRUE)
```

Challenge: BugNav getting stuck

The problem with using BugNav is that in some situation, the robot will get stuck and unable to get out from the loop. For example, if the robot is put inside a box with a tiny gap as the exit, the robot will keep following the wall in clockwise direction and unable to identify the gap to exit the box.

Several hard coded solutions have been implemented to avoid this. Repeated movements were added to the list of previous moved locations and if repeated movements were detected then the robot will generate random movements in order to break out the loop. A 3x3 array is used to store the list of prohibited directions, thus was successful in avoiding local repeated movement for most matches.

7.1.2 Fleeing

As this year's game is based on zombie apocalypse scenario, with the zombies acting as a 3rd player that attacks both teams, the ability for our robots to flee was important as the majority of games in the tournament were decided based on who survived the longest.

As the team followed an Agile methodology, there were many iterations of the Fleeing whereby features were added incrementally. These are described below as test-driven development.

Challenge: Implementing Fleeing strategy

The major backbone for the fleeing is to decide which direction to flee to and feed that into the BugNav to actually do the movements, several conditions were considered in order to decide on the direction.

First, the robot checks for where the majority of the enemies are located and give an average direction opposite to that, i.e average of the enemies locations vs our current location. This is used to decide the best direction to run to. However, this also meant that if we are surrounded by enemies, due to the symmetry, the robot will decide not to move, so special cases are hard-coded for this.

Secondly, while the Archon is fleeing it will spawn inexpensive units every 200 rounds in order to distract the chasing zombies or enemies. This was test-driven development and proved to be very effective. The spawning of inexpensive units provided a way to escape completely from chasing units 85% of the time.

Challenge: Adding neutral units and resource picking to flee strategy

Finally, units that are fleeing were coded to ensure that they maximise the collection of parts on the map. This was again test-driven development which required many attempts to become effective. This feature was important as fleeing was the only time when the Archons explore the map for resources and acquiring neutral units. Additionally, the number of units and resources was crucial in tied break wins.

7.1.3 Scouting

Challenges: Implementing Scouting

In line with the algorithm presented in the Design Section, the team attempted to implement a Scout which would explore the map, identify relevant pieces of information (such as enemy location) and communicate that information to allied units.

It became quickly apparent that an effective scout strategy was very difficult to engineer. While exploring the map in a random way was effective in discovering map features and enemy locations, communicating this information in a productive way was more difficult. As units act independently, using scouting to inform the behaviours of a mass of military units relies on effective inter-unit coordination. Several attempts to coordinate unit behaviour were made but few

were effective due to the limited signaling range of units and the difficulty of using such signals to inform a larger-scale strategy.

Instead resources were diverted to developing an effective turret/scout combination. As discussed in the Design Section, a scout's larger sight radius can be used to increase the effective range of a turret. This dynamic can be used very effectively in a "turtling" strategy.

7.1.4 Signalling

Challenge: Implementing Signalling Strategy

The early experiments with this used one quad-bit (or crumb) [23] to signal an enemy had been spotted north, south, east, or west of the transmitting unit. Receipt of this signal caused all available units in the broadcast range to move towards the spotted enemy. This initially proved quite successful, with our robots overwhelming disorganised and isolated enemy forces. However, as our opponents became more organised this strategy became less effective. Consequently we experimented with expanded 2- and 4-crumb implementations, allowing us to broadcast substantially more information. Although the expanded information transfer was useful in coordinating behaviour, we ultimately decided not to deploy this in the final product. The 4-crumb system in particular was ineffective as it required four turns to transmit a complete message, which is a comparatively long time in the game context. During these four turns both transmitting and receiving units had to remain within range of each other, but their rapid movement often resulted in them failing to stay within range. Thus, the rate of successful message transmissions was relatively low. Furthermore the transmission and decoding of messages, already a relatively expensive operation, became too resource intensive. Ultimately the team decided to use implicitly coordinated behaviours, without the messaging system.

Challenge: Battlecode Coordinate Space

The maximum and minimum coordinate space used to describe the game board was limited to the range of -100 and 500. Thus, we partitioned the integer range into sections allowing us to describe both the coordinates of a feature, such as locations of enemy Archons, and a description of what that feature is. For example, a scout exploring the board may discover an enemy Archon. It would broadcast the (X, Y) coordinates plus an offset. A receiving unit would receive the signal and evaluate which offset had been used, then decode the coordinates and store them appropriately.

7.1.5 Automated Test Suite

The testing suite design was split into three stages therefore the implementation followed the same three design stages. The first stage of the testing suite in which a large number "headless" were run required the use of a bash script to enable the running of the Battlecode headless client ant. This script initially created an XML "replay" file for each game that was played but this was later optimised so that all the games would be in a single XML replay file. This optimization was very important for stage 2 of the automated testing suite as it allows for the collection and analysis of a greater number of games and reduced overhead time for opening and closing each file. The second stage of the testing suite was implemented using Python as it facilitates for the data to be extracted from the XML reply files and processed into a dictionary. The final stage was to create a script that is able to extract the relevant information, this was achieved with Python as well to utilise the dictionary created in stage 2 and also the python "matlibplot" library to plot graphs and charts that show the success indicators.

Challenge: Computing success metrics

At present, the suite produces statistics detailing the player's win percentage per map, the value of the player's army and economy at each round, and a measure of damage inflicted over time

(shown below). These measurements give the team more insight into the precise nature of each match beyond a simple win/loss value. Extending this test suite to examine other areas of interest will be discussed further in the future development section.

Figure 5: Graph of units per round on 4 maps

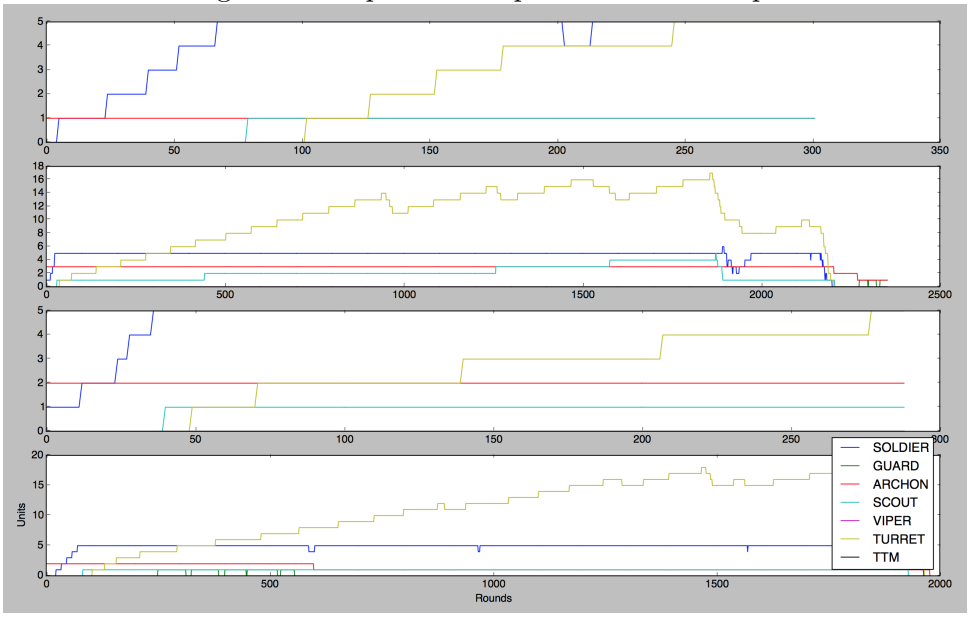
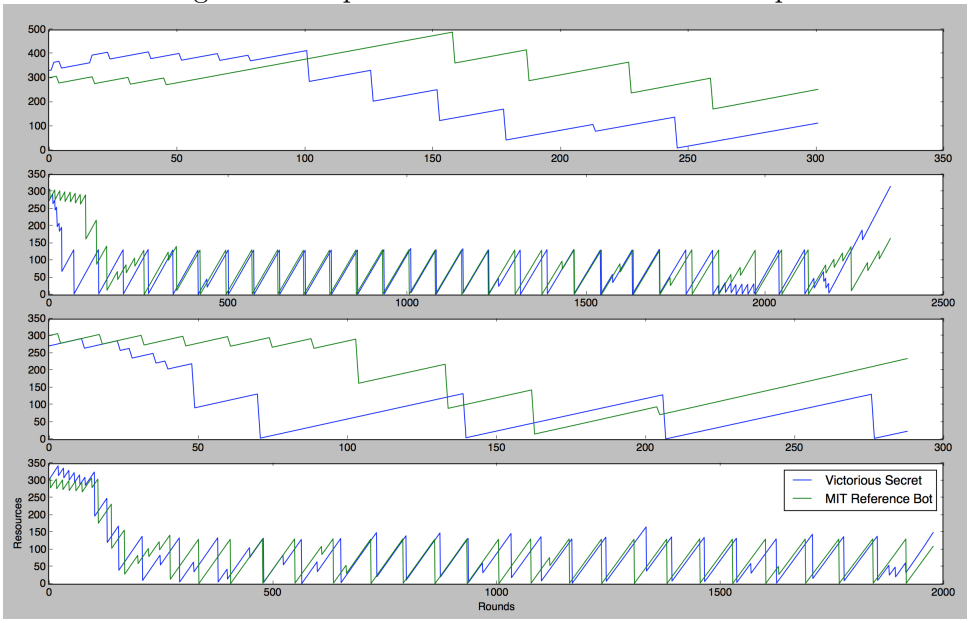


Figure 6: Graph of resources over time on 4 maps



7.1.6 Offensive and Defensive Bots

Both robots that entered the competition provided the basic functionally of producing units, engaging enemy units and implementing a strategy as necessary.

7.1.7 Offensive Strategy

The strategy started in a similar way to the defensive strategy, bringing the Archons together and defending them with Guards and Turrets in a circular formation. However, unlike the

defensive strategy, once the lead Archon had decided the attacking power of nearby units was sufficiently great, it would signal that all units should switch out of defensive behaviour and into an aggressive one. The lead Archon would also at this point signal a location that all units were to converge to.

The unique aspect of this strategy was the way the units coordinated their movements with only a limited ability to communicate. The desired movement pattern was to have two lines of units, a front line of Guards closely followed by a line of Turrets. This structure was designed to use the strengths of each unit to balance out their weaknesses. Turrets have the longest firing range and do the most damage of all units, but they are slow, have few hit points, and have a minimum firing range. This leaves them vulnerable to lightning skirmishes from fast units who move quickly and can get within the minimum firing range. In contrast, Guards have have the shortest firing range but one of the highest amounts of hit points. Thus, by placing the guards in front of the Turrets, the Turrets can use their superior range to attack enemy robots whilst the Guards prevent enemy robots from getting too close. The risk with this strategy is mainly from attacks from the flanks or behind.

Challenge: Coordination

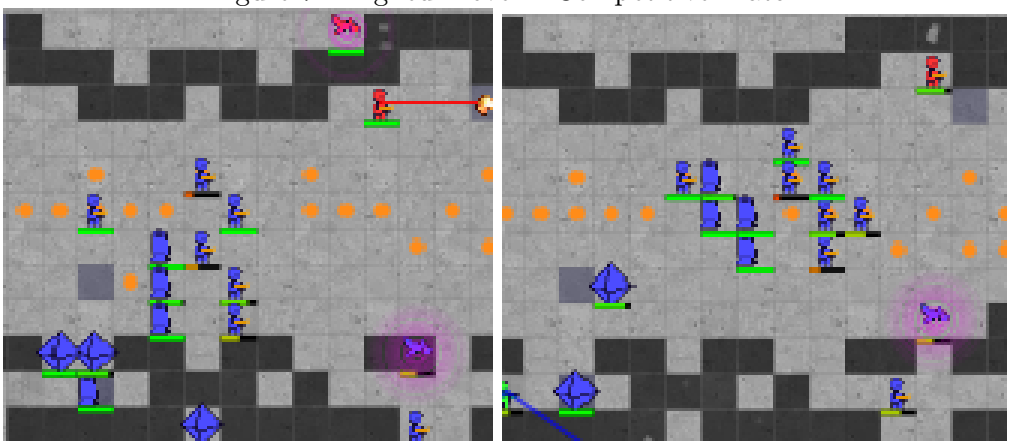
The coordination problem, of getting the robots to line up and then march forward in a consistent manner, was solved through a novel algorithm exploiting the fact that all robots share the same target rally point and that robots can mutually sense each other - i.e. if robot A can sense robot B, then robot B can sense robot A. Each robot observes the nearby turrets that are within its sensible range and identifies the robots that are closest to the target rally point, designating them the leading edge. It then identifies the set of map locations that share the same radius as the leading edge, selects the nearest free one, and moves towards this. If there are no points free, the robot moves towards the line, picks a direction randomly, and follows the edge of the wall until it finds a free location. The robots on the leading will not move if they can sense any robots that are not also on the leading edge, and that are not currently able to move. Thus, they wait until everyone is lined up and ready to go. Once everyone is ready, they all move one step closer to the objective at the same time. Guards follow a similar process, except they always try to move to be two grid spaces closer to the objective than the leading edge of turrets. Thus, they will only advance after the Turrets have completed a march.

Algorithm 3 Aligned Move

```
1:  procedure ALIGNED MOVE
2:      leading edge ← Turret closest to rally point
3:      if self in leading edge then
4:          if robots in sensor range = leading edge then
5:              if robots in sensor range are all ready then
6:                  move to rally point
7:          else
8:              free space ← free space on leading edge.
9:              if free space is empty then
10:                 if self is adjacent to leading edge then
11:                     move along leading edge
12:                 else
13:                     move to leading edge
14:             else
15:                 move to closest free space
```

This strategy was supposed to be particularly effective against Turtling strategies, where we know the opponent has fortified a single location as we know that they will not be able to perform a flanking operation. In practice matches between our Attacking and Defending strategies, this achieved good results. However, in the competition matches this strategy performed poorly. Partly this was due to the strategy getting flanked in practice as opponents did not choose the

Figure 7: Aligned Move in Competitive Match



Turtling strategies with sufficient frequency and partly as a result of unfavourable movements from the unpredictable Zombie opponent. As such, the strategy won only one of the five competitive matches played in the final round. The images below show two frames from one of the competitive matches that demonstrates the Turrets and Guards moving from left to right towards the enemy unit (in red).

7.1.8 Defensive Strategy

The first robot 'Victorious Secret' entered as the defensive bot implementing a successful turtling strategy. The bot in the first round would select an Archon as the leader through a calculation that decides which is least likely to be near a zombie den, and therefore in the 'safest' area. The other Archons would then BugNav towards the leader as fast as possible and a home base is established. A perimeter is then established around this 'homebase' and Archons build as many units as possible, exhausting the available resources in a first-come-first-served scheduling procedure.

Challenge: Turret Deployment

The most important algorithm in the Defensive strategy was the deployment of the turrets around the Archons, or in some cases the 'homebase'. In fact, when the turrets are 'unpacked' they can attack incoming opponents but not move. In order for turrets to be moved they need to be 'packed'. Thus, every round turrets need to be 'packed' or 'unpacked' depending on their location. Through extensive experimentation, it found that a checkerboard pattern was most effective. The pseudo-code for turret deployment is presented below:

Algorithm 4 Turret deployment algorithm pseudocode

```
1:  procedure TurretDeploy
2:      int radius = 2
3:      if turret not on checkerboard then
4:          if turret unpacked then
5:              pack
6:      else if  then
7:          if position available on checkerboard then
8:              move to closest available position
9:          else increment radius
10:     else
11:         if position available on checkerboard then
12:             move to closest available position
13:         else increment radius
```

7.1.9 Other Implementations

Challenge: Archons positioning

In the later stage of the competition, the team realised the Archon positioning at the beginning of the game was very important. Thus, a leadership selection code was implemented where in the first round the bot would select an Archon as the leader through 'clever' calculations on which is least likely to be near a zombie den. The other Archons would then BugNav towards the leader as fast as possible and a home base is established. The bot was then able to spawn many soldiers and turrets.

7.2 Machine Learning

The team decided to use a Q-learning framework to pursue the learning objective in which the relevant Q-function was implemented as a neural network.

The objective was to learn an optimal strategy policy against an opponent with a static policy. Specifically, the team aimed to teach a robot to flee from an opponent. In keeping with the objectives of the project, the opponent was the MIT supplied Reference Bot. Although there is an element of randomness in the strategy of the Reference Bot, the overall strategy is static. Thus, although there will be inevitable noise in our state transition estimates, the policy to learn remains fixed.

The team pursued this in two parallel streams, sharing operational and knowledge resources, but differing in the technology used. The two machine learning packages used were Tensorflow, a python library developed by Google, and Neuroph, an open source Java library.

Challenge: Removing Bytecode Limit/Importing External Packages

The Battlecode server places two restrictions on the robots. Firstly, each robot has only a limited number of computations it is allowed to perform, referred to as bytecode limit. Secondly, the ability to import external packages is severely curtailed.

The restricted computation available prevents any form of online learning. Thus, when using TensorFlow the team used offline learning, obtaining state and reward information from stored replay files.

When using Neuroph the team altered the source code of the battlecode server to lift the two restrictions. Thus, any robot developed this way would not be competition legal. However, this meant they were able to explore larger and more sophisticated neural nets during in game play. Though the bot developed in this way would not be submittable, the insights gained regarding the performance of different network architectures would feed back and inform development in TensorFlow.

Both approaches used the same basic structure, playing a simplified one-on-one combat game, with a small number of state variables and limited available actions.

7.2.1 TensorFlow Development Process and Methodology

The first step in implementing a Q-learning solution was processing game replay data into a form useful for training. Using the previously discussed headless client and testing suite, the team was able to run large numbers of matches, and convert the resulting xml replay files into python readable lists and dictionaries.

Once the data was converted into a python readable form, second layer of data processing was used to generate state information from the replay data. Each state was comprised of player health, X_POSITION, Y_POSITION, ENEMY_X_POSITION and ENEMY_Y_POSITION. We decided that this was the minimum amount of information that would be needed for a Q-learning algorithm to learn a flee behaviour.

It is important to note that each state corresponds only to one round in a match. It has been discussed by others that a more useful approach is for each state to represent a series of state action pairs, as rewards are often only received from a series of actions.[20] However, in order to keep the design as simple as possible in the first instance, we mapped each state to one round of play.

Once state information has been captured, we are left with a series of state action pairs. We then calculate the reward received for these state action pairs. The exact specification of the reward function is often a subtle art. However, for the sake of experimentation we chose a relatively arbitrary reward function: (player_health in state t2 minus player_health in state t1) plus 1. This function rewards the unit for not loosing health on the next round. This immediate reward was added to a discount factor times the predicted q value of the next state provided by the neural network.

This reward function is run on every state/action pair that appears in the replay file. We now have a data set that can be used to train a neural network to represent a Q-function.

The neural network went through two different designs. The first had only one layer. The second two. The one layer network was used as a starting point as it was simpler. A simple feed forward network was used in which each node is a simple matrix manipulation. The inputs are the state information and the action information each in the form of a vector. The network generates a single output being the Q-value of the state action pair.

In order to implement back propagation and training, TensorFlows Adam Optimizer was used. Once the network was trained, the adjusted weights were written to a text file.

The java player is responsible for reading in the trained weights from the text file. These weights are then used to calibrate the robots internal Q-function. This Q-function is then used to choose an optimal action at each state the robot encounters

Just as in any unsupervised learning program, there is a trade off between exploitation and exploration. At first the robot moves completely randomly to generate data. Once the network is trained, the robot will move randomly 30% of the time and 70% of the time according to the estimated Q-value of the next state.

7.2.2 TensorFlow Results

With both the single and double layer network, the behaviour converged very early to a simple behaviour. The unit would move in the opposite direction of the enemy, but was unable to change directions. Thus the unit would eventually run into a map boundary where the enemy unit would easily destroy the target unit.

We experimented with different network structures and hyper parameters but were unable to capture more complex behaviours. It is likely that such simple behaviour is due to the way state and reward was structured. In the game rewards are often received for a series of actions. For example, if a unit is corned the best strategy is to move in some direction towards the enemy. The network structures tested were unable to capture this long term dependence, and so could not overcome the short term negative reward required to reach the superior future state.

7.2.3 Neuroph Development Process and Methodology

Neuroph provides implementations of many core network structures and learning rules. In this case the team decided on a Multi-Layer Perceptron (MLP). Alternatives included using Adeline, Convolutional (CNN) or Hopfield nets. The team rejected Convolutional networks, as used by Deepmind with great effect to play several Atari video games, as the small state space did not require the compression and feature extraction that a CNN provides. Hopfield and other single layer nets were rejected as we wanted a multi-layer architecture that would be capable of expressing complicated decision rules.

The choice of activation function is a key determinant in the efficiency of learning. Selecting a sigmoid, hyperbolic tan, or non-smooth functions like the ReLU, was a theoretical challenge. The ReLU function has been shown to perform as well or better than the hyperbolic tan function in experimental work [9], where as the sigmoid is strictly between zero and one, corresponding to the action mapping that the model is supposed to generate. Due to the difficulty of selecting between activation functions, experiments were performed with multiple activation functions.

As the Neuroph implementation lifted all the constraints imposed by the official game, the state space, network structure, and action sets were all larger than those attempted with Tensorflow. The state was SELF_HEALTH, SELF_DELTA_HEALTH, SELF_CORE_READY, SELF_WEAPON_READY, ENEMY_HEALTH, ENEMY_DELTA_HEALTH, ENEMY_CORE_READY, ENEMY_WEAPON_READY, DISTANCE_TO_ENEMY, DIRECTION_OF_ENEMY, and an extra state ACTION for the action taken.

The actions allowed were DO_NOTHING, MOVE_NORTH, MOVE_NORTH_EAST, MOVE_EAST, MOVE_SOUTH_EAST, MOVE_SOUTH, MOVE_SOUTH_WEST, MOVE_WEST, MOVE_NORTH_WEST, ATTACK_TARGET. Since the games were one-on-one, there is only one available target to attack.

To ensure that there was sufficient exploration of the state space the robot only used the network to decide which action to take approximately 50% of the time. The remaining times a random action was chosen, with a 50% chance of being DO_NOTHING and then uniformly for the remaining actions.

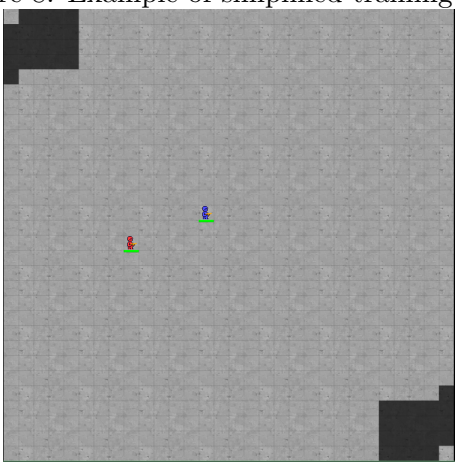
A number of different network structures were tested. A table summarising the networks is given below.

Table 1: Network Structures

Name	**Activation Function**	**Number of Layers**	**Nodes Per Layer**
2 Layer	Tanh	2	11, 9
2 Layer	ReLU	3	11, 9
3 Layer	Tanh	3	11, 11, 9

Games were played on one of 10 maps. Each map was consistent in that they were one-on-one matches between soldiers. The placement of the soldiers was varied between all maps, both in terms of the direction and distance between the robots. There were no other units on the map. The maps were kept relatively featureless, to further simplify the problem.

Figure 8: Example of simplified training map



Simulations were run in batches of 100 games, 10 per map. The network was updated at the end of each batch with propagation learning. The history was randomised, batch processed, and

limited to 100 iterations. Batches were not re-used across training cycles. The simplified nature of the map meant that each game was played relatively quickly.

7.2.4 Neuroph Results

After completing each round of playing and training with the partially random bot, a further 10 games were played with a completely non-random bot. No learning was conducted on these games, providing clean out of sample results. After each training round we computed the mean squared prediction error. The results below present both the in- and out- of sample data.

Figure 9: Prediction error, in-sample

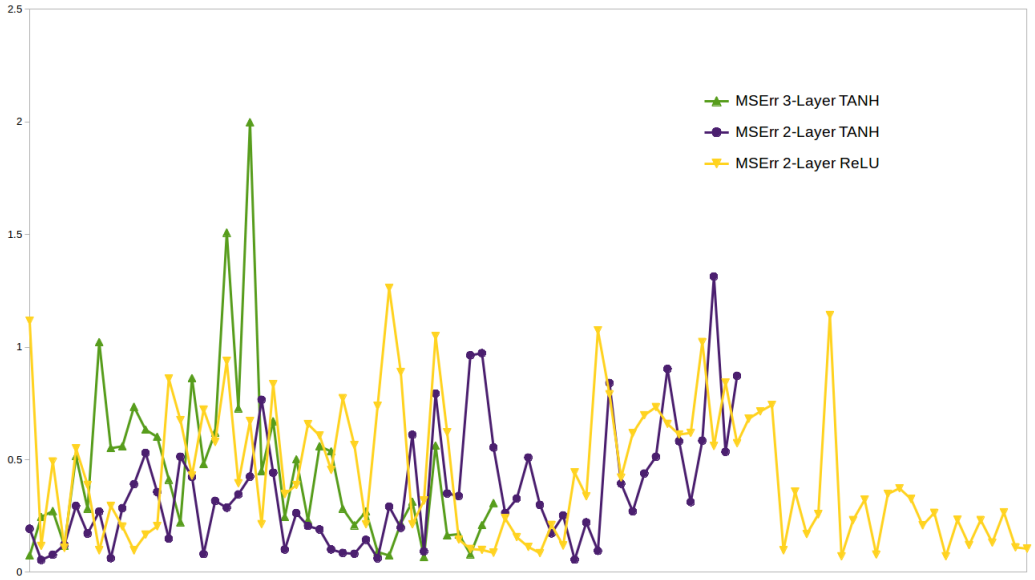
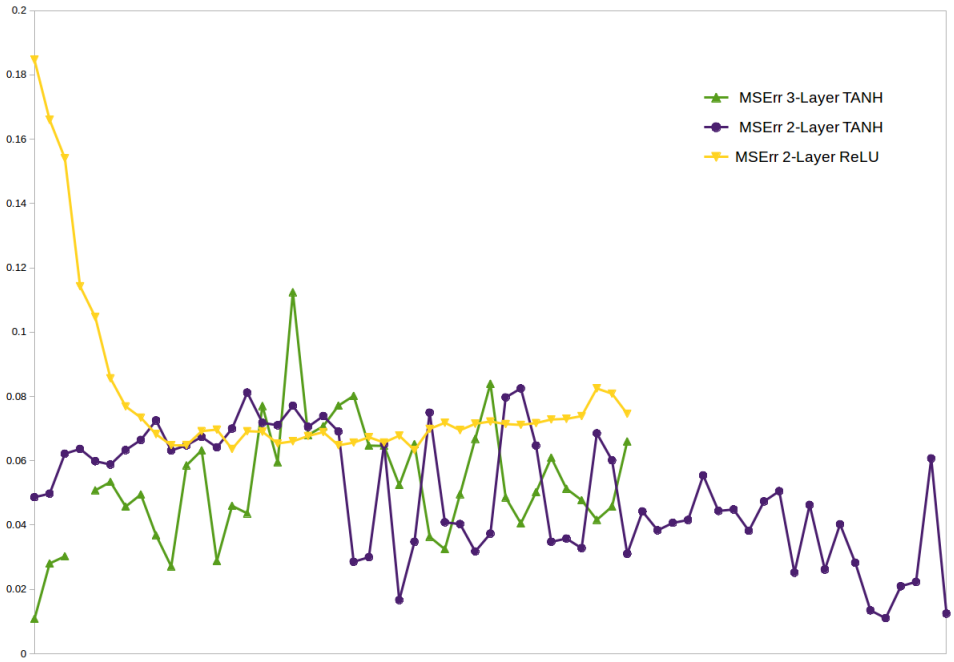


Figure 10: Prediction error, out-of-sample



The prediction errors show the expected convergence behaviour only for the two layer ReLU network. After each round of training the accuracy improves until it reaches a plateau. In

contrast, the three layer networks both started with lower mean squared errors but failed to make much, if any, improvement.

As the results above show, the neural nets managed some degree of convergence, and the turns survived steadily increased. However, the nets quickly plateaued, unable to improve their Q-value predictions. Despite the encouraging early results, further work is required to select optimal activation functions and network structures.

7.3 Final Product Evaluation

The project as a whole has been successful. The team has been able to achieve all of the minimum and medium requirements as well as compete in the MIT Battlecode competition reaching a reputable top 60 position. The super requirements pushed the team to new frontiers in computing success indicators, creating the testing suite and implementing machine learning techniques.

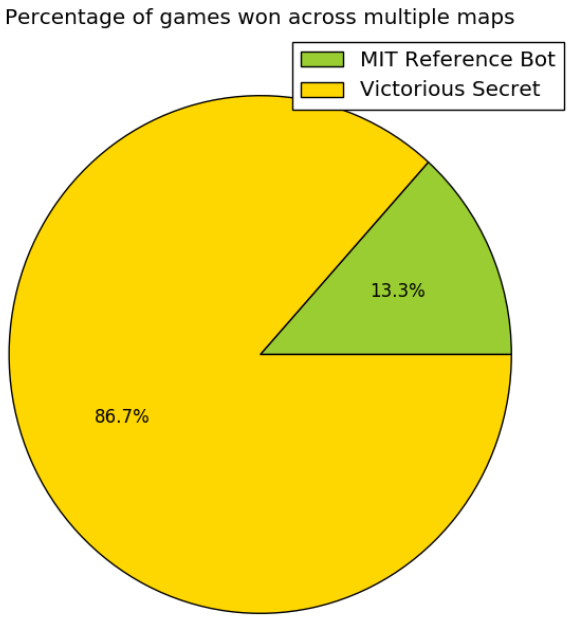
7.3.1 Comparative Performance

There were three core goals for this project. The first was to enter a working bot to the official MIT Battlecode competition. The second was to achieve a 70% win rate against the MIT reference bot. The final objective was to use machine learning techniques to develop robot strategy.

The first goal was met with two bots entered into the final competition. One used the offensive strategy and the other used the defensive strategy. Although we submitted working bots, the performance against other opponents was disappointing. The offensive strategy won only one out of five games. The defensive strategy performed better, but still only placed in the middle of the rankings. Therefore, it is clear that there were deficiencies in our strategies that could have been improved.

After the relatively disappointing performance in the competitions the second goal was achieved well, with the robot defeating the reference player in over 80% of matches. This was substantially above our target, and demonstrates the progress that was made between the first and second goals.

Figure 11: Pie chart with wins against MIT reference bot



The final objective of using machine learning to develop robot strategy was only partially met. Although multiple network structures and technologies were tested, the ultimate performance of the networks was substantially poorer than even a simple human designed strategy. The networks

quickly converged to comparatively poor results, despite the fact that they were trained on a simple one-on-one combat scenario. Despite this, the team remain confident that with more time and computing resource it should be possible to train a network that could solve this.

7.4 Further Development

7.4.1 Strategy Transitions

Transitions between defensive and fleeing strategies were successfully implemented and used to great effect during the tournament. However, with more time it would have been ideal for more complex switches between all three strategies of defense, offense and flee. These will provide scenario based switches that depend on the current state of the battle which allow the transitions to implement the most effective strategy to ensure the success of the game.

7.4.2 Zombie Den Annihilation

During the project, the team had a limited amount of time to test and implement strategies that we did not consider to be essential to the success of the robot in the tournament. However, one of the strategies that were created but not fully tested was the zombie den annihilation. This was used by the top performing teams to destroy zombie dens before zombie's were spawned by sending a scout to find all the zombie dens and then send soldiers/turrets to destroy the dens. In future development it would be beneficial and contribute greatly to the robots performance in winning more matches if this was implemented, tested and added to the robot.

7.4.3 Improved Offensive Strategy

The offensive strategy implementation was largely unsuccessful. The unwieldy and complex movement of the attacking force left it vulnerable to flanking attacks. Thus, a more flexible and dynamic offensive strategy should be developed. Superior scouting and intelligence gathering are also importent for optimising the ordering of attacks.

7.4.4 Improved Defensive Strategy

The defensive strategy relied heavily on being able to bring all the Archons together. In the events where this was not possible, it was common for one or more Archons to be starved of resources. This would prevent them from building any defensive units, leaving them extremely vulnerable to attack. In future development the team suggests the implementation of smarter sharing of resources between Archons can ensure a fair number of units are spawned from each Archon.

7.4.5 Improved Signalling/Scouting Strategy

In the competition, scouts were mainly used to improve the vision of the turrets as their attack range is greater than their sight range, discover enemy units and identify zombie dens.

However, after watching other teams' implementation during the course of the Battlecode competition, the scouts could have been used as decentralised headquarters to coordinate attacking units. Multiple scouts were also used by other teams to generate a map of each game. During one of the games we witnessed in the tournament, scouts were used to send spam messages to opposite team, thus leading to the opposing losing control to their units as their communication became scrabbled. Another further development would be to use scouts to lure zombies away from our units or towards an opposing teams.

7.4.6 Machine Learning Implementation

The project presented an interesting way for the team to interact with various machine learning techniques. These techniques are often complex and require a certain level of experience to implement successfully. Any attempt to develop this are further would likely require more work in the theoretical and technical background before a successful solution could be found.

8 Appendix

8.1 Progress

8.2 Log

Meeting 1

Attended: Pete, Karlson, Mo, Will

Location: Room 219A

Time: 23rd Nov, 12pm

Agenda -

1. Team Name?

Victorious Secret

2. What to learn? areas for preliminary reading, lectures on battlecode, logics, path-finding algorithms, maybe some MLs?

The next 2 weeks, the 10 lectures on the battlecode websites

Path-finding algorithms - A* search, Dijkstra

AI planning - SATPlan, GRAPLAN, POP

Q learning - Reinforcement learning

Review winning bots

9th Dec - Get preliminary readings done and make a basic bot

Split the learning after 2 weeks into logic , pathfinding or planning, ML, so each member has his edge

3. Timeline, when we will finish preparations and start cracking on it? And aim to finish by when?

Dec - Battlecode registration opens

9th Dec - Get preliminary reading done and the basic bots done.

Around 5th Jan (Last year figure) - Competition begins, Spec release

12th Jan (Last year figure) - Sprint tournament deadline

20th Jan (Last year figure) - Seeding tournament deadline

27th Jan (Last year figure) - Final Submission deadline

6th Feb (Last year figure) - Open tournament deadline

- Sprint Tournament - One week after spec release, you're given a chance to win small prizes in this tournament. The goal is to get an idea of the meta-game, and a chance to test your bot prototypes.
- Seeding Tournament - One week after the Sprint Tournament, this tournament determines your positioning in the Qualifying Tournament.
- Qualifying Tournament - One week after the Seeding Tournament, this tournament determines the contestants going into the Final Tournament, and showcases the final strategies of all the competitors. Final submissions must be in by this Tournament. Historically, the top 16 teams have advanced to the Final Tournament.

- Newbie Tournament - As part of the Final Tournament, the top newbie teams also compete for a prize.
- Final Tournament - On January 31st, the top teams compete for glory and fame.

4. Supervisor?

Go for Stephen Mugglement - Send the proposal. arrange a meeting on Thursday afternoon.

5. Get Tim to make a repo

Next meeting - Thursday with Stephen Muggleton, plus discuss on milestones

Meeting 2

Attended: Pete, Karlson, Mo, Will, Tim, Robin

Location: MLC, Meeting room 2

Time: 16th Dec, 4 pm

Agenda -

1. What to do during Christmas?

1. MIT Lectures - 9 of them, put them in resources
2. Strategies report - will put them into resources
3. at least able to use bug nav

1. Know Bug nav, A*,
2. Good coding style
3. Split the team into two and look at two bot
4. Download the bots and see how it works

2. Setted up group calendar
3. Have a mock battle!!! A vs B
4. Next meeting with Stephen? After c++ exams, Karlson will arrange

A Team- Pete, Will and Mo

B Team - Karlson, Robin and Tim

Pete will download all the resources and put them into the Wiki page

Meeting 3

Attended: Pete, Karlson, Will, Tim, Robin, Supervisor

Location: Stephan's office

Time: 14th Jan, 11 am

Agenda

1. Met up with Stephan and gave him the update and overview of the sprint bot that Pete made.
2. Asked Stephan on pointers on next step

1. How to implement ML into our game?
2. Inductive programming? Suggestions from Stephan

3. On Pete's bot

Plan

1. Everyone make a simple bot in one day.
2. Everyone make a map
3. Focus on the competition then maybe work on ML on long term

Meeting 4 - Mock battle

Attended: Pete, Karlson, Will, Tim, Mo

Location: Lab

Time: 15th Jan, 11 am

Agenda

1. Everyone's bots fight each other
2. Tim's bot won
3. Discussed on the strategies and how to split the work

Plan

1. Karlson and Mo works on Flee
2. Will works on Scout
3. Pete works on combat
4. Tim works on turtle

Meeting 5 - Seed Tournament Post mortem

Attended: Pete, Karlson, Tim, Mo, Robin

Location: 219A

Time: 21st Jan, 1 pm

Result

Great work, got up to top 50-60 out of 260 teams. Only lost the last round by a tiny pixel of health (only able to seen by zooming into the chrome browser)

Agenda

1. How to improve the bots

1. Team now split into offensive and defensive strategies
2. Pete/Will handle offensive bot, the rest handle the defensive bot

    2. On the defensive bot:

        a, Kill zombies den when you are turtling if dens is closed enough

        b. Get the turrets and soldier higher to more attack and small radius in turtle

        c. Improve gathering Archons at the beginning to build stronger base (including pick up part, pick archons, identify zombies dens)

        d. Flee-ing need to switch back to defence

    3. On the offensive bot:

        a. Guard micro - dont move towards enemies

        b. mixed in some vipers?

        c. identify zombies dens

d. Use vipers against turtling
e. Attack big zombies first
f. units circle archon when finished fighting

4. Test Soilder or Guards - which one is more effective?

Meeting 6 - Report 1 Discussion - 30mins

Attended: Pete, Karlson, Tim, Mo, Robin, Will

Location: 219A

Time: 1st Feb, 1 pm

Agenda

How to split the work for the write up

1. Karlson, Mo - Version Control - Deadline - Tuesday
2. Robin, Will - Dev Method - Deadline - Tuesday
3. Timothy, Pete - Requirements - Deadline - Tuesday

Requirements?

1.

1. Minimum - Working Bot - done
2. Automated Testing Suite

2. Good - Beat MIT teaching assistant bot
3. Super

1. Machine Learning with Flags
2. Neutral Network

Meeting 7 - Work

Attended: Pete, Karlson, Tim, Mo, Robin, Will

Location: 219c

Time: 9th Feb, 1 pm

Agenda

All communications on Slack - everyone install Slack to do work communication

Code standard

- Standardise the code standard in the project (Mo/Karlson uses statics all the time, the rest uses object)
- Re-organise the code a little bit
- Use generic methods as much as possible

Testing

Jobs (2 weeks) -

- Refractor Codebase to make generic strategy class and static (4)

- use Javadoc
- comments the code

- unit testing
- Test Client (2)
- Automate running of the MIT headless client
- Change parameters in maps and player starting locations
- Extract results from XML replay, what results
- who win and lose?
- error

Weekly Tuesday meeting - 3PM
Weekly Tuesday Hackathon - Tuesday from 11am
Meeting 8 - Work
Attended: Pete, Karlson, Tim, Mo, Robin, Will
Location: 219C
Time: 16th Feb, 3 pm
Agenda: Everyone Reports on progress

1. Work done so far

1. Karlson/ Mo separated BugNav from Flee and started on the javadoc on BugNav
2. Robin and Tim cleaned the code, but still have some work to do
3. Will done the Javadoc and Pete started on the attack code.

2. Get refactoring done today and everyone do it together - (Pete, Timothy and Robin)
3. Start headless clients - (Mo, Karlson, Will) by Friday
4. Extra interim meeting on Friday to check progress

Meeting 9
Attended: Pete, Karlson, Mo, Robin, Tim
Location: 219C
Time: 23rd Feb, 3 pm
Agenda: Progress, Meeting with Stephen, Report 2

1. Refactoring is completed
2. Automatic running of the MIT headless client - done
3. Beat the reference bot-

1. Ask for reference bot
2. get other example bot
3. Scout - identity zombie den (Pete and Karlson)
4. Fight - Kiting and Viper (Pete and Karlson)
5. Fight - Moving turtle (Tim and Robin)
6. BFS vs BugNav and Dig (Mo and Karlson)

Meeting 10

Attended: Pete, Karlson, Mo, Will, Tim Robin

Location: 219B

Agenda: Unit-Testing, Report Two

Summary

1. Tim and Robin done some tweaking on the turtles
2. Pete done quite a few Unit-testing , 50% coverage in his codebase
3. Will and Mo finished Headless client

Report 2

Tournament Report - Progress

1. related to schedule in the report
2. make revised schedule
3. headless client - Mo and Will
4. bot performance - Pete and Mo

Testing strategy (Everyone read up on this), so we can discuss in the next meeting

1. summary of testing result
2. coverage achieved
3. Everyone stopped what they are doing, do code coverage
4. 24-29th Hack Week Feb for the group project and finished for report 3

Meeting 11 -9th may 2016

Attended: Pete, Karlson, Mo, Will, Tim Robin

Location: 219A

Agenda: Final report

1. Team split into two, Pete and Will to work on the using Q-learning on the robot
2. The rest works on the report
3. Mo is doing introduction, Group work and Specification
4. Robin is doing methodology
5. Karlson works on the design and final product
6. Tim has interviews and will join back at the beginning of the week

8.3 Battlecode 2016 Overview

Battlecode is a turn-based strategy game whereby the goal is to destroy or outlast the enemy team. This year the game was a zombie invasion.

8.3.1 Zombies

Zombies periodically spawned from "dens" scattered around the map. The zombies move toward and attack the nearest player-controlled unit. More and stronger zombies are spawned as each game progresses. Units that were killed by zombies are re-spawned as zombies, and will attack their former allies. [8]

There were four kinds of zombie units. [8]

Figure 12: Zombies

8.3.2 Player unit types

Figure 13: Archons

Archons: Each team starts with a number of Archons, a leader unit which can construct other units. Whichever team loses all their Archons first loses the game. All the other units are built by Archons. [8]

Figure 14: Soldier

Soldiers: cheap but mediocre combat unit

Figure 15: Guards

Guards: melee unit with a lot of health

Figure 16: Turrets

Turrets: expensive and stationary units with a long-ranged powerful attack. Turrets can transform into a mobile TTM (Turret Transport Mode), which can move around but cannot attack until it transformed back into a turret. Transforming takes several turns.

Figure 17: Scouts

Scouts: cannot attack, long vision range, could fly over obstacles

Vipers: can infect other units, dealing damage over time and turning them into zombies

Figure 18: Vipers

8.3.3 Resources

Units were built by Archons from a resource called "parts." Each team received a passive income of parts, and there were also parts distributed on the ground that can be collected by Archons. There were also "neutral" units sitting on the map that you could recruit to your team by touching them with an Archon.

8.4 Progress

Minimum Requirements			
Task	Priority	Sprint	Status
Implement Bug Nav for pathing (Offence/Defence) [8]	5	1	completed
Implement Fleeing (Defensive)	5	1	completed
Implement Turtling (Defensive)	5	1	completed
Implement Napoleon (Offensive)	5	1	completed
Attack Micro (Offensive)	5	1	completed
Implement scouting and messaging system	4	1	completed
Switching between strategies	4	2	completed
Gathering Archons at the beginning to build stronger base	3	2	completed
Implement Archon specific micro behaviour	4	2	completed
Testing whether Soldiers or Guards are more effective	2	2	completed

Medium Requirements			
Task	Priority	Sprint	Status
Automate running of the MIT headless client	5	3	completed
Change parameters in maps and player starting locations	2	3	completed
Extract results from XML replay files	4	3	completed
Output description of winning conditions during the game	3	4	completed
Improve pathing using DFS or BFS and compare to Bug Nav	3	4	completed
Implement Viper Strategy	1	4	uncompleted
Identify Zombies Dens using scouts	4	4	uncompleted
Implement kiting in attack micro	2	4	uncompleted
Pathing and formation (e.g moving turtle, etc)	5	4	completed

Super Requirements			
Task	Priority	Sprint	Status
Literature review of relevant machine learning techniques	3	5	completed
Compute success metrics	5	5	semi-completed
Implement meta-level program for offline learning	3	6	discarded
Implement program induction techniques	5	6	discarded
Implement neural nets and reinforcement learning for combat micro	3	7	completed

Table 2: Sprint Schedule

Sprint	Schedule	Minutes
Sprint1	10th Jan - 17th Jan	- Mo and Karlson successfully implemented the pathing algorithms - Will and Pete started on the scouting - Submitted a trial bot into the sprint tournment
Sprint2	20th Jan - 27th Jan	- Mo and Karlson finished on the Flee-ing - Tim and Robin finished on the turtling strategy - Need to implement strategy switching, assign to Robin and Tim - Pete and Will completed scouting and started on the attack micro - The team submitted two bots into the competition
Sprint3	5th Feb - 13th Feb	- Submited Victorious Secret to Final tournments - Tim completed gathering of Archons at the base - Mo and Karlson added neutral units activation and picking up resources to Archon behavior - Tim and Robin put more turrets into the turtling - Pete completed Napolian strategy
Sprint4	20th Feb - 29th Feb	- Will done on scouting, improving turret's vision to shoot further - Pete made a offensive bot, not turtle - Mo and Karlson made a simple scouts strategy that identify zombies, but it is not good enough to be submited to the tournments - Will and Mo started on the MIT headless client as well as extracting results from replay files - Karlson worked on the DFS/BFS and compared to the original path-ing strategy
Sprint5	1st March - 8th March	- Tried to implement kiting in attack micro, uncompleted, Pete and Will - Tried to implement viper strategy, uncompleted, Karlson and Mo - Will and Pete started on Literature reviews on using Machine Learning on Battlecode - Everyone started unit-testing their code using IntelliJ - Mo and Karlson reached test coverage 81% - Pete and Will reach coverage 60% - Robin and Tim reached coverage 73% - Other tasks in Sprint 5 suspended due to Report Two Deadline
Sprint6	10th March-17th March	- Suspended due to Exams for Mo, Timothy, Robin - Will and Pete started on using Machine Learning in Battlecode
Sprint7	8th May - 13th May	- WIll and Pete decided to use reinforcement learning in Battlecode instead of program induction - Final report write up - Split the team into two, with Pete and Will to finish super requirements(Neural Net Learning) - The rest works on the final reports

References

[1] Boris Beizer. *Black Box Testing: Techniques for Functional Testing of Software and Systems (Computer Science)*. Wiley, 1995.

[2] Ross Cowan. *Roman Battle Tactics 109BC - AD313*. Osprey, 2007.

[3] Glenn Seemann David M Bourg. *AI for Game Developers*. O'Reilly Media, 2004.

[4] MIT Battlecode Dev. Battlecode 2016 game specs:, November 2016.

[5] T.D. T.-T. Hoang E. Dereszynski, J. Hostetler and M. Udarbe. Learning probabilistic behavior models in real-time strategy game. 2011.

[6] Chris Sims Hillary Louise Johnson. *Scrum: a Breathtakingly Brief and Agile Introduction*. Dymaxicon, 2012.

[7] Jeff Langr. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf, 2015.

[8] Greg McGlynn. Battlecode 2016 post-mortem: future perfect, Feb 2016.

[9] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Frnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010.

[10] Fidelis Perkonigg. Module 530: Software engineering practice and group project - lecture 7 - prototypes and agile methods.

[11] Fidelis Perkonigg. Module 530: Software engineering practice and group project - lecture 8 - software testing.

[12] Kim Hamilton Russ Miles. *Learning UML 2.0*. Dymaxicon, 2012.

[13] Alberto Uriarte-Florian Richoux David Churchill Mike Preuss Santiago Ontanion, Gabriel Synnaeve. A survey of real-time strategy game ai research and competition in starcraft.

[14] Ken Schwaber. *Agile Project Management with Scrum (Developer Best Practices)*. Microsoft Press; 1 edition, 2004.

[15] Armando Solar-Lezama. Combinatorial sketching for finite programs. Technical report, IBM T.J. Watson Research Center, 2003.

[16] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, B.S. (Texas AM University), 2003.

[17] G. Synnaeve and P. Bessiere. A bayesian model for opening prediction in rts games with application to starcraft. 2011.

[18] R Rivest T Cormen, C Leiserson. *Introduction to Algorithms*. Dymaxicon, 2012.

[19] Hctor Muoz-Avila Ulit Jaidee. Classq-l: A q-learning algorithm for adversarial real-time strategy games.

[20] David Silver-Alex Graves Ioannis Antonoglou Daan Wierstra Volodymyr Mnih, Koray Kavukcuoglu and Martin Riedmiller. Playing atari with deep reinforcement learning.

[21] Anders Walther. Ai for real time strategy games. 2006.

[22] Ben Weber and Michael Mateas. A data mining approach to strategy prediction.

[23] Eric W. Weisstein. Crumb.

[24] J. Young and N. Hawes. Evolutionary learing of goal priorities in a real-time strategy game. 2012.