A Coalition based Agent Design for Heroes of Newerth

Eirik M Hammerstad
Erik Kvanli

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Pinar Öztürk, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Video games have become more and more advanced, yet their artificial intelligence components are often a source of criticism. Even though academic AI has come very far, it is rarely seen in video games due to computational complexity. Video games usually use most of their computation time on graphics and physics, and since they require quick response times, the AI is left with scarce resources to use advanced techniques. This project examines the use of case-based reasoning and multi-agent cooperation to improve the existing AI in a commercial video game. The game is called Heroes of Newerth and is developed by S2Games. We focus on making group decisions in this real-time environment which is partially observable. We also use case-based reasoning techniques to improve agents' decisions. Our goal is to implement an agent with these properties, which is both more challenging and fun to play against.

Preface

This project was a master thesis project conducted under the Department of Computer and Information Science at NTNU during the spring of 2014. We would like to thank our supervisor, Pinar Öztürk for her help, giving us important feedback during the entire semester. We would also like to thank S2Games for creating a fun and entertaining game, and providing an API for developing modifications to the existing AI. At last, we would like to thank the Heroes of Newerth Bot Community for answering our many questions.

Sammendrag

I dette prosjektet undesøker vi om konsepter fra forskningsområder innen agent og multiagent systemer kan benyttes i moderne videospill. Vi har implementert et system som benytter seg av case-based resonnering og teknikker fra multiagent samhandling. Systemet vårt er implementert ved hjelp av det eksisterende AI rammeverket til Heroes of Newerth. Vi har undersøkt om dette kan føre til at agenter i Heroes of Newerth utfører bedre handlinger.

Contents

1 Introduction **3**
1.1 Motivation . . . 4
1.2 Research Goals . . . 5
1.3 Research Methods . . . 5
1.4 Report Outline . . . 5

2 Introduction to Heroes of Newerth **7**
2.1 Game Breakdown . . . 7
2.2 Existing AI in Heroes of Newerth . . . 8
2.2.1 Agent Behaviors in Heroes of Newerth . . . 9
2.2.2 The AI Code . . . 10

3 Background Theory **15**
3.1 Agent Systems . . . 15
3.2 Multi-Agent Systems . . . 17
3.2.1 Cooperation . . . 18
3.3 Machine Learning . . . 18
3.3.1 Case-based reasoning . . . 19

4 Related Work **21**
4.1 Cooperation . . . 21
4.2 Machine Learning . . . 23
4.3 Commitment . . . 26

5 Our Solution **29**
5.1 Methodologies . . . 29
5.2 Case-Based Reasoning Module . . . 32
5.2.1 Case-base implementation . . . 32
5.2.2 Reconsideration process . . . 34

5.2.3 Case 35
5.2.4 Behaviors 36
5.2.5 Comparison 39
5.2.6 Population of the case-base 40
5.2.7 Learning process 40
5.2.8 Files used in the case-based reasoning module 41
5.3 Cooperation 41
5.3.1 Architecture 42
5.3.2 Cooperation in Heroes of Newerth 42
5.4 Cooperation Protocol 45
5.4.1 Coalition description 45
5.4.2 Forming coalitions 46
5.4.3 Reconsidering existing coalitions 46

6 Experiments 49
6.1 Limitations 49
6.2 Evaluation Methods 50
6.3 Scenario 50
6.3.1 Scenario 1 51
6.3.2 Scenario 2 52
6.4 Experiments 52
6.4.1 Experiment E1: Testing case-based module as an action selection mechanism 53
6.4.2 Experiment E2: Testing coalition performance against the original agents without cooperation 53
6.4.3 Experiment E3: Testing coalition performance against the original agents 54
6.4.4 Experiment E4: Testing coalition performance with a case-based module as an action selection mechanism 54

7 Results and Discussion 57
7.1 Results and Discussion: Experiment 1 57
7.1.1 Results and discussion 58
7.2 Results and Discussion: Experiment 2 63
7.2.1 Results and discussion 64
7.3 Results and Discussion: Experiment 3 66
7.3.1 Results and discussion 66
7.4 Results and Discussion: Experiment 4 68
7.4.1 Results and discussion 69

8 Conclusion 71

9 Future Work 73

A Resources 75

List of Tables

2.2 Behaviors an agent in Heroes of Newerth can have 11

3.1 Properties of task environments 16
3.2 Properties of the environment in Heroes of Newerth 16

5.2 Environment values used in rating calculation of the behaviors . . 40

7.1 Case-based reasoning with the first sub-experiment 61
7.2 Case-based reasoning with the second sub-experiment 61
7.3 Amount of matches played, won and lost against the original AI without cooperation 64
7.4 Amount of coalitions suggested, formed and dissolved against the original AI without cooperation 64
7.5 Amount of agents joining and defecting the coalitions when playing against the original AI without cooperation 65
7.6 Amount of matches played, won and lost against the original agents 66
7.7 Amount of coalitions suggested, formed and dissolved against the original agents 67
7.8 Amount of agents joining and defecting the coalitions when playing against the original agents 67

A.1 Resources 76

List of Figures

2.1 Schematic of Forest of Caldavar, a map in Heroes of Newerth . . . 8
2.2 Heroes of Newerth's decision system for behaviors 9

3.1 A general learning agent, taken from [1]. 17
3.2 Case-based reasoning cycle, fetched from [2] 20

5.1 Case-based reasoning as action selection 33
5.2 Case flow 35
5.3 Proposed architecture to replace the central command's group-action selecter 43
5.4 Protocol for suggesting and forming a coalition 47

7.1 Gold per minute with the first approach 58
7.2 Experience per minute with the first approach 59
7.3 Gold per minute with the second approach 59
7.4 Experience per minute with the second approach 60
7.5 Gold per minute in a match where Arachna played Arachna 62
7.6 Experience per minute in a match where Arachna played Arachna 63

Chapter 1

Introduction

A video game is an electronic game which involves human interaction through a user interface to generate visual feedback on a video device. There are many different video game genres, including first-person shooters, role-playing games, and real-time strategy games. The real time strategy genre involves building structures and maneuvering units under a player's control in order to secure areas of the map and destroy the opponents' assets. A sub-genre of real-time strategy games is the multiplayer online battle arena (MOBA), where 2 teams play against each other until one team is defeated. Heroes of Newerth is a game from this sub-genre and it's the area of our study in this project.

Artificial intelligence (AI) is a broadly researched field, but it is not the current focus of the video game industry today. This has several reasons, e.g. a higher focus on graphics- and content quality. The lack of focus in this field may cause the AI in games to exhibit too simplistic behavior giving the human players who play against a computer agent an unrealistic feeling, creating an unsatisfactory game-experience. The most common approach to agent design in real-time strategy games is to give the AI increased income or other kinds of unfair advantages, which creates uneven games.

In this project we will examine the use of case-based reasoning and multi-agent cooperation to improve the existing AI in Heroes of Newerth, and hope that this can make the AI more realistic and fun to play against.

1.1 Motivation

The current AI in Heroes of Newerth is simple and easy to understand, making it's actions and behavior easy to predict. The team-play and interaction between different agents (Heroes) are predefined and hard-coded. The current strength of the two teams are not considered when team-wide actions are to be taken. At the moment a computer-controlled team is forced to conduct a team-wide action at a randomly selected time. The current state of the game is not taken into account. A more sophisticated system that allows creating coalitions and executing actions as a group, where the strength and the position of the enemy as well as other factors are taken into account, could be a great way to improve the users' experience when playing. There are however many factors to take into account in designing such an AI component; The environment is challenging as it is both partially observable and changing in real-time, and the two teams have opposing goals. In order to develop an AI that works in such an environment, we will have to focus on the role and construction of coalitions in video games. It is important to keep in mind that we have a time-constraint imposed by the environment, and since the number of possible coalitions might grow huge quite fast, we will have to do some simplifications as well as domain-specific optimizations. When making group decisions the goals and desires of the opposing side should be taken into account explicitly. However, a hero does not have explicit information about the plan of its opponents. We will therefore use opponent-modeling to close some of the gaps created by the uncertainty of decision making. We will use techniques taken from the field of machine learning, and adapt them to our project. As a measurement of success we will match our AI vs the existing AI of the game. It is also an important aspect for us as long-time computer gamers to keep the AI realistic, and not give it any unfair advantages against a human opponent.

1.2 Research Goals

Research Goal 1 (RG1) - Case-based reasoning as action selection mechanism

> Investigate how well case-based reasoning performs as an action selection mechanism in Heroes of Newerth, which is a dynamic, partially observable, real-time environment.

Research Goal 2 (RG2) - The performance of coalition formation

> Investigate how well the use of coalitions for achieving tasks in collaboration will work in Heroes of Newerth. Coalitions should be used to perform actions together with other agents.

Each research goal will be evaluated against the already existing AI in Heroes of Newerth.

1.3 Research Methods

Our research started with a design for agents in Heroes of Newerth using coalitions and case-based reasoning, which we studied in the previous semester. The architecture presented in this thesis has its roots in the project we delivered last semester. In order to answer our research goals, we implemented agents which decided the actions to be taken using a case-base. We also implemented a cooperation service, which provided the agents the ability to suggest and form coalitions in order to achieve tasks in collaboration. We used the original AI of the game as a benchmark for our research, and played our modified agents against it. A normal Heroes of Newerth game can last more than an hour, so we created our own lightweight client in order to run the game without graphics, which decreased the time needed for implementation, testing and learning. We then ran our experiments, which can be found in chapter 6. The results from our experiments can be found in chapter 7.

1.4 Report Outline

Our report starts by describing the game, Heroes of Newerth, in chapter 2, before we are examining background theory in chapter 3. Chapter 4 takes a closer look at work relevant to our project, and chapter 5 explains the architecture and implementation of our solution. After that chapter 6 defines the experiments we are conducting in order to investigate our research goals. We then present the results from these experiments along side our discussion of the results in chapter 7. We finalize our report by giving a conclusion in chapter 8, before we mention some ideas regarding what could be done further in chapter 9.

Chapter 2

Introduction to Heroes of Newerth

This chapter introduces the video game Heroes of Newerth which is a multiplayer online battle arena (MOBA) game launched in 2010 by S2Games. S2Games released code to create bots for Heroes of Newerth on May 1st 2013 with their 3.0 update of Heroes of Newerth[3]. We will start by giving a short game breakdown with the most important aspects of the game, before we take a closer look at the current state of the AI in Heroes of Newerth.

2.1 Game Breakdown

The game is played with two teams of five players - the Legion against the Hellbourne. Each player plays a specific hero throughout a game, picked at the start of each game. In this report a hero will be the equivalent of an agent in computer science. We will refer to them as agents for the remainder of this thesis, unless it does not fit the situation. During the game, the agent will gain experience and "levels" if he is near an enemy unit when it dies. Each agent has 4 unique abilities, which can be upgraded as the agent gains levels. The agent will also gain gold periodically, as well as after killing enemy units. The gold can be used to buy items which enhances the strength of the agent. In order to win a team has to destroy the central structure of the enemy team, which is located in the middle of their base. To get to the central structure of the enemy, a team has to

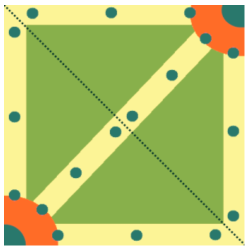

Figure 2.1: Schematic of Forest of Caldavar, a map in Heroes of Newerth

perform some sub-goals, like destroying defensive buildings protecting the enemy base. Cooperation is a method often used to achieve such sub-goals.

The most popular Heroes of Newerth map is Forest of Caldavar (schematically shown in figure 2.1). It has two main bases, located in the top right and the bottom left corners and represented by orange quarter-circles. Three lanes go out of each base, each of which meet an opposing lane halfway across the map. Every 30 seconds, starting at 1 minute and 30 seconds, a wave of faction-controlled creeps will spawn in the base of each team, one wave per lane. They will make their way towards the enemy base following their respective lanes, attacking everything in their path. The smaller blue circles, three for each team in each lane, represents defensive structures belonging to the team with base on the respective side of the map. The defensive structures, also called towers, provides a safe haven for agents in their team. All structures and units provide nearby vision to all of their team-mates. The area between each of the lanes are referred to as *the jungle*. The jungle contains paths and shortcuts between lanes, which can create dangerous situations, such as an agent moving from another lane in order to flank their enemies and attempting to kill them. The jungle is also home to the neutral creeps, or jungle creeps, which can be attacked for experience and gold. Neutral creeps holds no grudge against the two factions, and will only attack if provoked.

2.2 Existing Artificial Intelligence in Heroes of Newerth

This section describes how the computer-controlled agents (often called bots) in Heroes of Newerth are currently implemented. We will also describe how the AI code is structured, and explain the most important part of its code.

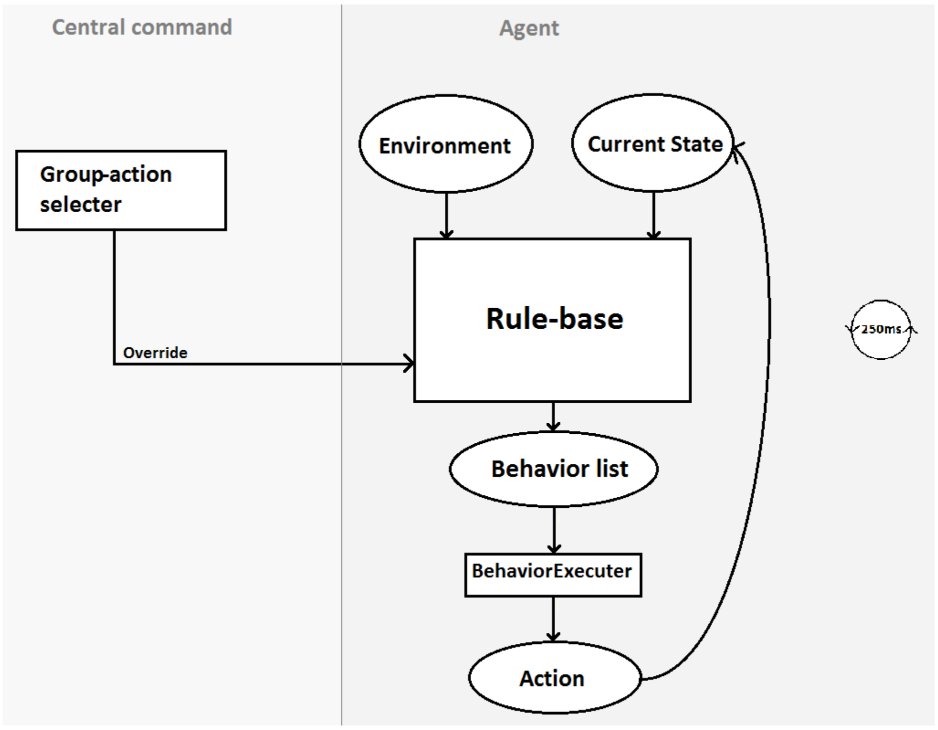

Figure 2.2: Heroes of Newerth's decision system for behaviors

2.2.1 Agent Behaviors in Heroes of Newerth

During a game of Heroes of Newerth each agent continuously decide what to do. The process of deciding upon an action is presented in figure 2.2. Both the current state of the agent as well as the environment is used by a rule-based system to produce a list of behaviors. A list of all standard behaviors are listed in table 2.2. The behaviors enclosed with parentheses are special cases of the behaviors with the same name (but without the parentheses), e.g. *hitbuilding* has different utility for hitting defensive buildings, which may attack back, from buildings which don't attack. Each behavior has three attributes:

Name
The name of the behavior is used to identify it from other behaviors.

Utility
The utility of a behavior is calculated by taking into account the environ-

ment as well as the current state of the agent.

Execution
The execution of a behavior tells an agent how to act if that particular behavior is chosen.

The rule-base outputs a list of behaviors sorted by their utility in descending order. The agent tries to execute the behavior with the highest utility. If he is unable to do so, e.g. he tries to shop, but is nowhere near a shop, he will try the next behavior in the list, and so on, until one behavior is executed. The agent re-evaluates the utility of the behaviors every 250ms.

Each team also has a central team-controller, which decides when the agents will cooperate. It does so by either sending out a command to group up to push an enemy, or to group up to defend against an enemy push. Both of these commands will override any behavior selected by the rule-base. The time to push is decided by the controller by selecting a random number in the range between 7 and 10 including both numbers, and waiting that many minutes. After a push is finished, usually because at least half the team dies, a new time between 3 and 6 minutes is selected for the next push.

Agent-specific behavior

Table 2.2 only lists standard behaviors which all agents can use. Developers have however the option to add specific behaviors to an agent, depending on which abilities that agent possesses. An example could be if that agent had an ability which affect a large area. That ability has to be handled in a different way from an ability which only affects a single unit. Developers also have the freedom to override the original behaviors.

Whenever a hero buys an item, the game engine checks whether or not that item only provides passive effects, or if it can be used to target enemy or friendly units. If the item can be used actively on units, a behavior is added to the buyer's list of possible behaviors to execute. The item-behavior stays in the behavior list until the item is sold or until it expires (some items have a finite number of charges, which can be depleted).

2.2.2 The AI Code

The AI code is split into 8 files for general AI, as well as two additional files for each hero the AI can choose. The two hero files are one .lua file, which specifies the behavior of that specific hero and one .bot file which is an XML file which

Name of behavior	**Utility range**	**Factors influencing utility**
HarrassHero	0-100	Distance, relative health points, range, attack damage, distance to enemy tower
RetreatFromThreat	0-100	Damage over last 1.5s, agent targeted by creeps or tower
HealAtWell	0-100	Current health points
DontBreakChannel	0 or 100	Depends if the agent is channeling an ability
Shop	0 or 99	Depends if an agent is done buying
PreGame	0 or 98	Depends on the game-time
HitBuilding	0,36 or 40	Depends on which building it is, range to building, and if the building is invulnerable
TeamGroup	0 or 35	If the team-controller tells the agents to group
(HitBuilding)	0, 23 or 25	If we can attack a building without being targeted back
AttackCreeps	0 or 24	If a creep can be killed with a single attack
TeamDefend	0 or 23	If the team-controller tells the agents to defend
PushBehavior	0 or 22	Depending on enemies dead, current attack damage
(AttackCreeps)	0 or 21	Kills a friendly creep, if it can be killed with one attack. This prevents enemies from killing it.
PositionSelf	20	Always 20, takes care of positioning the agent if nothing else happens.

Table 2.2: Behaviors an agent in Heroes of Newerth can have

links the lua file with the correct agent in-game. The 8 other mentioned files are as follows:

behaviourlib.lua

Contains all the behaviors of the agents(called bots in the code), their utilities, as well as the functions to be executed. It also contains some supporting functions.

botbraincore.lua

Contains all the functionality for controlling a single agent, as well as item functionality and skill related functions.

core.lua

Contains functions used by both teambotsbrain.lua and botbraincore.lua. Mainly mathematical functions and drawing functions.

eventslib.lua

Takes care of processing of all (combat-) events that might occur.

illusionlib.lua

Helps agents handling illusions of their own units, e.g. by using special items.

junglelib.lua

Contains functions which can be used to make an agent attack creeps in the jungle.

metadata.lua

Contains functions for storing and loading metadata, such as who are in the top, middle and bottom lane.

teambot/teambotsbrain.lua

Handles ”team-decisions”, by defining when to group up as a team, as well as how to defend as a team when being attacked.

Chapter 3

Background Theory

In this chapter we will introduce theoretical background for fields in which our project depends. We will start with agent-based systems. After that, we will describe multi-agent systems. At the end of the chapter we discuss machine learning, with focus on case based reasoning.

3.1 Agent Systems

An agent is an entity that perceives its environment through sensors and acts upon it with actuators [4]. The sensors are used to gather information about the surrounding environment and may be e.g. cameras or microphones for robots, which simulate the eyes and ears on humans. Examples of actuators are computer screens and robotic arms.

The level of intelligence in agents may vary. Russel and Nordvig [4] describes various approaches, e.g. a reactive agent only perceives its environment and acts upon its perception, without any reasoning of previous events or planning. Another approach may be a learning agent shown in figure 3.1, which tries to make improvements based on previous actions and their observed surroundings.

The environment in which an agent act can be categorized along various properties. Russel and Nordvig [4] identifies the following properties listed in table 3.1. Properties of the environment of Heroes of Newerth is described in table 3.2. Agents behavior is closely dependent on the properties of the environment.

Task environment	**Categories**	**Description**
Observability	Fully observable, partially observable, unobservable	What the agent's actuators are able to perceive.
Amount of agents	Single agent, multi-agent	How many agents are present in the environment.
Determinism	Deterministic, stochastic	Whether or not the next state of the environment completely depends on the action executed and the current state.
Episodic	Episodic, sequential	Whether or not the next state is dependent on the action taken in the previous state.
Staticity	Static, dynamic, semi-dynamic	If the environment can change while the agent is deliberating.
Discreteness	Discrete, continuous	How time is connected to the percepts and actions of the agent.
Knowledge	Known, unknown	If the outcomes or outcome probabilities for all actions are given.

Table 3.1: Properties of task environments

Task environment	**Category**
Observability	Partially observable
Amount of agents	Multi-agent
Determinism	Stochastic
Episodic	Sequential
Staticity	Dynamic
Discreteness	Continuous
Knowledge	Unknown

Table 3.2: Properties of the environment in Heroes of Newerth

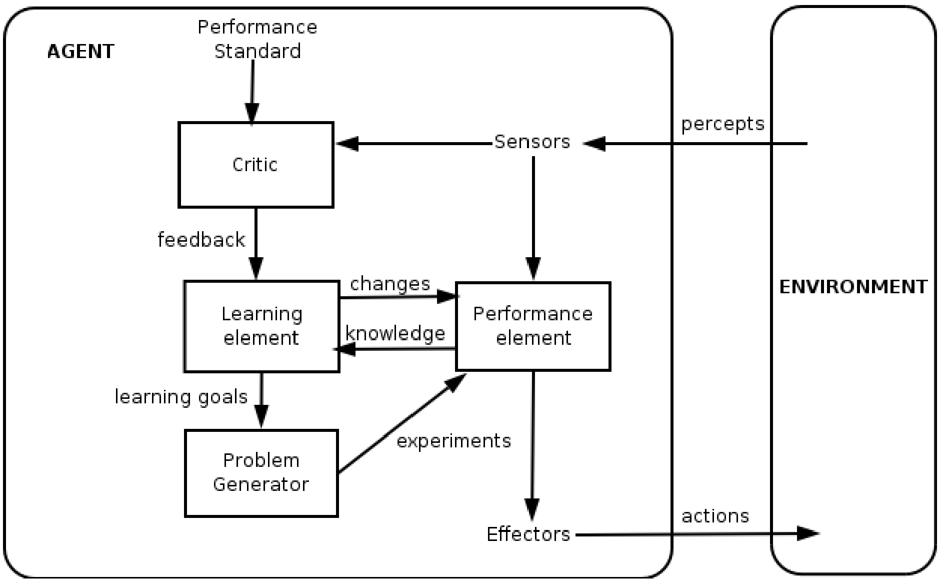

Figure 3.1: A general learning agent, taken from [1].

3.2 Multi-Agent Systems

A multi-agent system (MAS) is a system where multiple agents acts together in the same environment. They are often used to solve tasks which are too difficult for a single agent to accomplish on its own. In order to do so, MAS can utilize methods such as negotiation, cooperation, forming coalitions and bargaining[5]. Agents may also have different types of relationships, for example two agents may be coworkers or have a leader-worker relationship.

Agents in a MAS also have characteristics which defines them. All agents does not necessarily share the same characteristics, but most share the following two[6]:

Autonomy
: The agent is autonomous if it has the freedom to act and think for itself, at least to some extent.

Situated
: The agent is situated in some environment where it acts towards its objectives.

Other characteristics may include the ability to learn from past experiences, or the ability to interact with other agents (or humans).

3.2.1 Cooperation

Cooperation is a method an agent can use to work together with other agents in a MAS. It is the act of working together for a common purpose, even though the individuals who cooperate don't share the exact same goals. There are many ways to cooperate - a task can be distributed to several agents, who solve their part, and share the results; agents can detect that they have joint intentions and decide to coordinate their efforts; or multiple agents can plan the activity of their group ahead of time[5].

The first example mentioned in the previous paragraph described cooperative distributed problem solving(CDPS)[7]. CDPS can be broken down into two separate activities: task sharing and result sharing[8]:

Task sharing
Task sharing takes care of decomposing a problem into sub-problems and allocate them to agents.

Result sharing
Result sharing takes care of sharing the results from the sub-problems in order to find a solution to the original problem.

Contract Net[9][10] is a well known high-level protocol for task sharing. In a MAS it achieves efficient cooperation by following the following steps:

1. Recognize that we have a problem which we need help with solving.
2. Announce to other agents that we have a task we need help with.
3. If any of the other agents finds the task suiting, it will place a bid for that task.
4. The agent who announced the task has to decide which of the bidding agents will be awarded the task.

3.3 Machine Learning

Machine learning, a branch of artificial intelligence, deals with the construction and study of systems that can learn from data. Tom M. Mitchell provided a widely quoted, more formal definition of Machine learning as:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [11]. We are planning to use case-based reasoning, a machine learning method, for agents decision making which we will describe shortly here.

3.3.1 Case-based reasoning

Case-based reasoning(CBR) is the process of solving new problems based on solutions of similar past problems. CBR has been formalized for purposes of computer reasoning as a cyclic four-step process. Figure 3.2 shows the case based reasoning cycle. The steps are as follows[2].

Retrieve
: Here similar cases are retrieved. Where each case consists of a problem, the solution and optional annotations about how the solution was derived.

Reuse
: The most similar case is chosen and its solution is used. If there are several similar cases the average solution is chosen and all case explanations are used.

Revise
: Revise the proposed solution.

Retain
: Store the resulting experience as a new case in the memory, which can be used to solve future cases.

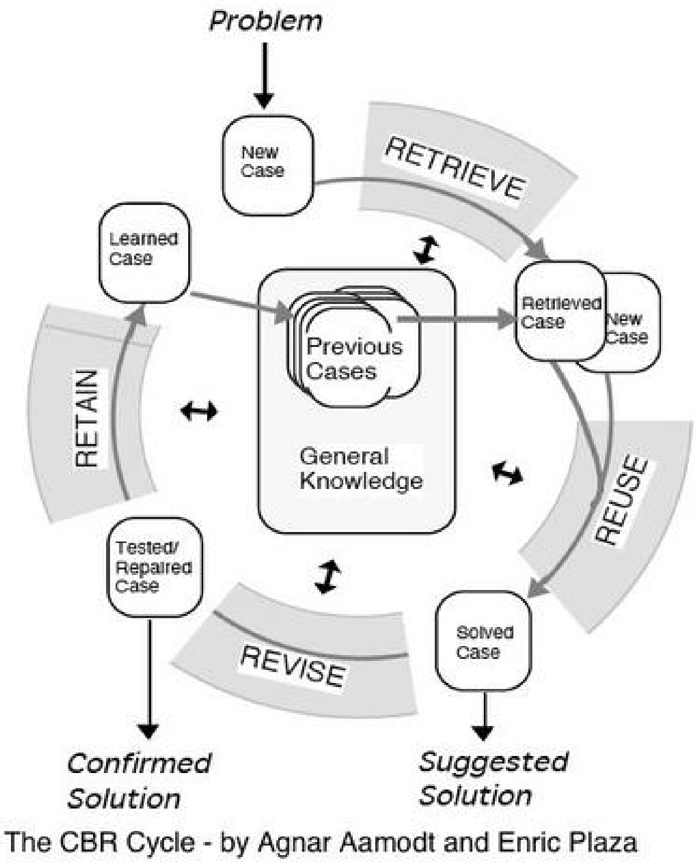

Figure 3.2: Case-based reasoning cycle, fetched from [2]

Chapter 4

Related Work

Our main focus in this project is cooperation in multi-agent environments, more specifically real time video games, and case-based reasoning. This chapter will be used to explain cooperation and case-based reasoning related approaches, some of which are taken in video games.

4.1 Cooperation

Cooperation is making the agents communicate together and decide upon an action, and also how the decisions can be made in real-time to fit systems such as video games. There is not much research on the subject of cooperation in multi-agent systems such as computer games, therefore we considered the subject of cooperation in multi-agent environments in general. This section contains different topics regarding cooperation in multi-agent systems. At the end of this section we will discuss them and tie them together with real-time video games such as Heroes of Newerth.

Organizational design

One topic addressed by Horling and Lesser [12] is the organizational design of an agent system. In order to solve tasks in an efficient matter, groups of agents need a strategy on how they coordinate themselves - who does what, and when should it be done. Horling and Lesser, as well as Fox [13], identifies that organizations can be used to reduce uncertainty and complexity. Organizational paradigms include hierarchies, holarchies, coalitions, teams, congregations, societies, federations, markets and matrix organizations. Different organizations has different characteristics and provide different benefits, as well as different drawbacks. Horling and Lesser states that there is no universal organization suitable for all problems and domains.

Mashhadi and Monsefi [14] defines one model using coalitions. They point out that the forming of coalitions has been viewed by literature as an optimization problem. They propose an algorithm which estimates Shapley values in order to reduce time consumption. Their model defines players as a many-to-one relation between agents and tasks, where one agent can participate in one task and many agents can cooperate with the same task. Their approach utilizes an iterative algorithm to determine the optimal set of coalitions. The algorithm starts with the complete set of players, and eliminates one by one depending on who has the lowest contribution (based on estimated Shapley values) until they are left with only the players that contribute above average. Another approach is due to Shehory et al.[15] whom also had a focus on computational tractability in order to reduce time consumption. They showed that it is possible to get close to optimal efficiency when allocating groups, by implementing coalition formation in a real-world multi-agent system. They had to make a few relaxations to meet complexity demands, such as the case of too many new tasks arriving at the same time, where they decided to temporarily disable the coalition reconsideration process.

Multiple coalitions

There may be restrictions related to cooperation in multi-agent environments. One restriction often found in literature is that a single agent can only participate in one coalition. This is a restriction Shehory and Kraus [16] looks closer at, when they present a distributed algorithm for task allocation among autonomous agents. They first provide an algorithm where a single agent can only participate in one coalition, before they generalize this algorithm to take into account overlapping coalitions. Agents does not necessarily use their entire capacity on a single task, and can this way partake in several tasks at a time. The agents can join multiple coalitions at a time as long as their maximum work capacity is not reached. Once done with a task, agents may join new coalitions as long as their work capacity allow it.

Discussion on cooperation

In the domain of real-time video games such as Heroes of Newerth, it is important to select a structure which allows for autonomous agents to dynamically cooperate towards a common goal. Horling and Lesser [12] sum up the advantages and disadvantages of the different organizational structures. They point out that a coalition is a structure which is usually goal-directed, with a short life - created for a specific purpose and terminated when it is no longer needed. Coalitions can have a flat structure, or it can have a "leading agent". All of these traits relate very well to how humans play Heroes of Newerth, cooperating for sub-goals as small groups, and dissolving the group when a sub-goal is no longer obtainable at that moment. Humans either cooperate based on consensus, or a single person can take a leading role based on personal characteristics. Horling and Lesser also point out that the greatest disadvantage is the cost of a coalition might outweigh the benefits of it. As Heroes of Newerth has a small number of agents which can form coalitions (five at most in each coalition, from a pool of 10 agents total), we feel that this will not be a major drawback. Mashhadi and Monsefi [14] suggest an algorithm for coalition formation which shows promising results, but the agent/task does not account for multiple coalitions. Shehory et al.[15] neither accounts for multiple coalitions.

Unlike [14] and [15], Shehory and Kraus [16] take into account overlapping coalitions. They also implement their algorithm in a non-super-additive environment, and it is used in a real-world application. They do however not account for the fact that there will be an opposing side which has opposing goals, an assumption not specified by any of the mentioned articles, which is the case in Heroes of Newerth.

4.2 Machine Learning

In this section we review some learning approaches in artificial intelligence. We want to use machine learning in order for the agents to be deliberate.

Artificial Intelligence techniques have successfully been applied to several computer games. However, in the vast majority of computer games traditional AI techniques fail to play at a human level because of the characteristics of the game. Most current commercial computer games have vast search spaces in which the AI has to make decisions in real-time, thus rendering traditional search based techniques inapplicable[17]. An extra complexity factor in Heroes of Newerth is the partially observable environment. This is the union of what can be seen by all objects currently under the client's or an ally's control. Different locations on the map could have different status, such as: unknown (no information), known (partially visible) and visible (any unit on this location is seen)[18]. Goals and actions needed by an agent are going to lead from the initial setup to the goal of winning through numerous situation as described in [19]. The article also divides tasks of a game into different managers which works in parallel, these managers are: strategy manager, income manager, production manager, tactics manager and recon Manager. A divide and conquer technique like this is suitable for Heroes of Newerth because of the complexity of the game.

Tan[20] suggests that cooperating agents which share information about current sensor readings, past experiences and current learned policies significantly outperform independent learning agents. Others research about one joint policy table, which agents have access to in turn, even though they are separate learners.

Case-based reasoning

There has been done some work on case-based planning using WARGUS, a fanmade modification of the video game Warcraft 2. Ontanon et al. [17] explains the learning done for the AI to have better behaviors. The AI in this game learns by watching expert demonstrations and save each case executed in a case base, which is used for behavior generation. These behaviors then creates a plan and the AI does an action according to sensor input.

The Case-based AI in Tactician(CaT) is used to implement an approach that learns to select which tactic to use at each state in the WARGUS game, as described in [21]. The paper presents learning curves that demonstrate its performance and quickly improves with training even though the adversary is randomly chosen for each WARGUS game. During a game, CaT records a description for each building state, along with the score and tactic selected in order to create a certain building. The tactics the program chooses is based on countering the enemies tactic which is gathered from games. When the game is over it checks whether the case from the game exists. If so, it updates the case's performance, otherwise it creates a new case and adds description, tactic and performance. Using this strategy they went from 22.9% wins to 82.4% wins after 100 games.

Weber and Mateas [22] have done research on the field of case-based reasoning in WARGUS. They did this by setting a generalized state for the current game state, then recall past solutions by checking distance to similar cases before selecting one using weighted random selection, which is performed.

CBR is used by Szczepański and Aamodt[23] to control the micromanagement of the units. Micromanagement is actions, manually chosen by the player during a game. They used a system which updated actions every second together with change of macro guidelines (behaviors) if something special happened. The use of behaviors made the system much less complex, because every situation included in behaviors didn't need a check every second. Their retrieval used a weighted nearest neighbor algorithm. They trained their system against the existing AI, with an expert human to learn the system each time it lost. Their work shows that CBR is possible for micromanagement, though they simplified their situations, by using only a two dimensional playable area with mirror matches (where each team possesses the same amout and type of units).

Exploration

When choosing actions, a greedy approach is to always use only the best option, but this may only be good in the short run as described by Sutton and Barto [24]. They bring up the action of exploring, which is a non greedy action. On single runs, exploitation is optimal choice to maximize the expected reward, but exploration may produce the greater total reward in the long run. The article explains how it's important to balance the selection of exploring and exploiting according to how many "plays" are left.

Team learning

Weiss and Sen [25] explain how to predict which actions led to the outcome of a situation. They also bring up the subject of team learning which is an important question, where team learning consist of a single or more learners involved, but the learners are discovering a set of behaviors for a team of agents. Berenji and Vengerov[26], investigate a cooperative learning setting where multiple agents employ communication to use and update the same policy. They also research the fact that parallel update of the table reduces the chance of optimal behaviors in the end. This is equal to learners exchanging learned information.

Discussion on learning

The lack of AI development in the gaming industry, as described by Graham [27], makes an intelligent AI solution rather valuable. This is also the fact in Heroes of Newerth where moderators at the forums[28] claims that currently, only 1 developer is maintaining the AI code 2 days a week, and the main development as of June 2014 is done by the Heroes of Newerth community.

Heroes of Newerth is a partially observable video game with a lot of interacting elements, as described in chapter 2, which makes the game highly complex. Because of this complexity, it's hard to create methods and strategies to master this game. Because of the complexity issue, the division of tasks is essential to gather enough information, and to handle everything in a proper way, as solved in [19] by dividing the game WARGUS into different managers with different responsibilities. This division would simplify the process for each manager involved. Cooperative learning is suggested to be effective as described in [20] and [26], but because of the fact that each agent controls an unique hero as described in chapter 2 we figured this is close to impossible, this is because we would need a general understanding of each hero which would be a master degree on it's own.

Molineaux et al. [21] provided a system which learned tactics based on previous games. Using this to control the AI agents in Heroes of Newerth would not work, because the existing AI selects the best action at the moment, and not an overall tactic for each game. The only possible tactics to track in Heroes of Newerth, as we can see it, would be item builds and skill builds.

The micromanagement system explained in [23] fits for the problem of controlling the AI agents, but they only tested their system on a smaller much less complex map, which is unlike anything a normal Heroes of Newerth game provides.

Exploration versus exploitation as described in 4.2 could be useful to our system. Though too many random decisions would result in huge losses, resulting in less information learned. [22] uses the Euclidean distance to find the distance between 2 cases. This could prove useful for our system in locating the most similar cases. In addition to finding the closest case, it's important to figure which action led to the current outcome, which is described in [25]. We could try this, though the complexity of Heroes of Newerth might hinder us.

4.3 Commitment

A topic addressed by Jennings [29] is that of commitment and conventions. A commitment is a promise to partake in a cooperative action, while a convention describes when an agent should reconsider his commitments. He argues that commitments are central in providing predictability for the agents, while conventions help the agents to be flexible and deal with a dynamic environment.

Closely related to commitment is the topic of trust, which Ramchurn et al. [30] discuss. They present a state of the art to show how agents in uncertain and dynamic environments can act and interact safely. They divide trust in two ways: trust on the individual level and trust on system level. Trust on individual level can, among others, be achieved by evolving and learning strategies, such as Axelrod's tit-for-tat strategy in the prisoners dilemma[31]. It can also be achieved by maintaining a model of reputation about the other agents. Trust on system level can be achieved by protocols (such as the Vickrey auction, where the dominant strategy will be to bid the true value of an item), a global reputation mechanism, or security mechanisms (such as public key encryption to determine identities) [30].

Discussion on commitment

In the domain of commitments and conventions, Jennings [29] presents us with a very broad theory on the subject. He does however not include any functional or implementational architecture. The latter holds for Ramchurn et al. [30] as well on the topic of trust.

Chapter 5

Our Solution

This chapter aims to give an overview of our system, both how we designed it and how we implemented it. Our system has two distinct parts. The first part is a case-based reasoning module for agents in Heroes of Newerth. It aims to replace the rule-based action selection mechanism described in section 2.2.1. The second part is a cooperation module for the agents in Heroes of Newerth. It will use a cooperation service, which we provide, to make the agents form coalitions towards common goals.

We will start by describing the methods we used to implement the system, as well as some of the problems we encountered. After that we will discuss the case-based reasoning module, before we discuss the cooperation module. We will end this chapter with the cooperation protocol used by the agents.

5.1 Methodologies

This section is going to explain the steps we followed on our way to a finished product. We start by describing why and how we created our development environment, before we mention some of the problems we encountered. The different resources we used are described in appendix A.

Setting up the environment

The current practice to test changes when developing on the AI code of Heroes of Newerth is to start a new match. Since Heroes of Newerth is a modern video game, and it takes a while to start a new match. As we will see in the next section, the case-based reasoning module learns cases from matches played. In order to populate a sufficiently large case-base, we would need to run a lot of matches, each lasting normally 30-60 minutes. It would therefore be greatly beneficial to have an automatic test-environment before we started developing. We set up our development environment the following way:

Finding a suitable IDE
First we needed a suitable integrated development environment(IDE), which could provide us with syntax highlighting and basic auto-completion. We are familiar with Eclipse from previous courses at NTNU, and discovered that it had a plugin for lua. This made Eclipse with lua development tools our choice.

Get source code
We also needed the source-code for the AI in Heroes of Newerth. The current retail-version can be extracted from a file in the directory where Heroes of Newerth is installed. Since we were doing more than just developing new agents (also called bots), we soon discovered that there were some errors in the current retail-version of the code. The errors would only get fixed when the game came with a new version (also called patch), so we decided to get a newer version. The newer version we decided upon was the community-branch of the AI code found on GitHub.

Create dummy-client
As mentioned earlier, it would be greatly beneficial to have an automatic test-environment. Heroes of Newerth does not come with a client which allows to run the game without a graphical interface. We would also like the game to exit after a match was over, but the current client only exits to the main menu. We therefore decided to build our own dummy-client which mimicked the behavior of the original client.

In order to do so, we started the dedicated server provided by S2Games, and launched a match between two AI agents. Using a program called RawCap we listened to all UDP datagrams and TCP packets being sent between processes on the computer. After the match was over, we opened and analyzed the data traffic with a program called Wireshark. After running several matches and analyzing the traffic afterwards, we managed to discover a pattern in the data being sent. Using C we created our own dummy-client which responded to the data sent by the dedicated server. Doing so reduced the overall resources used on our computer, which in turn allowed us to turn up the speed of the match. The dedicated server allows for match-speeds between 1 and 100, but too high speed will case the server to stutter and lose data. With the dummy-client we managed to raise the speed from 1.5 to 5.

The dummy-client would also exit when a match finished, or if something unexpected happened.

Automatic running of matches
We were now able to start a match, and run it at increased speed. One of our goals with our development environment was to run multiple matches in sequence in order to populate the case-base. However, the dedicated server did not accept arguments at start-up, which meant that human interaction was required to start a match. To circumvent this problem, we used a program called AutoHotkey to inject a string of data into the currently active program. The string contained parameters of the match, such as which agents to use, that the game-mode was BotMatch, etc. After the string was injected, we made AutoHotkey inject the "Enter" command, which would prepare the match.

Get results
We would also need to store results when a match was over. The results could be either match-data for our experiments, or cases for our case-base. Heroes of Newerth has a hidden database API which their user interface developers make use of. We copied and modified their code in order to store our data in different files. Unfortunately lua saves the data in a non-readable format, so in order to read it we had to start a new match and make the agents write the data as chat messages to each other.

Encountered problems

Some of the problems we encountered have already been mentioned when we described how we set up our development environment. The following paragraph describes two more problems we encountered.

Current state of the AI code
As we stated earlier, we discovered that there were some errors in the current retail-version of the code. This was an ongoing problem during development. We also had problems with the code being very tightly coupled. As we were trying to tie our system together with the original code, we

ran into errors. The errors could originate from a global variable set at an earlier point, a function doing other, or more, things than we expected, etc. Solving one error, usually lead to us finding ten new. We ended up with adding our code to the original AI files to prevent most of the errors.

Linux

Heroes of Newerth is a video game which runs on multiple platforms. It was created for both Microsoft Windows, Mac OS X, and Linux. Our preferred choice when developing would be Linux, as we are familiar with software developing using this platform. Unfortunately everything runs perfectly on Linux with the exception of matches with the game-mode BotMatch. This forced us to use Microsoft Windows. Fortunately there exists a linux-like environment for Microsoft Windows, called Cygwin.

5.2 Case-Based Reasoning Module

In order to win Heroes of Newerth, it is important for agents to make wise decisions. Wise decisions can help an agent to stay alive and gain advantages, by means of gold and experience. Currently the original agents in Heroes of Newerth makes decisions based on a rule-base, as explained in section 2.2.1. By introducing case-based reasoning to the agents in Heroes of Newerth we hope to increase the decision making abilities of the agents and make the game more interesting to play. In order to achieve this, we intend to remove the old rule-based system and replace it with a case-based reasoning module connected to a case-base, as shown in figure 5.1. We intend to fill the case-base with cases learned through multiple matches. Our case-based reasoning module will be explained further in this section.

5.2.1 Case-base implementation

Heroes of Newerth has a hidden database API which S2Games' user interface developers make use of. We modified their code in order to be able to store data to files on our own. For each agent used in our experiments, we will store a single file containing the agent's entire case-base. How the database connection is established is shown in listing 5.1. The Database.New function either loads a file if it already exists, or creates a new one. The ReadDB call reads the database containing the case-base to memory. This is done at the start of every match.

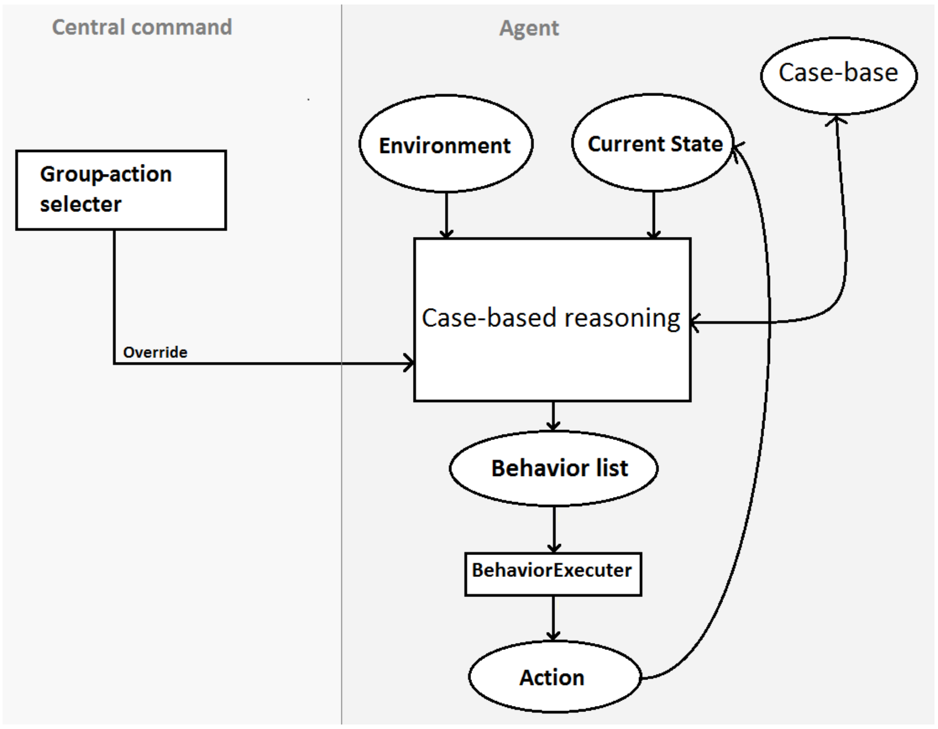

Figure 5.1: Case-based reasoning as action selection

```
casebase.database = Database.New(filename)
casebase:ReadDB()
```

Listing 5.1: "New database connection"

We also created a way to save new cases to the case-base using the database API, shown in listing 5.2. This code would load the case-base for an agent, where the name of that agent is the input to the LoadDatabase function. It would then append the cases from the current match to the case-base before flushing it, which would write the case-base to file. This would be done every five minutes of the match for each agent. We decided to only write to file every five minutes to save system resources.

```
local cases    --Cases from the match
casebase:LoadDatabase(agentname)
for _, case in ipairs(cases) do
  casebase:SetDBEntry(case)
end
casebase:Flush()
```

Listing 5.2: "Save cases to the case-base"

5.2.2 Reconsideration process

The agents in the current AI of Heroes of Newerth reconsiders their behavior lists every 250 ms. We early discovered that the time-consumption of the case-based reasoning module prevents us from reconsidering that often. The reconsideration process using the case-based reasoning module is triggered by:

Every frame
Every frame, which is the smallest unit of time we can use (50 ms), we check if the current behavior is still possible to execute. If we are unable to do so, the reconsideration process is triggered.

Task is completed
After a task is completed, e.g. an agent is killed when the HarrassHero behavior is active, the reconsideration process is triggered.

State changed
If the state of the game has changed drastically, for example someone healing an agent so it no longer has to return to the base to heal, a reconsideration process will be triggered.

Periodically

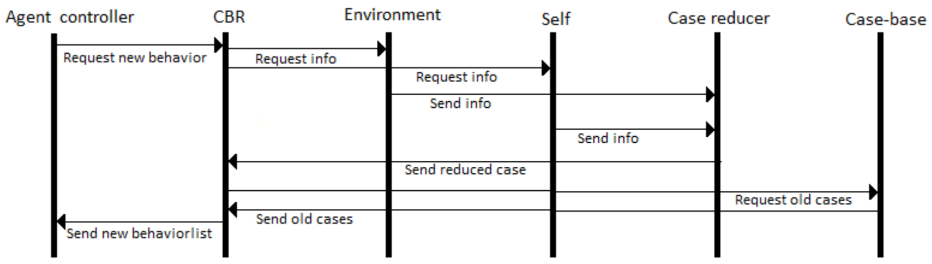

Figure 5.2: Case flow

We also check periodically in case neither of the former three points trigger. Instead of every 250 ms, we check every few seconds.

5.2.3 Case

The way we use a case is depicted in figure 5.2. We start by collecting a lot of information from the environment nearby: every unit, building and agent nearby. The environment variables are stored in a big case, as shown in listing 5.3. After that, the big case is reduced to a smaller format, as shown in listing 5.4, which makes it easier to compare to other cases. The case-based reasoning module then compares the new case to old cases in order to decide how to order the behaviors. The CBR module picks out the closest matching case for each behavior, and orders that list based on how close the cases were, before returning a behavior-list. How close the cases are to each other will later be referred to as the behavior's utility.

```
local bigCase = {}            --Big case with all variables
bigCase.enemyheroes = {}      --List of enemy heroes
bigCase.friendlyheroes = {} --List of friendly heroes
bigCase.friendlytowers = {} --List of friendly towers
bigCase.enemytowers = {}      --List of enemy towers
bigCase.friendlycreeps = {} --List of friendly creeps
bigCase.enemycreeps = {}      --List of enemy creeps
bigCase.self = self           --Reference to self
bigCase.time                  --Current time in milliseconds
return bigCase
```

Listing 5.3: A case containing all environment information

```
        local case = {}
        hero_level_diff = alliedHeroLevels - enemyHeroLevels
        hero_dmg_diff = alliedHeroDamage - enemyHeroDamage
        hero_ms_diff = alliedMovementspeed - enemyMovementspeed
        case.herostrength =  hero_level_diff + hero_dmg_diff
                           + hero_ms_diff
        case.towercount = (friendlytowerCount
                           - enemytowerCount) * 4
        case.creepcount = (friendlycreepCount
                           - enemycreepCount) * 2
        case.selfhealth = self:GetHealthPercent()
        case.allyhealth = friendlyHealthPercent
        case.enemyhealth = enemyHealthPercent
        case.time = currentGameTime
        return case
```

Listing 5.4: Reduction of big case to small case

5.2.4 Behaviors

In the original AI, each agent periodically chooses behaviors from a list of many behaviors(see section 2.2.1). Our design base its approach on the original, by creating a behavior list using the case-based reasoning module. We will be using the following behaviors:

- HarrassHero
- RetreatFromThreat
- HealAtWell
- HitBuilding
- AttackCreeps
- UseAbility
- Push
- PositionSelf

During implementation, we discovered that the agents needed substantially more data to learn behaviors than we first anticipated. This was primarily due to the "PositionSelf" behavior, which in the original code is used to position an agent if no better behavior is possible. It turned out that our agents were unable to learn this behavior properly before the data-sets became too big for our computers to handle. We therefore decided to try two different approaches to learning:

First approach

The first approach is what we just described. We used the original rule-based action selection mechanism to evaluate whether or not the behaviors could be executed. We kept this approach in order to compare with our second approach.

Second approach

The second approach involved removing PositionSelf from the case-base. Instead of trying to learn it, we decided to use the original system's rule-based action selection mechanism to decide if our agents should use the PositionSelf behavior. If PositionSelf was the highest rated behavior in the behavior-list from the rule-base, we decided to use this behavior. If something else was rated higher, we used the case-base to decide what to do.

UseAbility behavior

Agents have the options of buying items, some of which can be used actively on friendly or enemy units. In order for the agents to learn how to use items, we created a general behavior for all items. The original AI code stores a list of all the behaviors an agent can execute. If an item is bought, a behavior related to this item will be added to the list. If a behavior in this list is not one of the listed behaviors in table 2.2, we assume it is an item-related behavior. If the UseAbility behavior is picked from the case-base, one of the item-related behaviors will be executed.

DontBreakChannel behavior

DontBreakChannel is a special behavior which we have to take into account. It is only used if an agent is channeling an ability, which means that doing anything will cancel the ability. Doing this is in general very bad, and we want to avoid this ever happening. In order to do so, we always check if an agent is channeling an ability, and stop the reconsideration process if it currently is.

HealAtWell behavior

Early in the process of populating the case-base, we discovered that the agents were wasting a lot of time running home to the well, located in their base. The well heals friendly nearby agents, and running back is not necessary if an agent has a high amount of health. In order to prevent agents running home all the time, we made it impossible to execute the HealAtWell behavior if the agent's health was above 40 percent.

Execution process

Once the case-based reasoning module has its behavior list, we have to find a behavior to execute. The process of finding a behavior to execute using our first approach is shown in listings 5.5. We do this by going through the list, starting with the behavior with highest utility, and check if it can be executed.

```
if unitSelf:IsChanneling() then
  --Do nothing
else
  for curBehavior in behaviorList do
   if
    canBeExecuted(curBehavior) and
   (curBehavior.Behavior["Name"] == "AttackCreeps" or
    curBehavior.Behavior["Name"] == "HarrassHero" or
    curBehavior.Behavior["Name"] == "HitBuilding" or
    curBehavior.Behavior["Name"] == "PositionSelf" or
    curBehavior.Behavior["Name"] == "Push" or
    curBehavior.Behavior["Name"] == "RetreatFromThreat" or
    (curBehavior.Behavior["Name"] == "HealAtWell" and
                self:GetHealthPercent() > 0.4) or
    curBehavior.Behavior["Name"] == "UseAbility")
                then
      bSuccesful = curBehavior.Behavior["Execute"](self)
    end
  end
end
```

Listing 5.5: Execution process of a behavior using our first approach

For our second approach, we don't check if the behavior name is PositionSelf, but rather add the code from listings 5.6 to the end of the function. If none of the behaviors in the list can be executed, we execute the PositionSelf behavior.

```
  if bSuccesful == false then
    behaviorLib.PositionSelfBehavior["Execute"](self)
  end
```

Listing 5.6: Execution process using our second approach

5.2.5 Comparison

When we were deciding which case were most equal to other cases, we had to do a comparison between the new case and all the cases in the case-base. To compare cases we tried 2 methods. Either of these methods will be mentioned in the results and discussion.

Weighted nearest neighbor

This method compares two cases by looking sequentially at each of the varibles. For each variable, the variable with the lowest value is divided with the highest value, yielding a value between 0 and 1. Some comparisons of variables may be more important, therefore they can be weighed differently. These weights sum up to 100. The equation is shown in equation 5.1

$$
\begin{aligned}
d(p,q) = z * \sum_{i=1}^{n}(weight[i] * X_i/Y_i) \qquad (5.1) \\
Where\ X = q_i > p_i => q_i\ else\ p_i \\
And\ Y = p_i > q_i => q_i\ else\ p_i \\
And\ z = numberOfVariables/100 \\
And\ \sum_{i=1}^{n}(weight[i]) = 1
\end{aligned}
$$

Euclidean distance

This method compares two cases, where all variables are real numbers. It compares the variables in each case one step at a time, and measures the Euclidean distance between them, which is the distance between the two points. This is an approach taken by Sutton and Barto [24]. A mathematical description of the Euclidean distance is shown in equation 5.2.

$$d(p,q) = \sqrt{z * \sum_{i=1}^{n}(q_i - p_i)^2} \qquad (5.2)$$

Behavior	**Kills / death**	**Hero damage**	**Experie -nce**	**Gold**	**Health**	**Building damage**
HarrassHero	X	X	-	-	X	-
RetreatFromThreat	-	-	-	-	X	-
HealAtWell	-	-	-	-	X	-
HitBuilding	-	-	-	-	X	X
AttackCreeps	-	-	X	X	X	-
PushBehavior	-	-	X	X	X	X
UseAbility	-	-	X	X	X	-

Table 5.2: Environment values used in rating calculation of the behaviors

5.2.6 Population of the case-base

In order to populate the case-base, we decided upon an automatic approach. We keep track of several agent-related variables during the match, e.g. how much experience and gold the agent has earned per minute. Table 5.2 shows which variables we keep track of, depending on which behavior is currently active. Keeping track of these variables makes it possible for us to calculate if an agent has improved his situation while having a certain behavior. If the agent has improved his situation, we save that case, unless we already have a similar case in the case-base.

5.2.7 Learning process

In the beginning of the population process, the case-base will be entirely empty. In order to learn new behaviors, we create a random behavior list. We continue to create this random behavior list, but not all the time. The more cases the case-base contain, the chance of creating a random list gets lower. This process is an attempt to mimic the epsilon greedy process, as explained in section 4.2. This means that our CBR system would find a behavior and in X%, where X is a number representing the learning rate, of the cases another behavior ordering is chosen at random. Once the case-base contained sufficient amount of cases, the random creation of behavior lists would stop.

5.2.8 Files used in the case-based reasoning module

To make the case-based reasoning module work as a standalone module we had to modify and create the following files files:

case_builder.lua
: This file contains methods for creating a case. It collects environmental information as well as details regarding the agent itself. The file has functions to structure all the required information.

casebase.lua
: Connects our code with the database API supplied by the game engine.

cbr_comparator.lua
: Contains logic for comparing cases to each other, rating them, as well as reducing them.

cbr_hero_progress.lua
: Contains functions for saving how much experience and gold the agents earn. We had to create this file since the game engine refuses to give this information for the opposing team.

cbr_io.lua
: Contains helper functions which simplifies the use of the casebase.lua functions.

botbraincore.lua
: This is the original botbraincore.lua which controls a single agent. We had to do some changes in this file in order to implement our new design.

teambotbrain.lua
: This is the original teambotbrain.lua which controls team actions. We had to do some changes in this file in order to implement our new design. During the learning-phase we disabled the group and push behaviors from this file.

5.3 Cooperation

In order to make the behaviour of the agents more dynamic and interesting, we will implement a cooperation service which will allow the agents to group up and perform actions together. In this section we will describe how we intend to implement this, before we discuss how the agents can interact. We will base the interactions on how the original AI of Heroes of Newerth cooperate, which will lay a foundation for our cooperative protocol described in the next section.

5.3.1 Architecture

As described in section 2.2.1 and shown in figure 2.2, the existing AI in Heroes of Newerth decides when the agents are going to cooperate by using a central command. The central command waits 8-10 minutes after match start, and instructs all agents on the team to group up and push a lane together. This instruction overrides any other action the agent wish to perform, which hinders the agents from being autonomous.

We wish to provide the agents with means of cooperating while remaining autonomous. Figure 5.3 shows our proposed architecture. Instead of the central command, each agent will have his own cooperation module which will communicate with a cooperation service.

Cooperation service

The cooperation service is located outside the agents, and provides each agent with the ability to post information to the other agents. It resembles the blackboard in a blackboard architecture[32]. It will only provide agents with the means of communication, and will not issue any commands or force the agents to do anything they don't want to.

Cooperation module

The cooperation module will contain a decision mechanism required for the agent to cooperate. It will decide how and when it should call for coalition, and it will decide when to join other agents' coalitions. It should also determine if it doesn't want to continue cooperating, and if cooperation is benefitial for the agent at that time.

Based on the discussion we had in section 4.1, we decided to form coalitions in order to cooperate. A protocol for how this was done will be described in section 5.4.

5.3.2 Cooperation in Heroes of Newerth

In this section we will take a closer look at how we implemented cooperation in Heroes of Newerth. We will start by looking at what we used from the original code, and then take a look at a few things we had to keep track of in the cooperation module.

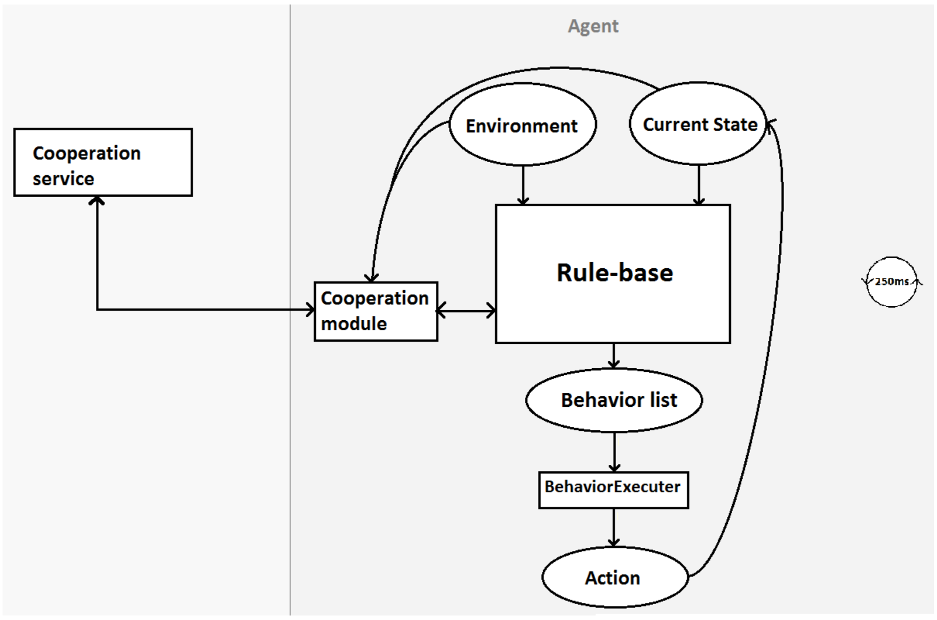

Figure 5.3: Proposed architecture to replace the central command's group-action selecter

The original AI in Heroes of Newerth are designed to cooperate in two different fashions. Based on the behaviors from table 2.2, we have the following behaviors which can make the agents cooperate:

- TeamGroup
- TeamDefend

We decided to base our work on these two behaviors. These behaviors are forced upon agents by the central command. We wanted our agents to decide on their own whether or not they wanted to cooperate. If an agent considers no cooperative interaction to be useful for him, the default action will be to stay in lane and continue what he was doing. For the remainder of this project we will only look at the *TeamGroup* behavior.

We identified two different characteristics which was needed by the agents in order to cooperate properly:

Locality
Is the action limited to an area, or is it team-wide?
Possible values: Local or global.
An example of a local action would be to defend a tower against an enemy push, while a global action could be to group up and siege an enemy tower.

Time until
How long the cooperative interaction is going to take place. This allows for a team to make plans, instead of just suggesting immediate actions. *Possible values:* As soon as possible, within 1 minute, 1-3 minutes, more than 3 minutes.

These were two variables we had to keep track of in the cooperation modules of the agents. Locality was needed if we wanted to be able to push without having the entire team. The time variable made the agents able to synchronize when to meet up before they were going to push. If it was omitted, the agents would run to the lane they were going to push, and would start pushing before everyone were there, which made them susceptible to being killed by the enemy.

On top of this, the agent has to consider the current state of the game, with special focus on himself and his surroundings. It is pointless to cooperate with your team on the other side of the map, if it leaves your part of the map open and vulnerable. Cooperating under such circumstances may potentially lead to great damage done by the enemy team.

5.4 Cooperation Protocol

As mentioned in section 5.3.1, we decided to use coalitions as our means to cooperate. This section formally describes how agents cooperate. It describes how agents initialize coalitions, how they reconsider them, and how they defect from them. First, however, we have to define what a coalition is, and describe how an agent evaluate whether a coalition will be beneficial or not.

5.4.1 Coalition description

In order for a coalition to be an effective means of cooperation, we needed to store the following information:

ID
In order to identify a specific coalition. This field is unique to every coalition.

Lane
Defines which lane to group and push.

Owner
Specifies which agent initiated the coalition.

AgentList
A list of all the agents who have joined the coalition - including the owner.

Active
Due to limitations in the AI-framework which we will discuss later, only one coalition could be active at a given time. An active coalition is one where multiple agents have decided to work together towards a common goal.

As mentioned in section 5.3.2, we will only take a closer look at the *TeamGroup* behavior. If an agent has the *TeamGroup* behavior as his current behavior, he will group up in a lane specified by the coalition. When all agents in the coalition have grouped up, they will push that lane together. Since we are only considering the *TeamGroup* behavior, the goal of the coalition will implicitly be given:

The goal of a coalition will be to kill the closest defensive building which stands in the way of the enemy main structure in the lane specified by the coalition. If no such defensive building exist, the goal will be to kill the enemy main building.

5.4.2 Forming coalitions

The easiest way a coalition is formed is depicted in figure 5.4. It starts with one agent wanting to form a coalition, which will only happen if it gets the ”push” behavior as the highest rated behavior from the rule-base (see figure 2.2 and table 2.2). The other agents on the same team are notified that a new coalition has been suggested, and asked to send their reply to cooperate or not. If the agent who considers to join also gets ”push” as the highest rated behavior, it will decide to join this coalition. If it doesn't want to join the coalition, it simply ignores it, and the suggesting agent will consider it as a no. A coalition with more than two agents is considered active, and the agents in that coalition will start to cooperate towards the coalition's goal. This procedure resembles the Contract Net protocol described in section 3.2.1, except that we allow every bidding agent to join the coalition.

Our reason for requesting all other agents on the same team to join every single coalition, instead of sending a request to nearby team-mates, is because it may be hard to calculate which agent is able to help. Heroes of Newerth has several ways to increase agents' range of influence. Examples include the teleport scroll, which allows an agent to teleport to a friendly structure, or global abilities, which can be targeted across the entire map. We feel that calculating all these special cases will require more computation than sending a few extra messages. The amount of agents a message is broadcasted to will maximum be 4, since teams consist of up to 5 agents.

We can't assume that all attempts to form a coalition will work without problems. One case happens when there are not enough agents agreeing to participate. In this case, the agent that initiated the cooperation process will have to identify that this coalition will never become active, and remove it from the cooperation service described in section 5.3.1. We decided to give each coalition containing only one agent a lifespan of fifteen seconds.

5.4.3 Reconsidering existing coalitions

An agent re-evaluates its current behavior every 250 milliseconds, as described in section 2.2.1. If the new behavior is not equal to the old one, the agent will have to reconsider all ongoing coalitions and all calls for coalition. In this subsection we only discuss ongoing coalitions. There are two possible actions of the reconsideration process:

- Continue to cooperate in the ongoing coalition

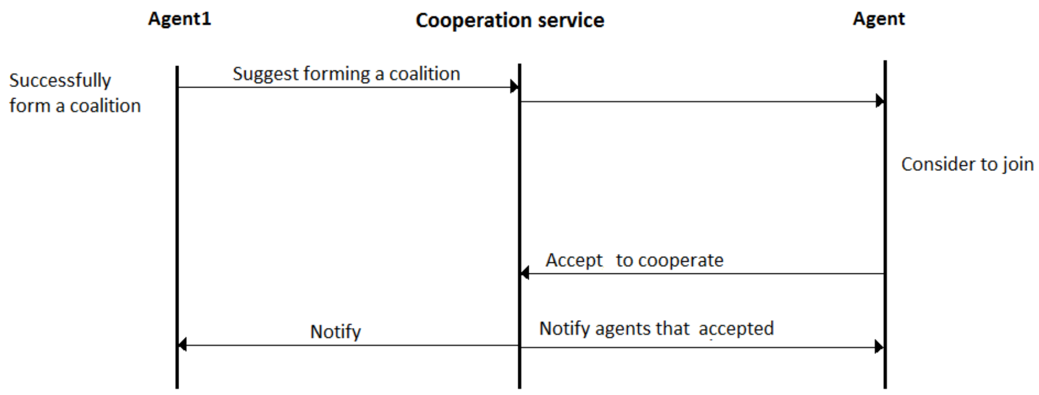
Agent1
Cooperation service
Agent
Successfully form a coalition
Suggest forming a coalition
Consider to join
Accept to cooperate
Notify
Notify agents that accepted

Figure 5.4: Protocol for suggesting and forming a coalition

- Defect from the coalition

The agents will choose the former point unless any of the following events happen:

Half of the agents in the coalition are dead or have defected
As mentioned in section 2.2.1, the original game stops a group action if half of the agents are dead. We decided to continue using it, and extended it to include defected agents. We also included that if only one agent remains in a coalition, the coalition will be dissolved. The original game did not need this rule, since a group action always consisted of the entire team.

The agent becomes critically low
In order for an agent to defect from a coalition, it has to become critically low - below 10% health points. If an agent remains in the coalition with this amount of health points, it will be an easy target and have a large chance of being killed by the enemy, which is undesirable for the team.

The latter point was added in order to address the topic of trust and commitment mentioned in section 4.3. It is meant as a mechanism which generates trust on an individual level, and will mimic a normal person's conscience. What we wish to achieve with this mechanism is to make it harder to defect from a coalition than to join one, so that agents have to keep their promise. We still don't want to keep agents from defecting if a coalition is clearly disadvantageous.

If an agent decides to defect from a coalition, the agent has to notify the cooperation service. The cooperation service will then notify all participating agents that a defection has occurred. They will then have to reconsider their participation as described in the former subsection.

Chapter 6

Experiments

This chapter describes the experiments we performed in order to test our system and evaluate our research goals from section 1.2. We start by presenting some limitations we encountered, before we introduce our evaluation method, as well as the scenarios we used for testing. The data sets we produced from simulations will not be available in their entirety, since they are too large.

6.1 Limitations

Using Heroes of Newerth for implementation and testing presented us with some limitations. The most significant limitations we encountered were:

Real time
: The game is a real time game, which force us to do calculations very fast if we want to obtain a good user experience. We also had to take steps such as not running any other software on our machines while simulations were ongoing to have as much system resources available as possible.

AI framework
: The existing framework sets the limits for our connections with the engine of the game, which prohibits us from reaching different types of information which we would need to get the best possible result. Examples include how much gold per minute and experience per minute our opponents gain.

Hardware demands and gameplay
: Heroes of Newerth is a modern computer game, and requires a modern

computer with a graphics card to run. A single match usually takes 30-60 minutes to complete. It is possible to speed up the matches, but this will demand a lot from our hardware.

Arguments at start-up
Heroes of Newerth is also not designed for doing simulations by running multiple matches in succession, like we intend to use it - you can't make it join a match on start-up without any user interaction, etc. This is why we created the dummy-client described in section 5.1.

6.2 Evaluation Methods

We have a number of different results we want to store after each match to measure the success of the experiments. The most important measurement of success is whether or not a match was won. Other important measurements include:

For evaluating case-based learning as an action-selection mechanism

- Experience gained per minute
- Gold gained per minute

For evaluating coalitions as a means to collaborate

- Coalitions suggested
- Coalitions formed
- Coalitions dissolved
- Coalitions joined
- Coalitions defected from

We will also watch some matches to see how the agents behave. During these matches, we will observe how they compare to humans based on our own experience.

6.3 Scenario

This section contains the scenarios we use for our experiments. Each scenario consist of two teams playing against each other. One of the teams will be using our suggested architecture, while the opposing team will be made up of original agents from Heroes of Newerth. This will give us a good way of comparing whether or not our solution is successful.

6.3.1 Scenario 1

Scenario 1 is a relatively simple scenario where two players play against each other on opposite teams using the same hero. This should give a good indication if any changes to the architecture of the agent improve or decline the performance of the agent. Two equal agents should over a span of multiple matches perform close to equal.

Players

- P1 as Arachna
- P2 as Arachna

  OR

- P1 as Defiler
- P2 as Defiler

  OR

- P1 as Glacius
- P2 as Glacius

  OR

- P1 as Witch Slayer
- P2 as Witch Slayer

  OR

- P1 as Chronos
- P2 as Chronos

Teams

- T1: P1
- T2: P2

6.3.2 Scenario 2

Scenario 2 is a slightly more complex scenario. In this scenario two teams of five play against each other. Each team will consist of five different heroes, and the opposing team will use the same heroes. This should also give a good indication if any changes to the architecture of the agents improve or decline the performance of the agents. Two equal teams should over a span of multiple matches perform close to equal.

Players

- P1 as Arachna
- P2 as Chronos
- P3 as Defiler
- P4 as Glacius
- P5 as Witch Slayer
- P6 as Arachna
- P7 as Chronos
- P8 as Defiler
- P9 as Glacius
- P10 as Witch Slayer

Teams

- T1: P1, P2, P3, P4, P5
- T2: P6, P7, P8, P9, P10

6.4 Experiments

The experiments are made to test our research goals and how well the modified architecture performs against and with other agents. We will base the performance measure upon how often a team manages to defeat the other, since that is the ultimate goal of a match in Heroes of Newerth. The performance will be measured over a series of matches. We ran each experiment over a period of two days, and since the duration of a match can differ, the amount of matches played can differ as well.

All matches were played on the same desktop computer with the following specifications:

CPU Intel® Pentium® Dual-Core E5500 @ 2.80GHz

RAM 4.00 GB

GPU ATI® Radeon® X1950 Pro

Operating system Microsoft® Windows® 7 Professional

6.4.1 Experiment E1: Testing case-based module as an action selection mechanism

This experiment is designed to test how well our case-based module functions as a method of action selection for agents in Heroes of Newerth, in other words it will investigate Research Goal 1. More specifically, we will compare the performance of agents with a case-based action selection mechanism against the original agents, which make use of a rule-based action selection mechanism.

We will use both Scenario 1 and Scenario 2 to evaluate the case-based module as an action selection mechanism. We will perform 2 experiments of this type, one for each approach described in section 5.2.4, only the best performing of these system are going to be tested with Scenario 1. To evaluate this experiment we are going to watch the learning curves as the exploration rate is reduced, as described in section 5.2.7.

6.4.2 Experiment E2: Testing coalition performance against the original agents without cooperation

This experiment is designed to test how well our design using coalitions performs as a means to make group decisions in Heroes of Newerth. More specifically, we will compare the performance of agents using coalitions to make group decisions against the original agents in Heroes of Newerth, but the original agents will not be able to cooperate. The original agents usually make use of a central command to cooperate, which we will disable. This experiment aims to investigate Research Goal 2, as specified in our research goals. We hope that this experiment will give a good indication whether or not our design is able to make the agents able to cooperate.

The outcomes will be evaluated based on how many matches are won by the team using coalitions. We will also take a closer look at how many coalitions are suggested, how many coalitions were created, and how many agents decided to join the different coalitions. Since the use of coalitions relies on multiple agents to be of any use, we will only evaluate this experiment using Scenario 2.

6.4.3 Experiment E3: Testing coalition performance against the original agents

This experiment is also designed to test how well our design using coalitions performs as a means to make group decisions in Heroes of Newerth. In this experiment we will compare the performance of agents using coalitions to make group decisions against the original agents which make group decisions after an arbitrary amount of time has passed. The original agents make use of a central command, while our system will be without a central command or a designated leader. This experiment aims to investigate Research Goal 2, as specified in our research goals. While experiment Experiment 2 aims to investigate whether or not our design is able to make the agents cooperate, this experiment aims to investigate if our design can be effective against an already working solution from a state of the art video game.

The outcomes will also be evaluated based on how many matches are won by the team using coalitions. We will yet again take a closer look at how many coalitions are suggested, how many coalitions were created, and how many agents decided to join the different coalitions. Since the use of coalitions relies on multiple agents to be of any use, we will only evaluate this experiment using Scenario 2.

6.4.4 Experiment E4: Testing coalition performance with a case-based module as an action selection mechanism

This experiment is designed to test how well our entire design works. That includes both the case-based action selection mechanism as well as the coalitions to make group decision. This experiment aims to evaluate Research Goal 1 and Research Goal 2. In this experiment we will compare the performance of agents using coalitions to make group decisions against the original agents which make group decisions after an arbitrary amount of time has passed. The original agents make use of a central command, while our system will be without a central command or a designated leader. The original agents will make decisions based on the existing rule-based system, while ours would use our case-based module as a decision system. Only the best of the two approaches explained in Experiment E1 is tested in this experiment.

The evaluation of this experiment is based on watching a single match.

Chapter 7

Results and Discussion

The purpose of this chapter is to explain what we found out during our experiments. Each section in this chapter represents an experiment from chapter 6. We will present the results of our experiments before we discuss them.

7.1 Experiment E1: Testing Case-Based Module as an Action Selection Mechanism

This experiment aims to evaluate how well a case-based reasoning approach works as an action selection mechanism for agents in Heroes of Newerth. To evaluate this, we ran 3 sub-experiments. The first sub-experiment used the first approach described in section 5.2.4, utilizing Scenario 2. The second sub-experiment used the second approach, while utilizing the same scenario. The approach which yielded the best result was used further to evaluate the third sub-experiment. This sub-experiment utilized Scenario 1.

The two first sub-experiments were run over a period of two days. During the simulation we had to slow down the speed of the matches, since the CBR-module had computational problems following the large amount of cases generated. The third sub-experiment was run once for each of the agent pairs, e.g. one match of Arachna vs Arachna, totaling five matches.

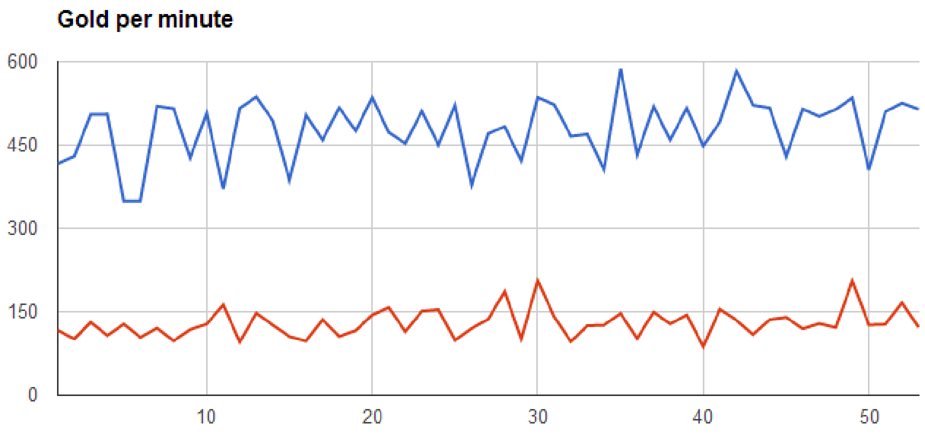

Figure 7.1: Gold per minute with the first approach

7.1.1 Results and discussion

Both of the two first sub-experiments ended up with 53 matches played. The gold and experience earned per minute by the agents are summarized in figure 7.1, figure 7.2, figure 7.3, and figure 7.4. The first two figures show the results using the first approach from section 5.2.4, while the last two figures show the results from the second approach. All graphs show the average value for each team during the matches.

The X-axis of each graph represents the number of matches played, while the Y-axis is a numerical representation of gold or experience earned per minute. The red graph in all of the figures portrays the performance of our agents, while the blue graph portrays the original AI's performance.

After each sub-experiment, we ran a single match in order to watch our agents in action. In the first sub-experiment where we used the first approach, we observed that the game was stuttering due to heavy computational load. It was only a minor grievance, and did not appear to affect the agents in any way. The agents using our design behaved very aggressive, which either led them to run back to the base in order to heal, or die a lot. In either case, the agents spent a significant amount of time away from battle. This did in turn make them lose a lot of potential experience and gold. After a while, the enemy team became too strong for our agents to handle, and our agents lost the match.

We also observed some random behavior where they walked in circles, retreat-

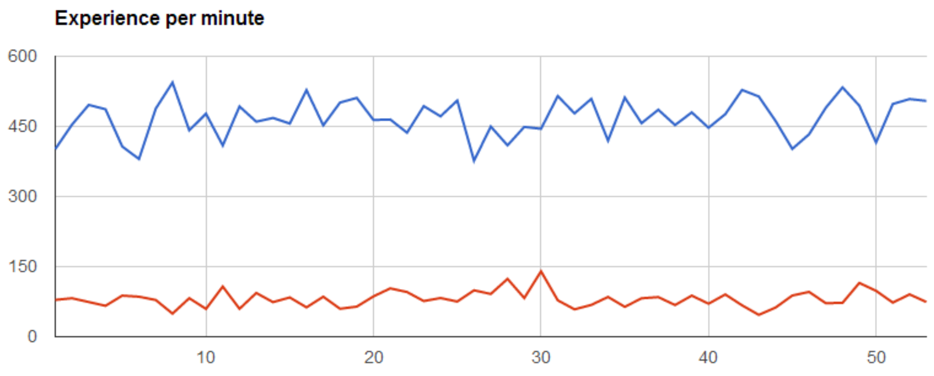

Figure 7.2: Experience per minute with the first approach

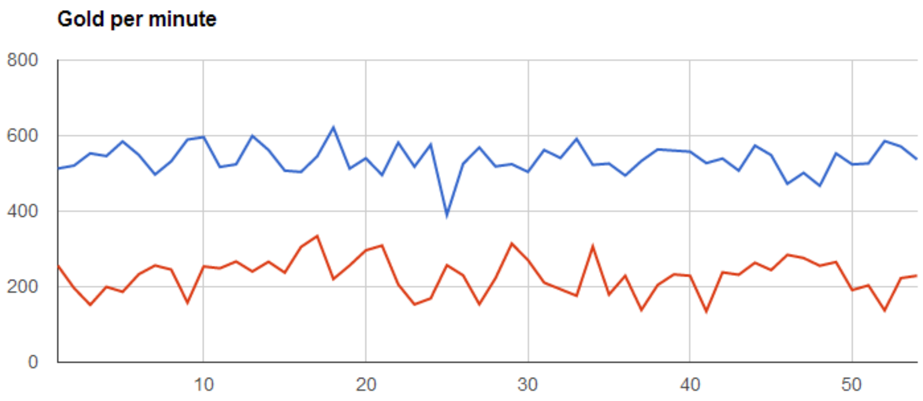

Figure 7.3: Gold per minute with the second approach

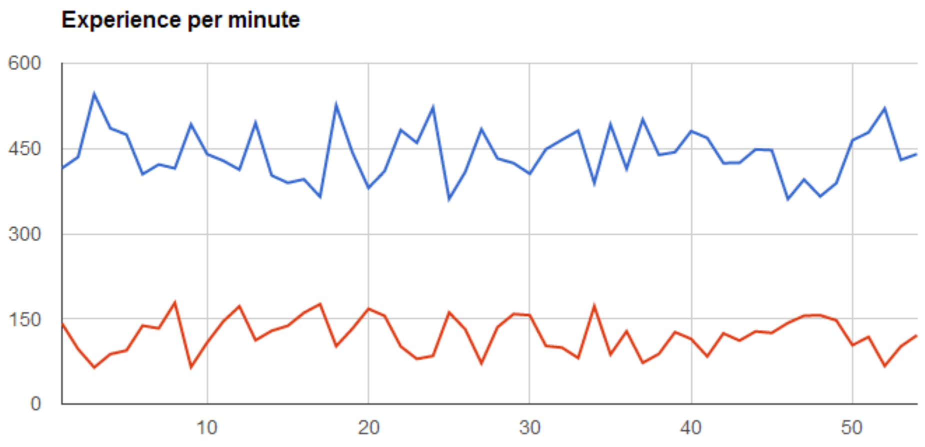

Figure 7.4: Experience per minute with the second approach

ing when in danger, and immediately returning to battle once out of harms way.

During the second sub-experiment we experienced similar results. The game was stuttering, the agents was overly aggressive, and ended up losing. However, the beginning-phase of the match we observed was different. The agents using our design managed to kill the enemy several times, which led to our agents having the upper hand. They were however unable to make use of this advantage, and after a while the enemy team capitalized on errors made by our agents. We decided to watch several matches, and this was a recurring behavior. We therefore decided to use the second approach during the third sub-experiment, along with the case-base we have generated for this sub-experiment.

For each of the sub-experiments we counted the use of behaviors. The results can be found in table 7.1 and table 7.2. We can see that the first approach used a much higher amount of "AttackCreeps" and "HealAtWell", while the second approach had a higher amount of "HarassHero". This compares well to our previous observations.

The third sub-experiment ran a single match for each of the agent-pairs in Scenario 1. We limited the agents to only use the middle lane (see chapter 2). After all the matches were played, we analyzed them and discovered that they all had a very similar pattern. We therefore decided to only include data from one of the matches, Arachna against Arachna, which can be seen in figure 7.5 and figure 7.6. Once again, the red line represents the agent using our design,

Hero	**Push**	**Harass Hero**	**Retreat**	**Use Ability**	**HealAt Well**	**Hit Building**	**Attack Creeps**	**Positon self**
Arachna	503	519	78	214	495	0	706	3270
Chronos	963	357	68	10	163	0	164	1610
Defiler	261	542	31	150	647	0	1357	3600
Glacius	237	396	57	417	302	0	1119	3376
Witch Slayer	367	514	84	334	1147	0	1781	7657

Table 7.1: Case-based reasoning with the first sub-experiment

Hero	**Push**	**Harass Hero**	**Retreat**	**Use Ability**	**HealAt Well**	**Hit Building**	**Attack Creeps**
Arachna	193	802	268	10	103	4	379
Chronos	0	299	612	1	244	4	222
Defiler	25	166	405	9	132	4	372
Glacius	0	386	425	56	103	5	452
Witch slayer	0	220	524	7	276	2	411

Table 7.2: Case-based reasoning with the second sub-experiment

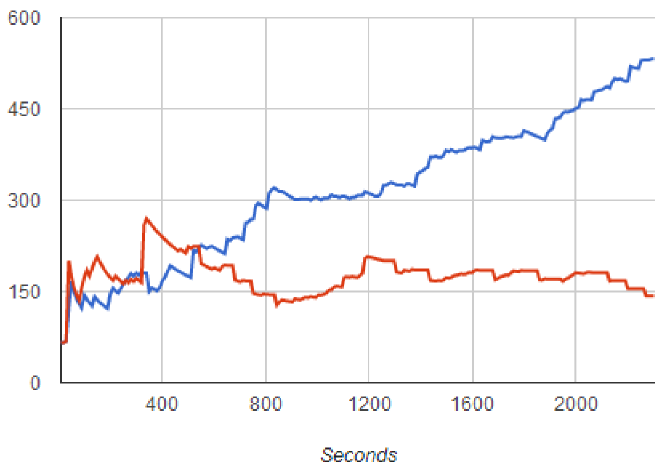

Figure 7.5: Gold per minute in a match where Arachna played Arachna

while the blue line represents the original AI. In these figures, the Y-axis is a portrayal of the performance of the agents, while the X-axis is the match length in seconds.

Like the matches we viewed during the second sub-experiment, these matches also displayed good trends in the start. Up until around 400 seconds, the agents using our design was in the lead, both when it came to experience and gold. The fact that the agents managed to learn aggression shows the potential of case-based reasoning as an action selection processes.

When looking at all of the sub-experiments, the agents using our design did not manage to win any matches. From figure 7.1, figure 7.2, figure 7.3, and figure 7.4 we see that there is no improvement from the first match to the last match. This means that the agents did no better than random when it came to winning matches. However, when we look at figure 7.6 and figure 7.5 we can see that our agents have managed to learn how to behave in the early stages of a match. This leads us to believe that a case-based reasoning approach can be used as an action selection mechanism in agents in Heroes of Newerth. Our approach did however not yield satisfactory results. We believe this is partly because we were unable to fully represent the complex environment provided by Heroes of Newerth.

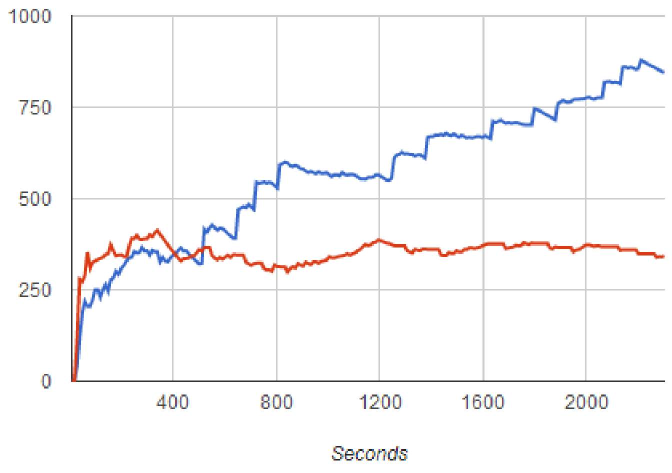

Figure 7.6: Experience per minute in a match where Arachna played Arachna

7.2 Experiment E2: Testing Coalition Performance against the Original Agents Without Cooperation

This experiment intends to evaluate whether or not our design using coalitions to cooperate works. We have tested our design against the original AI in Heroes of Newerth without their central command for cooperation. We disabled the cooperation in the original agents by commenting out some lines of code, as shown in listing 7.1 (a double hyphen is a comment in lua). The matches were played over a period of two days, running at triple speed. This resulted in a total of 159 played matches.

	Total amount	**Percentage**
Won	136	85.53%
Lost	23	14.47%
Total	159	100%

Table 7.3: Amount of matches played, won and lost against the original AI without cooperation

	Total amount	**Average per match**
Suggested	29033	182.60
Formed	3759	23.64
Dissolved	3046	19.16

Table 7.4: Amount of coalitions suggested, formed and dissolved against the original AI without cooperation

```
--StartProfile('Group␣and␣Push␣Logic')
--if self.bGroupAndPush ~= false then
--      self:GroupAndPushLogic()
--end
--StopProfile()
--
--StartProfile('Defense␣Logic')
--if self.bDefense ~= false then
--      self:DefenseLogic()
--end
--StopProfile()
```

Listing 7.1: Code showing how cooperation was turned off in the original AI taken from teambotbrain.lua

7.2.1 Results and discussion

This section presents the results and discussion from our experiment, showing how many matches we won and lost, as well as how many coalitions were suggested, initiated and dissolved. We also present information regarding how many agents decided to join each ongoing coalition.

The amount of matches won and lost, as well as the percentages are shown in table 7.3. Table 7.4 shows how many coalitions were suggested, formed and dissolved, both in total and in average per match. Table 7.5 shows how many agents joined and defected from the coalitions which were formed. If an agent didn't defect from a coalition, it was either because that coalition was active at

	Agents joined	**Agents defected**
2	2542	2297
3	161	220
4	100	244
5	956	285
Total	3759	3046

Table 7.5: Amount of agents joining and defecting the coalitions when playing against the original AI without cooperation

the end of the match, or because that agent died.

When we examine the number of matches won against the amount of matches lost, it is clear that the agents using our design has won significantly more matches than they lost. The results exceeded our predictions, and ended up at over 85% matches won. It should be clear that the implementation of our design is capable of achieving tasks in collaboration in Heroes of Newerth.

The amount of call for coalitions we found during the matches played also exceeded our expectations. During the 159 matches we played, a total of 29033 coalitions were suggested, which is 182.6 coalitions in average per match. Out of these, only 3759 coalitions were formed, which is 12.95% of the total suggested coalitions.

Since this amount of suggested coalitions was so high, and the percentage of joined coalitions were so low, we decided to spectate a match in order to see how the agents played. We discovered that very few coalitions were suggested in the start of the match. As the match progressed, and the the agents got stronger, we saw that calls for coalitions were being made. About half of the coalitions were answered, usually by the team-mate which already was in the same lane as the agent that initiated the coalition. This behavior continued for a while, sometimes with an agent from a different lane joining in.

The coalitions usually led to a tower being killed, which granted the entire team a gold-bonus. The agents used this gold-bonus to buy new items, and gained an upper hand against their opponents, which essentially were the same agents with less items. After a few towers were pushed down, and the gold lead started to become significant, coalitions with the entire team started to occur. Since the team using coalitions were already much stronger, and the enemy team did not cooperate and defend together, in the end the coalition resulted in a push which ended the match. This was after a few coalitions were disbanded because the agents underestimated the strength of the enemy towers, and became critically low.

	Total amount	**Percentage**
Won	108	75.00%
Lost	36	25.00%
Total	144	100%

Table 7.6: Amount of matches played, won and lost against the original agents

By looking at the amount of agents joining coalitions, we see that the ones where two and five agents join stand out. This observation also coincides with our observations during the match: it seemed like the coalitions were used in two different phases of the match. First, there was a phase where coalitions were mainly used locally by two and sometimes three agents. Then there was a phase where the entire team, or almost the entire team, decided to cooperate. We also saw that the amount of coalitions where all five heroes defected were significantly fewer than the ones formed. We think this either was due to those coalitions leading to a victory, or because agents died in the course of those coalitions.

7.3 Experiment E3: Testing Coalition Performance against the Original Agents

This experiment intends to evaluate whether or not our design using coalitions to cooperate is effective against the existing agents in Heroes of Newerth. We tested our design against the original agents, which use a central command to make group decisions. The original agents are able to defend as a team, or group up and push as a team. The matches were played over a period of two days, running at triple speed. This resulted in a total of 144 matches played.

7.3.1 Results and discussion

This section presents the results and discussion from our experiment, showing how many matches we won and lost, as well as how many coalitions were suggested, initiated and dissolved. We also present information regarding how many agents decided to join each ongoing coalition.

The amount of matches won and lost, as well as the percentages are shown in table 7.6. Table 7.7 shows how many coalitions were suggested, formed and dissolved, both in total and in average per match. Table 7.8 shows how many agents joined and defected from the coalitions which were formed. If an agent

	Total amount	**Average per match**
Suggested	29301	203.48
Formed	3766	26.15
Dissolved	3071	21.33

Table 7.7: Amount of coalitions suggested, formed and dissolved against the original agents

	Agents joined	**Agents defected**
2	2567	2320
3	143	247
4	126	247
5	930	257
Total	3766	3071

Table 7.8: Amount of agents joining and defecting the coalitions when playing against the original agents

didn't defect from a coalition, it was either because that coalition was active at the end of the match, or because that agent died.

As with experiment E2, the amount of won and lost matches indicates that the implementation of our design is capable of achieving tasks in collaboration in Heroes of Newerth. We were able to win 75% of the matches, which yet again exceeded our expectations. Compared to E2, the percentage has gone down by 10%, which indicates that the original agents with a central command for cooperation was a stronger opponent than those without cooperation.

Again the amount of call for coalitions we found during the matches we played were very high. During the 144 matches we played, a total of 29033 coalitions were suggested, which is 203.48 coalitions in average per match. Out of these, only 3071 coalitions were formed, which is 10.48% of the total suggested coalitions.

We decided to spectate a few matches during this experiment as well, in order to see how the agents played. There were similar tendencies to E2, with few coalitions suggested in the start of the match. This time, the enemy team was the first to group up and push, and our team had to defend. In the early match, fighting nearby a friendly tower provides a great advantage. The tower has a larger attack damage than most agents, as well as a lot of health points. Since our team was defending with an advantage, they generally came better out of the situation than the enemy team - usually killing an enemy agent.

As the match progressed, and the the agents got stronger, we saw again that calls for coalitions were being made. About half of the coalitions were answered, usually by the team-mate which already was in the same lane as the agent that initiated the coalition. This behavior continued for a while, sometimes with an agent from a different lane joining in. These two and three man coalitions were usually met with the entire opponent team defending, so unlike in E2, few coalitions directly resulted in a tower kill. The coalitions did however force the entire enemy team to defend, which left the remaining lanes free of enemy agents. This sometimes led to towers being killed in lanes other than where the coalition was taking place. Other times, the agents in the coalition were overly aggressive, and died, which gave the enemy an advantage.

Once again we saw that coalitions with the entire team started to occur later in the match. The team who had been successful earlier in the match usually managed to win the team-fights in this period of the match, which led to that team winning the match.

If we take a look at the amount of agents joining coalitions, we see once again that the ones where two and five agents join stand out. This observation also coincides with our observations during the match: coalitions are still used in two phases - the two and three man coalitions early in the match, and the larger coalitions towards the end of the match. Yet again the amount of coalitions where all five agents defected were significantly fewer than the ones formed. We draw the same conclusion - that this has to do with either those coalitions leading to a victory or defeat, or because agents died in the course of those coalitions.

7.4 Experiment E4: Testing Coalition Performance with a Case-Based Module as an Action Selection Mechanism

This experiment aims to test both the case-based reasoning module together with our design for cooperation. In other words, this experiment was made to test both Research Goal 1 and Research Goal 2, as described in section 1.2. As described in the results from experiments E2 and E3, we have already discovered that our design for cooperation has proven satisfactory results. The case-based reasoning module did however only show promising results in the early stages of matches, as described in the results from experiment E1. We used the case-bases generated by the second sub-experiment in experiment E2.

7.4.1 Results and discussion

We observed a match using our entire design in order to evaluate the performance of our agents. As in experiment E1, the game was stuttering, but this time it was stuttering even more. Very early in the match, three agents formed a coalition, and managed to kill one of the enemy agents. This seems to be a combination of what we observed in the previous experiments. Like in experiment E1, the agents using our design was aggressive and managed to get an early advantage. The agents also formed a coalition using three agents, which was something we experienced in experiment E2. Unlike the agents in experiment E2, the agents in this experiment formed the coalition at a very early stage. We believe this is a result of combining the cooperation design with the case-based reasoning mechanism.

Using coalitions this early led to our agents using a lot of time moving between lanes. This is not optimal early in a match, where agents' main gold and experience income is from enemy creeps. After several coalitions were created and dissolved, the enemy had gained a substantial advantage from our agents using so much time moving around. In the end the agents using our design ended up losing.

From this experiment we see the importance of when to cooperate. Cooperating too early can lead to positive results, such as killing an enemy agent. Even though that individual case can be advantageous, the overall outcome can be disadvantageous.

Chapter 8

Conclusion

The goal of this thesis was to investigate whether or not case-based learning and multi-agent cooperation could be applied to agents in Heroes of Newerth by expanding the original AI.

One of our goals was to investigate whether or not case-based learning could be efficiently used in a dynamic, partially observable, real-time environment such as Heroes of Newerth. As stated in Research Goal 1 we aimed to use case-based learning as an action selection mechanism for agents in Heroes of Newerth. Our experiments show that case-based reasoning is too time consuming for real-time games, unless it's run less frequently. They also show that a complex, partially observable environment needs many variables in order to be represented properly. Every variable left out would yield a weaker overall performance. We also observed some positive tendencies, especially in the early phases of a match. This leads us to believe that a case-based reasoning action selection mechanism can be applied to real-time video games.

Our second goal was to investigate if we could use multi-agent cooperation to improve the performance of the agents in Heroes of Newerth. Our second research goal, Research Goal 2, aims to find out how well the use of coalitions for achieving tasks in collaboration will work in Heroes of Newerth. We have found that using our methods have yielded a significant improvement against the original AI of Heroes of Newerth.

Overall our project has shown that using these methods for agents in Heroes of Newerth is feasible, and can improve the performance of the built-in AI.

Chapter 9

Future Work

There are many possibilities for improving our current solution. An optimal solution would include remaking the behavior system in the current AI to better fit CBR. We also discovered that changing how cases are compared could yield a huge difference in computation needed. Creating a more sophisticated method of comparing cases could yield increased performance, which in turn would allow for more cases in the case-base. We believe that more cases could make the agents perform better at later stages in matches.

The agents' cooperative abilities can be improved by including the *TeamDefend* behavior, explained in section 2.2.1. This can allow agents to defend more dynamically against enemy pushes.

Appendix A

Resources

Table A.1 explains the different resources we used throughout our Master degree.

RawCap	**RawCap** is a free command line network sniffer program for Windows that uses raw sockets. It has the ability to sniff packages going through localhost, meaning it sniffs packages sent to, from and inside the computer. RawCap saves the data read in datapackages, which is not user friendly.
WIRESHARK	**Wireshark** is a free and open-source packet analyzer program. It is used for network trubleshooting and has the ability to structure datapackages, making them readable.
H	**AutoHotkey** is a free open-source macro-creation and automation software utility which allows users to automate repetitive tasks. The program writes a bunch of choosen text on the users commands, either when it's started or when a selected hot-key is clicked.
Lua	**Lua** is a lightweight multi-paradigm programming language. Lua is designed as a scripting language.

	Eclipse is an integrated development environment (IDE) where one could run and compile code in programming languages such as Lua and more.
	Cygwin is a unix-like environment and command-line interface for Windows.

Table A.1: Resources

Bibliography

[1] General Learning Agent - Wikipedia. http://upload.wikimedia.org/wikipedia/commons/0/09/IntelligentAgent-Learning.png. [Online; accessed 11-December-2013].

[2] Agnar Aamodt and Enric Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI COMMUNICATIONS*, 7(1):39–59, 1994.

[3] Wikipedia - Heroes of Newerth. http://en.wikipedia.org/wiki/Heroes_of_Newerth. [Online; accessed 11-December-2013].

[4] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[5] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[6] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[7] E.H. Durfee, V.R. Lesser, and D.D Corkill. Cooperative Distributed Problem Solving. *The Handbook of Artificial Intelligence*, IV:83–147, 1989.

[8] Reid G. Smith and Randall Davis. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 11:61–70, 1981.

[9] Reid G. Smith. The contract net: A formalism for the control of distributed problem solving. In R. Reddy, editor, *IJCAI*, page 472. William Kaufmann, 1977.

[10] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput.*, 29(12):1104–1113, December 1980.

[11] Thomas M. Mitchell. *Machine learning.* McGraw-Hill, 1997.

[12] Bryan Horling and Victor Lesser. A Survey of Multi-Agent Organizational Paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2005.

[13] M. S. Fox. An organizational view of distributed systems. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 140–150. Kaufmann, San Mateo, CA, 1988.

[14] Habib Rajabi Mashhadi and Reza Monsefi. A novel algorithm for coalition formation in multiagent systems using cooperative game theory. In *The Iranian Conference on Electrical Engineering*, 2010.

[15] Onn Shehory, Katia P. Sycara, and Somesh Jha. Multi-agent coordination through coalition formation. In Munindar P. Singh, Anand S. Rao, and Michael Wooldridge, editors, *ATAL*, volume 1365 of *Lecture Notes in Computer Science*, pages 143–154. Springer, 1997.

[16] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1-2):165–200, May 1998.

[17] N. Sugandh S. Ontanon, K. Mishra and A. Ram. Case-based planning and execution for real-time strategy games. *Artificial Intelligence*, 2007.

[18] M. Buro and T. Furtak. Rts games as test–bed for real–time ai research. *Artificial Intelligence*, 2003.

[19] Alexander Nareyek. Artificial Intelligence for Computer Games. `http://www.ai-center.com/projects/excalibur/documentation/intro/aigames.html`, 2001. [Online; accessed 25-September-2013].

[20] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative learning. `http://web.mit.edu/16.412j/www/html/Advanced%20lectures/2004/Multi-AgentReinforcementLearningIndependentVersusCooperativeAgents.pdf`, 1993. [Online; accessed 02-November-2013].

[21] M. Molineaux D. W. Aha and M. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. *Artificial Intelligence*, 2005.

[22] Ben G. Weber and Michael Mateas. Case-based reasoning for build order in real-time strategy games. *Artificial Intelligence*, 2009.

[23] T. Szczepański and A. Aamodt. Case-based reasoning for improved micromanagement in real-time strategy games. *Artificial Intelligence*, 2009.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An Introduction.* The MIT press, 2005.

[25] G. Weiss and S. Sen. Adaptation and Learning in Multiagent Systems. `http://link.springer.com/chapter/10.1007/3-540-60923-7_16`, 1996. [Online; accessed 25-October-2013].

[26] H. Berenji and D. Vengerov. Advantages of cooperation between reinforcement learning agents in difficult stochastic problems. `http://ti.arc.nasa.gov/m/pub-archive/91h/0091%20(Berenji).pdf`, 2000. [Online; accessed 02-November-2013].

[27] R. Graham. Real-time agent navigation with neural networks for computer games. *Artificial Intelligence*, 2006.

[28] S2Games. Heroes of newerth bots forum. `http://forums.heroesofnewerth.com/forumdisplay.php?290-Bots`.

[29] N.R. Jennings. Commitments and Conventions: The Foundation of Coordination in Multi-Agent Systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.

[30] Sarvapali D. Ramchurn, Dong Huynh, and Nicholas R. Jennings. Trust in multi-agent systems. *Knowl. Eng. Rev.*, 19(1):1–25, March 2004.

[31] Robert Axelrod. *The Evolution of Cooperation.* Basic Books, New York, 1984.

[32] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, 3rd edition, 2012.