Aalborg University

Department of Computer Science

Title:

Behavior Based Fuzzy Logic for World of Warcraft

Topic:

Machine Intelligence

Project Period:

Februar 1st 2012 - June 8th 2012

Project Group: sw108f12

Søren Møller Larsen

Jon List Thorsteinsson

Supervisor:

Paolo Viappiani

Number of appendices: 3

Total number of pages: 69

Number of pages in report: 62

Number of reports printed: 4

Abstract:

This project covers the development of an autonomous agent capable of playing against human players in World of Warcraft, and is an extension of work done in the previous project.

An expert system that combines a behavior based approach with fuzzy logic has been developed, in which the fuzzy rules are weighted. The weights of the rules are adjusted with reinforcement learning to ease the configuration.

The experimental results shows that it is easy to configure an agent in the expert system and that the weights of the fuzzy rules are learned within a short period of time. The agent is far superior to the previously developed scripted agent and is capable of challenging experienced human players in a 3 versus 3 player scenario.

Summary

In the previous project we developed an autonomous agent capable of playing against human players in the video game World of Warcraft. The implementation consisted of several essential components, such as an advanced navigation system using navigation meshes and influence mapping, APIs for interacting with the game client and action selection based on scripted decision making. This project extends the agent by improving the action selection and navigation in close combat.

The previously developed action selection has been replaced with an expert system that combines a behavior based approach with fuzzy logic, in which the fuzzy rules are weighted. The expert knowledge in the fuzzy rules are configured as "If *Health[enemy]* IS *Low* THEN *Attack[enemy]*", which lets the expert user think in scenarios and intentions (attack the enemy with low health), rather than in specific actions and conditions, thereby easing the configuration of the agent. The behaviors ared defined as abstractions over several actions but for a single intention e.g. *attack* or *heal*, instead of having to configure actions and conditions for all possible scenarios. To reduce the problem of conflicting rules i.e. rules that dictate different behaviors in the same scenario, the rules are weighted based on their importance. These weights are learned with reinforcement learning to ease the configuration.

An experiment were conducted using teams of 3 characters, a *melee*, a *healer* and a *ranged* character, each indidually controlled. The results show that it is easy to configure an agent in the expert system and that the reinforcement learning learns the weights of the fuzzy rules within a short period of time. After 10 games the *Learning Fuzzy Team* (agents using fuzzy rules and reinforcement learning) starts to perform well against the *Fuzzy Team* (agents using rules with weights configured by expert users). Testing the teams in 100 fights shows that the *Learning Fuzzy Team* outperforms the *Fuzzy Team* with a 69% win rate. The agent is far superior to the previously developed scripted agent with a 96% win rate, and is capable of challenging experienced human players with a 51% win rate.

Preface

This report substantiates the result of Software Engineering group sw108f12's 10^{th} semester project at the Department of Computer Science, Aalborg University. The report is documentation for the project, proceeding in the period from the 1^{st} of February 2012 until the 8^{th} of June 2012.

Unless otherwise noted, this report uses the following conventions:

- Cites and references to sources will be denoted by square brackets containing the author's surname, and the year of publishing. The references each corresponds to an entry in the bibliography on page 63.
- Abbreviations will be represented in their extended form the first time they appear.
- When a person is mentioned as *he* in the report, it refers to *he/she*.

Throughout the report, the following typographical conventions will be used:

- References to variables and functions in code listings are made in `monospace` font.

Appendices are located at the end of the report. The source code for the software project, the report in PDF as well as online references are included on the attached CD-ROM.

Søren Møller Larsen

Jon List Thorsteinsson

Contents

Chapter 1

Introduction

The purpose of this project is to enhance the previously developed agent, which is able to play the video game World of Warcraft. The previous project focused on developing an API to interact with the game, the construction of a solid architectural development platform, and implementing pathfinding that uses a Navigation Mesh and Influence Mapping. Additionally a scripted decision making component was implemented for fighting in the game.

This project focuses on enhancing the decision making by combining Fuzzy Rules, Reinforcement Learning and Potential Fields to create a more intelligent way of choosing actions and reduce/automate the work needed in order to configure a new agent. The goal is to create a design that allows expert knowledge to easily be documented and interpreted allowing the agent to gain a better understanding of which behavior is appropriate in different scenarios during the fight, instead of just choosing an action based on scripted requirements.

Experimental results are done by letting the agents compete against human players, previously developed scripted agents and different configurations of the new agent to identify improvements of implementing the intelligent action selection.

The goals for this project can be summarized to the following statements:

- Use fuzzy logic to allow expert knowledge to be easily modeled and interpreted by the agent.
- Implement Potential Fields to improve the movement of the agent in the environment.
- Use reinforcement learning to automate and improve the configuration of new agents.
- Visualize the different techniques to ease tweaking and tuning of agents.
- Setting up a suitable test environment for both batched testing, training, and for playing against human players.

Chapter 2

Related Work

The Previous Project

In the previous project[Larsen and Thorsteinsson, 2011] we created an autonomous agent that was able to play with human players in the game World of Warcraft. The implementation consisted of developing: an API that can interact with the game client, an advanced pathfinding built on navigation meshes and influence mapping, and action selection based on a scripted decision making. This project focuses on enhancing the decision making using machine intelligence techniques, such as fuzzy logic and reinforcement learning. The World of Warcraft API, World of Warcraft Interaction API, Navigation System, and 3D visualization of the environment, all developed in the previous project, is used throughout this project.

AI for Games

In order for AI to work in a computer game, several components must be implemented. [Nareyek, 2004] identifies the two most important components, the moving and steering, and the decision making. They give an overview of various techniques for decision making such as finite state machines and decision trees, and mention the A* algorithm for pathfinding and influence mapping for terrain analysis. A finite state machine could in our case have been used to define the transitions between behaviors; however, it would make it much harder to incorporate expert knowledge as this creates dependencies between the behaviors and requires crisp conditions for taking a transition e.g., the transition *go from attack behavior to defensive behavior if health is less than 30 %*, creates a dependency between the attack and defensive behavior and uses a crisp percentage value. The mentioned Influence mapping and A* algorithm is also a part of the agents advanced navigation system developed in the previous project, but we preferred potential fields over influence mapping for close combat scenarios. [Nareyek, 2004] states that generalizing AI is very hard, as the decisions that needs to be taken differs a lot based on the game, which is also what we experienced during the design of the framework and the configuration of the agents.

Potential Fields

As positioning is an important aspect of the game, the agent needs to be able to move intelligently. The navigation system developed in the previous project provides a pathfinding

service, but does not provide a suitable method for positioning in close combat scenarios. [Goodrich, 2004] describes how potential fields can be implemented and used to find a path from one location to another, avoiding obstacles on the path using attractive and repulsive potential fields. As the agent already have a suitable pathfinding, repulsive potential fields for each enemy and attractive potential fields for each friend has been used to determine the optimal direction at any given point and the previously developed pathfinding has been used to find a suitable path in that direction.

Fuzzy Logic

Expert knowledge can be used to increase the performance of the agent, by giving it a better basis for decision making. [Hellmann, 2001] describes how fuzzy logic works with rules, predicates and membership functions and gives the foundation for the syntax of the fuzzy rules used in this project.

The agent has a large amount of actions to choose amongst and the right action is based on both the intention of using that action and different information from the environment, a behavior-based approach is suitable to abstract over the actions and focus on the intentions. [Bonarini et al., 2003a] describes how they have combined behavioral modules with fuzzy logic, using *WANTTO* and *CANDO* conditions. This is similar to our approach, though their *CANDO*s are based on a fuzzy evaluation with a threshold whereas our *CANDO*s are dependent on whether the behavior contains a usable action.

In order to associate fuzzy predicates with objects in the environment the agent must build an internal representation of the environment. [Bonarini et al., 2003b] uses fuzzy logic to control a robot playing soccer, and what they call *concepts* to describe and map observations about the environment to specific domain specific objects i.e., something round and red is mapped to a Ball object. Fuzzy predicates related to a Ball object can now be evaluated using the internal object representation. This is similar to our approach where we create predicates for an object class rather than a specific object. A predicate for an object class is a predicate that is not associated to a specific object in the environment, but to the set of objects in the environment that are included in the object class i.e., objects that share certain traits.

Reinforcement Learning

[Wender and Watson, 2008] uses reinforcement learning to develop a policy for an agent to select a city location in the game Civilization IV. Reinforcement learning could potentially have been used in our project to create an action policy, as an alternative to using expert knowledge. In this project, reinforcement learning has been used to handle inconsistencies in the rule base by adjusting the weights on the fuzzy rules; adjusting the expert knowledge rather than learning the whole action policy.
Another alternative could be to use a Partially observable Markov decision process (POMDP) to calculate a policy, but as the state-space of the game is very large it would be infeasible to

compute. [Tan and Cheng, 2009] uses IMPLANT a combination of POMDP and MDP to reduce the amount of time it takes to compute. The IMPLANT is used in a very simple game and requires a model of the environment; which makes it infeasible to scale up to a game like World of Warcraft because of the complexity of the game.

Expert Systems

In this project we develop an expert system using Fuzzy Rules. Fuzzy CLIPS[1] is a Fuzzy Logic extension for the Open Source expert System CLIPS[2] developed in C, and uses a generalized way of implementing fuzzy rules in its own language. Jess[3] is an alternative and was originally constructed as an Java clone of CLIPS but has been further developed since, it bases on the same type of LISP syntax as CLIPS but does not support fuzzy rules.
Using an external framework is often a better idea than re-developing some functionality due to potentially decreased implementation time, and potentially increased quality. Building on top of one of these frameworks also means that developers familiar with the frameworks would have a better understanding of the application and how to use it.
Both of these framework could have been an alternative framework to build our expert system on; however, as our current framework is developed in C# and combines fuzzy rules with a behavior based approach and adds weight to the rules, we developed the expert system inside our current framework in a similar approach as the previously developed scripted agent[Larsen and Thorsteinsson, 2011].

[1] `http://awesom.eu/~cygal/archives/2010/04/22/fuzzyclips_downloads/index.html`
[2] `http://clipsrules.sourceforge.net/`
[3] `http://www.jessrules.com/`

Chapter 3

World of Warcraft

This chapter give a small introduction to the game World of Warcraft (WoW)[1], which is the game that the agent is developed to play. Throughout the report terms from the game universe are used, and this section provides a better understanding of what the game is about and what some of these terms cover. WoW is a massively multiplayer online role playing game (MMORGP), and has more than 11 million active players [Activision, 2011]. Like most role playing games, the gameplay of WoW consists of casting spells and swinging swords to either defend/rescue friends or to slay evil foes.

3.1 Players and Characters

A *player* is the human playing the game on their computer while a *character* is the in-game avatar that the player controls. A player can have multiple characters but only one character per player can be online on the game server at a time. To play the game, the player logs in and selects the character that they what to bring online and play with. The player navigates the character around in the 3D world using a third person, over the shoulder point of view.
Characters in WoW are divided in to classes, defining the combat skills of a character e.g., a character of the warrior class is skillful with swords and a character of the hunter class uses bows and talks to animals. The player chooses what class their characters should be, but once the class is selected it cannot be changed. There are ten different classes in WoW and each of these have their own skill set defining their role in combat scenarios.
Some character traits are independent of class: a set of attributes describing the strengths of the character, a *level* describing how far in the game the character has progressed, a limited amount *health points* (HP) and a limited amount of resources used to perform actions such as casting spells.
Health points represent how much life a character has; when a character is hurt by another character he loses HP. If the HP of a character reaches zero, the character dies.
The type of resource that a character uses to cast spells or perform other actions, depends of the class; however the idea of using a resource to perform an action is shared across classes e.g., a character of the Mage class uses mana as a resource to cast spells. *Power* is used as a general term of all the types of resources the different classes use.

[1] http://us.battle.net/wow/en/

The level of the character describes how far in the game the character has progressed, the current maximum level of WoW is level 85, at this level the characters can engage in the *end game* which consists of defeating great dragons and competing in player versus player(PvP) matches.

3.2 Spells

Each class has a preset arsenal of spells that characters of that specific class can use in combat. A *spell* is an ability that a character can perform for example *Fireball*, *Disarm*, *Fear* etc.
To describe the characteristics of spells, the spell *Chain Lightning* is used as an example, this spell is described with the in-game tooltip of Figure 3.1

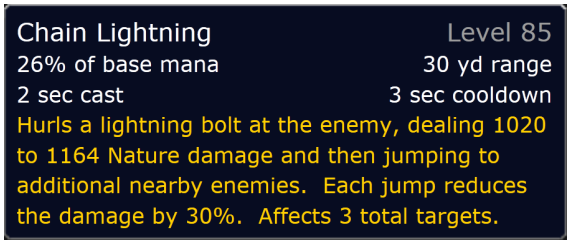


Figure 3.1: Chain Lightning Description Tooltip

The first line of the tooltip is the name of the spell; the next two lines describe important features of this particular spell and are explained below:

- *26% of base mana*: The amount of resources required to cast the spell, mana in this case. At some point when the character has no more mana this spell cannot be used before the character regains mana. Resource cost of spells varies a lot, powerful spells cost more than not so powerful spells.
- *30 yd range*: The range of the spell in game yards. To cast this spell the character must be within 30 game yards of the target of the spell.
- *2 sec cast*: The amount of time it takes from when the player presses the button till the character casts the spell. From a role playing point of view, a casttime is a way to illustrate that it takes some effort for the character to cast the spell. From a game balance point of view a casttime prevents a character from casting too many spells within a short period of time. A character must remain stationary while waiting for the casttime.
- *3 sec cooldown*: The cooldown is the amount of time that must pass before the spell can be cast again. Once the character has cast the spell, the spell cannot be cast in the next three seconds, the spell is hot and must cool down before it can be cast again. Some spells does not have a cooldown, when activated these spells apply a *global cooldown* to all the spells of a character, so that the character cannot cast any spells for the given amount of time i.e., casting an instant spells makes all spells of a character hot. The default time of the global cooldown is one and a half second. A cooldown is used to prevent characters from casting very powerful spell too often.

In addition to this most spells must be targeted at a character.

3.3 Roles

The class of a character dictates what spells that character is able to cast, but overall the classes can be separated into three categories defining the role of the character.

Melee: A melee character is a close combat fighter, his spells revolve around getting close to his target, preventing the target from getting away and to inflict damage to the target while it is close.
Ranged: A ranged character fights at range, his' spells revolve around getting away from melee characters, to keep melee characters at range and to inflict damage from a range.
Healer: A healer character is the medic of the game, his spells revolve around replenishing friendly players HP to prevent them from dying.

3.4 Client Interface

The screenshot in Figure 3.2 shows the client interface of WoW. There are the following marked

Figure 3.2: Screenshot of a World of Warcraft Client

areas in the screenshot:

A This area represents the status of the character controlled by the player. The green bar represents the health status and the blue bar represents the power status. The number

85 in the circle is the level of the character.

B This area shows the health status of all the characters in the team. In this way, the player is always able to see if a friendly character is low in health. Using this is particularly useful for healers.

C This area contains the icons representing the abilities usable by the character.

D The camera always points towards the character in the middle, controlled by the player. The player may use the mouse to swing the camera around and get another view angle.

3.5 Player versus Player

WoW is characterized as a so-called *theme park game*, i.e., a game in which there are a lot of different activities to engage in. One of these activities is PvP arena fights. PvP arena fights are gladiator fights, where two teams of characters are put in side an arena, and the team that survives, wins. There are three different ways to fight in an arena in WoW: two versus two, three versus three, and five versus five, and there are ten different classes in WoW so there are quite a few combinations of teams and classes. The agent developed in this project is developed for a fixed scenario: a three versus three game with a Hunter, a Druid and a Priest on each of the teams. The Hunter is a ranged class, the Druid is a melee class and the Priest is a healer class.

To be successful in PvP, a player must understand a couple of core concepts of the game, these terms and concepts are used in later chapters when describing how the agent executes them.

Crowd Control: Crowd control (CC) is a term used to describe the act of making a character unable to cast spells or unable to move for a certain amount of time. Every character has around 5 CC spells at their disposal. CC is obviously very powerful, and with the right timing CC can win a team the game.
The healer in is often the target of most of the CC spells in a game, because without the healer the rest of the team has no way to replenish HP and will eventually die from the damage coming from the other team. The duration of a CC spell is ten seconds, with a diminishing return attached, which means that a character cannot be crowd controlled indefinitely.

Bursting: To burst a character is to inflict a large amount of damage on that character for a short amount of time. This can be done if players of a team coordinate and activate damage-increasing spells at the same time and agree to attack the same enemy character. A common strategy is to CC the healer while bursting a character that that has low HP to kill him while the healer is sidelined.

Kiting: In video game terminology kiting is a term describing the act of keeping distance between a ranged and a melee character. The ranged character uses spells that slows or roots the melee character, that way the ranged character can inflict damage to the melee character while running and keeping distance, but the melee character cannot inflict damage to the ranged character. The term is a metaphor for running with a kite.

Kiting is most useful for ranged characters, but the healer can also to some extend *kite* the melee characters to avoid taking to much damage.

These concepts are all very important in order for the agent to perform well in PvP scenarios. Strategies, such as CC the enemy healer when another enemy is about to die, are configured using fuzzy rules. The potential fields allow the ranged and the healer agent to *kite* and increases their survivability.

Chapter 4

Background

This chapter contains a general description of some of the techniques used in later chapters and in the application.

4.1 Fuzzy Logic

Fuzzy logic is as an alternative to traditional two-valued based logic, where all propositions are either true or false[Hellmann, 2001]. It is used in scenarios where true or false values are not adequate, but a more nuanced view is needed. An example is if the agent should exhibit some predefined behavior in case another agent is injured. The answer to whether another agent is injured would in boolean logic be either true or false. Using fuzzy logic, the answer would be a value between 0 and 1 describing how true it is that the other agent is injured. Based on this more nuanced answer, and possibly similar answers to different queries, the agent is able to make a more informed choice, which motivates the usage of fuzzy logic.
In the field of machine intelligence fuzzy logic requires the application of the following three techniques:

1. *Fuzzification* of input values
2. Evaluation of fuzzy rules
3. *Defuzzification* of evaluation results

4.1.1 Fuzzification

In the fuzzification, the input values of the environment are mapped to degrees of membership to fuzzy predicates. For example 4 health points could be mapped to *poor health* while 100 health points could be mapped to *good health*. This mapping is done through the definition of the membership functions.

Predicates and Membership Functions

A predicate, sometimes called linguistic variable is a fuzzy variable that defines a degree of membership to associated membership functions.

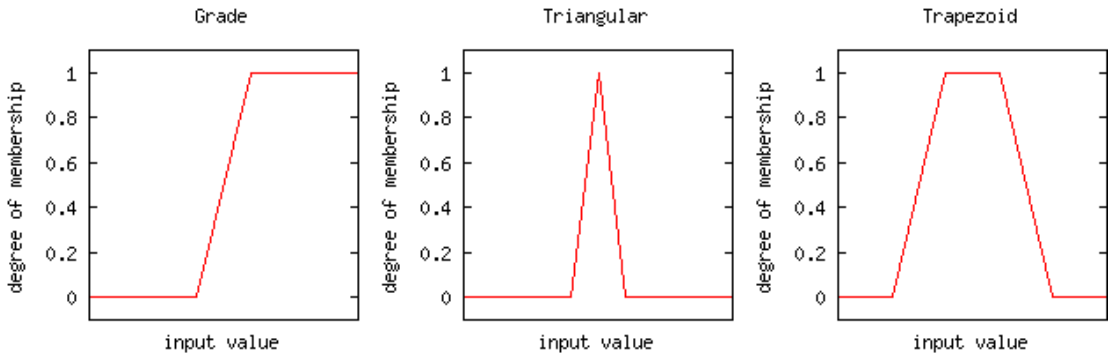


Figure 4.1: Common Membership Functions

For example, a *health* predicate could have the membership functions *poor*, *average*, and *good* associated to it, and each of these membership functions would map an input value to a degree of membership.
Membership functions are defined as functions of the input value and results in the degree of membership as a real value ranging from 0 to 1. Some of the typical membership functions are illustrated in Figure 4.1; for comparison a similar function for boolean logic is illustrated in Figure 4.2.
The *health* predicate described earlier could be defined using the membership functions illustrated Function 4.3 with the health points out the x-axis and degree of membership up the y-axis.

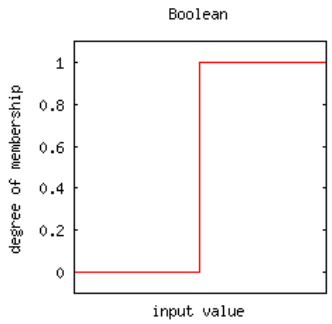


Figure 4.2: Boolean membership functions

4.1.2 Evaluation of Fuzzy Rules

The definition of fuzzy rules uses a familiar syntax: IF <*condition*> THEN <*action*>, and control of an agent can be done by defining behaviors with fuzzy rules. The *matchgrade* is the result of the membership function evaluation from all the conditions in the syntax. Imagine an agent that has the ability to attack, flee or do nothing, the fuzzy rules defining the behaviors for such an agent could look similar to the following:

- IF health IS poor THEN flee

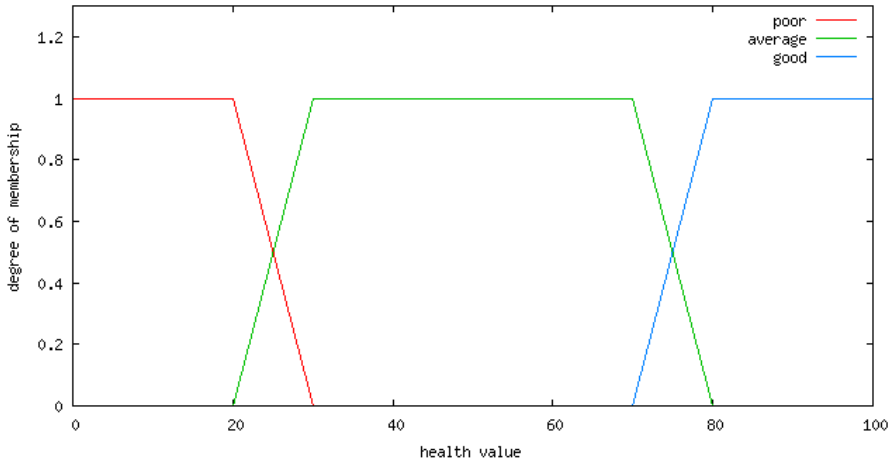


Figure 4.3: Health Membership Functions

- IF health IS average THEN do nothing
- IF health IS good THEN attack

Each of these rules will have a membership degree, a *matchgrade*, associated to them, depending on the input value and the definition of the membership functions. If the membership functions are defined as in Figure 4.3 and the input value of health is 27 the matchgrades of the three rules is approximately be as follows:

- **0.3** for *IF health IS poor THEN flee*
- **0.7** for *IF health IS average THEN do nothing*
- **0.0** for *IF health IS good THEN attack*

Logic Operators

Like in boolean logic, statements in fuzzy logic can include logic operators such as **AND**, **OR**, and **NOT**. In fuzzy logic the operators are defined as follows:

- A **AND** B = MIN(A,B)
- A **OR** B = MAX(A,B)
- **NOT** A = 1 - A

Where A and B both represent a degree of truth. The definition of **AND** means that the value of the operand that is the least true is used as the matchgrade for the whole statement, similar for the **OR** operator except that the value that is most true is used. The **NOT** operator simply inverses the value.
The matchgrade of a rule is the combination the matchgrades of the membership functions of predicates used in the rule. The rule *IF health IS poor THEN flee*, for example, has a matchgrade of 0.3 because when using 27 as the input value, the matchgrade for the membership function *poor* has a of the *health* predicate is 0.3 (see Figure 4.3).

4.1.3 Defuzzification

The matchgrades are used to select which action to perform; this is done through the process called defuzzification. One way to find the appropriate action, is to predefine a membership

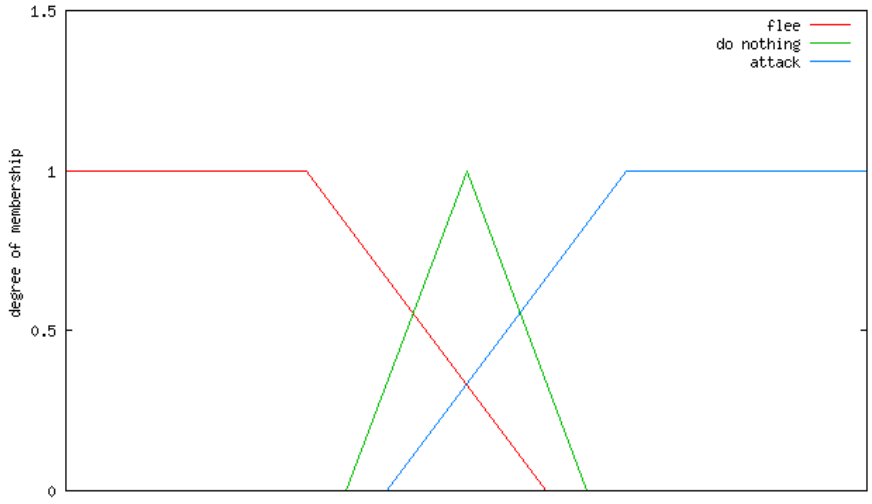


Figure 4.4: Membership Functions for Actions

function for every action, as illustrated in Figure 4.4, these membership functions illustrates the relationship between the actions. When the fuzzy rules are evaluated, the highest matchgrade on each action is used to fill in the membership in the graph as illustrated in Figure 4.5, now the action can be found by using the geometric centroid of the composite membership function represented as the entire dark area. The action is selected based on which part of the composite membership function the centroid is situated in, illustrated by the line above the centroid that results in the *do nothing* action with a matchgrade on 0.6.

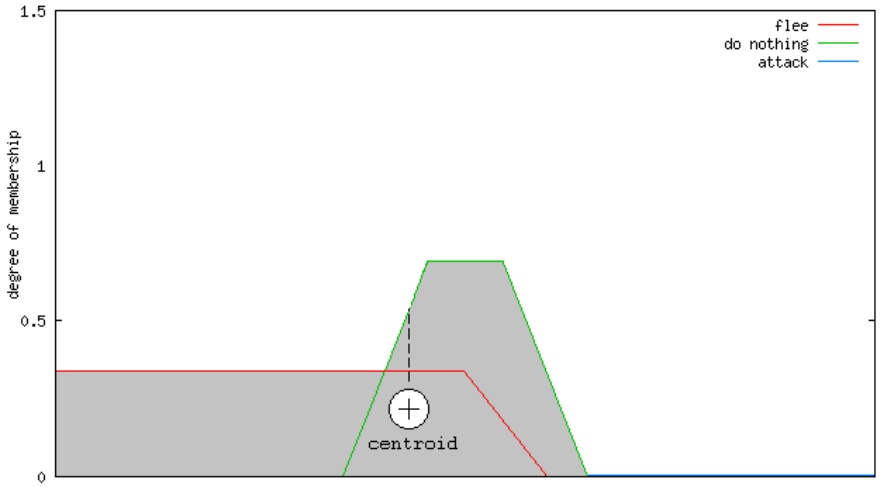


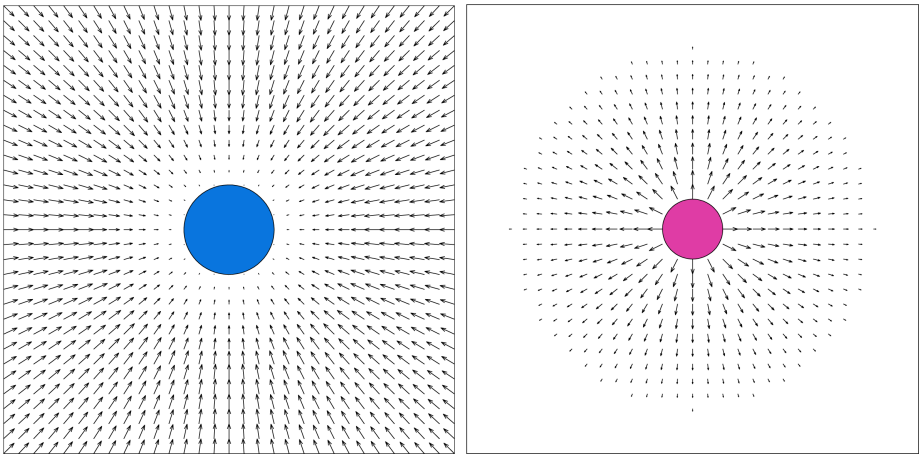
Figure 4.5: Composite Membership Function for Actions

The centroid approach is only useful if the actions are related, consider for example the rule

IF health IS good THEN dance, how would the membership function for *dance* relate the other membership functions and how would that affect the centroid calculation?
An alternative to the action selection is to take the action from the rule with the highest match-grade, which in this case would be *do nothing*.
In our project we adept the simple alternative, because all the actions that agents can perform cannot be described using a membership function relationship as in Figure 4.4. Just as it would be hard to relate a *dance* action to *attack* and *flee*, actions that the agent for this project can choose from are not immediately relatable.

4.2 Potential Fields

Potential fields are a technique used in robotics to control the movement of robots through an environment[Goodrich, 2004]. A potential field is similar to a magnetic field in that some points are attractive and others are repulsive. The agent is drawn towards points in the field that are attractive and it avoids points in the field that are repulsive. A way to visualize the field is to think of the agent as a ball rolling around in a hilly landscape, hills in the landscape are repulsive and valleys are attractive.
Different from other implementations, the agents developed for this project do not use the potential fields for pathfinding, but uses more traditional pathfinding algorithms. The potential field is instead used to guide the agent on a smaller scale and the pathfinding algorithms are used on a larger scale.



(a) (b)

Figure 4.6: Attractive and repulsive potential fields

4.2.1 Attractive Fields

Figure 4.6a is an illustration of an attractive field, the field has three properties: the radius of the goal (r), the spread of the field (s), and the strength modifier (α).
Figure 4.7 is a description of the field, where d is the distance from the field and θ is the angle between the agent and the goal. As can be seen in the formula the field is separated into

three different formulas. Inside the goal there are no forces which means that once something reaches the goal it will no longer move, outside the goal but inside the spread the force is determined by the distance to the goal with a more powerful pull the longer you get from the goal, outside the spread the pull uniform.

$$\begin{array}{cc} \text{if } d < r & \Delta x = \Delta y = 0 \\ \text{if } r \leq d \leq s + r & \Delta x = \alpha(d - r)\cos(\theta) \text{ and } \Delta y = \alpha(d - r)\sin(\theta) \\ \text{if } d > s + r & \Delta x = \alpha s \cos(\theta) \text{ and } \Delta y = \alpha s \sin(\theta) \end{array}$$

Figure 4.7: Formula for attractive potential field

The agent creates attractive fields around friendly characters in the game; the result of this is that the characters of one team are seldom very far away from each other, making it easier for the healer on the team to do his job.

4.2.2 Repulsive Fields

Figure 4.6b is an illustration of an repulsive field, like the attractive field it is defined using three properties: the radius of the obstacle (r), the spread of the field (s), and a strength modifier (α). The formula for the repulsive field (Figure 4.6b)is similar to the formula for the attractive field, it is separated into three parts: inside the obstacle the field is infinitely repulsive in the direction away from the center of the obstacle. Outside the obstacle, but inside the radius of the spread, the field is strongest near the obstacle and gets weaker the longer away from the obstacle the agent gets. Outside the radius of the spread, the field has no forces.

$$\begin{array}{cc} \text{if } d < r & \Delta x = -\text{sign}(\cos(\theta))\infty \text{ and } \Delta y = -\text{sign}(\sin(\theta))\infty \\ \text{if } r \leq d \leq s + r & \Delta x = -\beta(s + r - d)\cos(\theta) \text{ and } \Delta y = -\beta(s + r - d)\sin(\theta) \\ \text{if } d > s + r & \Delta x = \Delta y = 0 \end{array}$$

Figure 4.8: Formula for repulsive potential field

The agent creates repulsive fields around enemy characters; the result of this is that the healer and ranged characters are better at keeping a distance to enemy characters.

Chapter 5

Conceptual Design and Algorithms

This chapter describes the conceptual design and algorithms used to implement an action selection for the agent. Our approach for the action selection is a combination of fuzzy logic and a behavior based approach that allows expert knowledge to easily be represented and interpreted, enabling them to agent gain a better bases for deciding what behavior is appropriate in different scenarios during a fight. Online reinforcement learning is used to adjust the expert knowledge defined as fuzzy rules to reduce the workload of agent configuration. Finally potential fields are used to help the agent to move more intelligently in the fight, allowing for a more dynamic fight than if the agent were just standing still.

The application developed can be described as a cycle that starts with reading information from the game through the WoW API, and ends with sending keystrokes to the game through the WoW Interaction API, both APIs were developed in the previous project
[Larsen and Thorsteinsson, 2011], see Figure 5.1. The reaction time of the agent therefore depends on the number of application cycles per second. Below is a short description of the purpose of each component in the figure; the more technical parts of the description is described further in the rest of this section.

World of Warcraft API
Functionality: The API Translates memory of a WoW process into readable information.
Usefulness: Enables the agent to read information about the environment.
Input: Bytes from the memory of the WoW process.
Output: Convenient data structures representing the environment of WoW.

Fuzzy Rule Evaluator
Functionality: Fuzzify environment into predicates, evaluate and defuzzzify fuzzy rules.
Usefulness: Allows expert knowledge to be interpreted.
Input: Environment data from the WoW API.
Output: A prioritized list of behaviors.

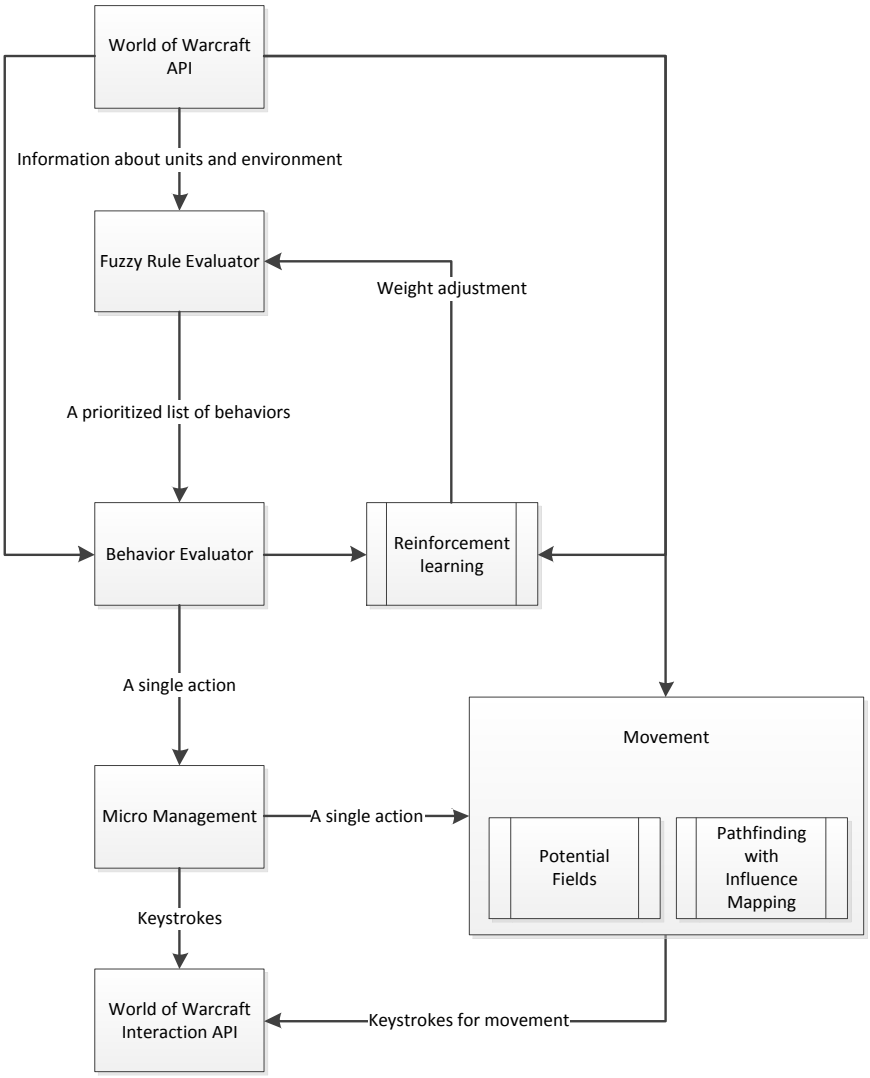


Figure 5.1: Structural Overview

Behavior Evaluator
Functionality: Finds an action based on the defuzzification result.
Usefulness: Allows an abstraction of actions, which eases the definition of rules and behaviors.
Input: A prioritized list of behaviors.
Output: A single action.

Reinforcement Learning
Functionality: Learns and adjusts the weights on selected fuzzy rules.
Usefulness: Eases the development of agents by automating rule weighting.
Input: A single action. *Output:* Adjust the weight on the rule.

Micro Management
Functionality: Ensures that low level requirements are met and translates actions into keystrokes.
Usefulness: Allows the agent actually to perform the action inside the game environment.
Input: A single action.
Output: Action keystrokes to the WoW Interaction API.

Movement
Functionality: Controls the agents movement based on the given action and surrounding characters.
Usefulness: Allows the agent to perform intelligent positioning in close combat scenarios.
Input: A single action.
Output: Movement keystrokes to the WoW Interaction API.

5.1 Fuzzy Rule Evaluation

When creating an expert system it is important that it is easy to convey the expert knowledge to the system. In the case of machine intelligence it is common to define knowledge by using a policy or a set of rules that an agent can follow to solve the problem.
In addition to developing an agent that plays WoW, the focus of this project is also to create a framework that enables a developer to easily describe the behavior of an agent. In the context of agent behavior, Fuzzy Logic rules are commonly defined using three components: predicates, membership functions, and actions, for example:

$$\text{IF } \textit{distance} \text{ IS } \textit{short} \text{ THEN } \textit{attack} \tag{5.1}$$

where *distance* is a predicate, *short* is a membership function and *attack* is an action. Defining a few rules for the different scenarios that an agent might encounter is straightforward and a rule base for an agent is highly understandable and manageable for AI developers.
In addition to these three components, this project introduces an *object class* component used together with a predicate or an action to describe the type of object to evaluate the particular predicate against or execute the action on. For example *IF distance[X] IS short THEN attack[X]*

where X is the object class that the *distance* predicate is evaluated against, and that the *attack* action is executed on.
Object classes are defined depending on the domain and depending on what kinds of objects the predicates and actions relate to. The interesting objects of the environment are identified and these are then divided into object classes. For the agent developed in this project, the interesting objects of the environment are the characters, these characters can be divided into three object classes: *friends*, *enemies*, and *self*. Using these three object classes more specific rules can be defined e.g.:

IF *distance[enemy]* IS *short* AND *health[self]* IS *Low* THEN *flee[enemy]* (5.2)

enemy covers multiple players in the environment and the one that the *flee* action is executed with is the enemy that has the best match, in the example above this is the enemy that is closest. If an object class is used with multiple predicates in the rule, the same object from the environment is used to evaluate the predicate.

IF *distance[enemy]* IS *short* AND *health[enemy]* IS *low* THEN *attack[enemy]* (5.3)

In the rule above the same object class is used with two predicates and with the *attack* action. If the environment contains three enemies, the rule above should be evaluated three times; once for each enemy. The rule is equivalent to the three regular rules below:

IF *distanceToEnemy1* IS *short* AND *healthOfEnemy1* IS *low* THEN *attackEnemy1*
IF *distanceToEnemy2* IS *short* AND *healthOfEnemy2* IS *low* THEN *attackEnemy2*
IF *distanceToEnemy3* IS *short* AND *healthOfEnemy3* IS *low* THEN *attackEnemy3*

At each evaluation the same enemy object is used to evaluate both the predicates i.e., at each evaluation, the *enemy* object class represents the same object in every predicate and in the action.
This is good for understanding, but sometimes it might be required to have different predicates in the same rule be evaluated against different objects. An example of such a rule is defined below:

IF *role[enemy0]* IS *healer* AND *health[enemy1]* IS *low* THEN *crowdControl[enemy0]*

The rule describes a scenario where the agent should crowd control an enemy healer in case another enemy has low health. Numbers are added to the object class to illustrate that they represent different objects.
This is a more explicit way of defining the fuzzy rules, which has a very small impact on the understandability.
The implementation of the behavior of an agent now only consists of defining multiple of such rules; an example of a small rule base is illustrated below.

a) IF *distance[enemy]* IS *short* THEN *attack[enemy]*
b) IF *distance[friend]* IS *short* THEN *help[friend]*

It is obvious from the example above that it is very easy to specify inconsistent information, in rule *a* the agent is told to attack enemies that are close by, and in rule *b* it is told to help

friends that are close by. In a scenario where there is an equally short distance to a friend and an enemy, the two rules will have the same matchgrade, which may not be preferable if the agent is better at helping than at attacking.
To give more control over how the agent behaves in scenarios like this, where the rules has an equal match i.e., where the agent's input values match more than one rule equally well, each rule is given a weight describing the importance of the rule.
This weight can then be used to determine which of the actions to choose e.g., if rule b is given a higher weight than rule a the agent will prefer to help friends in scenarios where an enemy is nearby rather than attacking the enemy. Previously we used the formula $\mu * \omega$ where μ is the matchgrade and ω is the weight, to calculate the final matchgrade, however, a concern we had was that if the weight is adjusted to 0, e.g. through reinforcement learning, the rule never gets selected. To solve this we came up with a formula that uses a contribution factor α to control the influence of the weight and by configuring α to a value close to 1, e.g. 0.9, ensures that the weight has a high influence but will never entirely eliminate a rule.
The weight's contribution is determined using the following formula:

$$M = \mu(1-\alpha) + (\mu * \omega * \alpha) \tag{5.4}$$

where M is the final matchgrade, μ is the original matchgrade from the fuzzy rule, α is how much the weight affects the original matchgrade, and ω is the defined weight of the fuzzy rule. If α is set to 0, the weight does not affect the rule at all, whereas if α is set to 1 and ω on a given rule is 0, the matchgrade will always result in 0 regardless of the original matchgrade.
Weighting rules give more control over the behavior of the agent, but also introduces a new value for a developer to consider. This trade-off can be mitigated by using learning techniques to learn and adjust the weights, Section 5.3 describes how reinforcement learning techniques can be used to learn the weights.

Evaluation

The evaluation of a rule is done using Algorithm 1 where R is the rule to evaluate, C is a set of the used object classes in the rule, E is a set of the relevant objects of the environment, O is a set of specific objects that are used when evaluating a rule (this set gets populated through recursion), d is the depth of the recursion, and L contains the result of the evaluation. The algorithm uses another function: `EvaluateRuleWithSpecificObjects`, this function calculates the matchgrade of the rule by evaluating every predicate in the rule with specific objects rather that object classes and combines result.
The algorithm essentially expands a single rule into a set of rules that can be evaluated individually using a specific set of objects; the algorithm evaluates each rule with every possible combination of specific objects based on the object classes used in the rule and produces a list of the results.

5.1.1 Rule Evaluation Example

To further elaborate the concepts described above, a more concrete example of how rules are evaluated in the context of a fight in WoW is presented in this section. The example builds on a

Algorithm 1 Fuzzy Rule Evaluation

```
1: function EVALUATERULE(R, C, E, O, d, L)
2:     c = C[d]
3:     e = objects in E that belong to the object class c
4:     for all objects in e do
5:         o = copy(O)
6:         add current object to o
7:         if o.size == C.size then
8:             m = EvaluateRuleWithSpecificObjects(R, o)
9:             add m to L
10:        else
11:            EvaluateRule(R, C, E, o, d + 1, L)
12:        end if
13:    end for
14: end function
```

simplified version of the configuration that the agent is developed for; an arena-fight between two teams, each consisting of a healer character, a melee character and a ranged character (see Section 3 for a description of the different character roles).

The Environment

For this example the environment consists only of the six characters and the arena that they are fighting in, the arena is an enclosed circular area. The data that the agent can read from the environment is:

- Health: The HP of each of the characters.
- Position: The position of each of the characters in the arena.
- Role: The role of each character i.e. healer/melee/ranged.

Rule Base

The example is based on the GM character's point of view (the one with the red arrow), which means that the rules and the behaviors used in the example are created for a melee character. The rule base of the melee character consists of three rules:

a) IF *distance[enemy]* IS *short* THEN *attack[enemy]*

b) IF *health[self]* IS *low* THEN *defend[self]*

c) IF *health[enemy0]* IS *low* AND *role[enemy1]* IS *healer* AND *distance[enemy1]* IS NOT *long* THEN *crowdControl[enemy1]*

The third rule is a little more advanced than the other two, it translates to: "if there is an enemy with low health, and there is a healer not far away, then crowd control that healer", as described in Section 3 this is a common strategy used in arena fights in WoW.

Scenario

Figure 5.2 illustrates an example scenario in a fight. The out most circle represents the arena walls and each circle inside the walls represent a single character. There are two teams in the arena; the black team and the gray team and there are six characters in the arena: two healers (BH and GH), two ranged (BR and GR), and two melee (BM and GM). The bars above the characters indicate how much HP the character has; a full black bar means that the character has full health.

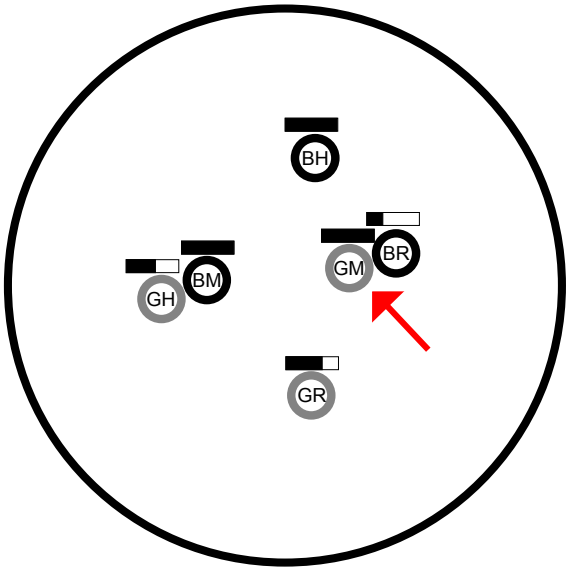


Figure 5.2: Example Arena Fight Scenario

Rule Evaluation

The rule evaluation in this example is based on a perspective from GM (The melee on the gray team, which the arrow points at), and rule a, b, and c listed above. Each rule is evaluated individually using Algorithm 1.

a) ***IF distance[enemy] IS short THEN attack[enemy]***

To evaluate this rule, the following input is passed to the algorithm:

R = IF distance[enemy] IS short THEN attack[enemy]

C = { “enemy” }

E = { BM, BR, BH, GM, GR, GH }

O = { }

L = { }

d = 0

This input results in three { character, matchgrade } pairs getting added to L ({ {BM, 0.3}, {BH, 0.4}, {BR, 1} }, each is the result of a fuzzy evaluation of the predicates in the rule using that particular character as the basis for evaluating the predicates.

b) ***IF health[self] IS low THEN defend[self]***
The input passed to the algorithm is the same as for the rule above except for R and C:

R = IF health[self] IS low THEN defend[self]

C = { "self" }

The evaluation results in $L = \{\{GM, 0\}\}$, the matchgrade is 0 because the gray melee (GM) character does not have low health. Having a matchgrade of 0 means that the rule is not relevant for the current state of the environment, and as such the action that the rule points to should not be executed.

c) ***IF health[enemy0] IS low AND role[enemy1] IS healer AND distance[enemy1] IS NOT long THEN cc[enemy1]***
Again only R and C are different:

R = IF health[enemy0] IS low AND role[enemy1] IS healer AND distance[enemy1] IS NOT long THEN cc[enemy1]

C = { "enemy0", "enemy1" }

Because this rule uses two object classes the algorithm will call itself recursively, this will result in a fuzzy evaluation for every possible combination of `enemy0` and `enemy1`, in this case this is nine evaluations. The only combination that will yield a matchgrade above 0 is the one where `enemy0` is the black ranged character and `enemy1` is the black healer character. Every other combination will result in a matchgrade of 0 or very close to 0, depending on the exact definition of the membership functions. The matchgrade of the rule using BR and BH will be close to 1, because BR has low health and BH is a healer that is not far away from GM in the figure.

The Result
Two out of three rules have a matchgrade, so now the weights are applied to find out which of the rules has the highest matchgrade and then what action to choose. If the weights and matchgrades of the rules are as follows:

- Rule a): weight = 0.5, matchgrade = 1
- Rule b): weight = 0.7, matchgrade = 0.8

using Equation 5.4 with an α of 0.9 the calculations of matches would be:

$$M_a = 1 * (1 - 0.9) + (1 * 0.5 * 0.9) = 0.55$$

$$M_b = 0.8 * (1 - 0.9) + (0.8 * 0.7 * 0.9) = 0.584$$

This means that the agent will choose the action that rule b) points to, on the character that gave the highest matchgrade with that rule i.e., the *CC* action on BH. According to the expert knowledge provided, this is the best action given the current state of the fight. This allows the agent to be able to find an appropriate action in different scenarios based on the configuration of the expert system.

5.2 Behavior Evaluator

In WoW the evaluation of how good an action is depends on a lot of information in the game, such as: the health and position of other characters, the currently active cooldowns, the roles of the different characters etc.
A typical character in WoW has around 50 different actions to choose from at any given moment, which makes it hard to create predicates for all the environment information and write suitable fuzzy rules that fit all the actions.

A solution is to use behaviors; these are abstractions of actions, which can be used to define higher level intents using actions. The behavior evaluator receives the collection of prioritized behaviors and finds the most suitable action from the given behavior.

A behavior represents an intention and the actions in the behavior are what carries out that intent. Our approach is to is to write each action of the behavior in prioritized order and with a set of preconditions, which must all be fulfilled in order for the action to be valid. The behavior evaluator iterates over the actions in a behavior and determines whether or not the actions are valid based on the preconditions and the state of the environment. The first valid action found in a behavior is the action that will be performed.
The first valid action found in a behavior is the action that will be performed
An example of this is an attack behavior with two actions:

- ***bleed***: causes 100 damage over 10 seconds and only one bleed can be applied at once.
- ***hit***: Causes 10 damage.

Both actions are chosen because they fit the intent of attacking. The precondition for *bleed* is that the target of the action must not be bleeding, since reapplying *bleed* will not cause the enemy to take extra damage. *Hit* has no preconditions but is placed after *bleed* in the prioritized list. This will cause the attack behavior to always have at least one valid action: bleed if it is not already applied and *hit* otherwise, causing the maximum amount of damage. Note that the actions may sometimes be dodged by the enemy, resulting in some uncertainties in the environment.

The pseudocode in Algorithm 2 shows the iteration over actions and preconditions.

5.2.1 Behavior Evaluation Example

Recalling the rules from the example in Section 5.1.1:

Algorithm 2 Finds a single action based on the defuzzification result

```
function GETACTION(input : prioritized list of behaviors)
    for all behavior in input do
        actions := GetActions(behavior);
        for all action in actions do
            if all preconditions are met for action then
                return action;
            end if
        end for
    end for
end function
```

a) IF *distance[enemy]* IS *short* THEN *attack[enemy]*

b) IF *health[self]* IS *low* THEN *defend[self]*

c) IF *health[enemy0]* IS *low* AND *role[enemy1]* IS *healer* AND distance[enemy1] IS NOT *long* THEN *crowdControl[enemy1]*

we define define three behaviors for the agent: *attack*, *defend*, and *crowdControl* to replace the actions of the same names, i.e. instead of using the fuzzy rule to point at an action, we point at a behavior.

Attack Behavior

An example attack behavior is given below, consisting of two actions: *hit* and *bleed*. These two actions are prioritized and preconditions are defined:

1. **bleed**, preconditions: *target character must not be bleeding already*
2. **hit**, no preconditions

Defend Behavior

A defend behavior is used when a character is under pressure, and should contain actions that mitigate incoming damage. The defend behavior is defined with the following list of prioritized actions:

1. **parry**, precondition: *target character must be melee*
2. **block**, no preconditions

Crowd Control(CC) Behavior

The intent of the CC behavior is to crowd control enemy characters, and should contain actions that incapacitate characters. The CC behavior in this example only contains one action:

1. **stun**, precondition: *target must not be CC'ed already*

Selecting and Evaluating the Behavior

In Section 5.1.1 the matchgrades of the rules with respect to the different characters were found, e.g.: rule a) with the black ranged character from Figure 5.2 has a matchgrade of 0.55. The behaviors that the rules points to are sorted by matchgrade and fed into Algorithm 2. Continuing from the example in section 5.1.1, where rule a) had matchgrade 0.55 and rule c) had matchgrade 0.584 the list of behaviors fed into the algorithm is: CC, Attack.
The *CC* behavior only has one action (*stun*) and the precondition for that action is that the target character cannot already be CC'ed. This means that if the target of the CC behavior is *not* CC'ed when this evaluation is executed the algorithm returns the *stun* action. If, however, the target of the behavior *is* already CC'ed the algorithm moves on to the next behavior in the list, which is attack, and will return the action *bleed* or *hit*.

Using this behavior abstraction means that the configuration of the agent can be separated into smaller chunks that are more manageable. The developers of the AI can specify the specific behaviors for specific scenarios rather than finding the most optimal configuration of the low level actions.

5.3 Reinforcement Learning

As described in Section 5.1, each rule in the system has a weight defining the importance of the rule, this is useful in scenarios where multiple rules have a similar matchgrade i.e. where rules are conflicting. In these scenarios the weight of the rules is used in combination with the matchgrade to find the most preferable behavior. Even through the rules are defined by an expert, it can still be difficult and time consuming to adjust the weights of all the rules. To avoid this difficulty reinforcement learning(RL) is introduced as a way to automatically adjust the weights. Note that in this case, RL is only used to weight the rules and has no influence on the action policy within the behavior.

RL is applied by using relevant statistics about the environment together with behavior specific reward functions to adjust the weights of the rules. The idea is that the environment changes are measured in a window that spans from when the behavior is activated till the selected actions in the given behavior, no longer has an effect on the environment; this duration is referred to as *lifetime*.
A model that stores all the information related to a single reward, such as which rule to update, the duration, and the measured data, is referred to as a *reward model*. When a new action is selected in the behavior evaluator, the RL identifies if the rule that selected the behavior and the target character are the same, if so, the *lifetime* of the *reward model* is extended based on the duration the new action. If a new rule or target character are found then a new *reward model* is created and the previous *reward model* is stored and used to update the weight of the relevant rule once its *lifetime* expires. This weight update consists of identifying the reward function to use, calculate the reward and use the learning rate and reward to modify the weight of the rule.

To address the issue where the RL converges into a local minimum that is not the global minimum, ϵ-*greedy*[Sutton and Barto, 1998] has been used in the behavior evaluator that allows the RL to keep exploring the search space. Using ϵ-*greedy* with the behavior evaluator means that the selected behavior is in some cases not the behavior with the largest matchgrade but a behavior chosen at random. An easy way to implement ϵ-*greedy* in the behavior evaluation process, would be to shuffle the list of prioritized behaviors generated in the fuzzy rule evaluation before passing it to Algorithm 2, as this algorithm assumes that the behaviors are sorted by priority.

The used RL approach gives the following challenges:

- Defining a proper learning rate: it is suitable to adjust the learning rate during the training period to have a high learning rate in the beginning and a low learning rate at the end of the training period, thereby allowing a very reactive weight learning in the beginning and fine tuning afterwards.
- Defining the lifetime for a reward model: in WoW the effect of casting a spell can last several seconds into the future, because of this, the effect of selecting a behavior can also last several seconds into the future, even past the selection of a new behavior. This means that it may not be suitable to only base the reward on the period of time the behavior is active. If a crowd control action is used, the behavior may be active for 1 second but the action affects the battle for 7 seconds, meaning that the reward should be based on longer period of time than just the time it took to execute the behavior.
- Defining behavior specific reward functions: different information is relevant to different behaviors. If an aggressive behavior is selected, the relevant information for the reward function to record could be the damage given to the target, whereas a crowd control behaviors should be rewarded for the period of time that the target of the behavior is inactive.

The following sections describe how each of these three challenges has been solved in our agent.

5.3.1 The learning rate

The following formula is used to update the weight:

$$\omega \mathrel{+}= \alpha\,(\text{Reward} - \omega) \tag{5.5}$$

where ω is the weight of the rule, α is the learning rate and *Reward* is a number between 0 and 1 given by the reward function. The learning rate defines how big an influence the reward has on the weight of the rule, a high learning rate results in a reactive weight, i.e. it changes a lot based on the reward a low learning rate results in less reactive weight. To enhance the convergence, the learning rate is high in the beginning of the training period and lowers as the rule is getting selected.

The learning rate can be formulated as the following:

$$\alpha = \alpha_{\min} + \max\left(0, (\alpha_{\max} - \alpha_{\min}) * \frac{C_{\max} - C_{\text{current}}}{C_{\max}}\right)$$

Where $\alpha_{\min}$ is the minimum learning rate and $\alpha_{\max}$ is the maximum learning rate. $C_{\max}$ is the maximum times a rule can be selected before it uses the minimum learning rate and C_{current} amount of times the rule have been selected so far. A illustration of this linear learning rate formula can be seen in Figure 5.3.

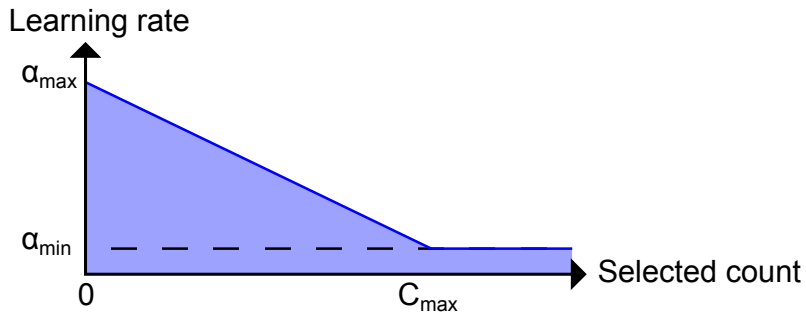


Figure 5.3: Linaer learning rate based on rule selected count

5.3.2 The lifetime of a reward model

The lifetime of the reward model is defined as the amount time between the behaviors has been selected until the effects of the behavior have ceased, and the rule weight is updated. The lifetime of the reward model is based on the amount of time that the behavior has an effect on the environment. Figure 5.4 illustrates this using the two behaviors CC and attack; the behaviors are selected at different times throughout the fight.

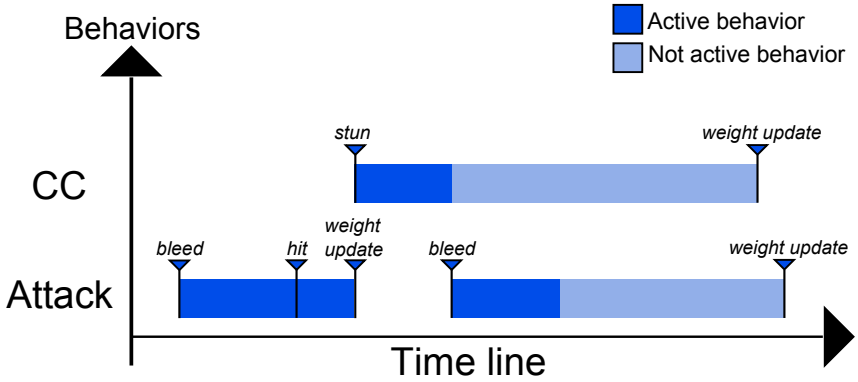


Figure 5.4: Reward Model Cycle

The darker area of the bar illustrates when the behavior is active, whereas the lighter color illustrates when a behavior is inactive but have not received the weight update yet i.e., it still has effects on the environment. The *CC* behavior has a long period of time without being active

because the action *stun* chosen in the behavior has a long duration that affects the lifetime. The given point in time each reward is calculated can be described with the formula:

$$t_{\text{end}} = \max_{a \in A} \left(t_{\text{start}} + \max(\mathit{CastTime(a)}, T_{\text{gcd}}) + \min(\mathit{Duration(a)}, T_{\text{md}}) \right) \tag{5.6}$$

where a is an action of the behavior and A is all the actions of the behavior. T_{gcd} is the global cooldown, i.e. the time it takes before a new action can be used. T_{md} is a maximum duration of an action, meaning the maximum time span that a lifetime is extended while not active; this value used because the duration of some actions are too long to measure a reward. t_{start} represents the point in time where the action was executed, *CastTime(a)* is the casttime of action and *Duration(a)* is the duration of action. T_{gcd} is determined by the game rules and ranges between 1 and 1.5 seconds. T_{md} is set to 4 seconds.
The formula ensures that the lifetime is based on both the time the behavior was active and the duration of the actions chosen by the behavior. An example of how this formula is used, is illustrated in the example in Section 5.3.4.

5.3.3 Defining behavior specific reward functions

The reward functions are constrained by returning a value between 0 and 1, which relates to the weight of the fuzzy rules. The reinforcement learning could be trained by using a long term reward based on if the agent either wins or loses the fight; however, this takes too long to converge. Instead, a short term reward, measured in the lifetime of the reward model by a behavior specific reward function, is used. The identification of which reward function to use is based on the intent of the behavior, the following four reward functions have been defined:

- **Attack Reward Function**: Behaviors that uses attack actions are rewarded for the damage done by the player to the target.
- **Heal Reward Function**: Behaviors that use healing actions are rewarded for the healing done by the character.
- **Crowd Control Reward Function**: Behaviors that uses crowd control actions are rewarded for a low level of activity of the target.
- **Defensive Reward Function**: Behaviors that uses defensive actions are mostly rewarded by an reduced amount of damage taken.

All the reward functions uses window-based statistics, these statistics measure the *damage done*, *healing taken*, *activity* etc. of each character in the lifetime or the reward model. All the values ranges between 0 and 1, where 1 is maximum potential and 0 is minimum potential i.e., when *activity* is 1 the player is constantly active and when *damage done* is 1 the player does the most amount of damage he is capable of. The normalization of damage and healing is based on the maximum measured value per second. This value is slowly decayed to reduce the significance of spikes in damage or healing, thereby giving a realistic normalization.
This gives a reward function that matches the intent of the behavior and combines this knowledge with window-based statistics to calculate the reward, which is usefull in short-term rewards.

5.3.4 Example of Reinforcement Learning

Continuing the example from section 5.2.1, a reward model is created based on the behavior chosen (the *CC* behavior) and the target character (the BH character from Figure 5.2) and the lifetime of the model is calculated using Equation 5.6.
Recall from section 5.2.1 that the only action of the *CC* behavior is the *stun* action. To calculate the lifetime of the reward model the *casttime*, *duration* and when the action was executed are needed. These are defined as: *casttime*: 0s, *duration*: 10s, executed at: $t_{\text{start}} = 100s$, furthermore T_{gcd} and T_{md} are defined as $T_{\text{gcd}} = 1.5s$ and $T_{\text{md}} = 4s$.
Filling these values in to the equation

$$t_{\text{end}} = (100s + \max(0s, 1.5s) + \min(10s, 4s)) = 105.5s \tag{5.7}$$

we see that the reward model expires $105.5s - 100s = 5.5s$ seconds after the behavior was chosen.
Once the model expires a reward is calculated using a behavior specific reward function, in this case the behavior chosen was *CC*.
A reasonable reward function for the *CC* behavior would be one that calculates a reward based on the activity of the target, because the intent of the *CC* behavior is to lower the activity.
The reward function could be defined as:

$$\text{Reward} = 1 - A$$

where A is a number between 0 and 1 describing the average activity of the target during the lifetime of the reward model; 1 being active all the time and 0 being inactive all the time.
When the reward model of the *CC* behavior expires and the measured activity is 0.2, the reward for choosing the behavior would be 0.8 and the weight of the rule can be updated using Equation 5.5.

In this section we establish how reinforcement learning can be used in combination with the designed fuzzy rule evaluation and behavior evaluator. Combining the calculated learning rate, the lifetime-based measuring, and the specific short term rewards makes it suitable for a practical applications such as WoW, which incorporates actions with *duration* and cannot be simulated.

5.4 Micro Management

WoW defines rules for when an action can be performed, if the conditions for these rules are not met, it is impossible for a character to perform the given action. The Micro Management component ensures that the conditions for the rules are met before the agent performs the actions. This section gives a brief description of some of the important conditions that must be fulfilled in order for the agent use a given action.

Ensure correct Shapeshift Form

Some actions require that character is in a certain shapeshift form, such as the ability *Bite* used by the class *Druid*, which requires the player to be in a *Cat* form. The Micro Management component ensures that the character is in the correct shapeshift form before the action is performed, by sending keystrokes to the interaction API.

Ensure the correct target

If the given action should be performed on a different target than the current target of the character, the Micro Management component identifies and sends the keystrokes necessary to shift the target to the desired one.

Ensure the correct Distance

If the character is not within the correct distance of the target of the action, the micro management will not enqueue the keys for using the action. It will instead issue a command to the movement component, remember the given action and execute it once the movement component has moved the character. This ensures that the agent does not enqueue keys that are not useful.

5.5 Movement

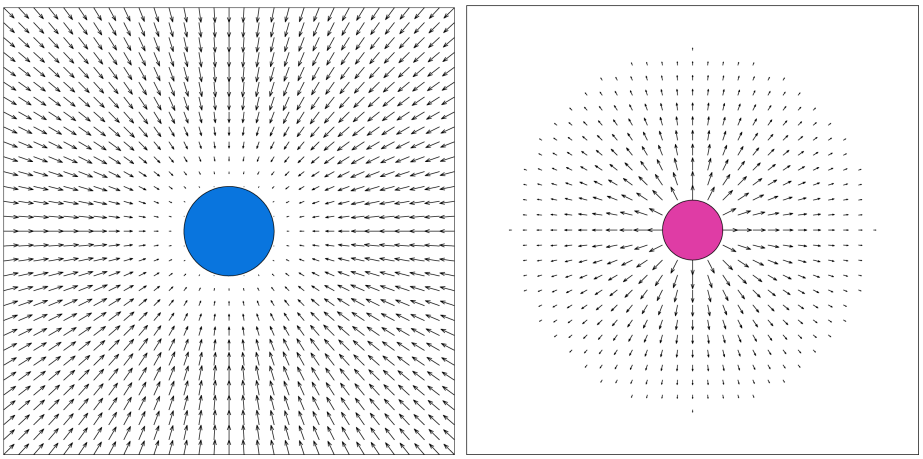
Some actions used by the agent requires certain positioning and some actions are allowed to be executed while moving, making movement and positioning an important aspect of how well the agent plays, bad positioning often indirectly results in a lost fight.
To accommodate this, the movement component is an autonomous component that handles the movement of the agent by analyzing the actions to determine where to move. The input to the component is all the actions a used by the agent and the output is movement keystrokes to the WoW Interaction API.

The conditions for moving correctly can be listed as the following:

- Stay or move in range of the character that the agent needs to perform an action on.
- Try to out-range as many opponents as possible.
- Try to stay near the friends.
- Stand still if the action requires it.

In the previous project we solved this problem on a larger scale with influence maps, these maps would affect a path calculated by the pathfinding algorithm and steer them around potentially dangerous points on the map. While this produced convincing results and has low performance cost in large scale fights with a fair amount of long paths, the resolution provided by the influence maps is not sufficient for close combat navigation.

Instead of influence mapping, potential fields are used to guide the navigation agent. These fields are used in a way that repels the agent from enemies (see Figure 5.5a[1]) but attract it to friends(see Figure 5.5b), which gives allows the agent to find an optimal movement direction at any given time in the game. The *strength* of the enemy fields may then be configured to ensure that the repulsive fields are only repulsive within the range of the enemy character; the closer to the enemy character is, the more repulsive the field is, as seen on the lengths of the arrows in the figure. As the movement potential fields does not tell anything about the game



(a) (b)

Figure 5.5: Attractive and repulsive potential fields

environment such as holes in the map, pillars and other obstacles, so once the direction has been found, using the potential field, a real pathfinding algorithm takes over and calculates a path to the desired location. [Larsen and Thorsteinsson, 2011].

5.5.1 Movement Example

Figure 5.6 illustrates an example scenario with three characters. The blue circle is friendly, the red circle is hostile and the black circle is the character that the example is based on. The potential field in the figure points in the most desirable direction to run for the character represented by the black circle.
Based on the potential field the character represented by the black circle "wants" to run away from the enemy character and towards the friendly character. This is only useful for the healer and the ranged characters, as the melee character needs to be close to the enemy.
Whether to follow the potential field or not is determined by the actions that the agent should perform, and on which character the actions should be performed.
If, for example, the black circle is a melee character, and the action selected is *hit the red circle*, then the potential field is ignored because the melee character needs to get close to the red circle in order to *hit* it.

[1] Figure is taken from [Goodrich, 2004]

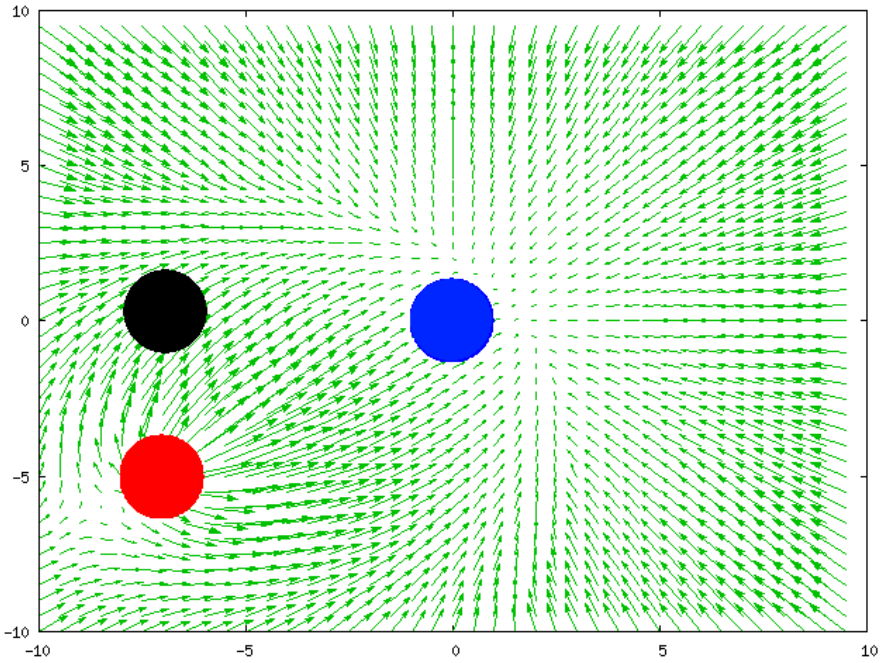


Figure 5.6: Potential Field Example

If instead the black circle is a ranged character and the action selected is *shoot the red circle*, then the potential field is used to move the ranged character away from the red circle and towards the blue circle while the character is *shooting*.

Using potential fields combined with the previous developed navigation system allows the agents to move freely, which improves the performance of the healer and ranged characters.

5.6 Visualization

One of the goals of this project is to visualize the different techniques to ease tweaking and tuning of agents. This section describes some of the ideas of how to get an overview of the fight and how to get an internal overview of agents in order to ease the tuning of the techniques used in the agent.

Figure 5.7 illustrates two mockups of the GUI. The left mockup is designed to give an overview of the current fight by displaying the state of each character of each team and a summary of wins for the previous fights. The top horizontal taps is used to change in between showing this overview and showing an internal overview of a single character.
The mockup to the right is the design for the internal overview of a single character. The vertical tabs is used to tab between different technical visualizations of parts used in the agent that controls this character, e.g. monitoring the weight of the fuzzy rules, the behavior evaluation process or the reinforcement learning. The content in the right mockup illustrates an example of how the fuzzy rule weights in an agent could be visualized.

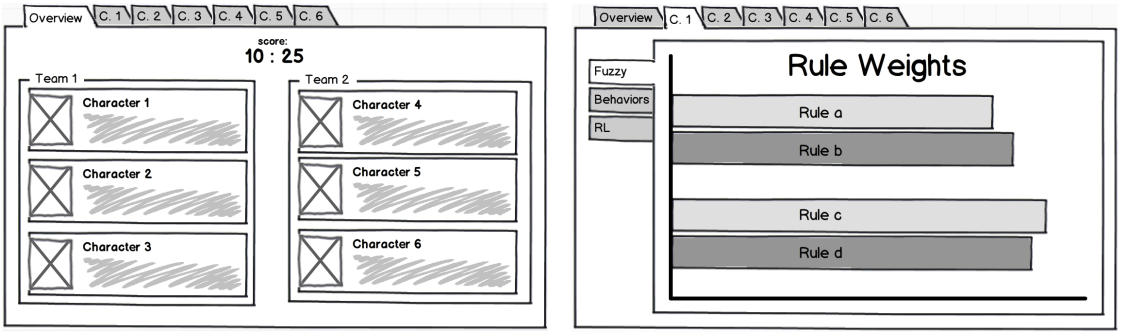


Figure 5.7: Mockup of the Application GUI

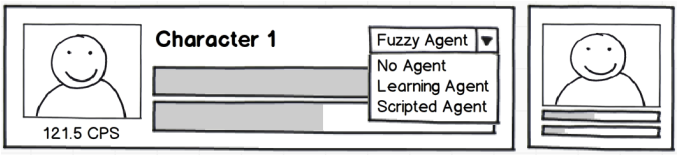


Figure 5.8: Agent Visualization

Figure 5.8 illustrates a mockup of a single agent, and contains the most important information:

- **Health & Power**: These are illustrated by the two bars and are used to identify the state of the character that the agent controls.
- **Class Icon**: The class of the character, e.g. *Hunter*, *Druid* or *Priest*, is illustrated by the image to the left.
- **Cycles Per Second(CPS)** The CPS can be seen below below the class icon and is used to measure the performance of the agent.
- **Target**: The square to the right contains the target class icon, health and power, and is used to identify who the character is targeting.
- **Agent Type**: A drop down menu to the top right is used to identify and change the type of the agent that controls the character.

The combination of using tabs and maintaining an overview gives a suitable visualization for tweaking and tunning the implemented agents.

Chapter 6

Implementation

This chapter describes the implementation of the framework and gives an overview of the structure in the solution. The whole solution is written as an application in C#, but could have been written in most other common languages. C# is chosen because it is a high level language that makes the process of developing simple and understandable, and we have a lot of previous experience using the .NET framework. Additionally the previous project was also written using C# making it even more compelling.

6.1 Fuzzy Rule Evaluation

This section describes the key parts of the fuzzy rule implementation developed for this project. The implementation builds on the concepts described in Section 5.1.

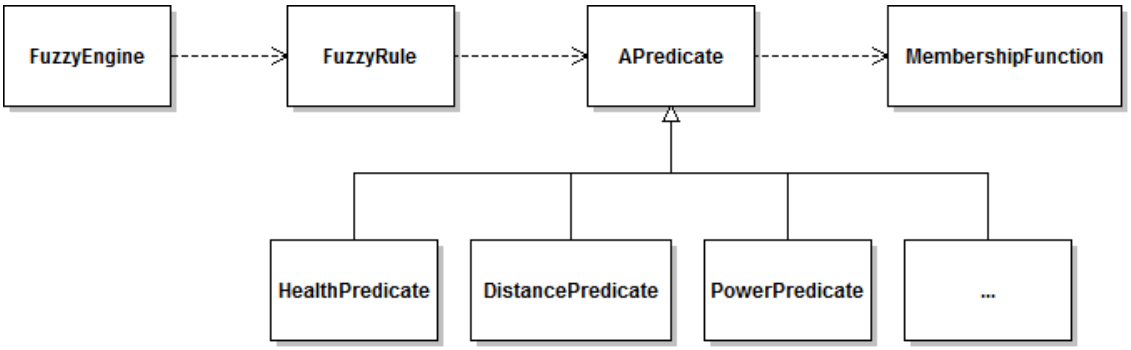


Figure 6.1: Fuzzy Logic implementation UML

The component consists of parts illustrated in the UML diagram in Figure 6.1.
The `FuzzyEngine` is responsible for the evaluation of the rules, the rest of the classes are used to represent rules, predicates and membership functions.
Every predicate used is defined in a single class extending the `APredicate` class which contains some common logic for predicates, the values of the predicates can be seen in Appendix 9. The `DistancePredicate` class, for example, is implemented by reading the distance between the character and the character passed to the predicate through object class in the fuzzy

```
<Rule Weight="0.8" Name="heal low friend">
  IF Health[friend] IS Low THEN Heal[friend]
</Rule>
```

Figure 6.2: Sample XML Fuzzy Rule

rule, this value is then fuzzified using the membership function attached to the distance predicate and the rule is evaluated using the techniques described in Section 5.1.
Each rule in the system is defined in a XML node with the weight and name of the rule as attributes on the node, the contents of the node is the rule itself Figure 6.2 is an example of rule.
Three agents are developed for this project: a melee agent, a ranged agent, and a healer agent, each of these agents have their own collection of rules defined in their own XML document. When an agent is instantiated this XML document is loaded and all the rules in the document are parsed and represented in the application using `FuzzyRule` objects.
An important goal of the project has been to ease the development of new agents, to complement this goal the ability to edit the rules live as the agent is active has been implemented. Whenever a change is detected in the file containing the rules, the file is reloaded and the `FuzzyRule` objects representing the old rules are replaced with new objects representing the new rules. This is very handy as it allows the developer to immediately see the effect of a change in a rule without having to recompile or restart the application.

The combination of configuring the expert system in XML and dynamically load changes gives a good foundation for modeling and evaluating the expert knowledge.

6.2 Behavior and Reinforcement Learning

This section gives an overview of how the behavior and reinforcement learning components communicate in the implementation, based on the design described in Section 5.2 and Section 5.3.

Figure 6.3 is an UML class diagram illustrating the implementation and dependencies of the two components.
Two behavior evaluators are implemented in the application, the *Original Behavior Evaluator* which uses the algorithm described in Section 5.2 to determine which action to choose, and the *Reinforcement Learning Behavior Evaluator* which uses a modified version of the algorithm. The modified algorithm has an ϵ*-greedy* mechanism build in, as described in Section 5.3, and it also interacts with the reinforcement component to update weights on rules based on rewards.

The class `LearningBehaviorEvaluator` uses an implementation of the *reward model* and reward functions to calculate rewards for expired reward models, based on the design described in Section 5.3. It also uses the `RuleWeightManager` to update the rule weights and the count for how many times a rule has been selected (C_{current}) used for calculating the learning rate, the formula is described in Section 5.3.1.

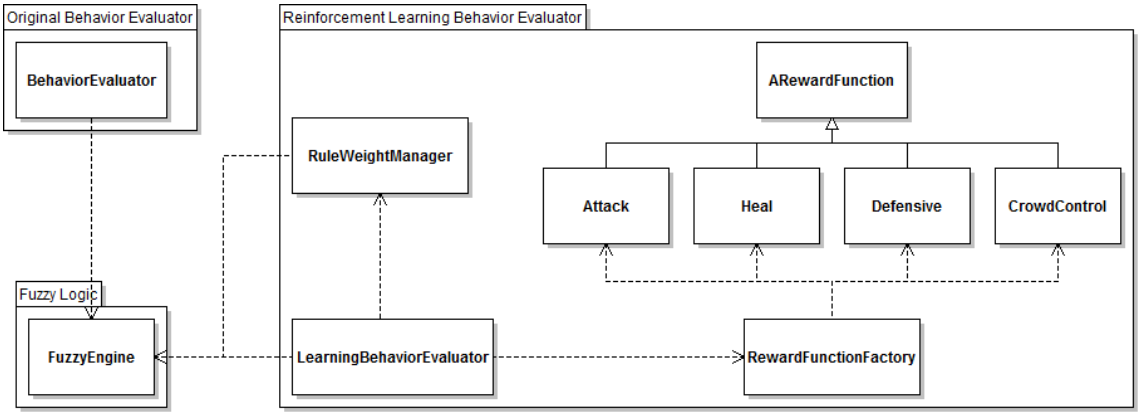


Figure 6.3: UML clas diagram of both the original and reinforcement learning behavior evaluator

The class `RewardFunctionFactory` is a factory used for creating behavior specific reward functions, described in Section 5.3.3. Each of the reward functions extends the abstract class `ARewardFunction`, which contains common reward function logic, such as calculating the part of the reward that is based on the overall combat and used in all the reward functions.
The abstract class `ARewardFunction` also contains an extra validation of the reward value produced by each of the reward functions to ensure that no reward is less than zero or greater than one.

This design enables the agent to use the original behavior evaluator or the RL behavior evaluator depending on the configuration, which is especially useful when conducting experiments, as some agents can be configured to run without RL.

6.3 GUI Framework

This section describes how the configuration tool is implemented to ease the monitoring and configuring of new agents, the design is based on the description in Section 5.6. The GUI is largely based on Extensible Application Markup Language (XAML)[1], as XAML stores the design of the GUI in XML in a very maintainable manner similar to HTML.

Views

Based on the mockups in Section 5.6, the type of views can be divided in two branches, one branch of views that gives an overview of all the agents at the same time and another branch that shows different aspects of the internal state of a single agent.
To ease the implementation of these, two interfaces has been defined: `IAgentView` and `IOverviewView`, as seen in Figure 6.4. Both interfaces extend the `IView` interface, which contains the name of the view and two methods, `OnShow` and `OnHide`, these methods are called when a view is shown or hidden allowing the view to start or stop any calculations that are needed to that particular view. This is done to minimize the amount of CPU cycles used in the views.

[1] http://msdn.microsoft.com/en-us/library/ms752059.aspx

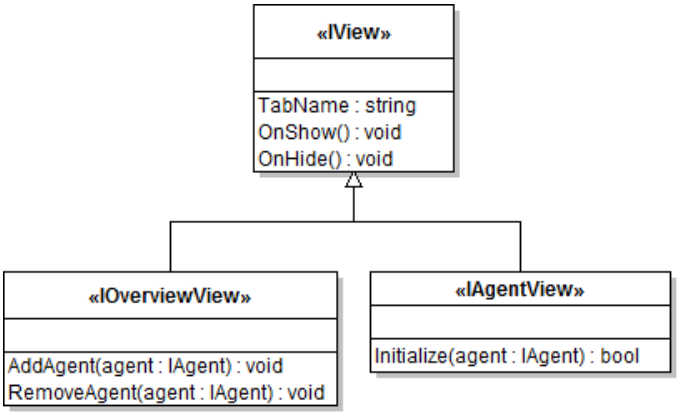


Figure 6.4: Interfaces used for GUI views

The `IOverviewView` contains two additional methods: `AddAgent` and `RemoveAgent`, allowing the view to maintain a list of references to all the agents that currently controls a character.
The `IAgentView` contains a single `Initialize` method, that allows the view to register the agent added to the view. Some views are specific to a certain kind of agent e.g., a view illustrating the rewards from a reward model, does not make sense for an agent that does not use reinforcement learning. The return value of the `Initialize` method controls whether the view is initialized.
A `ViewFactory` is implemented to automatically search the application for views that implement `IOverviewView` or `IAgentView` and initialize them for the relevant agents. This means that the only thing needed to create a new view, is to implement one of the two interfaces and the view is automatically integrated into the rest of the application, making it very easy to add new views.

Examples of Views

The *Battlestatus* view in Figure 6.5 gives an overview of current state of the fight, and is based on the mockup in section 5.6. It is an easy way to monitor the state of all the agents participating in the fight and it gives a much better overview than looking at each of the WoW clients. For each agent there is a small drop-down menu that allows changing of the type of agent controlling that character on the fly, this is very useful in scenarios where the performance of one agent type is tested against other agent types.

Figure 6.6 is a screenshot of one of the views that give an internal overview of an agent, in this case the fuzzy rule evaluation. When showed, the view subscribes to results from the fuzzy rule evaluator and the behavior evaluator, and shows the result in a visual manner, as seen in the screenshot. Green bar colors illustrate the active behavior, yellow bar colors illustrates inactive behaviors and red bar colors illustrates behaviors that cannot be chosen because the preconditions of the actions in the behavior are not met (in this case, the actions of the red *CCHealer* behavior is on cooldown). The matchgrade of the rules are illustrated by the partially filled bars.
The view also supports details such as expanding the behavior to see the matchgrade of other

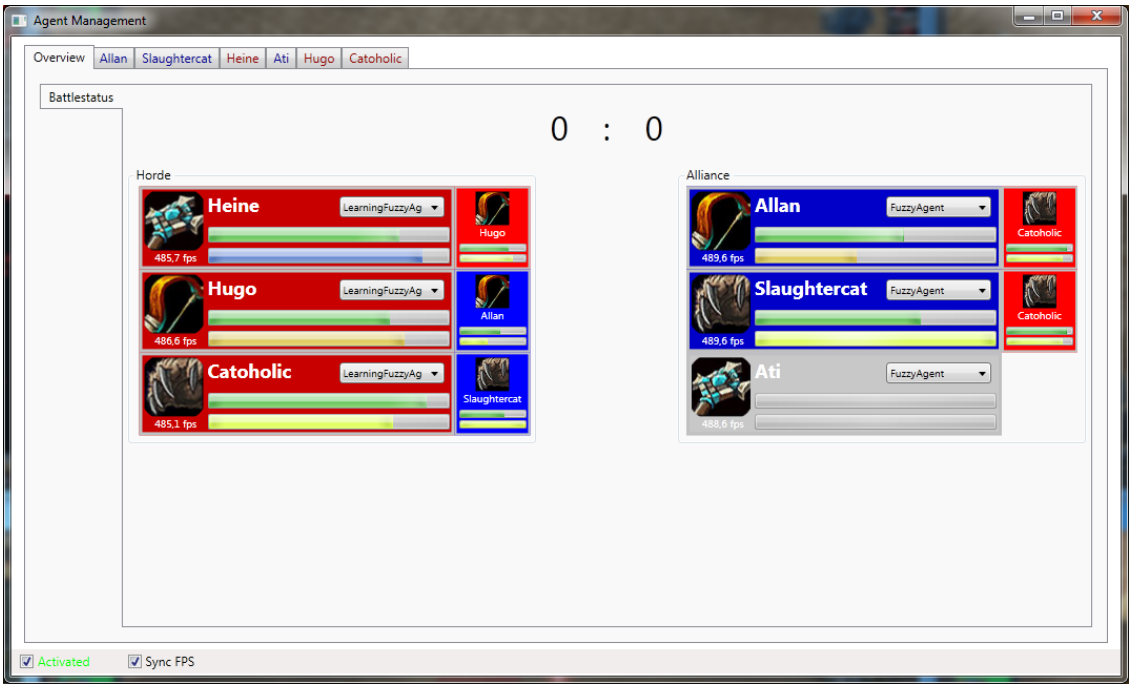


Figure 6.5: GUI for Battlestatus

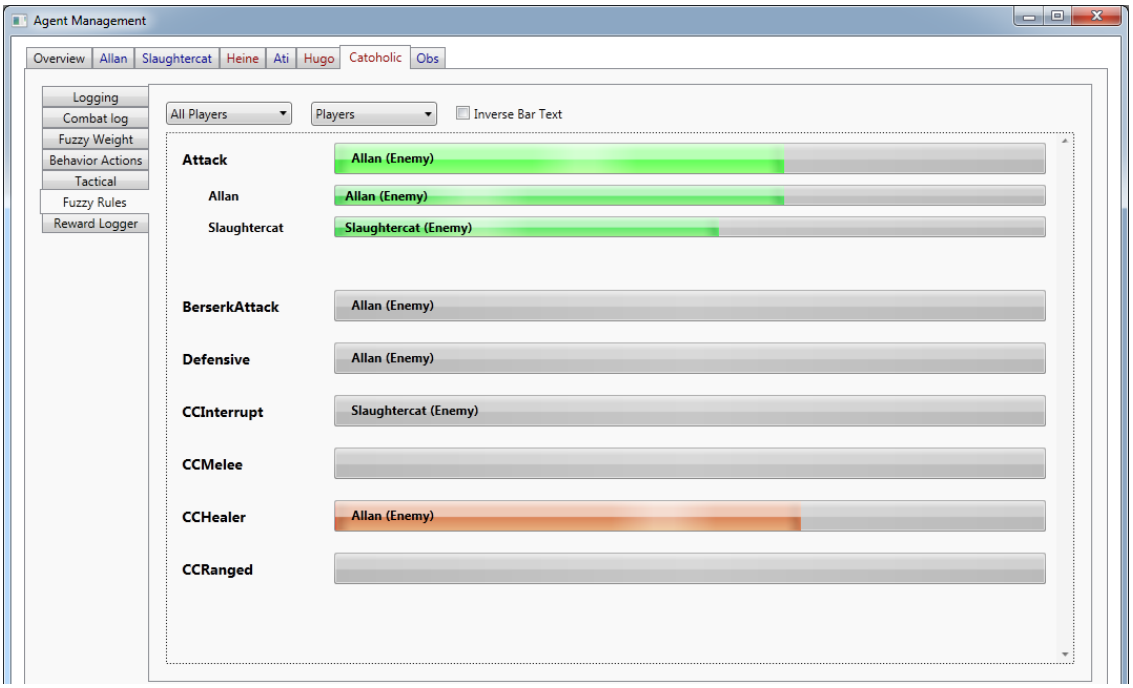


Figure 6.6: GUI for Fuzzy Rules

candidates to the object classes used in the rules. This view gave us an easy way to see how the tweaking and tuning of the fuzzy rules affects the choice of behavior by the agent.

The implementation allows the user of the application to get a good overview of the running agents, and gave an easy way to see how changing the configuration affected the agents.

6.4 Potential Fields

This section describes how potential fields has been implemented according to the design concepts in Section 5.5. The class diagram in Figure 6.7 is divided into two packages: *Potential Field Engine*; a general implementation potential fields, and *WoW Arena Potential Fields*, that is a WoW specific extension of the general implementation. The `GetPotential` method in the `PotentialFieldEngine` uses the *Location*, *Reach*, *Charge* and *Radius* to calculate the potential in a single location, see Section 4.2.

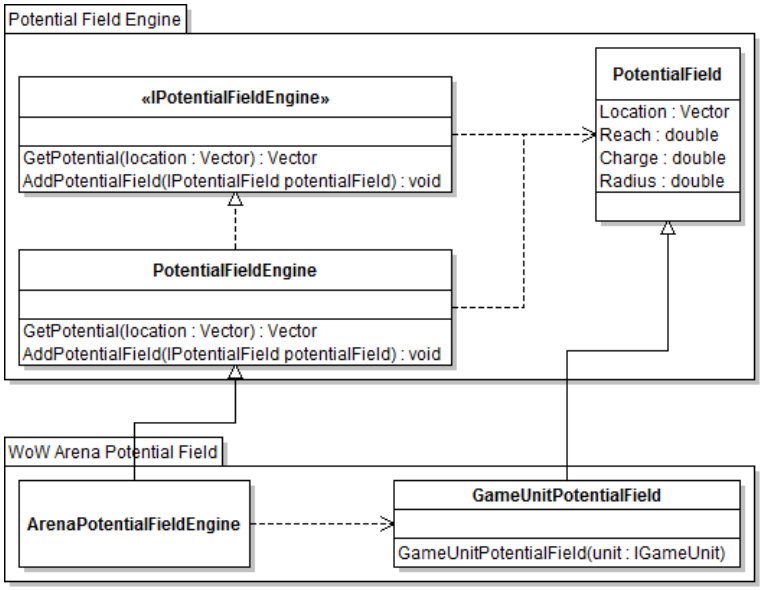


Figure 6.7: UML class diagram of the Potential Field Engine

The *WoW Arena Potential Field* ensures that each friend and enemy have its own potential field. It also adds an extra potential field in the middle of the arena so that every character is attracted to the middle, ensuring that none of the agents are leaving the defined arena.

The screenshot in Figure 6.8 is a screenshot of the potential fields visualized in the 3D engine that was implemented in the previous project. The Green color is the ground of the world and the red color is world objects; in this case the physical arena boundaries. The potential fields from both the center and all the friends and enemies are visualized with arrows on top of the ground. The *strength* of the potential is visualized by the arrow color; the more red an arrow is, the stronger the potential is.
The implementation of the potential field allows the agent to always know the best direction to walk, in order to get to a more safe location.

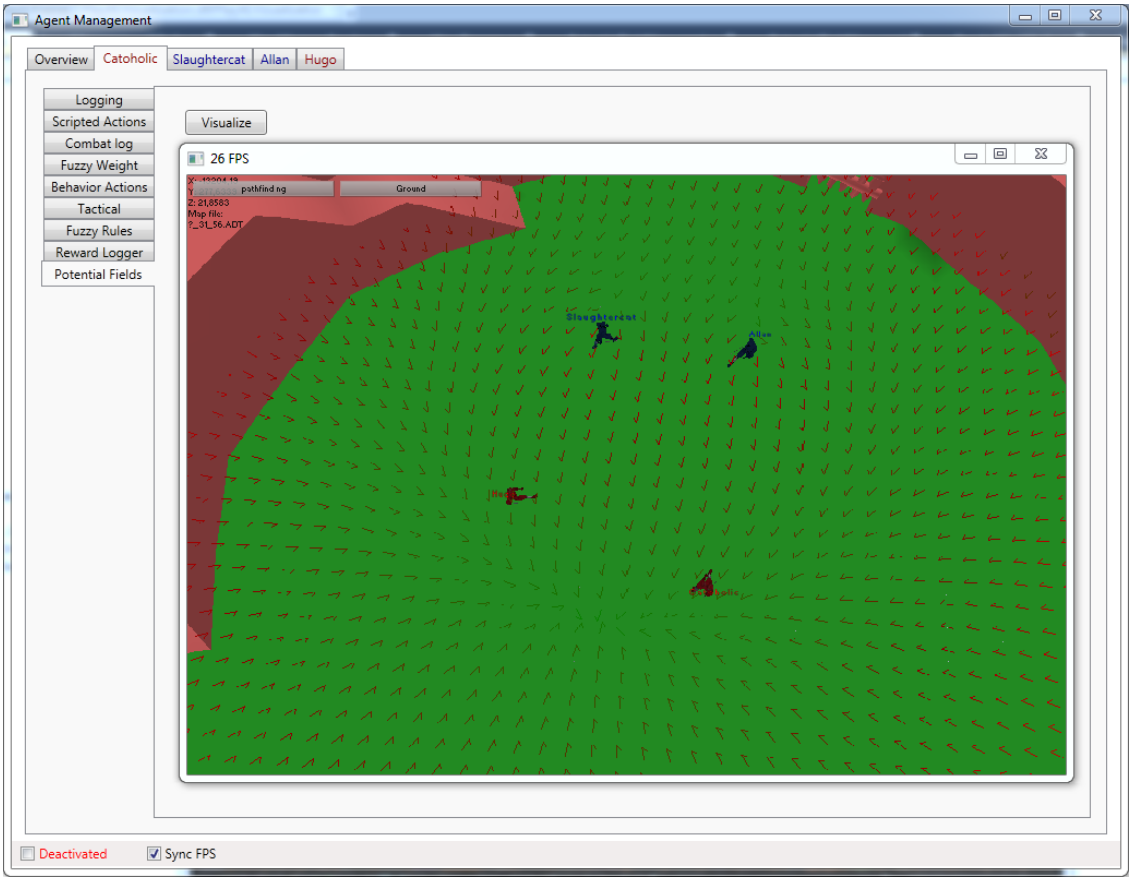


Figure 6.8: Screenshot of Potential Fields Visualization

Chapter 7

Experimental Results

This chapter gives an overview of the test environment and the competing teams, it describes the experience of configuring a new agent, and evaluates on both the convergence of the reinforcement learning and the match results of all the competing teams.

7.1 Test Environment and Competing Teams

The test environment consists of a circular arena, where each team has one character of each player type:

- **Healer**: A priest, configured to heal
- **Ranged Damage**: A hunter, dealing ranged damage
- **Melee**: A druid, configured to deal melee damage

By using one character of each player type for each team we obtain a more dynamic fight, as each team will have three different characters with different play styles. Moreover, the characters are configured in-game to be equally good, which ensures a fair fight.

The teams that will compete are the following:

- **Scripted Team**: Consists of three independent scripted agents, based on our implementation in the previous project. To make it fair, the potential fields have also been implemented in the scripted agents, allowing them to know how to move.
- **Fuzzy Team**: Consists of three independent fuzzy agents, using the fuzzy rules and weights configured by expert users.
- **Unweighted Fuzzy Team**: Consists of three independent fuzzy agents, using the fuzzy rules with no weights configured. All rules are defined by expert users, but without the possibility to change the weights to the rules; all weights are set to 1.
- **No Potential Fuzzy Team**: Consists of three independent fuzzy agents, using the fuzzy rules and weights configured by expert users, but uses no potential fields for navigation.

- **Learning Fuzzy Team**: Consists of three independent fuzzy agents that use reinforcement learning to adjust the weights of the rules. The agents have been trained against the *Fuzzy Team* before the battle.
- **Human Team**: Consists of three human players, each having previous experience in playing the given class against other human players in-game.

In the test scenario each team plays 100 fights against a *Fuzzy Team*, chosen because they proved to be a worthy opponent and thereby gives a good foundation for result comparison. Tests against non-human players are automated on a single computer, which saves a lot of time (100 fights takes around 5 hours) and ensures consistency.
The screenshot in Figure 7.1 shows how the application controls 6 individual WoW clients and how each character is controlled individually. The overview in the middle shows the victory history for both teams, and visualize each character's *health* and *power* status, allowing the supervisor of the test to identify how each team is managing without looking at each WoW client. The test environment supports the following:

- Runs on a single computer
- May run in the background, allowing the computer to be used meanwhile testing
- Counts and logs the fights won/lost for each team
- Automatically resets the environment; cooldowns, position, and everything else needed for all the characters before a new fight

Figure 7.1: Batched testing: A *Fuzzy Team* playing versus a *Fuzzy Team*

The constructed test environment is the foundation for all the testing described in this chapter.

7.2 Use case of a new Class Configuration

To give an idea of how the new agent is implemented in the framework, the following describe the steps of configuring a melee *druid*. The full configuration can be found in the Appendix 9 and Appendix 9.
The steps we used to configure each agent are the following:

1. Identify the behaviors needed by studying the responsibility and capabilities of the given role in a PvP scenario (see Section 5.2).

2. Add specific actions and preconditions for each behavior.

3. Test the behaviors individually. A behavior can for example be tested by placing an opponent in front of the agent and having only one rule that always select that specific behavior.

4. Define the fuzzy rules based on the intentions of the behaviors and the agent's personality, i.e. one agent may prefer to attack healers rather than ranged players.

5. Test the fuzzy rules by creating a fight that consists of the minimum amount of players that the agent needs to test all the cases in the fuzzy rules, i.e. the melee druid does not really care about the friendly players and is easiest to test alone against two opponents.

6. Set up a test environment for human players to play against the agents to find exploits or scenarios where the agent does not behave correctly. Create or modify rules and weights to cover the found exploits or scenarios, if any.

Some of the scenarios that we fixed during the first test of how well the *Fuzzy Team* played were the following:

- The druid selected defensive behaviors far too late, which ment that if the other team all attacked the druid, it would nearly always guarantee a win. This was fixed by adding an extra rule so that the druid went to a defensive stance both if his health was getting low or if he was taking a lot of damage.
- The druid did not select CC behaviors enough, this was fixed by increasing the weight of the rules for the crowd control behavior.
- The druid did not assist the other players if an enemy was about to die. This was fixed by adding an extra rule that made the druid prefer to attack enemies that were already being attacked, and to attack enemies that had very low health, regardless of the distance to the enemy.

Mentally the configuration felt much like learning to play the class. First, you study how the basics of the class works and what the capabilities and responsibilities of the class is related to the group. Next you find out what actions are appropriate in different scenarios, and lastly you play the game and adjust your game plan based on game scenarios. For the adjustment we used the tool developed to see how the agent reacted to the weights, see Figure 6.6, making it easy to see the effect of changing weights and adding rules. In general the expert knowledge

was very easy to describe using the fuzzy rules, making it preferable over a scripted configuration.

7.3 Evaluation

Several aspects of the agent has been evaluated. The following describes the evaluation of the effort required to configure an agent, the efficiency of the RL and the results of the fights.

7.3.1 Configuration Time

The configuring time of a scripted agent and fuzzy agent were approximately the same; however, it felt less complicated to implement the fuzzy agent as you think in behaviors and intentions while configuring, rather than the order of actions. To configure a scripted agent can be compared to configuring a single comprehensive behavior in the fuzzy agent, but because this behavior would have to take a lot of different scenarios into consideration rather than just a single intention as with the fuzzy agent, the complexity of the behavior increases a lot, making it harder to get an overview of the configuration. The hardest part of configuring the fuzzy agents was weighting the rules because it gets you out of the mindset of behaviors and intentions and requires knowledge of how the fuzzy framework works, i.e. instead of thinking in single scenarios and what behavior to select in that scenario, the expert user will have to get an overview of all the rules, determine scenarios that may conflict and configure the importance of each of these scenarios compared to each other. The reinforcement learning in the learning fuzzy agent eliminates this problem, as the weights will be learned automatically, but reinforcement learning comes with the trade-off of having to be trained before performing optimally.

7.3.2 Convergence of Reinforcement Learning

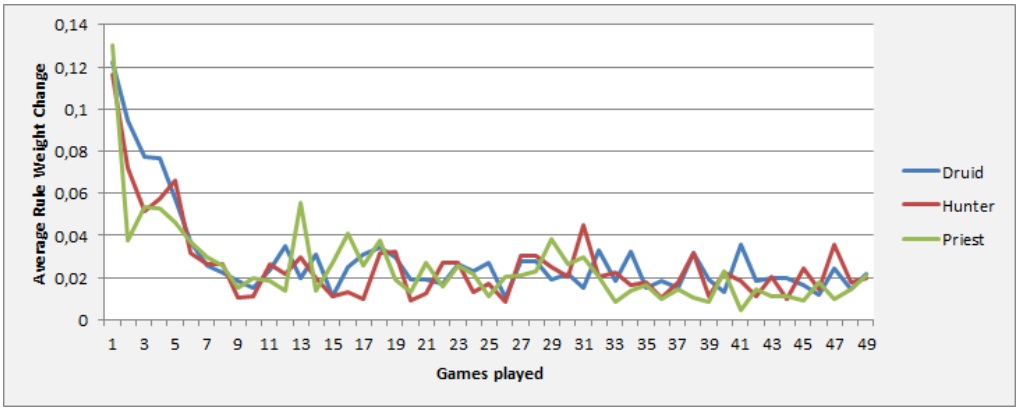


Figure 7.2: Convergence for each of the agents in the learning *Fuzzy Team*.

Figure 7.2 illustrates the reinforcement learning convergence for each of the agents in the *Learning Fuzzy Team* the data is recorded over 50 fights against the fuzzy team.

The learning rate is factored out of the data by inly using α_{max}, to show the real convergence rather than the one imposed by the decreasing learning rate As seen in the graph, the agents gets close to convergence after around 10 fights and starts to perform well against the *Fuzzy Team*.

Figure 7.3 illustrates the win rate during training and its clearly to see that after around 5

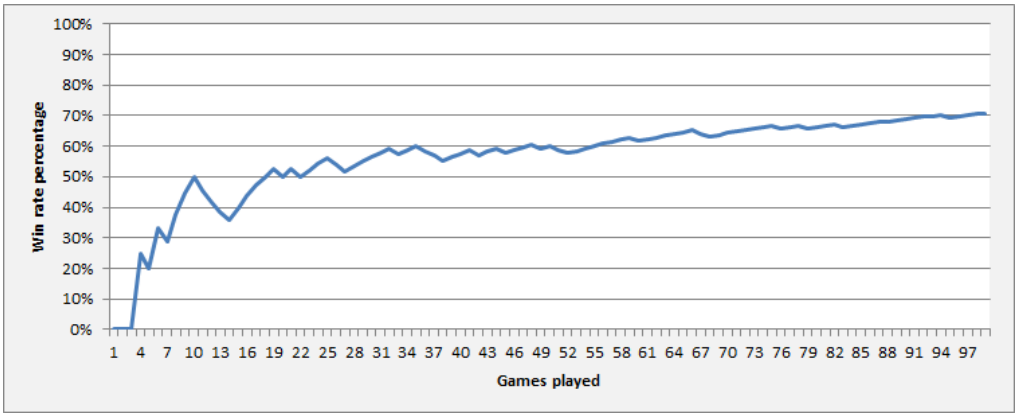


Figure 7.3: Win rate against *Fuzzy Team* during training.

fights, the *Learning Fuzzy Team* starts to challenge the *Fuzzy Team*. The reason why the *Learning Fuzzy Team* starts to compete so fast is because the agents on the team uses the same fuzzy rules and behaviors as the agents on the *Fuzzy Team*, meaning that the reinforcement learning does not have to learn the whole action policy, which would have taken a lot more time. This also limits the reinforcement learning, as it would never learn anything beyond the defined configuration.

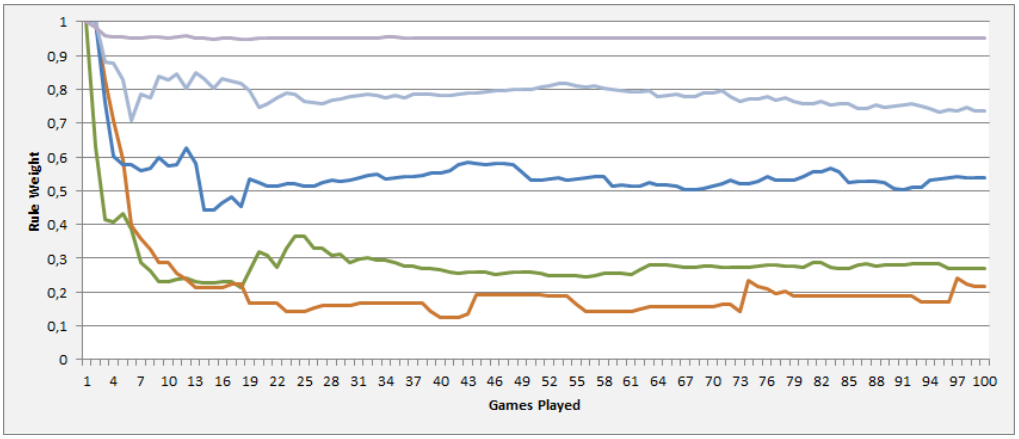


Figure 7.4: A hunters weight change while playing against the *Fuzzy Team* during training.

Figure 7.4 illustrates how the weights of all the rules are adjusted during the training for the hunter class, each line in the graph represents the weight change of a single rule over 100 fights and shows how favorable the rule is (note that only a subset of the hunter's rules are shown in the graph to avoid too much clutter). All weights are initialized to 1 and settles to

their respective weight during the training, as the graph illustrates.
The instability in the weight that sometimes occurs is when a rule covers several scenarios and only handles some of the scenarios well. The graph can therefore be used to find unstable rules and identify the scenarios that caused the weight change and configure a rule that covers that specific scenario better.

As a game takes around 3 minutes in average and it takes around 5 fights before the agent starts to play well, and the total time for a new agent to train is therefore around 15 min before reaching a playable level. Even though the training has to be done again whenever the configuration of an agent is changed, the reinforcement learning still reduces the total configuration time of a new class and simplifies the process.

7.3.3 Fight Results

Figure 7.5 shows the match results for all of the different teams competing against the *Fuzzy Team*. The *Unweighted Fuzzy Team*, *No Potential Fuzzy Team* and *Fuzzy Team* are included as control groups, as the *Unweighted Fuzzy Team* is expected to get a low win rate to ensure that weighting of the rules matters, the *No Potential Fuzzy Team* is expected to get a low win rate to ensure that movement and positioning matters, and the *Fuzzy Team* is expected to get approximately 50% win rate to ensure that one team of characters is not stronger than the other. A team could be stronger if the test environment is not configured correctly, e.g. the position of one team in the arena could be favorable or the characters of one team could be stronger due to in-game differences.

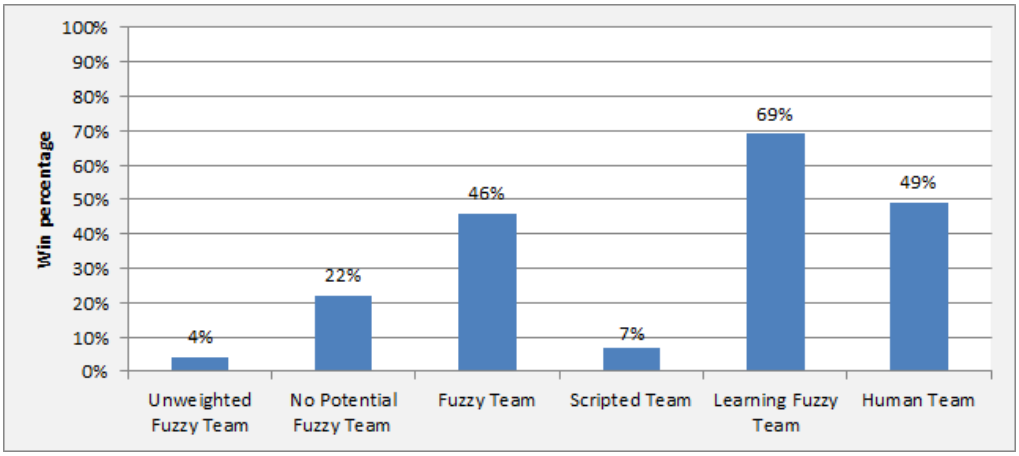


Figure 7.5: Win rate for competing against a *Fuzzy Team* for 100 fights

The three control teams play as expected, as the *Unweighted Fuzzy Team* gets a 4% win rate because no weights has been adjusted to differentiate between conflicting rules, *No Potential Fuzzy Team* gets a 5% win rate because the healer and ranged characters does not avoid the enemies, and the *Fuzzy Team* gets a win rate on 46 % showing that the win rate is approximately random.
The *Scripted Team* lost nearly all the fights because it uses a very simple single behavior model that does not support playing against multiple characters as well as the *Fuzzy Team* does.

The *Learning Fuzzy Team* gets a 69% win rate, outperforming the *Fuzzy Team*. Studying the learned weights shows that the main difference between the weights configured by the expert user and the learned weights areon rules defining which of the enemy players to attack, i.e. attacking the hunter was weighted higher in the agent configured by the expert user, but the reinforcement learning had learned that it was better to attack the priest.
The *Human Team* had a 49% win rate and the *Fuzzy Team* was both very entertaining to play against and was surprisingly good, even towards experienced humans. In order to win we had to coordinate strategies beforehand; as the agents have a very fast reaction time and does not miss click, they proved to be a real challenge to fight against.

Chapter 8

Conclusion

In the introduction of this report, the goals of the project are described. The following concludes on these different goals and evaluates the project.

Use fuzzy logic to allow expert knowledge to be easily modeled and interpreted by the agent
In this project we describe the design and implement of behavior based fuzzy logic agents playing WoW. The agents were tested in a 3 versus 3 scenario and different implementation of agents competed against a *Fuzzy Team*. In Section 7.2 we look at the experience of implementing a new class in the framework and conclude that the steps taken for the configuration felt much like learning to play the class, rather than configuring a technical application. As the behaviors were based on a intention, e.g. attack or heal, the complexity of configuring a behavior was simple compared to configuring the scripted agent because of the low amount of actions per behavior compared to the large amount of actions with lots of conditions in the scripted agent. The fuzzy rules allowed thinking in scenarios and the behaviors used to handle that scenario, rather than the order of actions. As the *Fuzzy Team* played against the *Human Team* the exploits found by the humans were easily correct by adding a few rules that covered those exploited scenarios. The only disadvantage was the weight configuration, as this is time consuming and requires some knowledge of how the fuzzy engine works, rather than only focusing of scenarios and behaviors. Looking at the match results in Section 7.3.3, we see that the *Fuzzy Team* is able to compete at a human level, and compared to the scripted agent developed in the previous project, the fuzzy agent is far superior.

Implement Potential Fields to improve the movement of the agent in the environment
As movement is an important aspect of the fight, we wanted the agent to seek towards the safest spot in the fight when possible. We combined the navigation from the previous project with potential fields, see Section 6.4, to allow the agents to move dynamically. The experimental results in Section 7.3 shows that the *Fuzzy Team* that uses potential fields has a 78% win rate against an equal team without potential fields, which proves both that the potential field works and the importance of positioning in a fight.

Use reinforcement learning to automate and improve the configuration of new agents

The weights is one of the elements in the configuration of agents that is time consuming and requires knowledge of how the fuzzy engine works, as described in Section 7.3.
To automate this, we used reinforcement learning to learn the weights and using reinforcement learning the agents were able to learn the weights of the fuzzy rules within very few games, as seen in the result in Section 7.3. The reason why the learning starts to converge so fast is because the expert knowledge is already configured as fuzzy rules and behaviors, meaning that the reinforcement learning does not have to learn the whole action policy, which would have taken a lot more time. On the other hand this also limits the reinforcement learning, as it would never learn anything beyond the defined configuration. We also noticed that decreasing the random chance in ϵ*-greedy* slightly increased the win rate of the *Learning Fuzzy Team*. An improvement to address this issue could be to not let the ϵ*-greedy* chose an entirely random action but use the matchgrade as the odds of an action to be chosen. In this way, the exploration has only a slight chance of picking low weighted fuzzy rules.
The *Learning Fuzzy Team* was also able to outperform the *Fuzzy Team*, as it learns that a healer is a better target than ranged.

Visualize the different techniques to ease tweaking and tuning of agents

When configuring an agent, a large part of the configuration consists of tweaking and tuning the fuzzy rules and their weights, as described in the use case in Section 7.2. To ease this process the a graphical tool was implemented, see Section 6.3 allowing us to easily gain an overview of the internal state of the agent, this eased the configuration of different aspects of the agent, such as behavior configuration, fuzzy rule weight adjustments and improving the reward functions for reinforcement learning. We also implemented a visualization of the potential fields in the 3D engine developed in the previous project that allowed us to confirm that the potential fields worked as intended. In general the visualization of both the overall combat and the internals of the agent was essential for both the implementation of new techniques, configuration of agents and the experimental results.

Setting up a suitable test environment for both batched testing, training, and for playing against human players

The test environment described in Section 7.1 was a huge help during the experimental results, as the test matches were very time consuming. The tests was mostly run on one computer in the background or run over night as we had no possibility of simulating or speeding up the game. The automated environment reset also ensured consistency in the results, as the characters always start with the same conditions without exceptions. This also made it more enjoyable for the *Human Team* as there were no downtime between the fights and nothing that you had to reset before a new fight against the agents started.

In this project we managed to construct an expert system that eases the implementation of agents and outperforms the previous developed scripted agents by far. The reinforcement learning reduces the configuration time and converges surprisingly fast and outperforms the agents with weights adjusted by the expert user, making the automatic learning of weights

compelling even though we were unable to simulate the training. In combination with the previous project we established an autonomous agent that uses expert knowledge for decision making and navigates around using advanced pathfinding, illustrating how to take an advanced game and use MI techniques to outperform traditional pathfinding and AI methods. The approach of behavior based fuzzy rules could also be used in other games that uses a single controlled character, such as first-person shooter games or other role playing games. As the implementation is done using patterns that eases switching out components, e.g. dependency injection, most of the framework is usable in any game by switching out the components that relates directly to WoW, such as the WoW API and WoW Interaction API.

Chapter 9

Future Work

The agent framework is designed to allow new techniques to be implemented and visualized, making it easy to integrate even more additional MI techniques. This section describes some of the ideas for future works.

Currently the behaviors are implemented using a prioritized list of actions with preconditions. Each behavior is configured by an expert user based on the intention of the behavior. The variety of classes in WoW shares some common behaviors, such as *attack*, *intensive attack*, *heal* and *intensive heal*, but the actions used for fulfilling these behaviors differs for each class. An improvement of the framework could be to automate the configuration of these common behaviors by selecting the actions that fits the intention and let the agent find an action policy with techniques such as reinforcement learning or MDPs.

The potential fields are used to guide the agent towards the best location in the fight, but there are only potential fields for the friends and enemies. An idea would be to add potential fields for objects in the environment that causes area effects such as traps for the hunter, which slows the enemies. In this way, the agents could exploit the effects on the ground and position even better.

When the human players competed against the agents, we noticed that the best way to play against the agents was to agree on a strategy (just like when two human teams compete), i.e. to communicate with each other in the team during the fight. Enabling the agents to do the same thing would open a whole new range of ways to configure the agents and possibly making them even better.

For reinforcement learning it is very usable to be able to simulate or speed up the game, allowing a faster and more automated training. The AI is often integrated in the game, which makes the simulation easier, but as our agent is built as a third party program to an existing game, we do not have access to the source of game. An idea would be to modify the game client and server, allowing the game to run faster in training periods, as we run both the server and the client. In addition to learning the weights of the rules, techniques for learning new rules could be adapted, allowing the agent to learn a play style beyond what is configured in

the fuzzy rules. In this way, the agent may learn rules that exploits the opponent and increases the performance.

Bibliography

[Activision, 2011] Activision (2011). Activision Blizzard Announces Record First Quarter Financial Results. http://investor.activision.com/releasedetail.cfm?ReleaseID=575495.

[Bonarini et al., 2003a] Bonarini, A., Invernizzi, G., Labella, T. H., and Matteucci, M. (2003a). An architecture to coordinate fuzzy behaviors to control an autonomous robot.

[Bonarini et al., 2003b] Bonarini, A., Matteucci, M., and Restelli, M. (2003b). Concepts and fuzzy models for behavior-based robotics.

[Goodrich, 2004] Goodrich, M. A. (2004). Potential Fields Tutorial.

[Hellmann, 2001] Hellmann, M. (2001). Fuzzy Logic Introduction.

[Larsen and Thorsteinsson, 2011] Larsen, S. M. and Thorsteinsson, J. L. (2011). Autonomous Agent for World of Warcraft Using Advanced Pathfinding.

[Nareyek, 2004] Nareyek, A. (2004). AI in Computer Games.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

[Tan and Cheng, 2009] Tan, C. T. and Cheng, H.-L. (2009). IMPLANT: An Integrated MDP and POMDP Learning AgeNT for Adaptive Games. The AAAI Press.

[Wender and Watson, 2008] Wender, S. and Watson, I. (2008). Using Reinforcement Learning for City Site Selection in the Turn-Based Strategy Game Civilization IV.

Appendix A

Fuzzy Rules

This appendix contains all the rules used in each of the agents, their specified weights, the weights learned through reinforcement learning and the percentage of how often the rule was selected during the training.

Weight	Learned Weight	Selected % in Learning	Rule
0.21	0,453	14,39%	IF Distance[enemy] IS Medium AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.11	0,199	2,48%	IF Distance[enemy] IS Far AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.22	0,257	4,63%	IF Distance[enemy] IS Medium AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.12	0,195	1,44%	IF Distance[enemy] IS Far AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.23	0,324	4,52%	IF Distance[enemy] IS Medium AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.13	0,199	1,09%	IF Distance[enemy] IS Far AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.71	0,547	16,52%	IF Distance[enemy] IS Medium AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.11	0,492	1,46%	IF Distance[enemy] IS Far AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.72	0,492	4,17%	IF Distance[enemy] IS Medium AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.12	0,542	0,82%	IF Distance[enemy] IS Far AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.73	0,519	3,88%	IF Distance[enemy] IS Medium AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.13	0,544	0,33%	IF Distance[enemy] IS Far AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.75	0,744	10,49%	IF Distance[enemy] IS Medium AND Health[enemy] IS Low THEN IntensiveAttack[enemy]
0.9	0,856	2,53%	IF DamageTaken[self] IS High AND Health[self] IS Low THEN Defensive[self]
1	0,963	0,11%	IF PlayerType[enemy] IS Healer AND DamageTaken[enemy] IS NOT High AND Distance[enemy] IS NOT Far AND Health[enemy1] IS Low THEN CCHealer[enemy]
0.8	0,952	5,76%	IF PlayerType[enemy] IS Healer AND DamageTaken[enemy] IS NOT High THEN CCHealer[enemy]
0.8	0,95	15,38%	IF PlayerType[enemy] IS Melee AND DamageTaken[enemy] IS NOT High AND Distance[enemy] IS NOT Far THEN CCMelee[enemy]
0.9	0,951	8,76%	IF PlayerType[enemy] IS Melee AND Distance[enemy] IS Near THEN CCMelee[enemy]
0.8	0,946	1,24%	IF PlayerType[enemy] IS Range AND DamageTaken[enemy] IS NOT High THEN CCRanged[enemy]

Table A.1: Hunter Rules

Weight	Learned Weight	Selected % in Learning	Rule
0.7	0,354	14,21%	IF Health[friend] IS Medium THEN Heal[friend]
0.9	0,424	19,42%	IF Health[friend] IS Medium AND DamageTaken[friend] IS High THEN Heal[friend]
0.8	0,579	6,68%	IF Health[friend] IS Low THEN IntensiveHeal[friend]
0.9	0,95	15,78%	IF Distance[friend] IS NOT Far THEN CCDispel[friend]
0.5	0,353	3,37%	IF Health[self] IS Medium THEN Heal[self]
0.7	0,444	4,75%	IF Health[self] IS Medium AND DamageTaken[self] IS High THEN Heal[self]
0.8	0,52	2,69%	IF Health[self] IS Low THEN IntensiveHeal[self]
0.7	1	0%	IF Distance[self] IS NOT Far THEN CCDispel[self]
1	0,95	10,58%	IF Distance[enemy] IS Near THEN CCFear[enemy]
1	0,483	0,33%	IF DamageTaken[self] IS High AND Health[self] IS Low THEN Defensive[self]
1	0,95	1,87%	IF Power[self] IS Low AND Distance[enemy] IS NOT Far THEN PowerShadowFiend[enemy]
0.3	0,419	20,33%	IF PlayerType[enemy] IS Healer AND Distance[enemy] IS NOT Far AND Health[self] IS High THEN ManaBurnAttack[enemy]

Table A.2: Priest Rules

Weight	Learned Weight	Selected % in Learning	Rule
0.21	0,379	8,49%	IF Distance[enemy] IS Near AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.11	0,248	3,93%	IF Distance[enemy] IS Medium AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.01	0,088	1,67%	IF Distance[enemy] IS Far AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.22	0,255	2,74%	IF Distance[enemy] IS Near AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.12	0,17	2,31%	IF Distance[enemy] IS Medium AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.02	0,057	1,16%	IF Distance[enemy] IS Far AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.23	0,273	3,62%	IF Distance[enemy] IS Near AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.13	0,146	1,71%	IF Distance[enemy] IS Medium AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.03	0,072	1,12%	IF Distance[enemy] IS Far AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.71	0,516	14,02%	IF Distance[enemy] IS Near AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.41	0,418	5%	IF Distance[enemy] IS Medium AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Healer THEN Attack[enemy]
0.72	0,511	3,75%	IF Distance[enemy] IS Near AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.42	0,407	1,34%	IF Distance[enemy] IS Medium AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Range THEN Attack[enemy]
0.73	0,507	2,83%	IF Distance[enemy] IS Near AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.43	0,209	0,4%	IF Distance[enemy] IS Medium AND DamageTaken[enemy] IS High AND PlayerType[enemy] IS Melee THEN Attack[enemy]
0.9	0,642	1,41%	IF Distance[enemy] IS Medium AND Health[enemy] IS Low THEN IntensiveAttack[enemy]
0.9	0,763	9,22%	IF Distance[enemy] IS Near AND Health[enemy] IS Low THEN IntensiveAttack[enemy]
0.9	0,53	4,9%	IF Distance[enemy] IS NOT Far AND DamageTaken[self] IS High AND Health[self] IS NOT High THEN Defensive[enemy]
1	0,595	2,68%	IF Distance[enemy] IS NOT Far AND Health[self] IS Low THEN DefensiveCritical[enemy]
1	0,949	12,62%	IF Distance[enemy] IS Near AND PlayerType[enemy] IS Healer THEN CCInterrupt[enemy]
1	0,948	1,91%	IF Distance[enemy] IS Near AND PlayerType[enemy] IS Melee THEN CCInterrupt[enemy]
1	0,952	9,17%	IF Distance[enemy] IS Near AND PlayerType[enemy] IS Range THEN CCInterrupt[enemy]
1	0,981	0,04%	IF PlayerType[enemy] IS Healer AND DamageTaken[enemy] IS NOT High AND Distance[enemy] IS NOT Far AND Health[enemy1] IS Low THEN CCHealer[enemy]
0.8	0,947	0,24%	IF PlayerType[enemy] IS Healer AND DamageTaken[enemy] IS NOT High AND Distance[enemy] IS NOT Far THEN CCHealer[enemy]
0.9	0,95	1,21%	IF PlayerType[enemy] IS Melee AND DamageTaken[enemy] IS NOT High AND Distance[enemy] IS NOT Far THEN CCHealer[enemy]
0.9	0,955	2,52%	IF PlayerType[enemy] IS Range AND DamageTaken[enemy] IS NOT High AND Distance[enemy] IS NOT Far THEN CCHealer[enemy]

Table A.3: Druid Rules

Appendix B

Melee Behaviors

This appendix contains the behaviors specified for the druid (melee) agent. Each table represents a single behavior. The actions in each table are sorted by priority, with the highest priority action at the top.

Action	Preconditions
Prowl	SelfCombat = False
Feral Charge(Cat Form)	
Pounce(Cat Form)	
Dash	SelfCombat = True,MinRange = 15
Faerie Fire(Cat Form)	SelfCombat = True
Tiger's Fury(Cat Form)	MaxEnergy = 10,SelfCombat = True
Savage Roar(Cat Form)	ForbiddenSelfAura = 52610,ComboPoints = 3
Rip	MaxStacks = 1,ComboPoints = 5
Maim(Cat Form)	ComboPoints = 5,MaxDiminishing = 1
Rake(Cat Form)	MaxStacks = 1
Mangle(Cat Form)	MinEnergy = 90
Mangle(Cat Form)	MaxStacks = 1
Shred(Cat Form)	

Table B.1: The Attack behavior

Action	Preconditions
Berserk(Cat Form)	SelfCombat = True
Feral Charge(Cat Form)	
Dash	SelfCombat = True,MinRange = 15
Faerie Fire(Cat Form)	SelfCombat = True
Tiger's Fury(Cat Form)	MaxEnergy = 10,SelfCombat = True
Savage Roar(Cat Form)	ForbiddenSelfAura = 52610,ComboPoints = 3
Rip	MaxStacks = 1,ComboPoints = 5
Maim(Cat Form)	ComboPoints = 5,MaxDiminishing = 1
Rake(Cat Form)	MaxStacks = 1
Mangle(Cat Form)	MinEnergy = 90
Mangle(Cat Form)	MaxStacks = 1
Shred(Cat Form)	

Table B.2: The IntensiveAttack behavior

Action	Preconditions
Survival Instincts(Bear Form)	
Barkskin(Bear Form)	
Enrage(Bear Form)	
Frenzied Regeneration	
Faerie Fire(Bear Form)	
Feral Charge(Bear Form)	
Demoralizing Roar	AlternativeCooldown = 30
Mangle(Bear Form)	
Lacerate(Bear Form)	

Table B.3: The DefensiveCritical behavior

Action	Preconditions
Faerie Fire(Bear Form)	
Feral Charge(Bear Form)	
Demoralizing Roar	AlternativeCooldown = 30
Mangle(Bear Form)	
Lacerate(Bear Form)	

Table B.4: The Defensive behavior

Action	Preconditions
Bash(Bear Form)	SelfForm = Bear,MustBeCasting = True
Skull Bash(Cat Form)	SelfForm = Cat,MustBeCasting = True
Skull Bash(Bear Form)	SelfForm = Bear,MustBeCasting = True
Maim(Cat Form)	ComboPoints = 1,SelfForm = Cat,MustBeCasting = True

Table B.5: The CCInterrupt behavior

Action	Preconditions
Hibernate	RequiredSelfAura = 69369,RequiresOnesOrMoreAura = 768,5487,MaxDiminishing = 1
Entangling Roots	MaxDiminishing = 1,RequiredSelfAura = 69369,NotTarget = True
Cyclone	MaxDiminishing = 1,RequiredSelfAura = 69369,NotTarget = True

Table B.6: The CCMelee behavior

Action	Preconditions
Cyclone	MaxDiminishing = 1,RequiredSelfAura = 69369,NotTarget = True

Table B.7: The CCHealer behavior

Action	Preconditions
Cyclone	MaxDiminishing = 1,RequiredSelfAura = 69369,NotTarget = True

Table B.8: The CCRanged behavior

Appendix C

Fuzzy Predicates

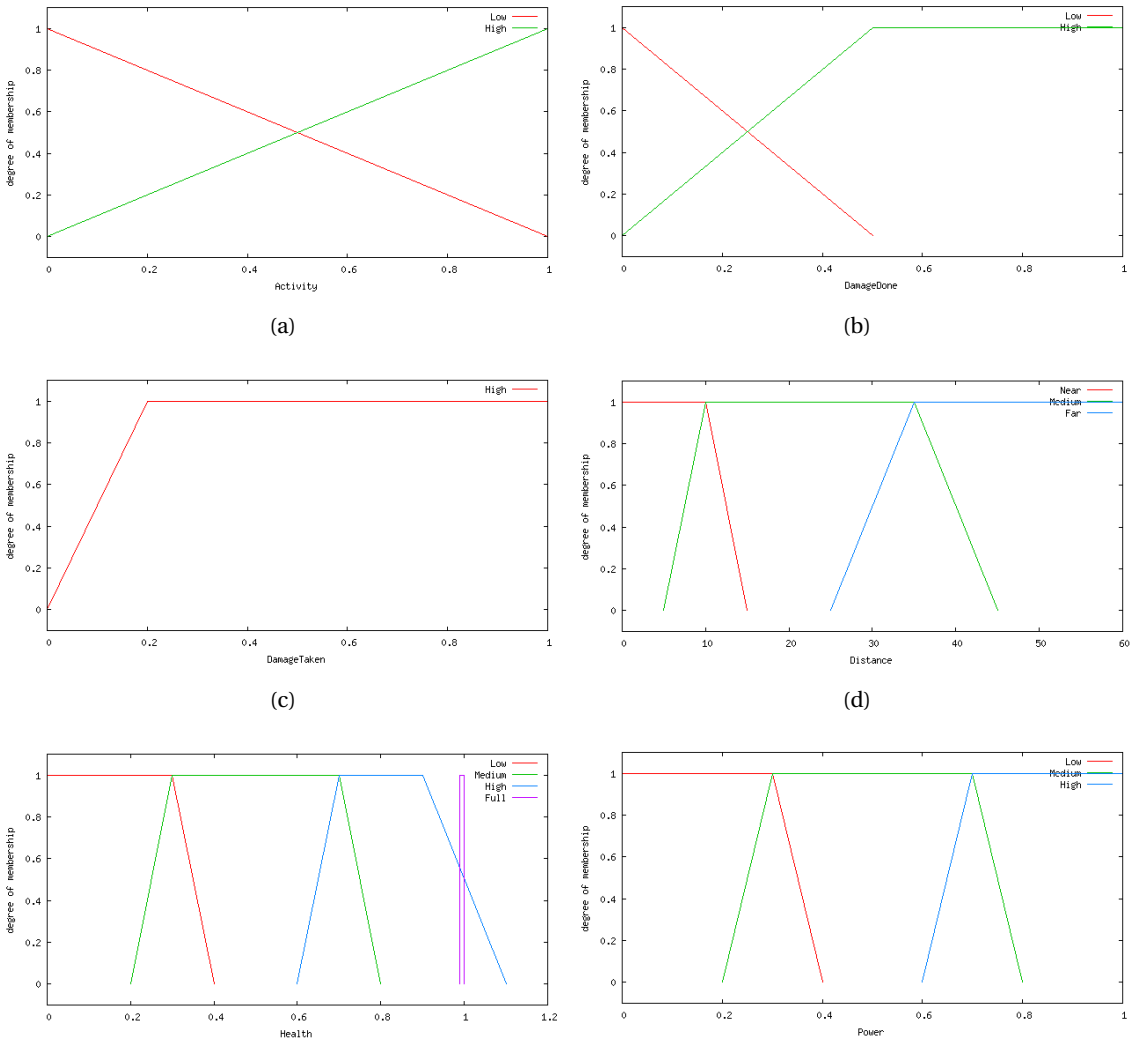


Figure C.1: Predicates used for all agents

This page is left blank for the purpose of containing the attached CD-ROM.