University of Alberta

Applying Agent Modeling to Behaviour Patterns of Characters in Story-Based Games

by

Richard Zhao

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2009
Edmonton, Alberta

Examining Committee

Duane Szafron, Computing Science

Vadim Bulitko, Computing Science

Michael Carbonaro, Educational Psychology

I dedicate this thesis to the loving memory of my grandparents.

Abstract

Most story-based games today have manually-scripted non-player characters (NPCs) and the scripts are usually simple and repetitive since it is time-consuming for game developers to script each character individually. ScriptEase, a publicly-available author-oriented developer tool, attempts to solve this problem by generating script code from high-level design patterns, for BioWare Corp.'s role-playing game *Neverwinter Nights*. The ALeRT algorithm uses reinforcement learning (RL) to automatically generate NPC behaviours that change over time as the NPCs learn from the successes or failures of their own actions. This thesis aims to provide a new learning mechanism to game agents so they are capable of adapting to new behaviours based on the actions of other agents. The new on-line RL algorithm, ALeRT-AM, which includes an agent-modeling mechanism, is applied in a series of combat experiments in *Neverwinter Nights* and integrated into ScriptEase to produce adaptive behaviour patterns for NPCs.

Acknowledgement

I am heartily thankful to my supervisor, Dr. Duane Szafron, who has been instrumental in guiding me through the research process from the very beginning. I thank my other thesis examiners, Dr. Vadim Bulitko and Dr. Michael Carbonaro, for their valuable questions, feedbacks, and suggestions.

I would like to thank Maria Cutumisu, whose work contributed to the basis of my own research. I thank my fellow researchers and M.Sc. students in the ScriptEase research group: Marcus Trenton, Neesha Desai, Christopher Kerr, Yi Yang, and AmirAli Sharifi. They have provided me with enormous amount of helpful advice throughout the year. I am also grateful of the support provided by the ScriptEase implementation team, in particular Jason Duncan, Joshua Friesen, and Robin Miller. Without their hard work I would not be able to complete my research.

Lastly, I offer my gratitude and blessings to everyone at the University of Alberta and the Department of Computing Science who supported me in any respect during the completion of this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

When the first computer games were made in the 1960s in the United States, they existed in limited circles away from public eyes. Today, the computer game industry has emerged from obscurity to the mainstream. In 2007, sales of computer game software in the United States generated 9.5 billion dollars in revenue [11]. Computer Role-Playing Games (CRPGs) are one of the best-selling genres of computer games, accounting for 18% of all sales in 2007 [11]. A computer Role-Playing Game is a story-oriented game in which the player assumes the role of an in-game avatar (a character in the game, referred to as the player character, or PC) and interacts with the virtual environment, following a plot-line set by game creators. Computer role-playing games are thus a special case of story-based games. The objective of the game usually asks for the player to accomplish a mission and, in order to do so, the player character must go through the world taking on various quests and gaining experience as the characters evolve.

In this virtual environment, all objects excepting the PC are controlled by individual pieces of programming code called scripts, and these scripts are interpreted by the game engine to determine how the game will progress. In the

earlier days of story-based games, game designers were also programmers and were able to write scripts themselves. As gaming hardware has become more advanced, larger and more complex games are expected by players and it is no longer feasible for one person to assume all the roles in a game-making team. As game designers become specialists in their area of expertise, e.g. as writers or artists, scripting has become the bottleneck in the creation of games due to the inability of these specialized game designers to express their ideas easily in the form of game scripts.

1.1 Behaviours in Games

In a story-based game, the PC usually interacts with many non-player characters (NPCs) who are computer-controlled avatars. These NPCs are fundamental to the game because they inhabit the virtual world and serve as the PC's guides, quest givers, friends, enemies, *etc.* The behaviours of NPCs are also controlled by scripts. In older story-based games such as *Ultima* [37], it was acceptable by players to leave the NPCs standing around with minimal behaviour scripted due to the limitation of gaming hardware and simplistic virtual representation. The original version of *Ultima*, shown in Figure 1.1, was created largely by a single person, Richard Garriott, in 1980. The world and the characters in it were represented by large pixels and nothing looked very believable. However, as increasingly realistic graphics were introduced into

games, players started to demand more realistic behaviours from NPCs. Most games today have manually-scripted NPCs and these scripts are usually simple and repetitive since there are hundreds or thousands of NPCs in a game and it is time-consuming for game developers to script each character individually. In Bethesda Game Studios' role-playing game *The Elder Scrolls III: Morrowind* [9], NPCs continue to stand around exhibiting minimal behaviours. Figure 1.2 shows a screenshot of this game. When the PC encounters NPCs in this world, the NPCs do not react at all to the presence of the PC, nor do the NPCs react to each other's presence. Bethesda Game Studios' newer game, *The Elder Scrolls IV: Oblivion*, has NPCs that visibly acknowledge the PC's presence but often ignore the PC's actions, for example, when a PC throws a fireball in an NPC's store.

Figure 1.1: Screenshot of *Ultima* [38], by California Pacific Computer Company

Figure 1.2: Screenshot of *The Elder Scrolls III: Morrowind*, by Bethesda Game Studios.

1.2 Generalized Behaviour Patterns

When designing a story-based game, the main storyline takes the utmost precedence. Therefore, most game development resources are put into objects, events, and NPCs related to the main storyline. Other NPCs are typically ignored or receive less attention. With increasing demand from players for realistic worlds, game designers have to start paying attention to side characters.

Developers could attempt to script everything by hand, as was done in the case of *Grand Theft Auto IV*. However, a budget of $100 million was spent on that game [19]. Such a budget is clearly not feasible for most game developers. The big challenge then is for game designers to provide game authors with tools which are able to automatically generate scripts for NPCs without having to write these scripts manually.

One approach to the scripting problem is to find common reusable patterns in the behaviours of NPCs across different games. For example, consider BioWare Corp.'s *Neverwinter Nights* (*NWN*) [21], where a particular game story is stored on the hard disk as a single package file consisting of NPCs, maps, and other objects. Special markers called "waypoints" can be specified on the map of an area and an NPC can be scripted to choose a "waypoint" at random, walk towards it and then move onto another "waypoint". As the NPC repeats this action, the player gains the impression that the NPC is wandering around in the game world. This behaviour can be generalized to a behaviour pattern called *wanderer*, which specifies that an actor should wander around the world in the fashion described. This pattern can be found in multiple story-based games and game designers can use this pattern in a new game of their choice.

For patterns to work within a particular game, the underlying pattern development tool has to be able to translate patterns into game engine-executable scripts. ScriptEase [26], a publicly-available author-oriented developer tool, uses such pattern libraries to generate NWScript code for *NWN*. The generated script

code has a C language-like syntax and contains instructions at the game level (e.g. move character C to location L). The actual execution of instructions is carried out by the *NWN* engine. ScriptEase contains a code library of four different pattern types. Encounter patterns [17] define interactions between objects in the game. Quest patterns [23] define the conditions and actions of a quest in the storyline. Dialogue patterns [27] constitute the conversations between characters in the game. Behaviour patterns [6], which define behaviours of NPCs, are related to the main topic of this thesis and are discussed in detail in chapter 4.

The use of ScriptEase patterns provides a level of indirection between the game designer and the underlying game engine. Game designers use a high level graphical interface to express their intents at an abstract level, while maintaining creative control to customize every detail of the game experience. ScriptEase eliminates the manual scripting stage of the process, reducing possibilities of human error.

1.3 Learning of Behaviours

A story-based game contains many NPCs. They interact with the player character and other NPCs as independent agents. As increasingly realistic graphics are introduced into games, players are beginning to demand more realistic behaviours from NPCs. Artificial intelligence (AI) for NPCs is thus attracting an increasing

amount of interest among game developers and researchers. Most games today have manually-scripted NPCs and scripts are usually simple and repetitive since there are hundreds of NPCs in a game and it is time-consuming for game developers to script each character individually. For example, in Bioware's *NWN*, most NPCs either have no scripts controlling their behaviour at all (resulting in nothing more than human-looking vending machines) or share a set of default scripts (they walk randomly to a set of pre-defined waypoints, defend themselves when attacked, etc.). In the official *NWN* campaign, 49 out of 61 NPCs have scripts attached in the Prelude, and only 19 out of 47 NPCs have scripts attached in the finale of Chapter One [4]. These NPCs show no personality of their own as they all behave in the same manner. Bethesda's *Oblivion* made improvements in this regard by featuring a proprietary "Radiant AI" system, so that NPCs in *Oblivion* have their own daily schedules. They will eat, sleep, go to interesting places and engage in conversations with one another. However, as is the case with most story-based games today, the NPCs in *Oblivion* have behaviours that are static – pre-determined by designers to act in their own specific schedule. Since it is hard for game designers to anticipate all possible scenarios during gameplay, the static behaviours sometimes result in an NPC who stares at an empty field or a wall for five in-game hours, as shown in Figure 1.3.

Figure 1.3: Screenshot of *The Elder Scrolls IV: Oblivion*, by Bethesda Game Studios, showing an NPC who stares at an empty field.

An NPC displaying more realistic behaviours is one that learns from past experience. For example, if an enemy is defeated by the player with ease, then this enemy should learn from the experience and not attack the player using the same tactics the second time. The noticeable lack of NPC learning in most story-based games today sometimes makes it easy for the player to exploit their computer counterparts. As soon as the player finds one method to dupe an NPC, the game loses its challenge because the same method can then be applied to all

subsequent NPCs of the same type. This is evident in the default *NWN* AI, where the NPC sorcerers can be easily defeated if the player remembers a fixed order of spells the NPC sorcerers use. In this case, the PC can choose an appropriate counter-strategy.

Learning is also very important in companion AI. A companion is a computer-controlled NPC who travels alongside the player character in the game and provides aid in various situations (e.g. battles or puzzles). To provide a better gaming experience for the player, companions should learn from the player's habits and adapt accordingly. For example, if the player character is not effective in disarming traps while the companion is, the companion should take initiative in such situations. On the other hand, if the player character is effective in disarming traps, the companion should not steal the fun/challenge away from the player. In essence, behaviour learning attempts to mimic what a real-life person would do when controlling the companion as an avatar.

A major problem facing the learning of behaviours in NPCs is the short life-span of an NPC. In a story-based game, there are usually hundreds, if not thousands of NPCs. Except NPCs that are essential to the storyline, such as major villains and allies, each individual NPC only occupies a very limited amount of screen time. If an NPC learns from past experience and disappears forever (leaves the player, is killed by the player, *etc.*), then the effort put into learning is wasted. To address this problem, this thesis introduces the concept of memetic intelligence.

Meme, a term originally introduced by the evolutionary biologist Richard Dawkins in his book *The Selfish Gene* [7], describes an abstract unit analogous to the gene that represents cultural ideas, symbols and practices. Memetics is the concept that these cultural ideas are transmitted from one mind to another, causing a replication of culture analogous to genetics. In the context of story-based games, all NPCs of a particular group (e.g. human sorcerers or dwarf fighters) can be considered a "culture", thus a transfer of knowledge between members of the group is feasible. This sharing form of intelligence is defined as memetic intelligence.

Memetic intelligence can be justified "in game" by means of information transfer. Members of the same culture tend to interact with each other frequently, thus any information gathered by one tends to pass amongst other members of the same culture by word-of-mouth or other methods of communication.

Applying the concept of memetic intelligence, I propose that all NPCs belonging to the same group share learning experiences. Therefore, each member of the group gains the experience from everyone in the group, effectively shortening the time the learning algorithm needs for each individual. In this sense, the time it takes for a learning algorithm to be effective is affordable if members of the same group reoccur in the game. This is common in story-based games, since there are usually a limited number of group types and the player will encounter a large number of NPCs from a particular group. While playing the game, players will have the feeling that the NPCs they encounter are learning as

the PC progresses.

NPC behaviour algorithms are based on a set of parameters which control the behaviour of the NPCs. With learning, the parameter values change as the algorithm is run. Arbitrary initial values for these parameters usually result in poor behaviour choices. However, the parameters need not be initialized when a player starts a game. Instead, a learning phase can be done off-line, before games are made available to the general public. Game designers can derive reasonable parameter values by training the NPCs with a set of rational behaviours during design time. This “off-line” learning can take days or weeks not otherwise available during game time. When a player starts a game, these pre-learned parameter values produce reasonable behaviours. As the game is played, the NPCs can adapt to the player's habits while the learning algorithm further adjusts these parameters on-line.

Game designers currently consider on-line learning algorithms too impractical to be incorporated into commercial games, due to the effectiveness of learning algorithms, the emergence of unexpected behaviours, the run-time overhead of learning algorithms, and the complexity of the scripting code implementing learning algorithms. This dissertation seeks to address these problems by introducing a learning algorithm that effectively learns reasonable behaviours such that the behaviours execute efficiently. I show that the scripts of the algorithm can be generated by ScriptEase. Chapter 2 describes past approaches to NPC behaviours, both in commercial games and academic research. Chapter 3

introduces a new learning approach to behaviours with agent-modeling and evaluate it using combat scenarios between NPCs in *NWN*. Chapter 4 presents ScriptEase behaviour patterns and describes how learning can be incorporated into the ScriptEase tool. Chapter 5 concludes this dissertation.

Chapter 2

Related Works

Attempts at making machines mimic human behaviour have been present for a long time. In 1950, Alan Turing proposed in his paper *Computing Machinery and Intelligence* [36] the question, "can machines think?" He then phrased the question in a concrete way which is known today as the Turing Test: a man *A* and a woman *B* stay in separate rooms, and an interrogator tries to tell them apart by asking each individual a series of questions. The objective of *A* is to try to prevent the interrogator from guessing the correct identities and the objective of *B* is to help the interrogator. All questions and answers are typewritten and this is the only method of communication. Turing then writes,

> "We now ask the question, 'What will happen when a machine takes the part of *A* in this game?' Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and a woman?" [36]

While the Turing Test continues to attract the attention of many computing science researchers, mimicking human behaviours in computer games is a relatively simpler problem. There are two fundamental differences: communication and comprehension. Instead of a computer having to communicate using the full spectrum of human language, NPCs in story-based

games need only to respond appropriately to a limited set of events. Instead of a computer needing to comprehend the full spectrum of human behaviour, an NPC need only understand behaviour appropriate to the domain of the story. Researchers have attempted different approaches to the problem of creating NPCs that mimic human behaviour convincingly. There are two main categories of approaches. *Static* behaviours are behaviours that do not change over time, while *learning* behaviours are those that evolve to become better according to the NPCs' preference. This chapter is divided into two sections, where different approaches to static and learning behaviours are discussed.

2.1 Approaches to Static Behaviours

Traditionally, NPC behaviours are defined by scripting codes attached to each individual NPC. Games such as *NWN* provide only minimal behaviours, allowing NPCs to perform meaningless actions until the PC approaches them and the plot must be advanced. These games give players a feeling that the whole world has no purpose except to respond to the player character's interests. However, few methods have been applied in commercial and published games to create more realistic behaviours because of complexities in implementing learning algorithms and the effectiveness of existing learning algorithms.

2.1.1 Cheating Behaviours

Many commercial game designers decided to simply cheat and create “fake intelligence” for characters. These NPCs appear to interact in an intelligent way with the player but, in reality, are simply disregarding the basic rules of the game world by “cheating.” A good example is *Grand Theft Auto IV* (*GTA4*) [14]. *GTA4* is an action-adventure game set in a fictional city based heavily upon modern-day New York City; the rules of the game world are similar to the rules of the real world. For example, characters don't have the ability to cast magical spells. Yet, as the player character does evil deeds, police cars will conveniently appear from around corners and start chasing the player character. By breaking the rules of the world and teleporting police cars to the player's location, the game gives the player a false sense that the police are exhibiting intelligent behaviours.

This method of cheating requires game designers to be able to predict all potential situations and manually design an ad-hoc solution for each one. For example, in one instance, an NPC gangster member is stealing a vehicle. The PC, as a rival gangster member, shows up. While normally the NPC would attack the PC, the NPC simply ignores the PC and continues to steal the vehicle. Unfortunately, the logical behaviour of interrupting the current actions upon seeing someone from a different faction is not anticipated by designers. *GTA4* is also a massive project with a development budget of about $100 million and a production crew of over one thousand people [2]. If designers working on such a

high budget game do not have the resources to plan for possible behaviours, designers of lower budget games cannot afford this resource-intensive ad-hoc cheating method. Attempts have been made in *GTA: Chinatown Wars* [13], the sequel to *GTA4,* to correct the particular behaviour mentioned above using a more principled behaviour-interrupt mechanism, so that the NPC would notice the PC. However, once a behaviour is interrupted, the NPC will not remember to go back later and finish the behaviour, since game designers have not created a behaviour-resume mechanism.

2.1.2 Finite State Machines

One non-cheating method to create intelligent behaviours is to employ finite state machines (FSMs). FSMs are one of the most frequently used methods for characters in first-person shooters [16]. In an FSM, each state represents a sequence of actions to be performed under certain conditions and state transitions occur as these conditions change. Figure 2.1 shows an example of an FSM, in which states are shown as boxes and transitions are shown as arrows. When a game starts, an NPC employing the FSM will start in the "search for a player" state, so that the first NPC action is to search for a player. If a player is found, then the "getting informed about player's location" transition condition is satisfied, and "move to assumed player location" will be the next active state. FSMs are used by the *Quake* [24] series of first-person shooter games.

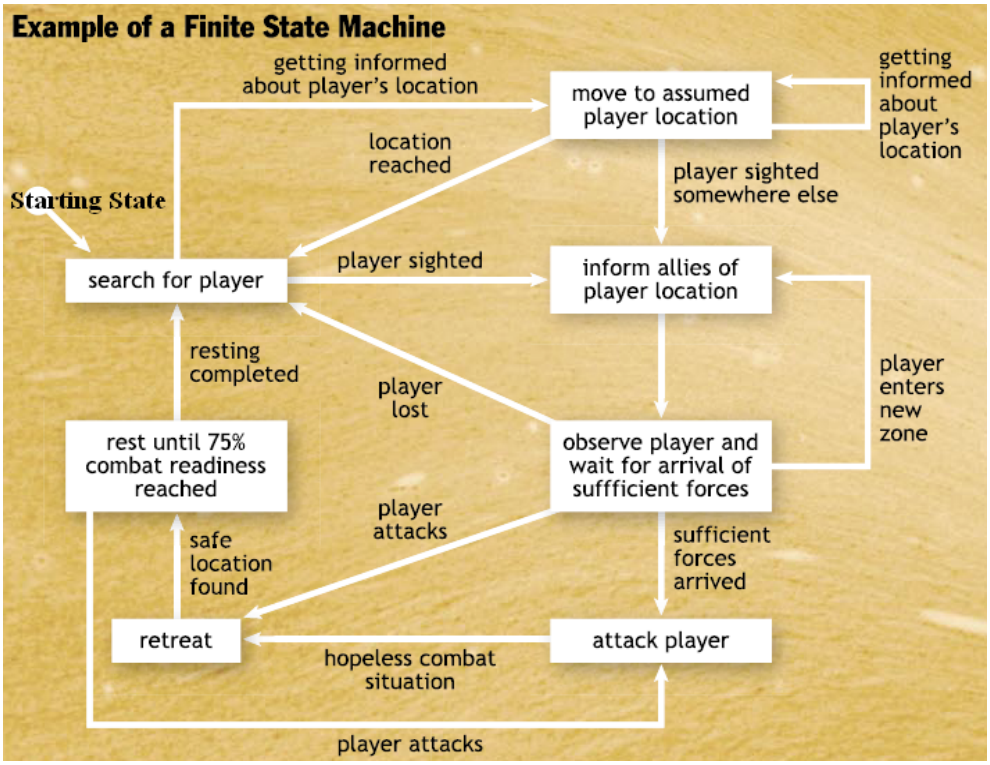

Figure 2.1: Example of a finite state machine, adopted from *AI in Computer Games* [20].

A more convenient version of the finite state machine is the hierarchical finite state machine, in which states are grouped into sets of states that share the same transition condition [16]. The sets form a hierarchy of choices from which the NPC will choose the action to perform. Although non-hierarchical FSMs have the same expressive power as their hierarchical counter-parts, grouping states together provides less redundancy in state transitions and an overall cleaner structure. *Halo 2* [15], a popular first-person shooter released in 2004, uses a more generalized version of the hierarchical FSM [16]. One difficult problem

inherent to FSMs and the hierarchical version is that the these behaviour models must be designed specifically for the NPCs that employ them, requiring that each transition or decision to be manually specified. This effectively makes the models inapplicable to a different set of NPCs without significant modification. Reusability, the likelihood that part of an architecture can be used again elsewhere with minimal or no modifications, is thus very limited with this method.

2.1.3 Goal-Oriented Approaches

Other games, such as the story-oriented *The Elder Scrolls IV: Oblivion* [10], takes a goal-oriented approach. In *Oblivion*'s pre-release interviews, designers have claimed that the game's NPCs are given goals that they must accomplish in a given day, and they must use their knowledge of the game world to find ways to accomplish these goals (e.g., to get food, NPCs can buy, hunt, or steal [10]). Beta testers have since found hilarious situations resulting from NPC behaviours, mostly involving NPCs killing each other for food or other necessities. As a result, the final release of the game features a much more restricted version of the behaviour system.

Similarly, in the popular life simulation game series *The Sims* [28], NPCs have basic *motives* which drive their choices of actions, such as “hunger,” “social,” and “hygiene.” An *advertisement* is attached to a game object to define how interaction with the object can satisfy these motives [12]. For example, the

"hunger" motive drives an NPC to obtain food and a refrigerator object advertises that the food it contains can satisfy the hunger motive. Luckily, NPCs in *The Sims* are not given the option to kill other NPCs in order to accomplish their own goals.

The behaviour system in *The Sims* suffers a similar problem to the finite state machine approach in that this system is designed to fit the settings of *The Sims*. In *The Sims*, the player acts in the role of a god who has control over the world. However, comparing to story-based games with complex levels of interactions between NPCs and the PC, characters in *The Sims* do not interact directly with the player. Applying the behaviour system in *The Sims* to a story-based game would therefore be difficult. Some newer games, such as S.T.A.L.K.E.R. [25], claim to use goal-oriented action planning for behaviours. It is not clear yet how successful this approach will be.

2.1.4 Beats and Drama Manager

Façade, an experimental game built by Michael Mateas and Andrew Stern, attempted to create a "fully-realized, one-act interactive drama" [18]. There are only two NPCs in *Façade*, as shown in Figure 2.2. The player can directly interact with both of them by typing in natural language sentences on the keyboard. Although it is not the first attempt at natural language processing, *Façade* is relatively successful in producing appropriate responses. Behaviours

are organized in hierarchies called story beats, which, in turn, are chosen by a global drama manager. The drama manager controls the overall flow of the story and creates rising and falling dramatic tension by selecting the appropriate beats for the characters. The drama manager uses player interactions to inform the selection of appropriate beats. The big drawback, according to its authors, is that *Façade* took three years to complete, and it is not certain how the architecture will adapt to a world containing more than two NPCs. Scalability, the capability of an architecture to increase performance proportionally with an increase in required resources, is very important in a typical commercially available story-based game with hundreds or thousands of NPCs.

Figure 2.2: Screenshot of the game *Façade*, showing the two NPCs, Grace and Trip. The question shown on the bottom has been typed in by the player.

2.1.5 A Pattern-based Approach

ScriptEase [26], a tool developed by researchers at the University of Alberta, is built with simplicity in mind. Its goal is to give non-programmers the ability to create stories in a computer game without having to write a single line of programming code.

ScriptEase takes a pattern-based approach. This means that NPC behaviours are generalized into a library of behaviour patterns which can be easily reused. Each behaviour pattern specifies a high-level behaviour of an NPC and its corresponding options. For example, a high-level *Patrol* pattern specifies that an NPC should patrol near a "patrol post". The only option of this pattern is a valid "patrol post", which can be any game object. This pattern is general enough to apply to any patrolling situation, including a guard patrolling around a city gate, or a mother dragon patrolling around her young child. At the same time, this pattern contains complete programming code to be attached automatically to the game story, and a game designer needs only to choose the NPC and the options.

To help with pattern library organization, ScriptEase groups behaviour patterns into categories. There are two main categories: *Proactive behaviours,* or behaviours that NPCs perform in the background, when nothing important is happening near them; and *latent behaviours,* triggered by events in the game. Latent behaviours take precedence over proactive behaviours. Within each

category, behaviours are further divided into *independent behaviours*, which an actor can perform alone, and *collaborative behaviours*, which an actor performs with a partner. These categories will be discussed in detail in Chapter 4.

ScriptEase patterns can be easily applied to multiple NPCs regardless of the creature type or class of the NPCs, so the limitations in reusability and scalability have been addressed. For example, the Patrol pattern can be applied to a guard or a dragon. However, as with all the methods described in this chapter thus far, the current ScriptEase behaviour patterns are static – no learning occurs. The more human-like an NPC behaves, the more believable they are to a player. As discussed in Chapter 1, an important application of learning is the behaviour of an companion to the PC. The companion is acting in a similar role as another player in a cooperative multi-player game, therefore acting unconvincingly would probably agitate the player. I believe that having NPCs that react intelligently with behaviours that evolve over time will more likely improve gameplay, since such behaviour is expected of a human player. The next section describes several methods for developing behaviour learning.

2.2 Approaches to Behaviour Learning

Learning algorithms are rarely seen in commercial games today [30]. Nevertheless, academic researchers have created various techniques to tackle the problem. Reinforcement learning (RL) techniques can be applied to behaviours in order to introduce the capability of adjusting behaviour selection during the course of a game.

Rich Sutton describes reinforcement learning as a process of trial-and-error such that "(the) learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them." [32]. In applying reinforcement learning methods to an NPC's behaviours, the expectation is that the NPC would be able to learn correct actions on its own, without the need of a human player to teach it every step of the way. The designer's assigned rewards replace intervention by the human player.

Two approaches will be discussed in detail below: the rule-based dynamic scripting approach by Spronck et al. [29], and the Sarsa(λ)-based reinforcement learning approach by Cutumisu et al. [5].

2.2.1 Dynamic Scripting Approach

Dynamic scripting is a rule-based learning algorithm. A *rule* specifies what an NPC should do when certain conditions are satisfied in a game and a *rule-base* is a collection of rules. Spronck implemented a dynamic scripting algorithm, which he later called DS-B [33], in the game *NWN*. Each NPC in *NWN* belongs to a single class. For example, an NPC can be a fighter or a sorcerer (there can be NPCs with multiple classes, but they are outside the scope of this discussion). The DS-B algorithm maintains a rule-base for each class in the game. Each time an NPC is created in the game, the rule-base associated with the particular class of the NPC is attached to the NPC. In this case, a rule is defined as an action with a condition (e.g. try to heal myself if my health is below 50%). A *script* containing a sequence of rules is dynamically-generated for each round and the script controls the behaviours of an NPC in the game. With dynamic scripting, the scripts generated depend upon the rules selected. Rules are probabilistically selected according to weights assigned to each rule. Rules with higher weights are more likely to be selected and the weights are updated according to a reinforcement learning algorithm.

The easiest way to evaluate the effectiveness of an algorithm is to test it in a quantitative experiment, in which the results of the experiment can be measured using concrete numbers. In a computer game, a combat scenario is well-suited to

such an experiment. It is not surprising that Spronck implemented and tested his algorithm in a combat scenario.

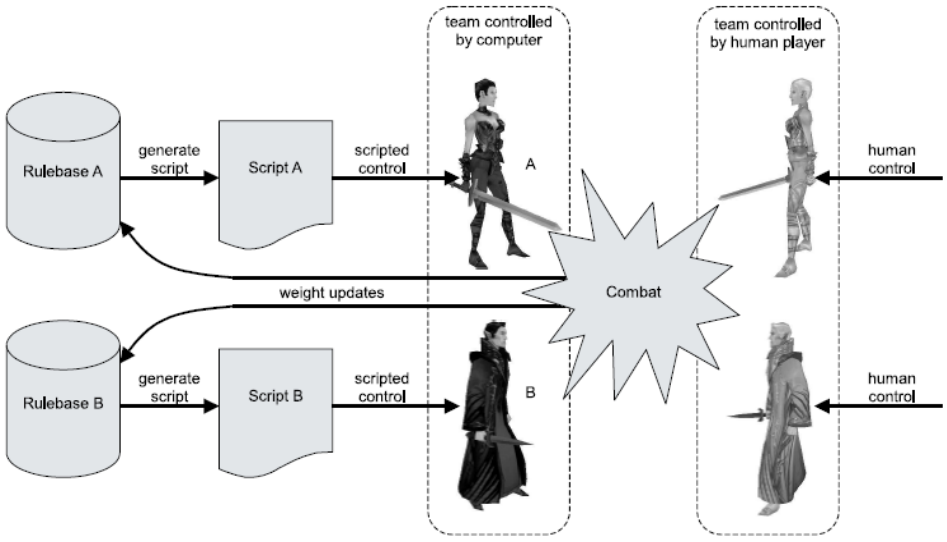

Figure 2.3: An illustration of dynamic scripting implemented in *Neverwinter Nights*, from Adaptive Game AI [29].

As shown in Figure 2.3, dynamic scripting is applied in a combat scenario with a fighter NPC (left A) and a spell-caster NPC (left B). For a fighter NPC, each script contains five rules from a rule-base of twenty rules. For a sorcerer NPC, each script contains ten rules from a rule-base of fifty rules. After each round of combat, the weights of the rules are adjusted according to whether the rule produced good results. The problem with this approach is that the rule-base is large and has to be manually ordered [35]. Moreover, once a policy is learned by this technique, it is not adaptable to a changing environment [5].

2.2.2 Sarsa(λ) and ALeRT

Sarsa(λ) [32] is an online single agent reinforcement learning algorithm. Using Sarsa(λ), a learning NPC maintains a set of states of the environment and a set of valid actions the NPC can take in the environment. A state can be anything from "whether it is now day time or night time" or "whether I am indoors or outdoors", to combat related ones including "whether I detect an enemy nearby" and "whether I have an active spell effect shielding myself from hostile spells." Examples of actions are "I move to this location" or "I attack the nearest enemy with my most powerful spell."

Time is divided into discrete steps and only one action can be taken at any given time step. At each time step *t*, the learning algorithm designer specifies an immediate reward, *r*, that is calculated from observations of the environment. The immediate reward is a numerical value evaluating the action taken by the NPC. A reward can be defined so that a positive number represents a good action and a negative number represents a bad action, and the scalar value of the reward represents how good or bad the action is. For example, the action "drop my weapon" when there is an enemy nearby is probably an undesirable action that will result in a large negative reward.

Generally, in reinforcement learning, rewards are only assigned at the end of an episode. In many situations, the immediate reward is not obtainable and it is

only obvious at the end of an episode whether the actions have led to favourable results. Having only delayed rewards at the end of an episode instead of immediate rewards at every time step will slow down the learning process. However, in both cases, the learning algorithm will converge to the same strategy. In computer games, the speed of learning is extremely important, so identifying an immediate reward is essential.

Which action to take at each step is determined through a policy. A policy π is a mapping of each state s and each action a to the probability of taking action a in state s at each time step. The value function for a policy determines how good an action is given that policy. The value function for policy π, denoted $Q^{\pi}(s,a)$, is the expected total reward (to the end of the game) of taking action a in state s and following π thereafter. Note that $Q^{\pi}(s,a)$ is the cumulative reward expected from the present until the end of the game in a non-discounted finite-horizon setting, not the immediate reward, r.

The policy π chooses the action with the maximum value given by Q^{π}. However, Sarsa(λ) uses an epsilon-greedy policy. This means that a random action will be taken epsilon-percent of the time to encourage exploration. For example, if epsilon (ε) equals 2%, then the actions determined through π will be taken 98% of the time, while 2% of the time the NPC will perform random actions. Epsilon is called the exploration rate.

In reality, there is no way to compute the exact values of $Q^{\pi}(s,a)$. The goal of the learning algorithm Sarsa(λ) is to obtain a running estimate, $Q(s,a)$, of the

value function $Q^{\pi}(s,a)$. $Q(s,a)$ is updated at each time step t and evolves over time as the environment changes dynamically. The estimate $Q(s,a)$ is initialized arbitrarily and is learned through experience. Sarsa(λ) uses the Temporal-Difference prediction method to update the estimate $Q(s,a)$, where α denotes the learning rate, γ denotes the discount rate, and e denotes the eligibility trace:

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

The subscripts represent time steps.

The learning rate α controls how quickly $Q(s,a)$ changes, thus how quickly the behaviour of the NPC changes. From the assignment operation above, it is clear that a bigger α will result in a bigger change. In traditional Sarsa(λ) learning, α is either a small fixed value or it decreases over time in order to guarantee convergence. Within a computer game environment, the slow learning of Sarsa(λ) poses a serious problem, since a player will get frustrated if it takes five hours for an NPC companion to learn that staring at a wall is undesirable. The ALeRT (Action-dependent Learning Rates with Trends) algorithm [5] modifies the standard Sarsa(λ) algorithm in three ways.

Firstly, trends in the environment are analyzed by tracking the change of $Q(s,a)$ at each step. The trend measure is based on the Delta-Bar-Delta measure [31] in which a window of previous steps is kept. *Delta-bar*, which represents the trend, is calculated by taking the average value of δ (the temporal difference error at each step) over the window. If the current change follows the trend, then the learning rate is increased; if the current change differs from the trend, then the

learning rate is decreased.

Secondly, as opposed to a global learning rate, α, ALeRT establishes a separate learning rate, $\alpha(a)$ for each action, a, of the learning agent. This enables each action to form its own trend so that when the environment is changed, only the actions affected by that change register disturbances in their trends.

Thirdly, as opposed to a fixed exploration rate or a monotonic decreasing exploration rate, ALeRT uses an adjustable exploration rate, changing in accordance with a positive or negative reward. If the received reward is positive, implying the NPC has chosen good actions, the exploration rate is decreased. On the other hand, if the reward is negative, implying the NPC has not been choosing good actions, the exploration rate is increased.

To compare the effectiveness of ALeRT to DS-B, the ALeRT algorithm was similarly applied to *NWN*, in a series of combat experiments with NPCs. ALeRT achieved good results with fighter NPCs. However, when I applied the ALeRT algorithm in duels between two sorcerer NPCs, ALeRT exposed one of its weaknesses. The complexity of the policy that a sorcerer NPC needs to learn exceeds the current capabilities of ALeRT. The next chapter describes how ALeRT can be exploited and how I improved ALeRT with an agent-modeling technique.

Chapter 3

Learning with Agent Modeling

I will start this chapter by describing the set-up of the experiments I performed that identified the need for agent modeling. As described in previous chapters, NPCs who are capable of adjusting their behaviours based on the game world and interactions with PCs are more likely to provide a more enjoyable gaming experience. Research in behaviour learning has started only in recent years and commercial games today seldom include NPCs with behaviour learning, due to factors such as the effectiveness and complexity of learning algorithms. Dynamic scripting (DS-B) [29] and ALeRT [5] are two different approaches to the problem of behaviour learning developed in academia for use in games. The effectiveness of the two algorithms were compared in a set of experiments using combat between a pair of fighters [5]. A combat scenario was chosen because the results of the experiment can be measured using concrete numbers of wins and loses. I used a similar combat scenario in which each team consisted of either a single sorcerer or a sorcerer and a fighter.

3.1 Implementation of ALeRT in Neverwinter Nights

The reinforcement learning algorithm ALeRT was implemented in an arena-combat environment using the commercial *NWN* game engine, as shown in Figure 3.1. In *NWN*, an agent is defined as an AI-controlled character (an NPC), and a player character (PC) is controlled by the player. Each agent responds to a set of events in the environment. For example, when an agent is attacked, the script associated with the event *OnPhysicalAttacked* is executed.

Figure 3.1: Spronck's arena [22] showing combat between two teams.

An episode is defined as one fight between two opposing teams, starting when all agents of both teams have been created in the arena and ending as soon as all agents from one team are destroyed. A step is defined as one round of combat, which lasts six seconds of game time.

Every episode can be scored as a zero-sum game. When each team consists of one agent, the total score of an episode is 1 if the team wins and -1 if the team loses. The immediate reward function of one step is defined as:

$$r = 2[\frac{\hat{H}_s}{\hat{H}_s + \hat{H}_o} - \frac{H_s}{H_s + H_o}]$$

H represents the hit points of an agent where the subscript s denotes the hit points of the agent whose actions are being selected by the learning algorithm and the subscript o denotes the hit points of the opposing agent. An H with a hat (^) denotes the hit points at the current step and an H without a hat denotes the hit points at the previous step. When an agent reaches 0 hit points, it dies.

Each fraction inside the square brackets represents the proportion of hit points of the learning agent compared to the sum of the hit points of both agents at one time step. A larger fraction means a more advantageous situation for the learner. Therefore, a positive reward represents an improvement in situation for the learning agent since the previous step, and a negative number represents a degradation. The multiplier of 2 outside the square brackets is needed for normalization (refer to discussion on the total reward in the next paragraph). The total reward of an episode (r_{total}) must be the sum of the rewards at each time step.

Let n be the final step of the episode. The total reward is calculated in Figure 3.2.

$$
\begin{aligned}
& r_{total} \\
&= r_1 + r_2 + \ldots + r_n \\
&= 2\left[\frac{H_s^{\,1}}{H_s^{\,1} + H_o^{\,1}} - \frac{H_s^{\,0}}{H_s^{\,0} + H_o^{\,0}}\right] + 2\left[\frac{H_s^{\,2}}{H_s^{\,2} + H_o^{\,2}} - \frac{H_s^{\,1}}{H_s^{\,1} + H_o^{\,1}}\right] + \ldots + 2\left[\frac{H_s^{\,n}}{H_s^{\,n} + H_o^{\,n}} - \frac{H_s^{\,n-1}}{H_s^{\,n-1} + H_o^{\,n-1}}\right] \\
&= 2\left[\frac{H_s^{\,n}}{H_s^{\,n} + H_o^{\,n}} - \frac{H_s^{\,0}}{H_s^{\,0} + H_o^{\,0}}\right] \\
&= 2\left[\frac{H_s^{\,n}}{H_s^{\,n} + H_o^{\,n}} - \frac{1}{2}\right]
\end{aligned}
$$

Figure 3.2: The total reward of an episode.

The subscripts on r and the superscripts on H represent time steps. At time step 0 (the start of an episode), $H_s^0 = H_o^0$ since both sides start with the same amount of hit points. Therefore, if the agent survives at the end of the episode (*i.e.* the opponent is killed), $H_o^{\,n} = 0$, $H_s^{\,n} > 0$, and $r_{total} = 1$. On the other hand, if the agent dies at the end (*i.e.* the opponent survives), $H_s^{\,n} = 0$, $H_o^{\,n} > 0$ and $r_{total} = -1$.

When each team consists of two agents, the total score can be defined similarly. The hit points of all team members are added together. The total score of an episode is 1 if the team wins and -1 if the opposing team wins. If the subscripts 1 and 2 denote team members 1 and 2, then the immediate reward function for one step is defined as:

$$r = 2\left[\frac{\hat{H}_{s1} + \hat{H}_{s2}}{\hat{H}_{s1} + \hat{H}_{o1} + \hat{H}_{s2} + \hat{H}_{o2}} - \frac{H_{s1} + H_{s2}}{H_{s1} + H_{o1} + H_{s2} + H_{o2}}\right]$$

The estimated value function $Q(s,a)$ is calculated using features from a feature

vector. The feature vector contains combinations of states from the state space and actions from the action space. Available states and actions depend on the properties of the agent. Two types of agents are used in the experiments, a fighter and a sorcerer. Fighters and sorcerers are common units in *NWN*.

The state space of a fighter consists of three Boolean variables:

1. the agent's hit points are lower than half of the initial hit points;

2. the agent has an enhancement potion available;

3. the agent has an active enhancement effect.

Each Boolean variable can take the value of true (1) or false (0), resulting in eight different states in total.

The action space of a fighter who is in combat against a single agent consists of four actions shown in Table 3.1.

Action Category	**Description**
Attack melee	Attack with the best melee weapon available
Attack ranged	Attack with the best ranged weapon available
Speed	Drink a speed enhancement potion
Heal	Drink a healing potion

Table 3.1: Actions of a fighter

The state space and the action space of a sorcerer are relatively larger because of a sorcerer's ability to use a large number of spells. The state space of a sorcerer in combat against a team that includes a sorcerer consists of four Boolean variables, to a total of 16 different states:

1. the agent's hit points are lower than half of the initial hit points;

2. the agent has an active combat-protection effect;

3. the agent has an active spell-defence effect;

4. the opposing sorcerer has an active spell-defence effect.

The action space of a sorcerer in combat against a single agent consists of the eight actions shown in Table 3.2.

Action Category	**Description**
Attack melee	Attack with the best melee weapon available
Attack ranged	Attack with the best ranged weapon available
Cast combat-enhancement spell	Cast a spell that increases a character's ability in physical combat, e.g. Bull's strength
Cast combat-protection spell	Cast a spell that protects a character in physical combat, e.g. Shield
Cast spell-defence spell	Cast a spell that defends against hostile spells, e.g. Minor Globe of Invulnerability
Cast offensive-area spell	Cast an offensive spell that targets an area (multiple creatures), e.g. Fireball
Cast offensive-single spell	Cast an offensive spell that targets a single character, e.g. Magic Missile
Heal	Drink a healing potion

Table 3.2: Actions of a sorcerer

In a more complex environment where both teams have a fighter and a sorcerer, ALeRT can also be applied. However, each action must be appended with a target. The targets are friendly fighter, friendly sorcerer, enemy fighter, and enemy sorcerer, depending on the validity of the action. This system can be extended easily to multiple members for each team.

3.2 Deficiency with ALeRT

A series of experiments was conducted with fighter NPCs, where one fighter used the ALeRT algorithm to select actions and the opponent fighter used other algorithms (the default *NWN* scripts, DS-B, hand-coded optimal strategy). ALeRT achieved good results in terms of policy-finding and adaptivity [5]. However, when I applied the ALeRT algorithm to duels between two sorcerer NPCs, the experimental results were not as good as I expected. The ALeRT algorithm did not produce consistent winning results. Section 3.5 Experiments and Evaluation explains the results in detail.

There may be several reasons for this. A fighter only has limited actions. For example, a fighter may only: use a melee weapon, use a ranged weapon, drink a healing potion or drink a speed enhancement potion. However, a sorcerer has more actions, since sorcerers can cast a number of different spells. Making each spell a unique action would mean that the learning algorithm has to choose between dozens of actions. Even after I abstracted the spells into eight categories, a sorcerer still had eight actions instead of the four actions available to fighters.

Not only does a sorcerer have a greater number of actions to choose from, the actions themselves are also more complex. An action that a fighter can take against another fighter produces similar outcomes regardless of the environment, e.g. stabbing an opponent with a sword will have the same expected damage throughout the combat. However, when a sorcerer casts a spell, the expected

damage depends on the state and previous actions of the agent the spell has targeted. For example, if the spell target is protected by a Minor Globe of Invulnerability, then a Fireball will do no damage.

3.3 Modeling an Opponent

With ALeRT, each action is chosen based on the state of the environment, where the state does not include information about recent actions of other agents. Although there is no proof that ALeRT always converges to a global optimal strategy, in practice it finds a strategy that gives positive rewards in most situations. In the simple case of a fighter, where the immediate reward of an action does not depend on the previous actions of the opponent, a winning strategy can be constructed that is independent of the other agent's actions. For example, in the four-action fighter scenario it is always favourable to take a speed enhancement potion in the first step. Unfortunately, where the previous actions of the other agents are important, it is not so easy to find a winning strategy. In the case of sorcerers, the spell system in *NWN* is balanced so that spells can be countered by other spells. For example, a Fireball spell can be rendered useless by a Minor Globe of Invulnerability spell. In such a system, any favourable strategy has to take into consideration the actions of other agents, in addition to the state of the environment. The task is to learn a model of the opposing agent and subsequently come up with a counter-strategy that can exploit such

information.

Uther and Veloso [39] described a method of Opponent Modeling Q-learning, by adding features based on the predicted current actions of other agents to the feature vector. To model agents in my experiments, I adapted this opponent modeling idea to ALeRT. I created ALeRT-AM, ALeRT with Agent Modeling.

3.4 ALeRT-AM

To incorporate agent modeling, I modified the value function $Q(s,a)$ to contain three parameters, the current state s, the agent's own action a, and the opposing agent's action a'. The modified value function is denoted $Q(s, a, a')$. At every time step, the current state s is observed and action a is selected based on a selection policy, ε-greedy, where the action with the largest estimated $Q(s, a, a')$ is chosen with probability $(1-\varepsilon)$, and a random action is chosen with probability ε. Since the opponent's next action cannot be known in advance, a' is estimated based on a model built using past experience. The ALeRT-AM algorithm is shown in Figure 3.3.

$N(s)$ denotes the frequency of game state s and $C(s, a)$ denotes the frequency of the opponent choosing action a in game state s. For each action a, the weighted average of the value functions $Q(s, a, a')$ for each possible opponent action a' is calculated, based on the frequency of each opponent action. This weighted average is used as the value of action a in the ε-greedy policy, as shown on the line marked by **.

Initialize Q arbitrarily, $C(s,\ a) \leftarrow 0, N(s) \leftarrow 0$ for all s,a

Initialize $\alpha(a) \leftarrow \alpha_{\max}$ for all a

Repeat (for each episode) until s is terminal

$\vec{e} = \vec{0}$

$s,\ a,a'$ ← initial state and action of episode

Let F represent the set of features from $s,\ a,\ a'$

Repeat (for each step t of episode)

For all $i \in$ F:

$e(i) \leftarrow e(i) + \frac{1}{\|F\|}$ (accummulating eligibility traces)

Take action a_t, observe reward, r_t,

opponent's action a'_t and next state, s_{t+1}

With probability 1 - ε:

** $a_{t+1} \leftarrow argmax_a \sum_{a'} \frac{C(s_t,a')}{N(s_t)} Q(s_t,a,a')$

or with probability ε:

$a_{t+1} \leftarrow$ a random action $\in$ A

$\delta \leftarrow r_t + \gamma \sum_{a'} \frac{C(s_{t+1},a')}{N(s_{t+1})} Q(s_{t+1},a_{t+1},a') - Q(s_t,a_t,a'_t)$

$Q \leftarrow Q + \alpha(a)\delta\vec{e}$

$\vec{e} = \gamma\lambda\vec{e}$

$C(s_t,a'_t) \leftarrow C(s_t,a'_t) + 1$

$N(s_t) \leftarrow N(s_t) + 1$

$\Delta\alpha = \frac{\alpha_{\max} - \alpha_{\min}}{\alpha_{\text{steps}}}$

if $\overline{\delta(a)}\delta > 0 \ \wedge \left|\overline{\delta(a)} - \mu_{\overline{\delta(a)}}\right| > f\sigma_{\overline{\delta(a)}}$

$\alpha(a) \leftarrow \alpha(a) + \Delta\alpha$

else if $\overline{\delta(a)}\delta \le 0$

$\alpha(a) \leftarrow \alpha(a) - \Delta\alpha$

end of step

$\Delta\varepsilon = \frac{\varepsilon_{\max} - \varepsilon_{\min}}{\varepsilon_{\text{steps}}}$

if $\sum_{step} r_{step} = 1$

$\varepsilon = \varepsilon - \Delta\varepsilon$

else

$\varepsilon = \varepsilon + \Delta\varepsilon$

end of episode

All variables are initialized to 0

The action with maximum value is chosen, where the value is calculated using a weighted average of opponent actions

The temporal difference error is also adjusted by a weighted average of opponent actions

The frequency of opponent action in current state is incremented; The frequency of current state is incremented

Figure 3.3: The ALeRT-AM algorithm

In Figure 3.3, the red boxes highlight the agent-modeling components of the algorithm. When agent modeling is implemented in the learning algorithm of the sorcerer, besides features from the state space and the action space, the feature vector also contains features from an opponent action space, consisting of estimated actions for the opposing sorcerer agent.

3.5 Experiments and Evaluation

For the two opposing teams in the experiments, one team is scripted with ALeRT-AM, while the other team is scripted with one of the following three strategies. NWN is the default Neverwinter Nights strategy, a rule-based static probabilistic strategy. DS-B represents Spronck's rule-based dynamic scripting method. ALeRT is the unmodified version of the online-learning strategy.

Each experiment consisted of ten trials and each trial consisted of either one or two phases of 500 episodes. All agents started with zero knowledge of themselves and their opponents, other than the set of legal actions they can take. At the start of each phase, each agent was configured with a specific set of equipment and in the case of a sorcerer, a specific set of spells. Opposing pairs of agents of the same class are configured identically. In a one-phase experiment, I evaluated how quickly our agents could find a winning strategy. In a two-phase experiment, I evaluated how well our agents could adapt to a different configuration (set of sorcerer spells). I decided to include 500 episodes in one

phase because in my preliminary experiments, more episodes using the same configuration did not produce more interesting results. It should be noted that the *NWN* combat system uses random numbers, so there is always an element of chance.

To keep consistency through the experiments, all RL parameters are kept the same for the experiments. Experiments with the fighter agent use the same parameter values as the experiments by Cutumisu et al. [5], so the results can be compared. With the sorcerer agent, most parameter values are kept the same as the fighter agent, except the maximum exploration rate. I believe that since the exploration rate determines how frequent an action is chosen at random, a larger number of actions require a corresponding increase in exploration. Some preliminary experiments were run to determine a desired maximum exploration rate. Values used for my experiments are shown in Table 3.3.

Parameter	**Value**
Maximum learning rate (alpha max)	0.2
Minimum learning rate (alpha min)	0.01
Number of steps from alpha max to alpha min	20
Maximum exploration rate (epsilon max) - fighter	0.02
Maximum exploration rate (epsilon max) - sorcerer	0.08
Minimum exploration rate (epsilon min)	0.005
Number of steps from epsilon max to epsilon min	15
Discount rate (gamma)	1
Eligibility trace (lambda)	0

Table 3.3: Parameter values used in the experiments.

3.5.1 Motivation for ALeRT-AM

The original ALeRT algorithm, with a fighter as the agent, was shown to be superior to traditional Sarsa(λ) and NWN in terms of strategy-discovering. It is also superior to Sarsa(λ), NWN, and DS-B for environmental-change adaptation [5]. I begin by presenting the results of experiments where ALeRT was applied to more complex situations.

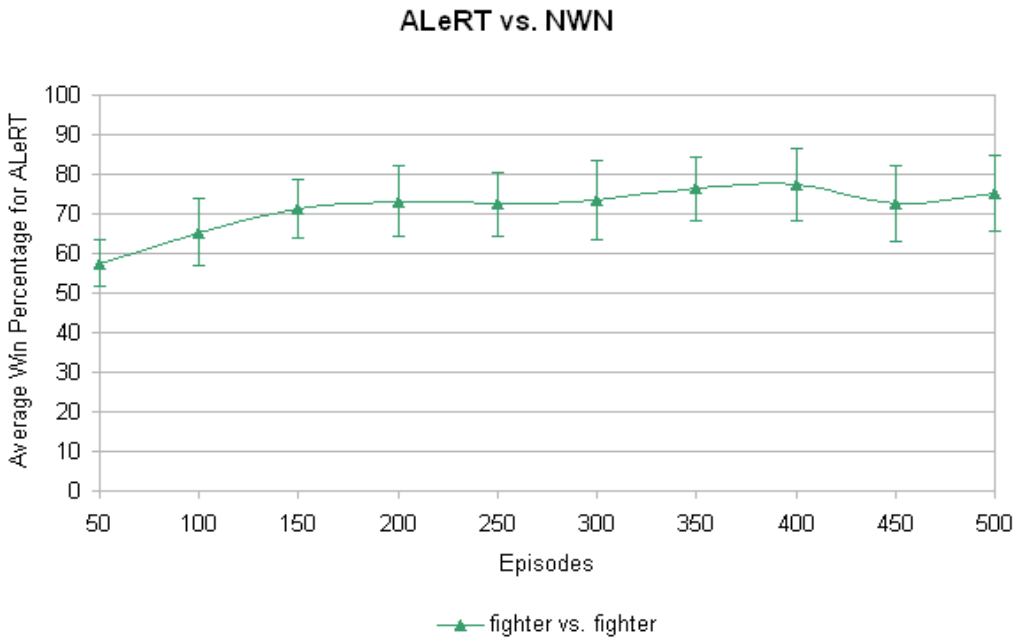

Figure 3.4: ALeRT against NWN for the fighter team.

Figures 3.4 to 3.6 show the result of ALeRT versus the default NWN algorithm, for three different teams, one fighter team, a new sorcerer team and a new sorcerer-fighter team. ALeRT is quick to converge on a good counter-strategy that results in consistent victories for all teams. Since for each team, the

default NWN strategy is a fixed strategy based on a set of static rules, once a counter-strategy is found, it will always work against the static NWN strategy.

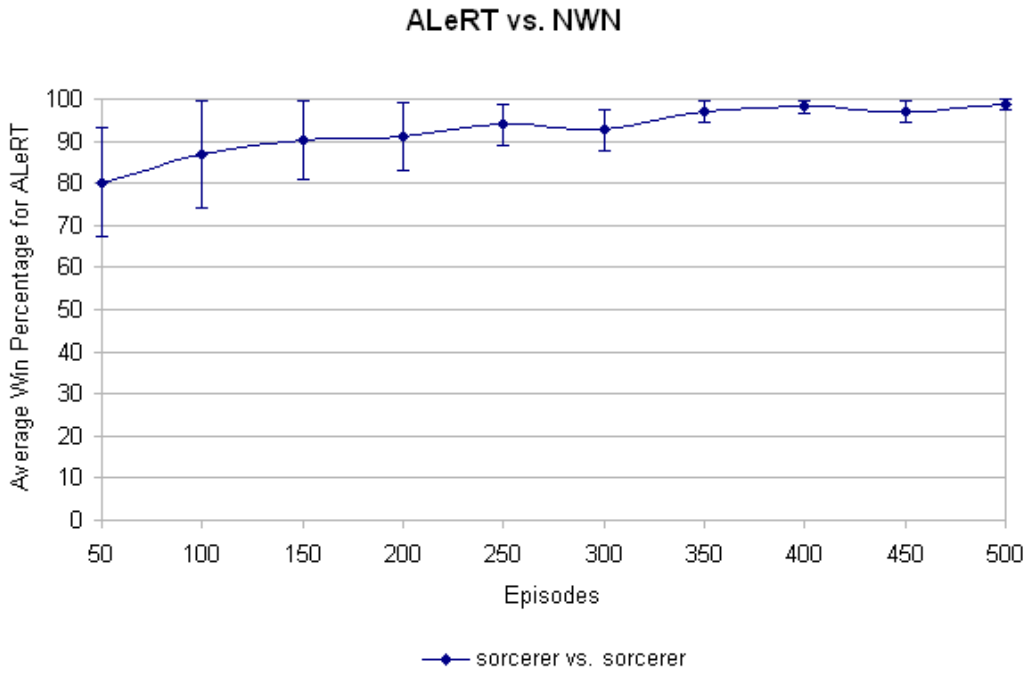

Figure 3.5: ALeRT against NWN for the sorcerer team.

The Y-axis on the graph represents the average number of episodes that the former team (in this case ALeRT) has won, as a percentage of the total number of episodes finished. The X-axis represents the number of episodes finished in the experiment, in 50-episode increments. Recall that each data point is the average of 10 trials. The bar on each data point represents one standard deviation of the 10 trials.

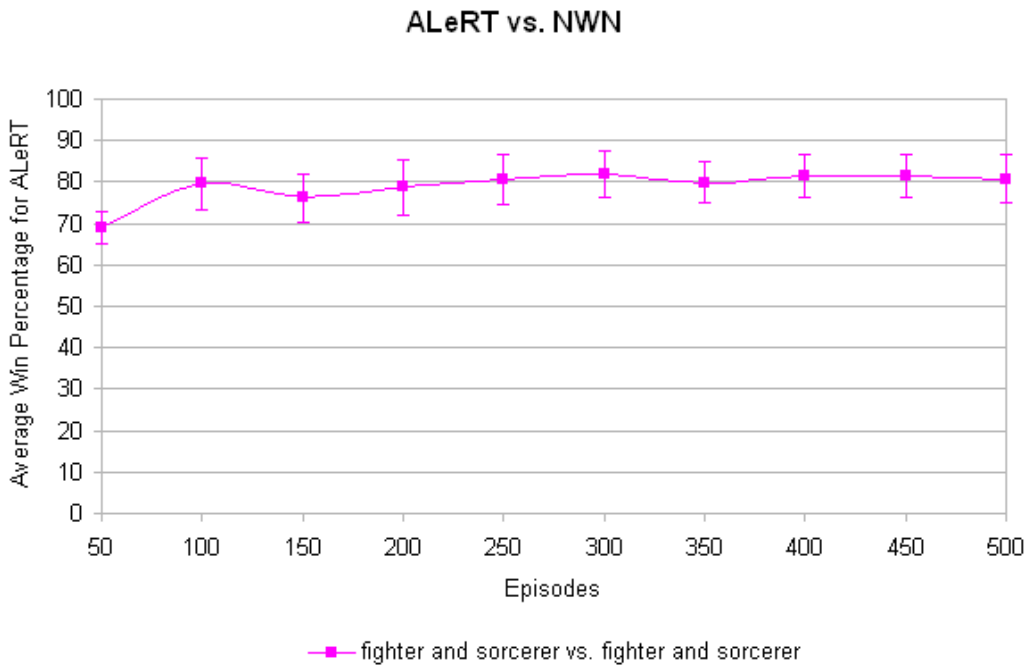

Figure 3.6: ALeRT against NWN for the fighter-sorcerer team.

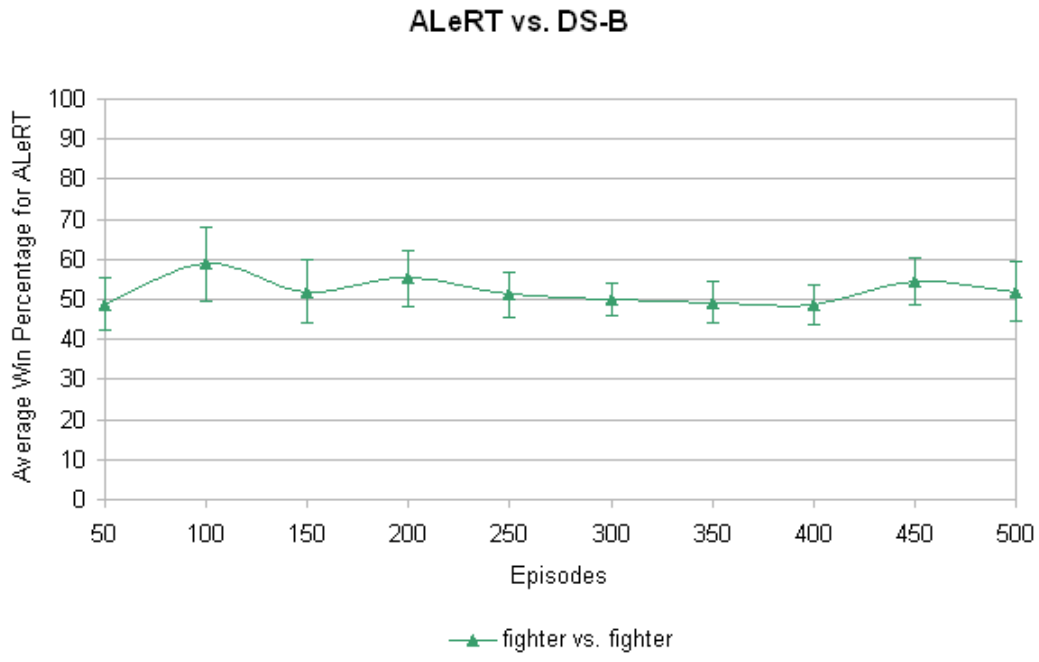

Figure 3.7: ALeRT versus DS-B for the fighter team.

Figures 3.7 to 3.9 show the results of ALeRT vs. DS-B, for the same teams. For the fighter team and the fighter-sorcerer team, both ALeRT and DS-B are able to reach an optimal strategy fairly quickly, resulting in a tie. The optimal strategy in this case consists of drinking a speed-enhancement potion and repeatedly hitting the opponent with a sword. This is the strategy to achieve the best results regardless of what the opponent does.

In the case of the sorcerer team, the results have higher variance with an ALeRT winning rate that drops to about 50% as late as episode 350. Why is the variance for the sorcerer team higher than the variance for the fighter-sorcerer team? It turns out that these results depend on whether an optimal strategy can be found that is independent of actions of the opposing team. A lone fighter has a simple optimal strategy, which does not depend on the opponent. In the fighter-sorcerer team, the optimal strategy is for both the fighter and sorcerer to focus on killing the opposing sorcerer first, and then the problem is reduced to fighter vs. fighter.

However, with a sorcerer team, there is no single static best strategy, as for every action, there is a counter-action. For example, casting a Minor Globe of Invulnerability will render a Fireball useless, but the Minor Globe is ineffective against a physical attack. The best strategy depends on the opponent's actions and the ability to predict the opponent's next action is crucial. If a best strategy can be found, it should consistently beat other strategies with lower variance. Table 3.4 shows the expected damage for each type of attack against potential defences.

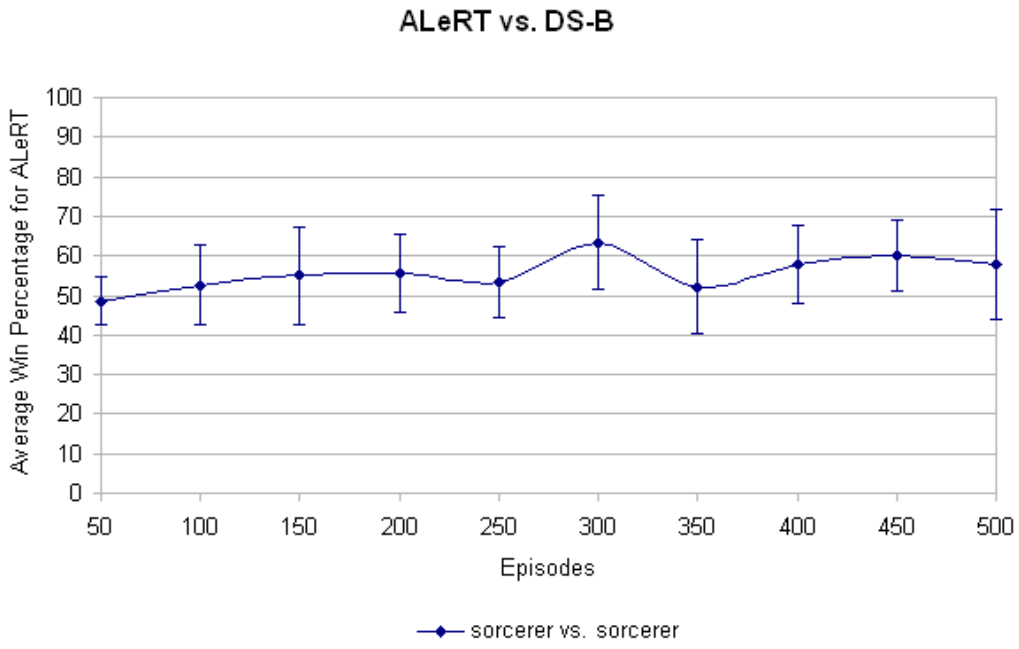

Figure 3.8: Motivation for ALeRT-AM: ALeRT versus DS-B for the sorcerer team.

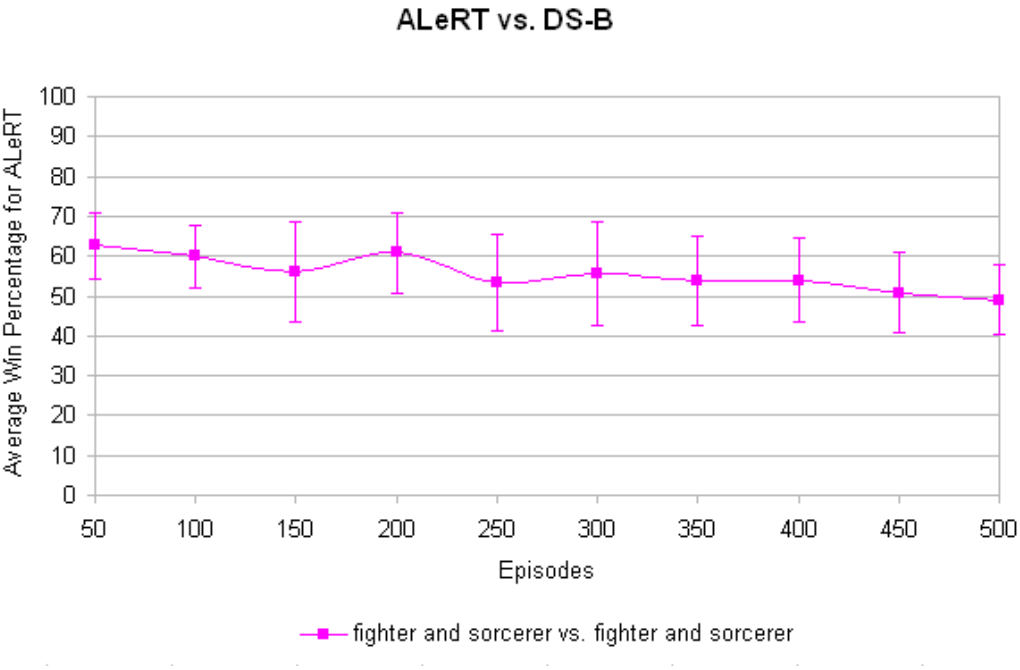

Figure 3.9: ALeRT versus DS-B for the fighter-sorcerer team.

Action	Expected Damage
Sorcerer attacks opposing sorcerer with "Dagger +2" (Attack melee)	3.2175 hit points
Sorcerer attacks opposing sorcerer with "Light Crossbow +1" (Attack ranged)	3.465 hit points (if in melee range of an opponent, invokes an "attack of opportunity", which means the opponent can make an extra attack automatically)
Sorcerer casts Fireball on opposing sorcerer (Cast offensive-area spell)	16 hit points (if opponent does not have the Minor Globe of Invulnerability effect or the Resist Elements effect)
Sorcerer casts Fireball on opposing sorcerer (Cast offensive-area spell)	1.5 hit points on the first hit only (if opponent does not have the Minor Globe of Invulnerability effect but has the Resist Elements effect)
Sorcerer casts Fireball on opposing sorcerer (Cast offensive-area spell)	0 hit points (if opponent has the Minor Globe of Invulnerability effect)
Sorcerer casts Magic Missile on opposing sorcerer (Cast offensive-single spell)	14 hit points (if opponent does not have the Minor Globe of Invulnerability effect or the Shield effect)
Sorcerer casts Magic Missile on opposing sorcerer (Cast offensive-single spell)	0 hit points (if opponent has the Minor Globe of Invulnerability effect or the Shield effect)

Table 3.4: Expected damage dealt by selected actions based on *NWN* rules.

3.5.2 Agent Modeling

With agent-modeling, ALeRT-AM achieves an approximately equal result with ALeRT when battling against the default NWN strategy (Figures 3.10 and 3.11). The sorcerer team defeats NWN with a winning rate above 90% while the fighter-sorcerer team achieves an 80% winning rate. Both ALeRT and ALeRT-AM can find the same winning strategy against the static NWN strategy. ALeRT-AM finds the winning strategy just as fast as ALeRT despite the fact that it has a more complex algorithm.

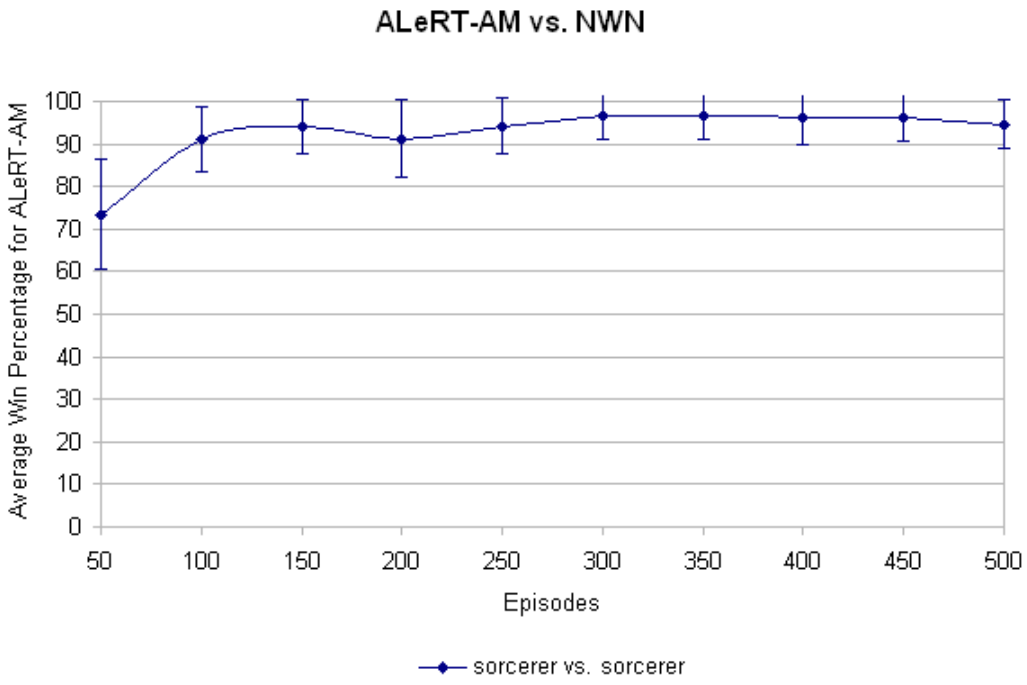

Figure 3.10: ALeRT-AM against NWN for the sorcerer team.

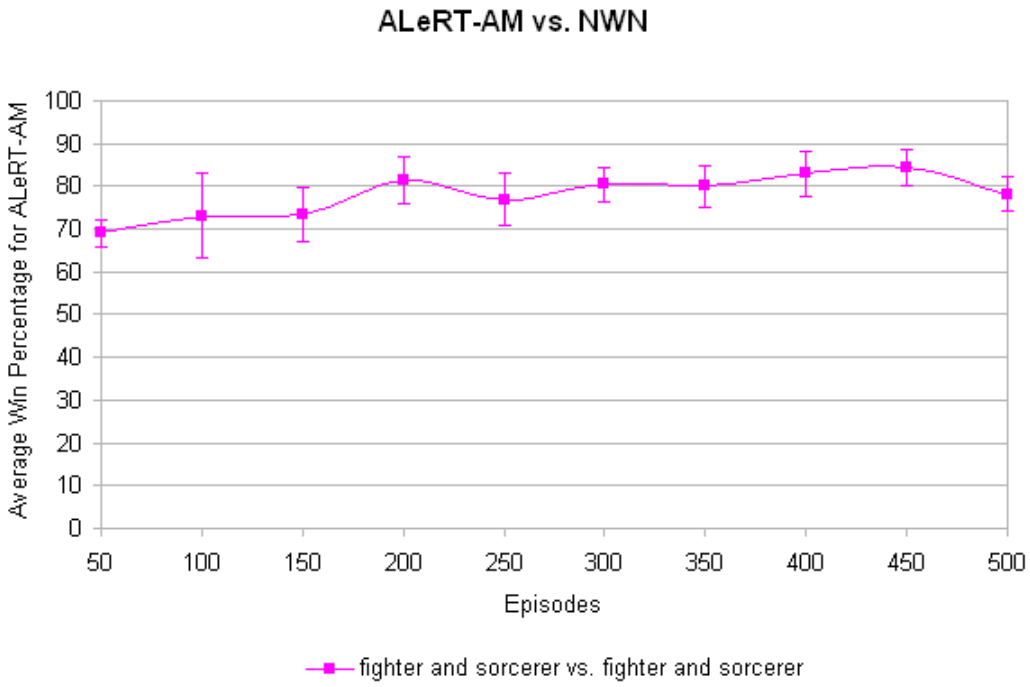

Figure 3.11: ALeRT-AM against NWN for the fighter-sorcerer team.

ALeRT-AM performs better than ALeRT against DS-B (Figures 3.12 and 3.13). The results for the sorcerer team have much lower variance. For the sorcerer team, ALeRT-AM derives a model for the opponent in less than 100 episodes and is able to keep its winning rate consistently above 62%. At one standard derivation, ALeRT-AM is clearly winning against DS-B, while ALeRT (Figure 3.8) is not. For the fighter-sorcerer team, ALeRT-AM does better than ALeRT against DS-B by achieving and maintaining a winning rate of 60% by episode 300. At one standard derivation, the winning rate is still generally above 50%.

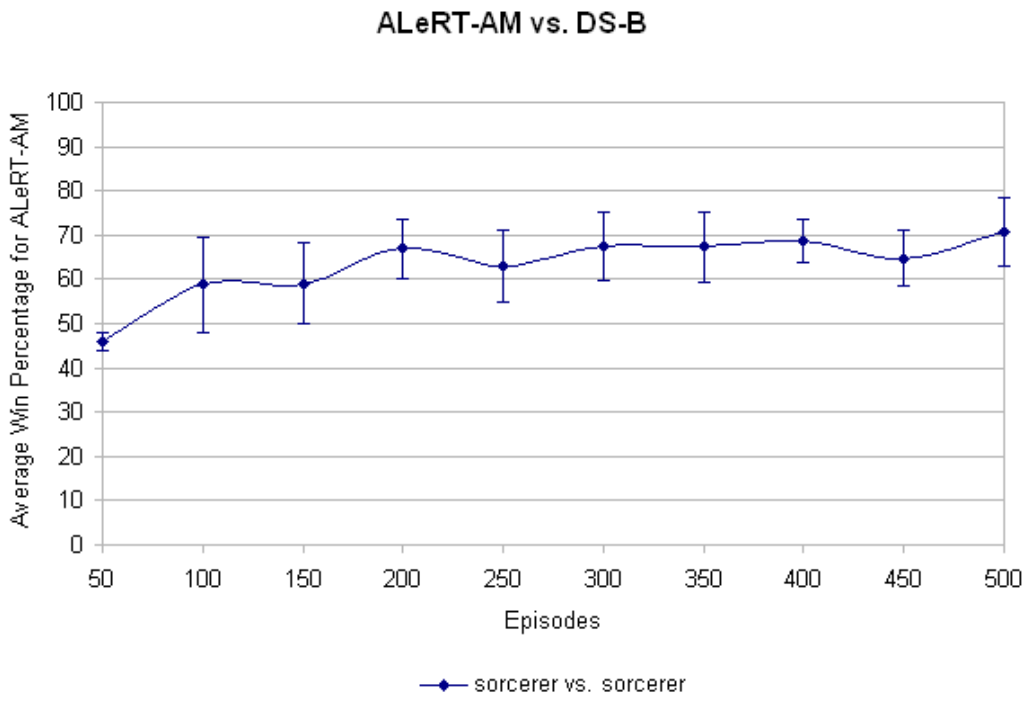

Figure 3.12: ALeRT-AM versus DS-B for the sorcerer team.

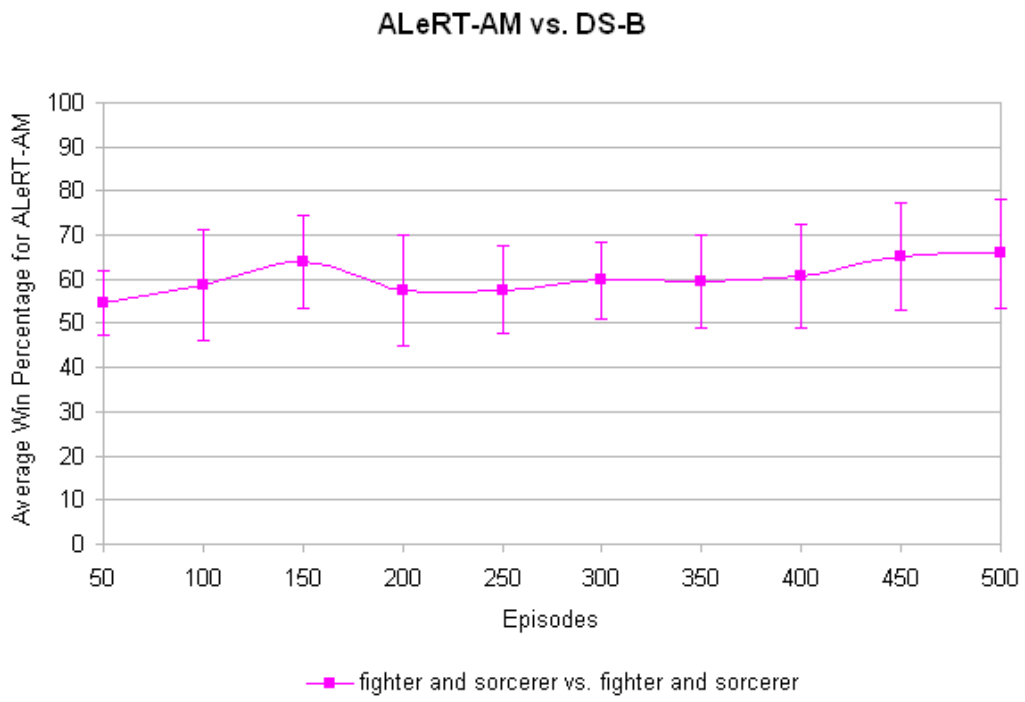

Figure 3.13: ALeRT-AM versus DS-B for the fighter-sorcerer team.

The ALeRT-AM algorithm was also tested again the original ALeRT algorithm. Figures 3.14 and 3.15 show the results for the sorcerer teams and the fighter-sorcerer teams. ALeRT-AM has an advantage at adapting to the changing strategy of ALeRT, generally keeping the winning rate above 60% and quickly recovering from a disadvantaged strategy, as shown near episode 400 in the sorcerer vs. sorcerer scenario (a turning point on the graph). A z-test to determine whether ALeRT-AM had a higher winning percentage (above 50% in Figures 3.14 and 3.15) was successful at 95% confidence for all episode sets after episode 200 (except for one point at episode 400) for the sorcerer team, and after episode 300 for the fighter-sorcerer team.

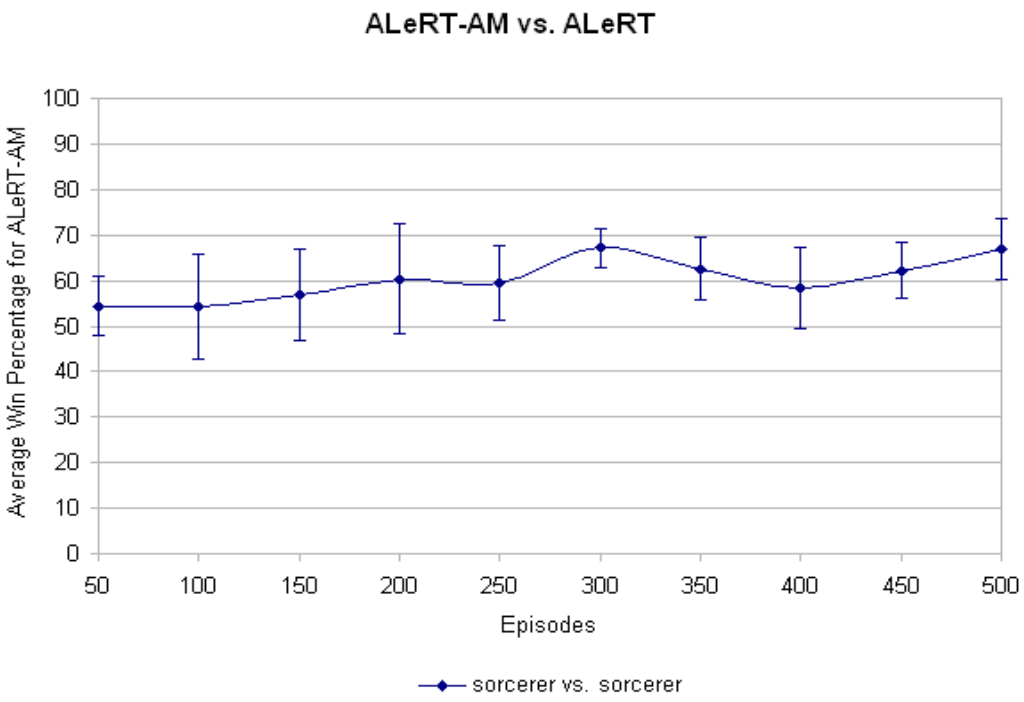

Figure 3.14: ALeRT-AM versus ALeRT for the sorcerer team.

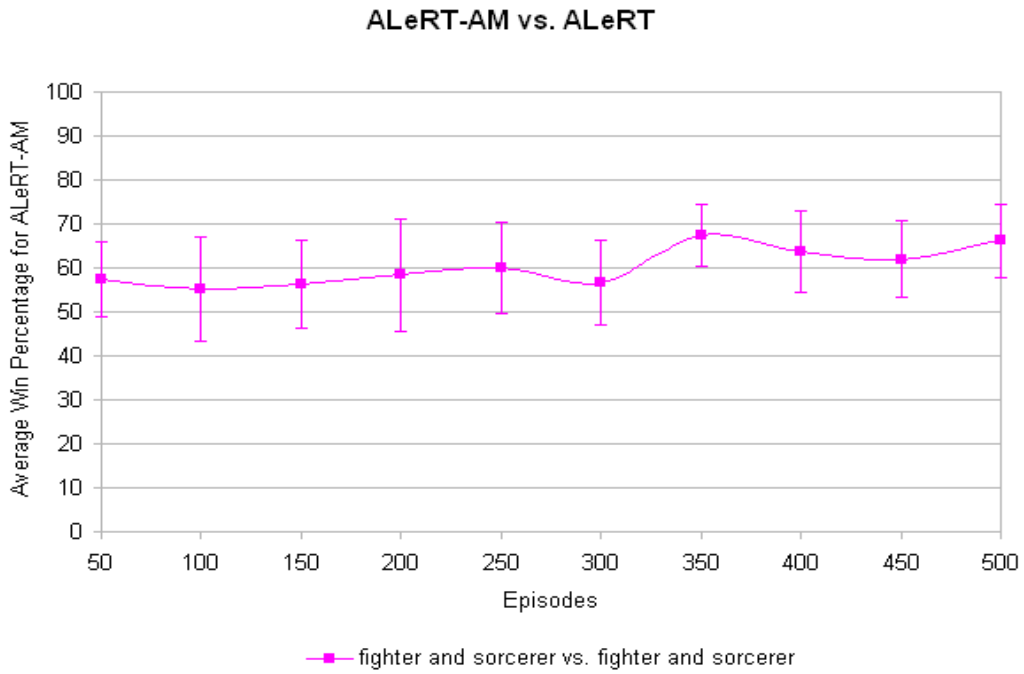

Figure 3.15: ALeRT-AM versus ALeRT for the fighter-sorcerer team.

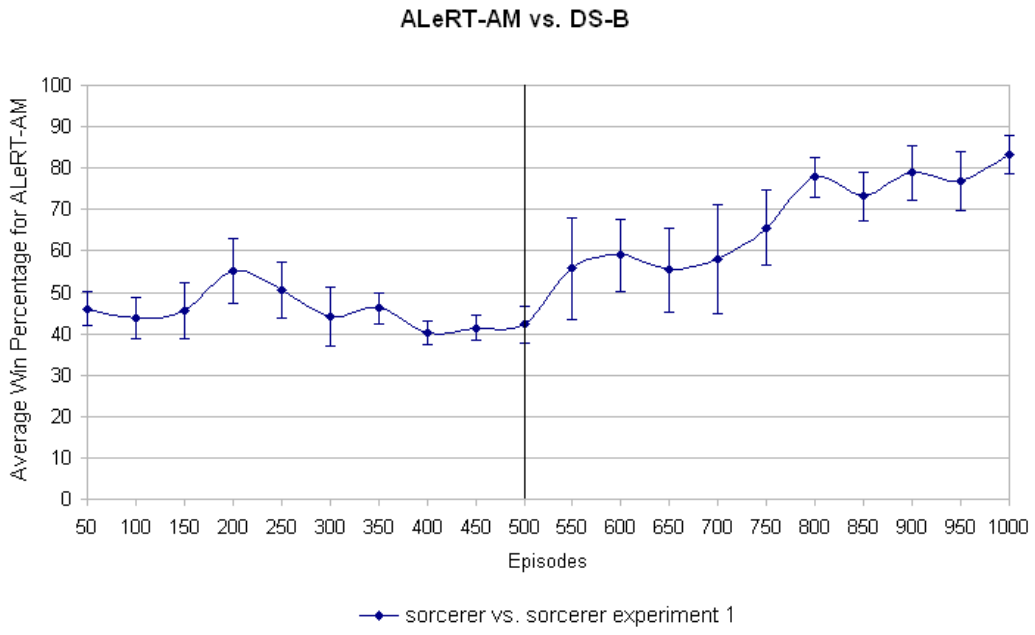

Figure 3.16: ALeRT-AM versus DS-B in a changing environment, experiment 1.

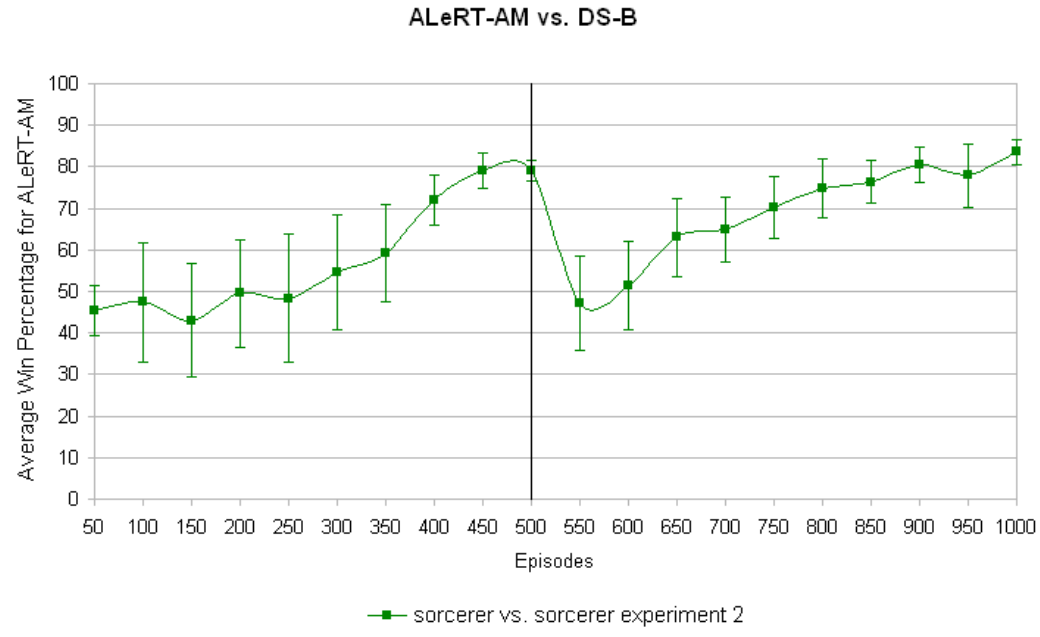

Figure 3.17: ALeRT-AM versus DS-B in a changing environment, experiment 2.

3.5.3 Adaptation in a Dynamic Environment

ALeRT was shown to be adaptable to change in the environment for the fighter team [5] by changing the configuration at episode 501 (the second phase). For a fighter team, a better weapon was given in the second phase. I demonstrate that the adaptability remains with agent modeling (Figures 3.16 and 3.17). For a sorcerer team, the new configuration has higher-level spells. I am interested in the difficult sorcerer case and two sets of experiments were performed. In the first set of experiments, both sorcerers have the Fireball spell, but no Minor Globe of Invulnerability in the first phase. In the second phase, both sorcerers gain the Minor Globe of Invulnerability spell. In the second set of experiments, both sorcerers have the Shield and the Magic Missile spells in the first phase. In the second phase, both sorcerers gain the Fireball and the Minor Globe of Invulnerability spells. In the first set of experiments, for the first 500 episodes, the single optimal strategy is to always cast Fireball, since no defensive spell is provided that is effective against the Fireball, and both ALeRT-AM and DS-B find the strategy quickly, resulting in an approximate tie. DS-B has a slightly higher winning rate due to the epsilon-greedy exploratory actions of ALeRT-AM and the fact that in this first phase, no opponent modeling is necessary, since there is a single optimal strategy. After gaining a new defensive spell against the Fireball at episode 501, there is no longer a single optimal strategy. In a winning strategy, the best next action depends on the next action of the opponent. The agent model

is able to model its opponent accurately enough so that its success continuously improves and by the end of episode 1000, ALeRT-AM is able to defeat DS-B at a winning rate of approximately 80%. In the second set of experiments, both the first 500 episodes and the second 500 episodes require a model of the opponent in order to plan a counter-strategy, and ALeRT-AM clearly shows its advantage over DS-B in both phases.

In the context of a real game, the speed of adaptation is a genuine concern. Both Figure 3.16 and 3.17 show that once a change in the environment is introduced, ALeRT-AM very quickly starts to explore possible new actions while remaining competitive against the opponent (at episode 550, ALeRT-AM is winning at approximately 50% within one standard deviation). Within another 200 episodes, ALeRT-AM dominates the opponent. This turn-around time can be shortened in a real game by having multiple NPCs utilizing the learning algorithm at the same time. Using the concept of memetic intelligence introduced in Chapter 1, these NPCs would share their learning experience amongst themselves. Since every episodic action produces a reward, it makes no difference which NPC actually performed the action, as long as the NPCs are of the same class and type.

3.5.4 Observations

Although ALeRT-AM has a much larger feature space than ALeRT (with sixty-four extra features for a sorcerer, representing the pairs of eight features for the

agent and eight features for the opposing agent), its performance does not suffer. In a fighter-sorcerer team, the single best strategy is to kill the opposing sorcerer first, regardless of what the opponent is trying to do. In this case, ALeRT-AM performs as well as ALeRT in terms of winning percentage for all three teams. Against the default static NWN strategy, both ALeRT and ALeRT-AM perform exceptionally well, quickly discovering a counter-strategy to the opposing static strategy, as is the case with the sorcerer team and the fighter-sorcerer team. I have also shown that ALeRT-AM does not lose the adaptivity of ALeRT in a changing environment where the agents' configuration changes.

In the sorcerer team, where the strategy of the sorcerer depends heavily on the strategy of the opposing agent, ALeRT-AM has shown its advantages. Both against the rule-based learning agent DS-B and the agent running the original ALeRT algorithm, ALeRT-AM emerges victorious and it is able to keep its victory by quickly adapting to the opposing agent's strategy.

When implementing the reinforcement learning algorithm for a game designer, the additional features required for agent-modeling will not cause additional work. All features can be automatically generated from the set of actions and the set of game states. The set of actions and the set of game states are simple and intuitive, and they can be reused across different characters in story-based games. In the next chapter, I will show how the learning algorithm can be implemented using behaviour patterns in ScriptEase and how it can be applied to various situations.

Chapter 4

Behaviours in ScriptEase

Recall from Chapter 2, ScriptEase is a pattern-based scripting tool developed by researchers at the University of Alberta. The current implementation of ScriptEase is able to generate scripting code for *Neverwinter Nights* (*NWN*) only, although the concept of pattern-based scripting is universal across story-based games.

A custom-made *NWN* story file called module000.mod has only one area inside a city. There is a door called MetalDoor at the side of a house, as shown in Figure 4.1.

ScriptEase uses a hierarchy of patterns to represent interactions in a story file. The basic components are *events*, *definitions*, *conditions,* and *actions*. Figure 4.2 shows a screenshot of the ScriptEase Pattern Builder with module000.mod opened for editing. It contains an example of each the above four types of components.

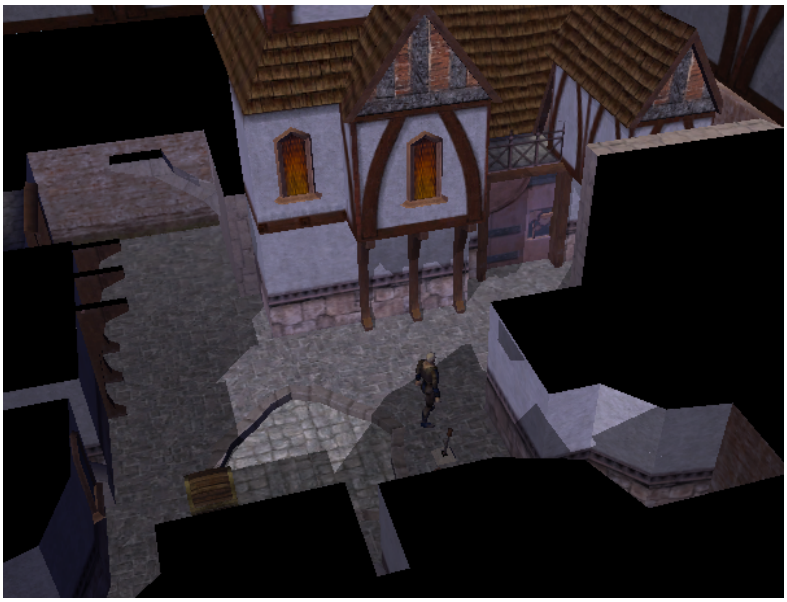

Figure 4.1: Screenshot of the module000.mod game story in *NWN*.

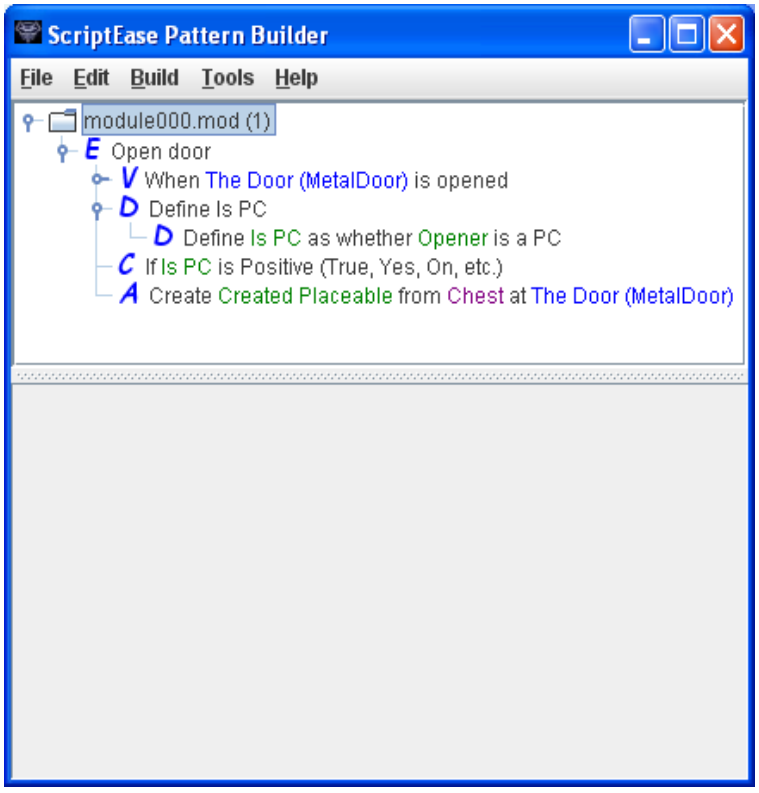

Figure 4.2: ScriptEase Pattern Builder.

An event is shown in Figure 4.2 as the line that starts with a blue V. The letter V is used to represent an event since the letter E is used to represent an encounter – the high level pattern that contains the event. This event happens when the specific door (MetalDoor) is opened. The first line that starts with a blue D is a definition block with a list of definitions. In this case, there is only one definition in the list called "Is PC", which can have the two answer: "Yes" if the opener of the door is a PC, or "No" otherwise. A condition is shown on the next line with a blue C. A condition specifies a requirement. In this case, "If '*Is PC*' is Positive" requires that the definition "Is PC" has the answer "Yes", or in plain English, the opener of the door is indeed a PC. The final line starting with a blue A represents an action. This action creates a Chest placeable object at the door. A placeable object, or simply a placeable, is any object that can be placed onto the game world using a map builder (the *NWN* map builder provided by BioWare is called the Aurora Toolset).

The event, the definition, the condition and the action are all under a line starting with a blue E. This is an *encounter* which encapsulates all these components into a single pattern called "Open door". An encounter is an interaction between objects in the game. In this specific example, the interaction is between a PC who opens the door and the door. The whole encounter, written in plain English, is: when a PC opens the MetalDoor, the game creates a Chest at this door.

An *encounter pattern* is an abstraction of a specific encounter. The abstracted encounter pattern for the encounter shown in Figure 4.2 is: when someone opens a door, something happens in the game. An abstract encounter pattern can then be used in many situations. ScriptEase has a library of pre-built patterns so that game designers can pick and choose the desired abstract patterns to create their story. The encounter pattern picker is shown in Figure 4.3. On the left side, game designers can pick the desired objects and on the right side, encounter patterns are grouped into categories for easy access. Once a designer selects an encounter pattern, the pattern instance is adapted for the particular story by setting options such as which door to open for the "Open door" pattern. Other adaptations include adding actions such as the "create chest" action. The encounter pattern is one of the four pattern types in ScriptEase. The other three, behaviour patterns, quest patterns, and dialogue patterns, are all more sophisticated patterns building upon the encounter pattern. The focus of my thesis is on extending the capabilities of behaviour patterns.

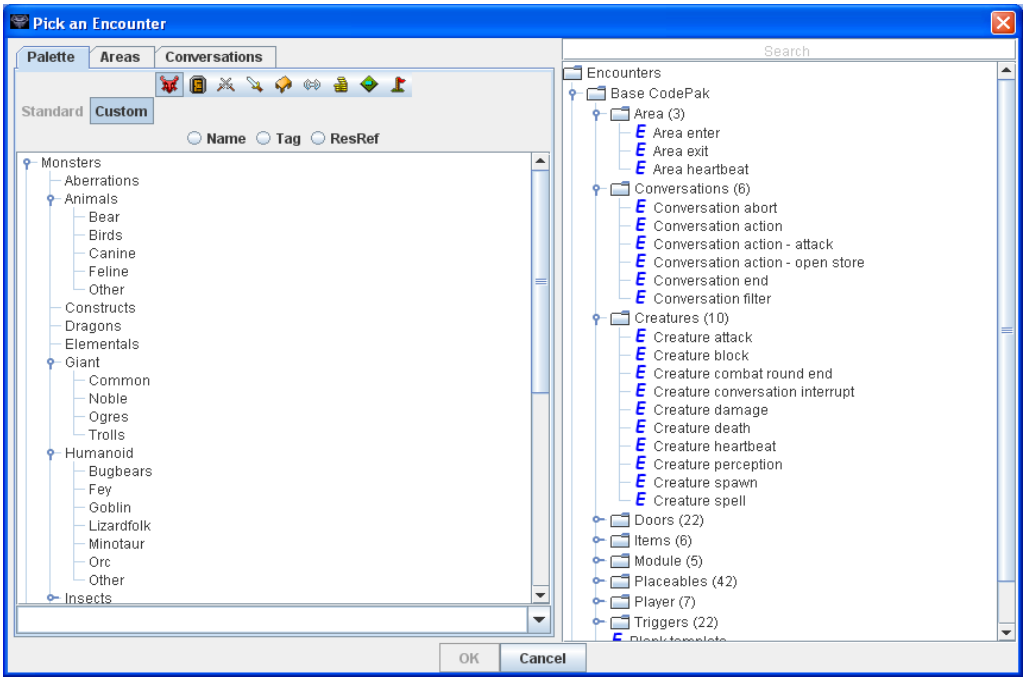

Figure 4.3: The encounter pattern picker.

4.1 Categories of Behaviours

Behaviour patterns define how NPCs interact with the PC, other NPCs, and other objects in the game. An NPC with a basic behaviour is mentioned in Chapter 1 as a wanderer, who simply wanders around in the game world. This NPC does not make an interesting character. A more interesting NPC should have multiple behaviours and the NPC should choose intelligently which behaviour to perform based on events happening in the game world and the NPC's internal state.

ScriptEase tackles the behaviour problem by categorizing behaviours in two

ways. A behaviour that happens in the background when nothing else important to the NPC is happening is called a *proactive* behaviour since it is initiated by that NPC. For example, one NPC in a game is a little girl. When the game starts, the girl runs around as a small child would usually do. This is a proactive behaviour since she is not doing it in response to an event in the game or another NPC. On the other hand, when a PC approaches her, if she stops running and waves her hand at the PC, her hand-waving behaviour would be called a *latent* behaviour. A latent behaviour is a behaviour triggered by an event in the game. In the example above, the approaching of the PC is the event that triggers the girl's latent behaviour. A latent behaviour always has higher priority and interrupts a proactive behaviour.

Behaviours are also divided into *independent* behaviours and *collaborative* behaviours. With independent behaviours, the NPC is able to perform all the necessary actions alone. The girl in the example above has two independent behaviours, the running-around *Wander* behaviour and the hand-waving *Exclaim* behaviour. Although the exclaim behaviour is triggered by another character, the PC, the behaviour itself can still be done independent of the PC (*i.e.* the PC can totally ignore the girl and it would not affect the girl's ability to wave her hand). If upon seeing the PC, the girl gets scared, runs away and returns with her dad (assuming her dad is another NPC in the game), this behaviour would be a *collaborative* behaviour. A collaborative behaviour has to be completed with a partner NPC. The girl would not be able to complete this behaviour without a

cooperating dad. The girl's calling-for-help behaviour is called a *latent collaborative* behaviour since it is initiated by an external event, the arrival of the PC. The dad's answering-help behaviour is called a *reactive collaborative* behaviour since he only performs this behaviour to react to a partner who initiated it. Since all reactive behaviours are collaborative, we simply refer to them as reactive behaviours.

Combining the proactive/latent categorization and the independent/collaborative categorization, there are five types of behaviours in ScriptEase: proactive independent, latent independent, proactive collaborative, latent collaborative, and reactive [4]. Figure 4.4 shows the different types in a tree diagram.

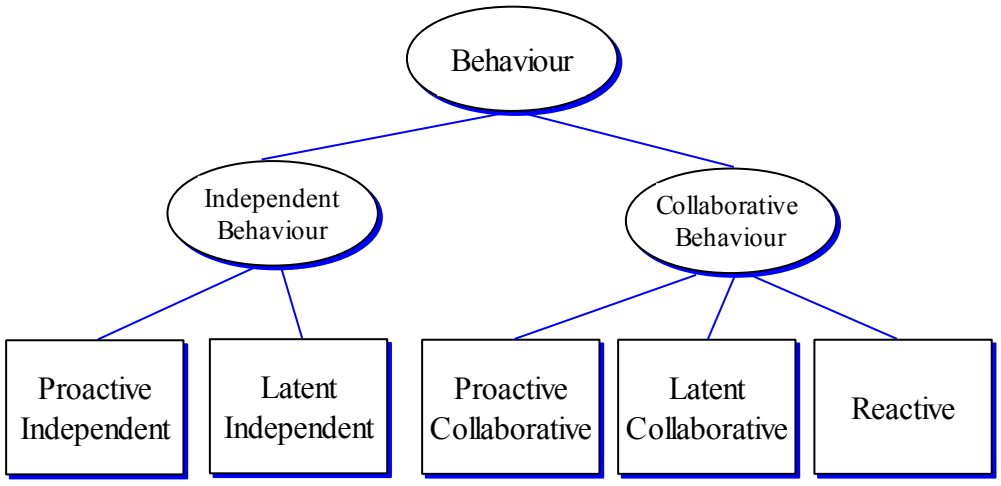

Figure 4.4: Conceptual behaviour types. The names in rectangular boxes are the actual names used in ScriptEase.

ScriptEase has a library of basic behaviours (refer to Appendix A). In the example mentioned above, Wander and Exclaim are two basic behaviours. Basic behaviours are implemented using a sequence of *tasks* which are in turn made up of actions. A task is a sequence of actions to be completed in order. If a task is interrupted before all its actions finish, when the behaviour resumes the interrupted task will start from the first action in its sequence again. The resumable nature of behaviours allows NPCs to finish their work even after an interruption by the PC or another character. The details of the resume mechanism are outside the scope of this thesis and are described by Maria Cutumisu [4].

The basic behaviours do not restrict themselves to be any of the five ScriptEase behaviour types. This is done so that a single basic behaviour can be used in multiple situations (latent, proactive, collaborative, etc.). The Wander behaviour can be independent proactive if the NPC wanders simply when nothing else is going on, or independent latent if the NPC wanders because of an explosion that stuns the NPC. Recall that reusability is one of the main measures of a scripting tool. Allowing general behaviours to be customized by game designers in various situations is important in maintaining reusability. With that said, the mechanism of putting basic behaviours into the five behaviour types uses *cues*.

4.2 Behaviour Cues and Role Cues

Behaviour cues are used to control the selection of proactive behaviours and

trigger latent behaviours based on events [4]. There are five types of behaviour cues in ScriptEase, each corresponding to one type of behaviour: proactive independent cue, latent independent cue, proactive collaborative cue, latent collaborative cue, and reactive cue. A behaviour cue can contain multiple basic behaviours, but all basic behaviours contained in a cue are considered behaviours of that type. Figure 4.5 shows two behaviour cues in ScriptEase. A cue is represented by a blue > while a basic behaviour is represented by a blue B. There is a proactive independent cue with two basic behaviours, Wander and Patrol, as well as a latent independent cue with an Exclaim behaviour. Since there are more than one behaviour in the proactive independent cue, one is chosen randomly. This latent cue triggers when the MetalDoor is opened. The basic behaviours in the proactive independent cue become proactive independent behaviours and the ones in the latent independent cues become latent independent behaviours.

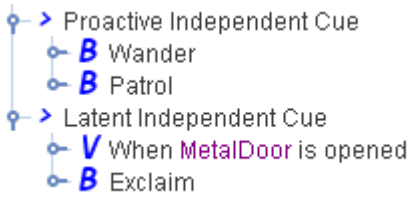

Figure 4.5: An example of two behaviour cues with three basic behaviours.

An NPC can have multiple types of behaviours, thus ScriptEase has to have the ability to put multiple behaviour cues on an NPC. This is done through a *role*. Figure 4.6 shows a role (represented by a blue R) with three behaviour cues. The

actor who uses this role, the NPC, can Wander, Rest, or Loiter. The blue M denotes a motivation. A motivation determines the dynamic probabilities of selecting each proactive cue. In this example, since there is no motivation, proactive cues are selected randomly, using static probabilities assigned by the game designer. If there is a latent cue, then when the event associated with the latent cue happens, the latent cue will always be chosen over the proactive cues.

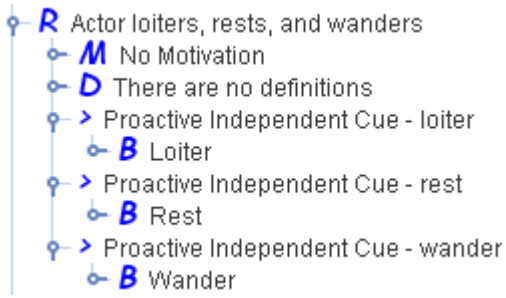

Figure 4.6: An example of a role containing three behaviour cues.

An NPC can also assume different roles during the course of a game. A simple example is by time. In the morning, an NPC who is a city guard may assume the role of a guard. By night, when the duty is done, the NPC may go to a tavern and assume the role of a tavern patron. Putting multiple roles on an NPC is done through a *performance*, the highest level behaviour pattern. A performance (represented by a blue P in ScriptEase) contains all the roles an NPC can take in the entire game. Using a performance, an NPC can take on different roles activated by *role cues*. Role cues are similar to behaviour cues, except they activate roles instead of basic behaviours. A role cue can contain multiple roles,

but if there are multiple roles, all cues inside the roles are combined into a single role. All role cues are latent independent, which means they are activated based on an event in the game (e.g. a certain time of the day has arrived). Figure 4.7 shows a complete performance of a guard.

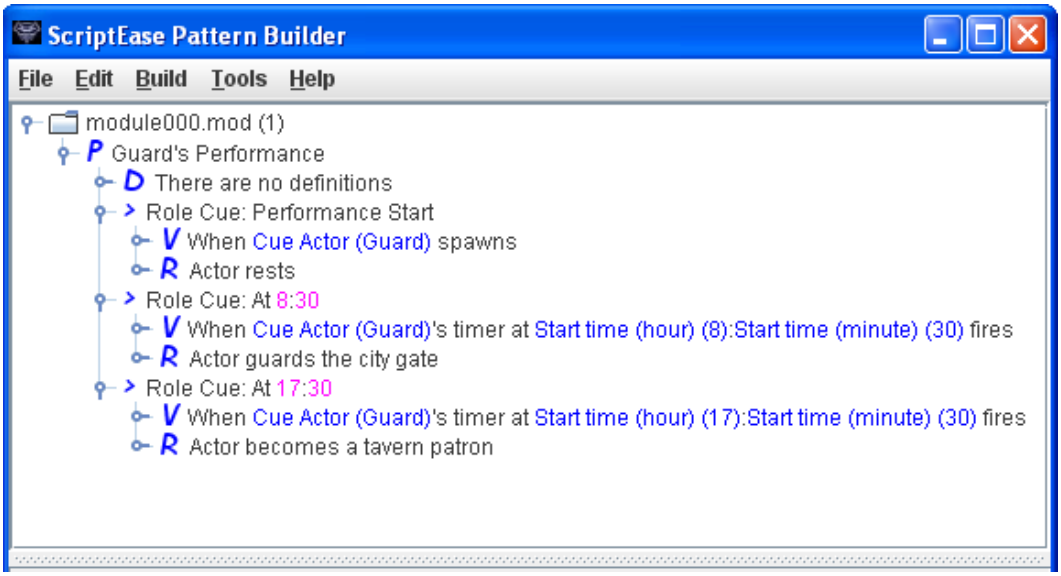

Figure 4.7: The performance of a guard.

There are several special latent behaviour cues and role cues. One is seen in Figure 4.7 as the *performance start* role cue. This role cue is present in every performance, and its associated event is the creation (spawn) of the NPC. NPCs that are statically placed by a game designer will activate this cue when the game story starts. Dynamically spawned NPCs will activate this cue when they are spawned. Another special cue is the *timer* cue (seen twice in Figure 4.7 at times 8:30 and 17:30). The timer cue is trigged when a certain time of the day is

reached. There is also a *range* cue which occurs when an NPC is within a certain distance of another creature (NPC or PC as selected by the game designer). Figure 4.5 shows an example of a range cue. Any latent independent cue can be used as a behaviour cue or a role cue.

4.3 Learning Algorithms in ScriptEase

I have described the structure of a behaviour pattern in ScriptEase. As mentioned above, all basic behaviours are inside behaviour cues. The way a behaviour cue is triggered depends on its type. Proactive cues are triggered when there are no active behaviours. The proactive behaviours in all proactive cues are selected probabilistically according to weights set by the game designer for each proactive cue. Latent cues are selected based on events in the game and are given priorities in case two events happen at the same time. All proactive cues are given priority 0 and latent cues always have higher priorities than proactive cues. A reactive cue triggers when another NPC identifies this NPC as a viable collaborator.

The current cue selection policy does not accommodate reinforcement learning (RL) algorithms. To integrate RL algorithms, a special role needs to be built. I call this role the ALeRT-AM learning role.

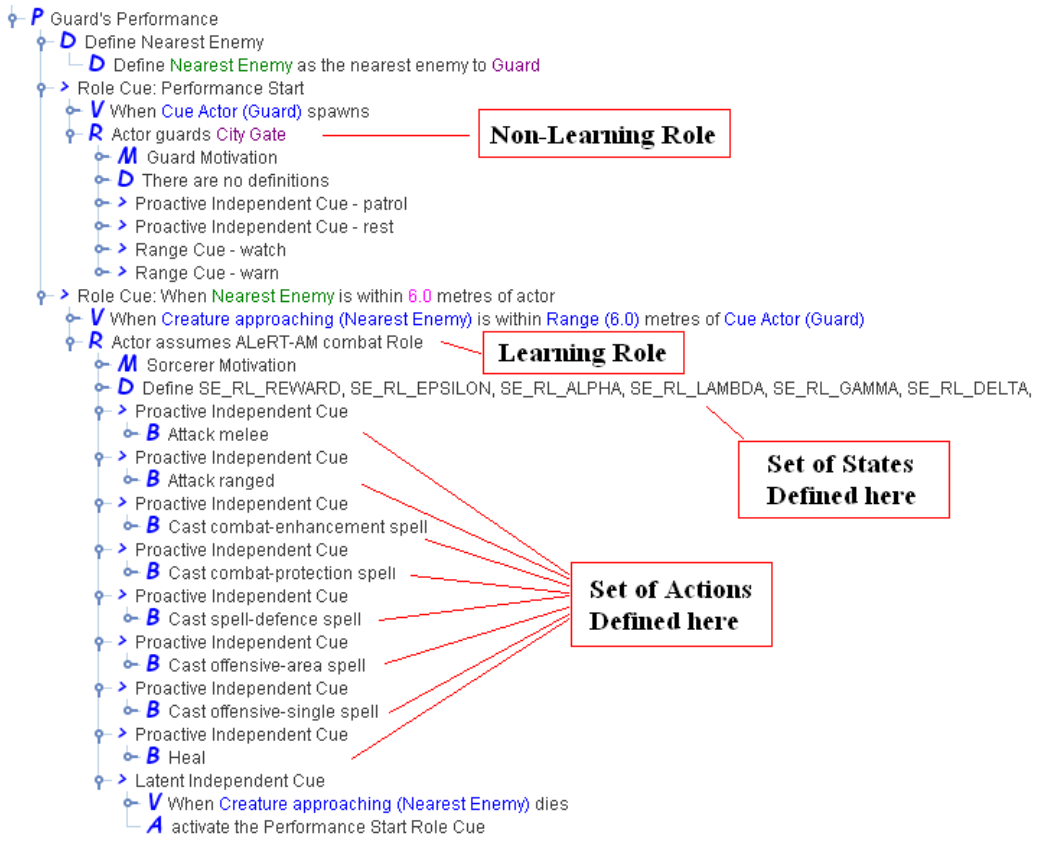

Figure 4.8: The ALeRT-AM role adapted in a combat situation with a sorcerer.

The ALeRT-AM role, shown in Figure 4.8 as the combat role, is activated by a range role cue: when an enemy is within 6 metres. The RL algorithm is embedded in this special role so that the scripting code for the learning algorithm is generated whenever this role is used. Game designers need to specify the possible actions for the NPC to take. Each of the actions for the learning agent is created as a behaviour in its own proactive independent cue, making the action a proactive behaviour. These behaviours will be used by the learning algorithm as the set of actions to choose from. Special definitions are used to create the set of

states, as well as parameters for the RL algorithm (exploration rate epsilon, learning rate alpha, etc.). All definitions are kept in the definition block (the line with "Define SE_RL_Reward, ..."). These definitions can be modified by game designers to suit their needs in various situations.

For agent-modeling in ALeRT-AM, a set of possible target-agent actions and a target-agent for the agent model can be specified through definitions as well Definitions called target-agent action definitions are used. These definitions are recognized by the learning algorithm as special definitions for the target-agent. In the specific example of an enemy sorcerer, the target-agent is defined to be an enemy sorcerer, and the set of eight actions are defined. This role is also created so that if no target-agent definitions are found, regular non-agent modeling ALeRT will be used as the learning algorithm.

With the integrated framework of the learning algorithm and ScriptEase behaviour patterns, it will take game designers relatively little effort to apply a learning algorithm to an NPC in game, thus paving the way for the creation of NPCs with more realistic behaviours. Although the implementation of the ALeRT-AM learning algorithm is currently done in the game *NWN* only, the architecture can be similarly applied to future story-based games, such as Bioware Corp.'s *Dragon Age: Origins* [8], to be released in 2009.

Chapter 5

Conclusions and Future Work

This chapter summarizes the topics of the thesis, discusses possible future work, and concludes by stating the contributions of the thesis.

5.1 Summary

It is increasingly important to have more realistic NPC behaviours in story-based games. Commercial game companies and the academic community have made various attempts at building NPCs with believable behaviours, with disadvantages in different situations. The pattern-based ScriptEase tool provides a new generative model for building NPC proactive, reactive and latent behaviours, with an implementation in the commercial game *Neverwinter Nights* (*NWN*). Furthermore, a general purpose reinforcement learning algorithm ALeRT has been proposed to give NPCs using ScriptEase behaviour patterns, the ability to adapt to changes in the gaming environment.

To demonstrate the effectiveness of ALeRT, it was implemented in a series of combat experiments between fighter NPCs in *NWN*. While the fighter NPC using

ALeRT was able to find a winning strategy quickly against fighters using other strategies, when I implemented ALeRT on NPCs of other classes such as sorcerers, I could not replicate such good performance. The reason is that with fighter NPCs, the winning strategy depended only on the actions of the learning agent itself. With sorcerer NPCs, ALeRT did not give consistent results when the winning strategy included actions that are dependent on the opponent's actions. I hypothesized that building an opponent model, or more generally, an agent model for NPCs in a game is an important factor in NPC behaviours. I modified the learning algorithm ALeRT to incorporate agent modeling and evaluated the new algorithm ALeRT-AM by modeling opponent agents in combat, in the same scenarios where ALeRT failed to achieve my expectations. The ALeRT-AM algorithm improved upon ALeRT by allowing the NPC using the learning algorithm to construct a model for the opponent and then use this model to predict the best action it could take against that opponent. In my experiments, the NPCs using ALeRT-AM succeeded in taking into consideration opposing NPCs' actions. ALeRT-AM achieved better results than ALeRT when competing against the same opponents, and ALeRT-AM emerged victorious when competing head-to-head against ALeRT. I have shown that the ALeRT-AM algorithm exhibits the same kind of adaptability that ALeRT exhibits when the environment changes.

I have also integrated the ALeRT-AM algorithm into the ScriptEase framework. With special ScriptEase behaviour patterns automatically generating scripting code for the ALeRT-AM algorithm, game designers can apply the

learning algorithm without having to worry about the technical details. Although my experiments focused on modeling opponent agents in combat, the same algorithm can also be applied beyond combat since game designers need only to use the behaviour pattern, and supply a set of states for the desired situations in the game, a set of legal actions the NPCs can take and a way to assign rewards to the NPCs.

While designing a game, all the learning of NPC behaviours need not happen from zero after a player starts the game. An initial learning phase can be done off-line at design time before the game is made available to players, so that when a player starts the game, the initial behaviours of NPCs are reasonable and as the game progresses, the NPCs will be able to adapt quickly to the changing environment. A typical story-based game consists of hundreds or thousands of NPCs. Using the concept of memetic intelligence proposed in Chapter 1, the learning experience can also be automatically shared among NPCs of the same type in the game, thus greatly reducing the time required for the learning algorithm to adapt.

5.2 Future Work

I developed the agent-modeling ALeRT-AM algorithm and explored situations with a single learning agent (fighter or sorcerer) and a team of two cooperating learning agents (fighter and sorcerer). This laid the groundwork for research into

multi-agent cooperations. Maintaining agent models for many different types of NPCs can be a challenging task.

The effectiveness of the reinforcement learning ALeRT-AM algorithm was only demonstrated in a combat scenario. A combat scenario was chosen because the results can be measured directly in terms of numbers of wins and loses. Although the ALeRT-AM algorithm outperforms previous algorithms in combat, the algorithm itself is a general purpose learning algorithm which can be applied in various situations. Combat is only a very limited application in games and as future work, non-combat situations can be explored. For example, the agent modeling technique can be used by a companion NPC to model the player character. A companion can apply the agent modeling technique effectively to predict the PC's actions and indirectly the player's habits.

Experiments can be conducted with human players to get an evaluation of the human conception of the learning agents. The player would control a character in the game and a companion would be available to the player. Two sets of experiments would be run, one with an ALeRT-AM learning companion and one with a static companion or a companion whose actions must be controlled by the PC. Players would be asked which one they prefer. Ultimately, the goal of the learning algorithm is to provide NPCs with more realistic behaviours to present a better gaming experience for players.

5.3 Conclusions

This thesis provides two important contributions. First, it introduced ALeRT-AM, a general purpose reinforcement learning algorithm that includes an agent modeling technique. With the agent model, the NPCs that employ the learning algorithm can adjust their behaviours based on what other NPCs are doing, thus providing responses that appear more intelligent. Second, the learning algorithm is integrated into ScriptEase, a pattern-based scripting tool, so that the learning algorithm can be applied by game designers who may have no knowledge of programming. This will open the door to a range of possibilities for the incorporation of learning techniques in future story-based games.

Bibliography

[1] AI in GTA IV: Nothing Spectacular. 2009. http://aipanic.com/ai-in-gta-iv-nothing-spectacular/.

[2] G. Bowditch. Grand Theft Auto producer is Godfather of Gaming. *The Sunday Times,* April 27, 2008. http://www.timesonline.co.uk/tol/news/uk/scotland/article3821838.ece.

[3] M. Cavazza. 2000. AI in Computer Games: Survey and Perspectives. *Virtual Reality,* Vol 5, No. 4: 223-235.

[4] M. Cutumisu. 2009. Using Behavior Patterns to Generate Scripts for Computer Role-Playing Games. PhD thesis, University of Alberta.

[5] M. Cutumisu, D. Szafron, M. Bowling, and R.S. Sutton. 2008. Agent Learning using Action-Dependent Learning Rates in Computer Role-Playing Games. *4th Annual Artificial Intelligence and Interactive Digital Entertainment Conference* (AIIDE-08), 22-29.

[6] M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, C. Onuczko, and M. Carbonaro. 2006. Generating Ambient Behaviors in

Computer Role-Playing Games. *IEEE Intelligent Systems*, 21(5), Sept./Oct. 2006, 19-27.

[7] R. Dawkins. 1989. The Selfish Gene – New Edition. Oxford University Press.

[8] Dragon Age: Origins. 2009. http://dragonage.bioware.com/.

[9] The Elder Scrolls III: Morrowind. 2009. http://www.elderscrolls.com/games/morrowind_overview.htm.

[10] The Elder Scrolls IV: Oblivion. 2009. http://www.elderscrolls.com/games/oblivion_overview.htm.

[11] The Entertainment Software Association. 2009. http://www.theesa.com/.

[12] K. D. Forbus. Under the hood of the Sims. 2009. http://www.cs.northwestern.edu/~forbus/c95-gd/lectures.

[13] Grand Theft Auto: Chinatown Wars. 2009. http://www.rockstargames.com/chinatownwars/.

[14] Grand Theft Auto IV. 2009. http://www.rockstargames.com/IV/.

[15] Halo 2. 2009. http://www.microsoft.com/games/halo2/.

[16] D. Isla. 2005. Handling Complexity in the Halo 2 AI. *Game Developers Conference* (GDC 05).

[17] M. MacNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford and D. Parker. 2004. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. *19th IEEE International Conference on Automated Software Engineering (ASE).* Linz, Austria, pp. 88-99.

[18] M. Mateas and A. Stern. 2003. Integrating Plot, Character and Natural Language Processing in the Interactive Drama Façade. *Technologies for Interactive Digital Storytelling and Entertainment (TIDSE)*, Darmstadt, Germany. March 24-26.

[19] The Most Expensive Game Ever. 2009. http://tektodo.com/2008/05/02/the-most-expensive-game-ever/.

[20] A. Nareyek. 2004. AI in Computer Games. *Queue,* Vol 1, No. 10: 58-65.

[21] Neverwinter Nights. 2009. http://nwn.bioware.com.

[22] NWN Arena. 2009. http://ticc.uvt.nl/~pspronck/nwn.html.

[23] C. Onuczko. 2007. Quest Patterns in Computer Role-Playing Games. M.Sc. thesis, University of Alberta.

[24] Quake. 2009. http://www.idsoftware.com/.

[25] S.T.A.L.K.E.R. 2009. http://www.stalker-game.com/.

[26] ScriptEase. 2009. http://www.cs.ualberta.ca/~script/.

[27] J. Siegel and D. Szafron. 2009. Dialogue Patterns - A Visual Language For Dynamic Dialogue. *Journal of Visual Languages and Computing,* 20 (3), 196-220.

[28] The Sims. 2009. http://thesims.ea.com/.

[29] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. 2006. Adaptive Game AI with Dynamic Scripting. *Machine Learning* 63(3): 217-248.

[30] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. 2003. Online Adaptation of Computer Game Opponent AI. In *Proceedings of the 15th Belgium-Netherlands Conference on AI,* 291-298.

[31] R.S. Sutton. 1992. Adapting Bias by Gradient Descent: An Incremental Version of Delta-Bar-Delta. In *Proceedings of the 10th National Conference on AI*, 171-176.

[32] R.S. Sutton and A.G. Barto. 1998. *Reinforcement Learning: An Introduction*. Cambridge, Mass.: MIT Press.

[33] I. Szita, M. Ponsen, and P. Spronck. 2008. Keeping Adaptive Game AI Interesting. In *Proceedings of CGAMES 2008*, 70-74.

[34] J. Tanenbaum. 2008. Believability, Adaptivity, and Performativity: Three Lenses for the Analysis of Interactive Storytelling. Master's Thesis, Simon Fraser University.

[35] T. Timuri, P. Spronck, and J. van den Herik. 2007. Automatic Rule Ordering for Dynamic Scripting. In *Proceedings of the 3rd AIIDE Conference*, 49-54, Palo Alto, Calif.: AAAI Press.

[36] A.M. Turing. 1950. Computing machinery and intelligence. *Mind*, 59: 433-460.

[37] Ultima. 2009. `http://www.uo.com/archive/`.

[38] Ultima screenshot. 2009. `http://www.mobygames.com/game/atari-8-bit/ultima/screenshots/gameShotId,310104/`.

[39] W. Uther and M. Veloso. 1997. Adversarial reinforcement learning. Tech. rep., Carnegie Mellon University. Unpublished.

Appendices

Appendix A – Basic Behaviour Catalogue

Each of the following pre-built basic behaviour pattern is from the ScriptEase pattern library.

- Approach - The actor walks near an object while speaking a random one-liner.
- Attack - The actor attacks an object while speaking a random one-liner.
- Beckon - The actor beckons a nearby creature by facing the creature and speaking a random one-liner.
- Beseech - The actor beseeches a nearby creature by walking near the creature and speaking a random one-liner.
- Cast fake spell - The actor casts a fake spell on an object while speaking a random one-liner.
- Challenge - The actor challenges a nearby creature by approaching the creature and initiating a dialogue with the creature.
- Check - The actor checks a container for an item while speaking a random one-liner.
- Converse-Listen - The actor listens to a random one-liner facing a partner, then respond.
- Converse-Talk - The actor speaks a random one-liner facing a partner, then listens to a response.
- Converse-Serve - The actor serves a drink to a customer.
- Converse-Served - The actor takes a drink from a server.
- Destroy - The actor destroys a creature casting a fake spell while speaking a random one-liner.

- Dispossess - The actor withdraws an item from a creature while speaking a random one-liner.

- Exclaim - The actor speaks a random one-liner facing a partner.

- Exclaim with animation - The actor speaks a random one-liner facing a partner while performing a random animation.

- Follow - The actor follows an object while speaking a random one-liner.

- Interact - The actor walks to an object and interacts with it (uses it) if the object can be interacted with, while speaking a random one-liner.

- Loiter - The actor walks/runs in a random direction and within a random distance of its origin location while speaking a random one-liner.

- Manipulate - The actor performs a skill on an object while speaking a random one-liner.

- Patrol - The actor patrols around a set of patrol posts with the same tag while speaking a random one-liner.

- Perform skill - The actor performs a skill on an object while speaking a random one-liner.

- Pose - The actor performs a simple animation while speaking a random one-liner.

- Rest - The actor rests on a placeable while speaking a random one-liner.

- Return - The actor returns to original location while speaking a random one-liner. The actor returns to its origin location if it is accidentally displaced during the game.

- Spawn - The actor spawns a creature casting a fake spell while speaking a random one-liner.

- Strike - The actor attacks an object repeatedly while speaking a random one-liner.

- Use - The actor walks to an object and uses it while speaking a random one-liner. The actor attempts to use a placeable, by performing one of the following actions on the placeable: bash, knock, unlock, or use.

- Vanish - The actor vanishes while speaking a random one-liner.
- Wander - The actor walks/runs a random distance in a random direction while speaking a random one-liner.
- Warn - The actor warns an intruder while speaking a random one-liner.
- Watch - The actor watches an intruder while speaking a random one-liner.
- Withdraw - The actor withdraws an item from a container while speaking a random one-liner.